DTIC FILE COPY

REPORT DOCUMENTATION PAGE

Form Approved
OPM No. 0704-0188

AD-A222 331

2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
April 1990	Release 1 - Oct 89 to Apr 90

4. TITLE AND SUBTITLE
Ada 9X Project Report
Ada 9X Revision Issues Release 1

5. FUNDING NUMBERS
MDA 903 89 C 0003

6. AUTHOR(S)
Cy Ardoin, Ph.D., Paul Baker, Ph.D., Robert Dewar, Ph.D., Doug Dunlop, Ph.D., Paul Hilfinger, Ph.D., Joseph Linn, Ph.D., Reg Meeson, Ph.D., Steve Michell, Karl Nyberg, Olivier Roubine, Ph.D.

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Institute for Defense Analyses
1801 North Beauregard Street
Alexandria, Virginia 22311

8. PERFORMING ORGANIZATION REPORT NUMBER

DTIC ELECTE MAY 31 1990 S D

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Ada Joint Program Office
The Pentagon (1211 S. Fern St. 3E113)
Washington, DC 20301-3080

Ada 9X Pjt. Office
AF Armament/FXG
Eglin AFB, FL
32542-5434

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
This is the first of three reports to be released under this title.

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This document contains the first release of the preliminary findings of the Ada 9X Requirements Team based on their analytical efforts from October 1989 to April 1990. The requirements process has been initially a bottom-up approach at distilling public revision requests into the next higher level - Revision Issues (RIs). This document contains the first release of these RIs. Two other releases are planned. The final Ada 9X Requirements will be completed in December 1990.

(KR)

90 05 30 079

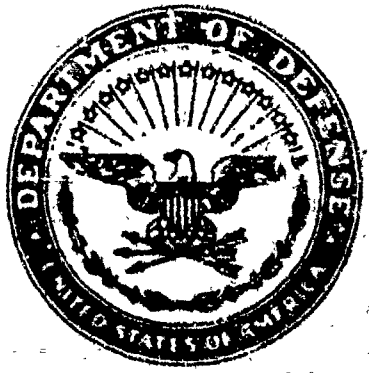
14. SUBJECT TERMS
Revision Issues, Ada 9X, Revision Requests, Ada Joint Program Office, Ada 9X Project Office

15. NUMBER OF PAGES
433

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UNLIMITED

Ada 9X Project Report



Ada 9X Revision Issues
Release 1

April 1990

Office of the Under Secretary of Defense for Acquisition
Washington, D.C. 20301

PREFACE

This document contains the first release of the preliminary findings of the Ada 9X Requirements Team based on their analytical efforts from October 1989 to April 1990. The requirements process has been initially a bottom up approach at distilling public revision requests into the next higher level - revision issues (RIs). This document contains the first release of these RIs. Release 2 will occur at the end of June. Release 3 will occur in mid-August. The final Ada 9X Requirements will be completed in December 1990. The work reported in this document was conducted under Institute for Defense Analyses (IDA) contract MDA 903 89 C 0003 on behalf of the Ada 9X Project Office under sponsorship of the Ada Joint Program Office.

The following individuals collaborated to provide the revision issues presented in this document:

Dr. Cy Ardoin - IDA
Dr. Paul Baker - CTA
Dr. Robert Dewar - NYU
Dr. Doug Dunlop - Intermetrics
Dr. Paul Hilfinger - Univ. Calif. Berkeley
Dr. Joseph Linn - IDA
Dr. Reg Meeson - IDA
Mr. Steve Michell - Prior Data Systems, Canada
Mr. Karl Nyberg - Grebyn
Dr. Olivier Roubine - Consultant, France

Accesion For	
NTIS CRA&I	☒
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution /	
Availability Codes	
Dist	Avail and / or Special
A-1	

CONTENTS

1. INTRODUCTION

The Ada 9X Project requirements development process is analogous to the open process by which requirements were developed for the design of the Ada language. That is, a small team of Ada practitioners [i.e. the Requirements Team] is responsible for analyzing revision requests and other input, and for formulating requirements for the revision of the Ada language while obtaining feedback from the Ada community.

The requirements development process began in October 1988, with the solicitation of requests from the Ada Community by the Ada Joint Program Office sponsored Ada 9X Project, and will end in December 1990, with the publication of the Ada 9X Requirements Document. Since the inception of the requirements process, the Requirements Team has had the benefit of comments on interim documents from the Distinguished Reviewer Group, workshops, and public forums. This document is an interim product of the requirements development process.

Since the Requirements Team was established in October 1989, the Team has reviewed all the revision requests collected from the Ada community and developed revision issues for most topics which are of concern to the Ada community, as evidenced by the revision requests and workshop input received. The content of this document is intended to provide status information to participants of the Ada 9X Project's Requirements Workshop (April 1990) and to the Ada community at large; and, to obtain feedback.

There are four appendices that accompany this document. Appendix A is an index of all the Ada 9X Revision Issues (RIs) in this document listed in numerical order by RI number. Appendix B is a list of all the RIs in this document indexed by page number. Appendix C is the list of RIs that are under consideration by the Distinguished Reviewers. Appendix D is the list of RIs under consideration by the Requirements Team and the Distinguished Reviewers.

Please forward comments related to this document or any matters regarding the Ada 9X Project to Chris Anderson, Ada 9X Project Office, AFATL/FXG, Eglin AFB, FL 32542-5434 or anderson@uv4.eglin.af.mil. Comments related to this release of Revision Issues should be sent **no later than 31 May** to be included in the next release which is scheduled for mid-June. Comments received after 31 May will be considered for the next release scheduled for late July.

2. SCOPE AND FORMAT

A Revision Issue (RI) is obtained by analyzing revision requests, workshop issues (e.g. May 89 Workshop) and Ada Issues (submitted to ISO WG9). The RIs in this document are the unedited and partially completed working papers of the Requirements Team as of 5 April 1990. As the requirements development process continues, these RIs will be revised to reflect reviewer comments and may be consolidated into higher level issues. Thus, the RIs in this document represent the lowest level of granularity in problem-resolution analyses. A bottom-up analytical process has been adopted by the Requirements Team to ensure that all issues have been fully considered.

The Requirements Team developed a RI Template, consisting of seven parts, to document their problem-resolution analysis. Additional parts are in a few RIs because the writer felt they were needed. Only three parts will be found in RIs that have not been fully analyzed.

The following discusses the content of each major part and the meaning of terms that are relevant to the analysis being presented.

RI Part 1 - Header: Contains five lines of text for for administrative purposes. They are:

!topic - self explanatory

!number - a unique number assigned by the Requirements Team member who has primary responsibility for writing and revising the RI.

!version - self-explanatory

!tracking - a section number or numbers from the Requirements Team Issues outline data base.

RI Part 2 - Issue: There may be more than one issue per RI. Each issue is initially classified by the Requirements Team and rated as to importance and desirability. The Issue classification and rating are important discussion topics in the consensus building process by which requirements are evolving.

A. Issue classification and the meaning of each class or label:

!Issue revision: a potential change to the Ada language

!Issue implementation: a predictable behavior for programs and compilers especially needed for real-time or distributed applications.

!Issue administrative: a proposed change in procedure or emphasis in documentation of implementation defined or dependent characteristics of Ada compilers.

!Issue presentation: a proposed change to the text or the organization of the Ada language reference manual.

!Issue secondary standard: a proposed capability which supplements the

primary Ada language standard.

!Issue out-of-scope: a proposed revision considered to be out-of-scope for Ada 9X for several reasons, such as:

research: a proposed change to the Ada language that requires future research and development to confirm its desirability.

un-needed: a proposed change to the Ada language that is already provided or an easy workaround exists.

problematic: a proposed change which is incompatible with existing Ada application code, would have a severe impact on Ada compilers, and is inconsistent with the Ada 83 design model.

B. Issue rating:

The Requirements Team initially rates the importance and desirability of an Issue by considering four aspects. Immediately following the !Issue (class) there are a series of words (some abbreviated) which are the initial rating for perceived user need, implementation impact, compatibility impact, consistency with Ada83.

Example:

!Issue revision important, moderate implementation,
upward compat, consistent

The following is the reference for the possible combinations of ratings.

Perceived user need
- not defensible
- desirable
- important
- compelling
- essential (reserved for an issue that is known to require resolution in order to obtain ANSI and ISO acceptance)

Implementation impact (on compilers)
- severe
- moderate
- small
- ? - don't know

Compatibility impact (on existing code)
- severe
- moderate
- upward compatible
- ? - don't know

Consistent with the Ada model (design)
- inconsistent
- moderately consistent
- consistent
- ? - don't know

RI Part 3 - References: In this part of an RI are listed all the revision requests (RRs), Ada commentaries (AI's), Workshop Issues (WI's), and informal papers or publications which are relevant to the Issue topic.

RI Part 4 - Problem: A summary statement of the problem(s) found in the references.

RI Part 5 - Rationale: A discussion of supporting points for the Issue, its classification and rating.

RI Part 6 - Appendix: Notes the Requirements Team wishes to keep for future reference when writing requirements.

RI Part 7 - Rebuttal: Views which diverge in some way from those expressed in the revision issue. These views are also an important reference for the Requirements Team when developing requirements.

3. GENERAL ISSUES CONCERNING THE STANDARD

The issues that may be developed for this section cover a range of perceived problems with the content and organization of the Ada Language Reference Manual or with compiler implementations. The issues in this section have not been developed but a listing of the revision requests which may serve as the references for development of revision issues is provided.

!topic possible presentation improvements in the lrm description of the language
!number RI-1004

!reference RR-0067
!reference RR-0091
!reference RR-0204
!reference RR-0206
!reference RR-0260
!reference RR-0267
!reference RR-0274
!reference RR-0277
!reference RR-0281
!reference RR-0292
!reference RR-0298
!reference RR-0300
!reference RR-0301
!reference RR-0305
!reference RR-0309
!reference RR-0326
!reference RR-0350
!reference RR-0436
!reference RR-0500
!reference RR-0501
!reference RR-0502
!reference RR-0622
!reference RR-0638
!reference RR-0683
!reference RR-0732
!reference RR-0757
!reference RR-0758
!reference RR-0769
!reference WI-0106

!reference AI-00582

!problem

There are a number of inputs to the requirements process that have suggested improvements in the description of the language in the reference manual. These need to be noted when the manual is revised.

4. FUNCTIONALITY OF THE LANGUAGE

4.1 Generality

RI-2002
RI-2102
RI-2021
RI-2029

!topic OOP (flexibility, maintainability)
!number RI-2002
!version 1.9
!tracking 4.1.1

!introduction

This RI is intended to cover some of the evolutions that are considered desirable to improve the capacity of Ada9X to support the development of large systems with a long life-time, to support incremental development and development by reuse and adaptation of existing software parts, and more generally to promote more flexibility in the development of programs. The requirements are phrased in fairly general terms, so as not to preclude any solution or to constrain the design in certain ways that may not be compatible with other language aspects. Nevertheless, it is expected that these requirements can be met by providing language features that are inspired by certain trends in modern language design, generally referred to as "object-oriented" language design.

!terminology

The word "entity" is used to designate a program object that is defined in terms of a number of data structures (also called "fields") and a number of "operations" that may be performed on the entity. The set of all fields and operations defined on an entity are referred to as its "properties". A set of entities that have the same properties can be considered as belonging to the same type, in the traditional Ada sense. It is indeed anticipated that the language changes that may be necessary to meet the requirements expressed here will be designed consistently with, and taking advantage of, existing Ada83 features.

Some of the properties of a given entity may be accessible from any program unit that can designate the entity (the "visible properties"), while certain properties may be only accessible from restricted parts of the program (the "private properties"). The bodies of all operations defined on an entity, as well as other aspects such as private fields, certain details of the representation of visible fields, and possibly other characteristics (e.g., initialization or finalization of an entity type) will be referred to as the "implementation" of the entity.

!Issue revision (1) important,severe impl,upward compat,mostly consistent

1. Ada9X shall provide a mechanism to define entity types to characterize a set of entities with common properties. It shall be possible to create entities of a given entity type just like it is possible to create objects of a given type. Ada9X shall also provide a mechanism whereby new entity types can be defined based on existing ones, with additional properties, or with different implementations of existing properties. As a minimum, it will be possible to use entities of such a new entity type anywhere entities of the original entity type(s) could be used, in exactly the same way. Ada9X shall also define the conditions under which entities of the original entity type(s) can be used where

entities of the new entity type can.

!Issue revision (2) important,severe impl,moderate compat,mostly consistent

2. Ada9X shall support the use of entities of a common entity type with different implementations, so that such entities can be used interchangeably.

!reference RI-2012 - Initialization/Parameterization

!reference RR-0125
!reference RR-0140
!reference RR-0167
!reference RR-0223
!reference RR-0394
!reference RR-0440
!reference RR-0441
!reference RR-0442
!reference RR-0505
!reference RR-0516
!reference RR-0525
!reference RR-0540
!reference RR-0599
!reference RR-0609
!reference RR-0660
!reference RR-0662
!reference RR-0750
!reference WI-0203
!reference WI-0204

!reference RR-0081
!reference RR-0668

!problem

Although intended to promote maintainable and reusable software, Ada83 is sometimes perceived as falling short of these objectives.

One of the typical problems encountered during maintenance is the modification of existing software to accommodate new cases, or to modify the way certain operations are carried out in certain cases.

Such changes generally imply adding new .alues to enumeration types, new variant in record types, and new cases in a large number of case statements. As a consequence, a large portion of the original program may have to be modified or recompiled.

In the area of reusability, Ada83 provides generic units, which allow some parametrization so that the same treatments apply to a large number of cases, but this does not lend itself to an adaptive reuse that would allow for slight modifications to the original components.

Another category of problems not easily handled by Ada83 is that encountered in systems with high flexibility requirements. Examples of such systems are systems with some degree of parallelism, or fault-tolerant systems with hardware and/or software redundancy. In such systems, it may be extremely useful to provide alternate implementations for a common abstraction.

Recent developments in object-oriented languages indicate a privileged area of investigation to solve such problems.

!rationale

The central idea behind the above requirements is that there is a need for an improved abstraction mechanism (herein called the entity) that "possesses" both data elements (i.e., a state) and operations. Note that this goes beyond the current abstraction facilities of Ada83, where one can define objects in terms of their data elements, and the operations that are applicable to these objects (i.e., the object does not "carry" its operations).

The main difference is that in the case of an entity, one can envision two entities of the same type that differ not only by the value of their data elements, but also by the "value" (i.e., the implementation) of their operations.

RI-2002.1 essentially calls for inheritance ("a mechanism whereby new entities types can be defined based on existing ones"), and for at least a minimal form of polymorphism ("different implementations of existing properties"). This requirement addresses the problem of software maintenance and reusability as follows: given an existing system using a number of entity types, new cases can be taken into account by defining new entity types based on existing ones, with new properties or with new implementations of the original properties, without entailing significant modifications of the existing code. These new entities can now be handled by the original system like the original entities, since they offer all the visible properties, but in addition, it will be possible to add new special treatments that will exploit their specificities.

RI-2002.2 does not necessarily call for a different set of mechanisms. It merely indicates that in addition to having alternate implementations through inheritance, it is also necessary to consider having alternate implementations through "variance". What this requirement actually calls for is a clear separation, at the level of entities, between the visible description of the entity behavior (the so-called contract model) and its actual realization. In Ada83, such a separation exists at the type level, i.e., all objects of a given type must have strictly the same "implementation". What is suggested here is that Ada9X should allow different objects of the same type to have different implementations (in the sense of different bodies for the same operations).

!guidelines

As these requirements essentially call for object-oriented extensions to Ada, this section indicates some orientations in order to obey some fundamental principles.

a. The extensions must not depart from the notion of strong typing. In fact, it is strongly suggested that the extensions should lead to the

introduction of new (or extended) forms of types, rather than that of an orthogonal mechanism.

b. The notion of an entity must remain as close as possible to that of an object; for instance, it should be possible to declare entities, and to use entities as components of other types (including entity types), or to in the definition of access types. It should also be possible to pass entities as parameters to subprograms, entries or operations.

c. The notion of "polymorphism" that is anticipated is restricted in that it comes from alternate implementations of the same operation specification. These requirements DO NOT call for a more general, untyped, form of polymorphism where an operation with a given name can be invoked on objects of arbitrary types.

d. Implicit in these requirements is the notion of "dynamic binding" of operations, which comes from the model of an object that "possesses" its operations as some form of component values: when an operation is invoked on an entity of a given type, the operation that will actually be executed is the one attached to the current VALUE of the entity, which is not necessarily the one defined on the type associated with the current DESIGNATION of that entity.

e. The requirements do not explicitly call for multiple inheritance, nor do they rule it out.

f. It is expected that whatever solution is provided, its integration in the language will be carried out to its fullest extent. In particular, the relationships between entities, entity types and generics or tasks will be considered, leading to possibilities such as formal entity type parameters, generic entities or active entities.

Finaily, solutions t· these requirements may also provide answers to other specific requirements sucli as, e.g., the desire to have procedure parameters or procedure variables.

!appendix

%reference RR-0125 Introduce object-oriented inheritance into the language

RR-0125 asks for inheritance; the solution given is based on package types.

%reference RR-0140 Provide more support for OOD

RR-0140 asks for OOP a la C++, noting that C++ has demonstrated that efficient mechanisms exist for OOP.

%reference RR-0167 · Allow classes of abstract data types

RR-0167 wants the type system to be three-level, i.e. object/type/class instead of the two-level object/type system that we have now. Polymorphic procedures are an important part of the discussion.

%reference RR-0223 Need to add inheritance to support object-oriented programming

RR-0223 asks for inheritance to be implemented essentially via package types coupled with a couple of pragmas for import/export visibility.

%reference RR-0394 Merge concepts of task and package into concept of an object

RR-0394 suggests a merging of the concept of a package and a task. The apparent idea is that the combined entity would allow controlled visibility of the declarative part AND would have an independent thread of control. Simula classes are very similar.

%reference RR-0440 Extend Ada to be truly object-oriented

RR-0440 wants OOP a la C++ and Smalltalk; a "supertype" concept is suggested but not further described.

%reference RR-0441 Extend Ada to allow for polymorphism

RR-0440 wants OOP a la C++ and Smalltalk; no solutions are suggested.

%reference RR-0442 Extend Ada to allow a package type hierarchy

RR-0440 wants OOP a la C++ and Smalltalk; package types are mentioned.

%reference RR-0505 Provide extendable record types, records as generic parameters

RR-0505 suggests that generic record parameters would help in solving the lack of inheritance.

%reference RR-0516 Provide more support for OOD

RR-0516 provides an argument that the original thinking of prohibiting inheritance in Ada83 was wrong.

%reference RR-0525 Extend Ada to allow for polymorphism and inheritance

RR-0525 contains material detailing how polymorphism and inheritance could be added to Ada using extensible records and a polymorphic reference type.

%reference RR-0540 Extend Ada to allow for inheritance

RR-0540 wants inheritance but not polymorphism or dynamic binding.

%reference RR-0599 Certain changes to derived/private types will help inheritance

RR-599 suggests that simple changes in derived types and private types give you essentially the right amount of inheritance.

%reference RR-0609 For oop, allow user-def override of "=", "/=", ":=" on all types

RR-0609 wants to redefine assignment and equality so that a user-defined type feels like a builtin type.

%reference RR-0660 Need constructors and destructors for package types

RR-0660 wants to add initialization and finalization for package types. The addition of package types is implicit in the discussion.

%reference RR-0662 Need package classes, inheritance in Ada for oop

RR-0662 (a companion to RR-0660) wants package types as the Ada9X solution for OOP.

%reference RR-0750 Add support for inheritance and polymorphism to the language

RR-750 wants OOP as modeled by Wirth's paper on "Type Extension".

%reference WI-0203 Allow interchangeable implementat'ns with same behaviour
%reference WI-0204 Provide classes, instances and inheritance
%reference RR-0081 Provide subprogram and package types
%reference RR-0668 Need package types to get, e.g., an array of packages

!rebuttal

There are four basic problems with the requirements as written. First, there are at least 3 requirements that are hidden in the !terminology.

1. visible vs private properties.
2. initialization.
3. finalization.

It doesn't seem right to have requirements on interchanging entity implementations when the definition of the implementation of an entity is so vague. This has the effect of increasing the implementation impact and decreasing the PUN.

Third, the terminology of the requirements does not seem to address dynamic binding even though it is said (in the !guidelines) to be implicit in the requirements. There is a large difference in English (not Ada) semantics between "different implementations of the same operation" and procedures that share the same parameter/result profile. It

seems wrong to promise the first and only give the second. It seems okay to only promise the second and the user can try to get the first by extralingual means.

Last, the notion of polymorphism allowed is not strong enough handle certain notions that are recurring in OOP application of which the SMALLTALK "bag" and the Simula "event-list" are two examples.

!topic User-defined assignment and equality
!number RI-2102
!version 1.7
!tracking 5.4.1.1 5.4.1.2 5.7.2.2

DRAFT DRAFT DRAFT DRAFT DRAFT DRAFT

!Issue revision (1) important,moderate impl,upward compat,mostly consistent

1. (Overloading of Assignment) Ada9X shall use allow an assignment operation for a limited, private type to be overloaded on ":=" to be used in assignment statements and initialization clauses for objects.

!Issue revision (2) compelling,moderate impl,upward compat,mostly consistent

2. (Abstract Data Types) An attempt shall be made to provide a mechanism in Ada9X by which (what Ada83 calls) a limited type may be bundled with an equality operation and appropriate assignment operations to yield a type ADT so that clients would see it as a nonlimited type. Capabilities would include

1. Explicit invocation of the equality and assignment operations provided for ADT;
2. Declaration of a nonlimited record or array type with ADT as the component type;
3. Specification of a subprogram with an OUT-mode parameter of type ADT;
4. Declaration a variable of type ADT with explicit initialization.

!reference RR-0070
!reference RR-0088
!reference RR-0160
!reference RR-0184
!reference RR-0413
!reference RR-0669
!reference REFER ALSO TO RR-0163
!reference REFER ALSO TO RR-0609
!reference REFER ALSO TO RR-0663
!reference RR-0201
!reference RR-0663

!reference RI-2025 Less Restrictions on Overloading; Enhanced Resolution
!reference RI-2035.1 "="(x,y:same_type) return STANDARD.Boolean
!reference RI-2035.4 Automatic Visibility of Equality
!reference RI-2012 Initialization/Parameterization of Types
!reference RI-2022 Finalization

!reference

Revision Study Note on User-defined Assignment B. Brosgol, J-L Gailly, 20 December 1989

!problem

Allowing users to define types with the flexibility afforded built-in types is an important goal. Programmer want to define their own types that have the same feel as the predefined types without giving up complete control over representation. For example, a type designer may choose to implement some type using a dynamic structure; lacking garbage collection facilities, the design may include reference counting on the data nodes. There are 4 major impediments to this:

1. lack of user-defined assignment(s).
2. lack of user-defined equality.
3. lack of sufficiently powerful type initialization.
4. lack of type finalization.

User-defined assignment would reduce the need for user-defined subtype and object constraints.

!rationale

[2102.1]
A number of users would be satisfied with a capability that allowed limited private types to look a little more like a predefined type even it did not compose so easily.

[2102.2]
The requirement speaks to the problem of bundling user-defined assignment and equality with a type definition. Such a mechanism could enchance the abstraction capabilities of the language and ease maintenance. The Initialization and Finalization are the topics of other Revision Issues (namely, RI-2012 and RI-2022, respectively). A major problem with respect to user-defined assignment and equality is to ensure that the language specifies all of the places where the user-defined operations would be used. For example, it must specify whether or not user-defined assignment is used for initialization expressions, for component assignment in a record assignment, for by-copy parameter transmission. The (lack of) strength of the requirement is due to the fact that the is a great potential for interaction between this requirement and others.

!appendix

There were several excellent discussion of this issue author by t.taft with respect to RI-2101.1 (which at the time was RI-2025.5). while not recorded here, they are certainly relevant, particularly at the revision (as opposed to requirements) level. The same is true of the referenced RSN. One very interesting idea was that the notion of "user-defined" might well be extended to attributes, selection, and indexing. The last of the ideas was o-o-s'd in RI-2025.5.

%reference RR-0070 Allow user-defined assignment for limited types

RR-70 wants to allow a user-defined procedure to replace the built-in assignment operation under user-control.

%reference RR-0088 Prohibit overloading of assignment operator ":="

RR-88 wants to not have user-defined assignment because of the problem that partial assignments are possible.

%reference RR-0160 Allow user-defined assignment for limited types

RR-160 wants to have user-defined assignment mirroring the ability to redefine "=" for limited types. Several good examples are given of why this might be a good idea.

%reference RR-0184 Need user-defined assignment operator for limited private type

RR-184 seems more interested in overloading := with a user-defined procedure than with the issue of user-defined assignment.

%reference RR-0413 Allow user-written ":=" for all types

RR-413 wants user-defined assignment. The author brings up an interesting point about not allowing IN OUT for scalars since subtype constraints might be violated.

%reference RR-0669 Allow user-written ":=" routines

RR-669 wants to define assignment for user-defined types. An important point is brought up about using the same assignment for initialization.

%reference REFER ALSO TO RR-0163 (5.2.1.2.2)

Wants to be able to write a strings package; user-defined assignment would help.

%reference REFER ALSO TO RR-0609 (4.1.1)

RR-609 want to be able to redefine ":=" and "=" for OOP.

%reference REFER ALSO TO RR-0663 (5.7.2.2)

RR-663 wants to define := and () for any appropriate types.

%reference RR-0001 Limited types are overly restricted

RR-0001 wants initialized constants and aggregates of limited types

%reference RR-0202 Relax parameter mode rules for limited types sp %reference RR-0578 Out-mode parameters of limited private types should be allowed

RR-0202 and RR-578 want OUT mode parameters for limited types.

%reference RR-0272 Limited types are of little true value

RR-272 counsels that LIMITED has so little value in its current form that it should be removed.

%reference RR-0392 Need "semi-limited" type with predefined := but no predefined =

RR-392 notes that there are times when a type should have its assignment operator visible but not its equality operator.

%reference RR-0541 Elim. restrictions on ltd. types, allow user-def :=, =, DESTROY ops

RR-541 wants user-defined ":=" and "=" and also notes that ":=" is much more useful if finalization is given also.

%reference RR-0670 Decouple "=" and "/="; don't distinguish private from ltd. private

RR-670 wants to allow "=" and "/=" to be independent; a new sort of limited type would be introduced where this was true.

%reference RR-0201 Liberalize overloading of operators to other character sequences

RR-0201 wants two items: (1) to define := for limited private types, and (2) to introduce more infix operators into the language that would then be eligible for overloading. Only (1) is covered here.

%reference RR-0663 Allow certain overloading of ":=" and "()"

RR-0663 wants to allow overloading of := and (), i.e. indexing, for limited private types. "()" is not covered here.

!rebuttal

!topic Call-back style; Subprograms as Parameters and Objects
!number RI-2021
!version 1.4
!tracking 4.1.2 5.5.2
DRAFT DRAFT DRAFT DRAFT DRAFT DRAFT

!Issue revision (1) compelling,moderate impl,upward compat,mostly consistent

1. (Subprogram and Entry Types) Ada9X shall provide types whose values are references to subprograms and (and entries viewed as procedures) sharing a common parameter/result profile and types whose values are references to entries sharing a common parameter profile. The basic operations of such types shall include (1) assignment and (2) application of a subprogram or entry value to a conformant argument list. Ada9X shall restrict the available values of subprogram and entry reference types based on safety considerations.

!Issue revision (1) compelling,moderate impl,upward compat,mostly consistent

1. (External Call-Back and Call-In) Ada9X shall support the calling of Ada subprograms in a situation where there is no Ada main program. Ada9X shall support the calling of Ada subprograms and entries from a nonAda program component at least for the case where there is an Ada main program and where these subprograms and entries have been passed as parameters to the nonAda component.

!reference RI-1012

!reference RR-0014
!reference RR-0064
!reference RR-0128
!reference RR-0180
!reference RR-0388
!reference RR-0422
!reference RR-0430
!reference RR-0503
!reference RR-0512
!reference RR-0641
!reference WI-0304
!reference WI-0506
!reference RR-0081
!reference RR-0629
!reference RR-0414
!reference RR-0563
!reference RR-0611
!reference RR-0647
!reference REFER ALSO TO RR-0642

!problem

There are many uses of subprogram variables but they fall into two general classes: late-binding with closure and late-binding without closure. The difference between the two is whether the subprogram variable implicitly carries some state information about the environment in which it was bound. Late-binding without closure is frequently used for passing subprogram arguments to procedures so as to create a sort of generic procedure with execution time binding. Another use of subprogram variables is to dynamic construct a dispatch table to associate behaviours with a given object. The dynamic binding aspects of object-oriented programming may be achieved in this way.

One can construct late-binding with closure from late-binding without closure. A frequently used form of late-binding with closure is called the "call-back" paradigm. In its most frequent form, there is a framework component is whose responsibility is to manage a complex interaction among various processes; the processes themselves are defined by clients of the framework. Often, the framework defines various decision points in the processing but leaves open how the decisions are made. This may be accomplished by associating a subprogram variable with a process. From the framework's point of view, the framework "calls-out" into the client when a decision is needed. from the client's point of view, the framework "calls-back" into the client. The call-back paradigm is frequently used in windowing systems and in discrete event simulation systems.

A number of systems are naturally structured using the functional and call-back programming styles. There are external standards, notably X-Windows, that utilize the call-back paradigm; for other languages, the call-back mechanism is based on the concept of subprogram references. Also, some authors of mathematics software indicate that such software is structured more naturally using subprograms as actual rather than generic parameters. Even when generics are used, such references cannot easily be incorporated into a data structure, such as a dispatch table.

In the above discussion, the emphasis was on subprograms, but similar problem exist for which entry variables are most appropriate. For example, one can imagine a situation in which the framework mentioned above is running concurrently with other Ada tasks. When a call-back occurs, one may want to ensure the variable references are synchronized; in such a case, an entry may be a more appropriate call-back realization than a subprogram. (Of course, one could always call an entry from a subprogram.) Also, one can imagine a situation where an entry variable represents a service; one may want to dynamically change the provider of the service during program execution.

!rationale

The use of subprogram and entry types in programming is a topic that has been extensively studied and subprogram types in some form have been incorporated in most procedural languages since 1967. It was originally thought that the inclusion of generics in Ada would greatly reduce the desire to have subprogram variables–that seems now to have been mistaken. The programming model offered by Ada would be significantly strengthened by the addition of subprogram and entry types. Also, the use of Ada with other programming languages will be significantly strengthened if external call-back and

call-in are supported.

!appendix

There is some discussion of the need to be able to execute a subprogram at a given address. Address clauses seem to have this property already except that perhaps the LRM does not make it clear what is required. There is a separate RI on this issue - RI-0104.

%reference RR-0014 Allow subprogram call by address

RR-0014 suggests that a mechanism is needed to supply a subprogram body by providing a machine address.

%reference RR-0064 Allow some form of subprogram callback

RR-0064 wants some form of call-back between modules, especially for parameterization of the interface. The generic solution is cumbersome because too many generics have to be instantiated to cover all of the cases.

%reference RR-0128 Provide subprograms as parameters to subprograms and entries

RR-0128 wants subprograms specifically as parameters to other subprograms and entries.

%reference RR-0180 There is a need for procedures as parameters for X-windows, etc.

RR-0180 suggests that procedure parameters would be a solution to the problems of (1) interfacing with X-Windows and (2) supporting the use of Ada as a target language for a source language with dynamic binding.

%reference RR-0388 Proposal for clean way of executing a subprogram by its address

RR-0388 suggests subprogram-bodies-by-address a la RR-0064. The RR goes on to discuss the need for subprogram parameters that are subprograms with polymorphic reference parameters.

%reference RR-0422 Allow subpgms as parameters and maybe also as values

RR-0422 wants subprograms as parameters and as types/objects. Functional programming and ADTs are mentioned as motivating factors.

%reference RR-0430 Objects of a subprogram "type" for subpgm parms, subpgm addresses

RR-0430 wants subprograms as parameters and as types/objects. However, the latter case presents dangling pointer problems whereas the first does not. A discussion of possible restrictions to deal with the dangling pointer problem is presented.

%reference RR-0503 Provide subprogram types for dispatcher-style programming

RR-0503 wants subprogram types. Numerous examples are cited of the nonportable means that are (and will be) used to circumvent this problem.

%reference RR-0512 Provide subprograms as parameters to subprograms

RR-0512 wants at least subprograms as parameters. Mentioned is the fact that generics require error-prone multiple level instantiations and that combinations of input classes require untoward numbers of generic instantiations. The primary use cited was for numerics work.

%reference RR-0641 Add subprograms as parameters to the language

RR-0641 wants subprograms as parameters; portability is cited as a driving motivation.

%reference WI-0304 Allow subprogram call by address

%reference WI-0506 Need mechanism to pass subprograms and entries to non-Ada pgms

%reference RR-0081 Provide subprogram and package types

%reference RR-0629 Need procedure and function types

RR-0081 and RR-0629 want subprogram types; no specific problems or solutions are given. RR-0081 also wants package types.

%reference RR-0414 Language needs subprogram types and subprogram objects

RR-0414 wants subprogram types and subprogram objects. Dissatisfaction with the two-rendezvous method and also with nonportable methods is mentioned.

%reference RR-0563 Need to allow subprogram types and variables

%reference RR-0611 Allow subprogram types, variables, constants, parameters, etc.

RR-0563 and RR-0611 wants subprogram types/objects. There are excellent discussions of the dangling pointer problem.

%reference RR-0647 Add procedure variables to the language

RR-0647 wants to allow procedure variables; the proposal explicitly mentions entries as literals of the procedure types.

%reference REFER ALSO TO RR-0642 (5.15)

RR-0642 wants a limited use label variable added to the language to support program saving program points in emulating low-level state machines. The mechanism should (according to RR-0642) have enough power to allow tail-recursion elimination to be simulated if the state machine being emulated is an interpreter for certain kinds of languages.

!rebuttal

!topic Support for control engineering programming
!number RI-2029
!version 1.1
!tracking 4.1.3

DRAFT DRAFT DRAFT DRAFT DRAFT DRAFT

!Issue revision (1) desirable,moderate impl,upward compat,mostly consistent

1. (Time-out on Operations) Ada9X shall provide a mechanism by which a sequence of operations is guarded by a time-out, i.e. if the sequence takes more time than a specified duration then the sequence of statements if abandoned.

!Issue revision (2) desirable,moderate impl,upward compat,mostly consistent

2. (Locking on Shared Variables) Ada9X shall provide a mechanism by which a task can acquire either shared-for-read-only or exclusive access to a shared variable for a sequence of statements so that no explicit release of the lock is required, i.e. the lock is released automatically whenever the user exits the sequence of statements. Ada9X shall provide a mechanism by which a task can ascertain if a variable is locked before attempting to acquire it.

!Issue out-of-scope (un-needed) (5) desirable,severe impl,upward compat,inconsistent

5. (Make Ada the Single Tool Needed for Real-Time Programming) Ada9X shall provide facilities for the following activities needed for real-time programming:

1. Inherent Prevention of Deadlocks;
2. Feasible Scheduling Algorithms;
3. Early Detection and Handling of Overload;
4. Determination of Entire and Residual Task Runtimes;
5. Application Oriented Simulation regarding RTE Overhead;
6. Event Recording;
7. Tracing.

[Rationale] While each of these activities is an important engineering aspect of real-time programming, these activities are more appropriately handled by tools other than the programming language.

!reference RR-0759
!reference RI-2022 Finalization
!reference RI-7005 Priorities and Ada's Scheduling Paradigm
!reference RI-7007 Alternate Task Scheduling Paradigms

!problem

Many in the control programming communities find Ada83's facilities for real-time programming insufficient for their needs. Problems of tasking and scheduling paradigms are found in RI-7005 and RI-7007. The focus here is on two specific problems:

1. lack of time-out on operations.
2. no efficient way to program a shared-variable monitor for the sharing paradigm where many readers are allowed simultaneous access but a writer must have exclusive access.

!rationale

[2029.1] In real-time programming, it is often imperative to abandon an operation that is taking too long. Ada has facilities that can do this and even provides direct support if the operation is an entry call. The workaround using the current facilities is felt to be too complex and error prone.

[2029.2] Many programmers design their control programs in terms of a shared variable usage paradigm where many readers can share a variable but only one writer. It is somewhat cumbersome to implement an appropriate monitor for such a variable if timed calls are used (by the exclusive requesters). In addition, the rendezvous required are perceived to be too inefficient by this user community.

!appendix

%reference RR-0759 Add real-time and verification facilities for control engineering

RR-759 contains a proposal to update Ada to achieve a number of important features for control systems including

1. Application Oriented Synchronization Constructs
2. Timeout of Resource Claims
3. Availability of current task and resource status
4. Inherent Prevention of Deadlocks
5. Feasible Scheduling Algorithms
6. Early Detection and Handling of Overload
7. Determination of Entire and Residual Task Runtimes
8. Exact Timing of Operations
9. Application Oriented Simulation regarding RTE Overhead
10. Event Recording
11. Tracing
12. Usage of only Static Features if necessary

(1) and (2) are covered by RI-7005 and RI-7007.

!rebuttal

4.2 Reliability

RI-1030
RI-1031
RI-1032
RI-1033
RI-1034

!topic early messages from implementations for apparent problems
!number RI-1030
!version 1.5
!tracking 4.2.1

!Issue revision (1) important,small impl,moderate compat,consistent

1. (Bad Pragma Means No Compile) A pragma that violates language-defined rules concerning placement and allowed arguments shall be illegal in Ada9X.

!Issue revision (2) important,moderate impl,moderate compat,consistent

2. (Tightened Legality Rules) The legality rules of Ada shall be constrained in an attempt to prevent or lessen the possibility of predefined exceptions being raised, erroneous execution, or incorrect order dependence. In particular, Ada9X shall provide additional legality rules (or alternatively, modify existing rules) for as many cases as possible where additional compile-time checking could reduce the possibility of run-time errors. Specific examples of potential changes along these lines include:

1. Making it illegal to assign a static expression to an object of a locally static subtype where the expression value does not belong to the subtype of the object (compile-time detection of the exception condition described in 5.4(4));
2. Extending legality rules 7.4.1(4) and 7.4.3(2) to improve compile-time detection of the erroneous execution described in 7.4.3(4);
3. Making it illegal to use a static expression with the value 0 as the right operand of the MOD operator (compile-time detection of the exception condition described in 4.5.5(12));
4. Making certain uses of multiple address clauses within the same the same declarative part (or package specification) illegal to allow compile-time detection of the erroneous execution described in 13.5(8);
5. Altering the scope and visibility rules of the language in some way to lessen the possibilities of the access-before-elaboration exception condition described in 3.9(8).

!Issue administrative (3) important

3. (Encourage Good Compiler Feedback) Ada9X compilers shall be encouraged to detect and report (via warnings or information messages) as many forms of potential errors and suspicious programming as possible. These diagnostics may address potential program problems such as:

1. statements whose execution can be shown to cause erroneous execution or to cause a pre-defined exception to be raised;
2. unsound or otherwise suspect programming practices such as no assignments to an OUT-mode parameter in a subprogram, no accepts

for an entry in a task body, and unreachable statements;

3. violations of assumptions implied by the use of a implementation-dependent usage-restrictive pragma;
4. use of a pragma that cannot be complied with by by the implementation.

!Issue out-of-scope (un-needed) (4) important,small impl,moderate compat,inconsistent

4. (Implementation-Defined Legality) An Ada9X implementation shall be allowed to reject a legal compilation unit if it can determine, at compilation time, that execution (or elaboration or evaluation) of a construct in the compilation unit will raise a predefined exception or cause erroneous execution or an incorrect order dependence.

[Rationale] It is the language, not implementations, that should define the legality of Ada programs. This is a fundamental Ada principle that should not be overturned. A more appropriate solution to this problem seems to be to modify the semantics of the language to catch, at compile time, more kinds of run-time errors (see 1030.2, above). As the quality of Ada implementations continues to improve, it is likely that more and more compilers will attempt to diagnose run-time error situations and issue warnings.

!Issue out-of-scope (un-needed) (5) desirable,moderate impl,upward compat,inconsistent

5. (Define Warning Cases in Language) Ada9X shall define a set of Ada programming situations for which each implementation shall be required to issue a warning message.

[Rationale] Currently, many of the better compilers being sold are already making a reasonable effort to diagnose potential programming problems by emitting warnings. Market pressures suggest that this trend will continue. The need for warnings from the compiler will be lessened by 1030.1 and 1030.2 above, which make certain Ada83 "warning situations" illegal. It is not clear that minimum standards for warning messages belong in a programming language standard. It is not clear that a single set of situations that warrant warning messages can be found that is independent of software development methodology.

!Issue administrative (6) important

6. (Compilation Mode for No-Compile on Warning) Ada9X compilers shall be explicitly allowed to provide a mode of operation (i.e., a compiler parameter or switch) that causes the compiler to reject (i.e., as though illegal) compilation units for which it generates warnings.

!reference RR-0165
!reference RR-0209
!reference RR-0211
!reference RR-0214
!reference RR-0216
!reference RR-0217
!reference RR-0242

!reference RR-0328
!reference RR-0616
!reference RR-0692
!reference RR-0754
!reference RR-0756
!reference RR-0771
!reference WI-0104
!reference AI-00031
!reference AI-00242
!reference AI-00340
!reference AI-00417

!problem

Many users have reported that they feel they do not get sufficient information from their Ada compilers concerning potential programming mistakes. In particular, these users have observed that:

1. incorrect pragmas are ignored, often silently;
2. even correct pragmas are often not binding on an implementation and may be ignored without warning;
3. obvious programming mistakes that the compiler knows about (or should know about) go unreported;
4. when issued, warnings (or information-level messages) are inadequate because users lack incentive to read warnings or lack time to hunt for real problems that may be indicated in a lengthy list of warnings;
5. implementation-defined usage-restrictive pragmas are quite dangerous because a pragma cannot affect the legality of a compilation unit.

There are two main consequences of these problems. One is that software problems are detected later rather than sooner. This can be a major inconvenience when the cost of correcting the problem (e.g., compile and link time) is significant. The second more serious problem is that it causes more of a need for testing and potentially allows errors to go undetected prior to the release of the software.

!rationale

[1030.1]
The language should recognize that many pragmas have important effects and therefore should not be simply ignored when they are incorrect. It seems counterproductive to consider a compilation unit legal even though it contains an incorrect (and therefore ignored) pragma that may affect the observable results of running the program.

[1030.2]
The best solution to the problems discussed above seems to be to tighten up the semantics of the language to catch as many error situations as possible at compile time. Take as an example the Ada83 rule that a function subprogram must contain at least one return statement. This is a semantic restriction placed in the language to help catch, at compile

time, the PROGRAM_ERROR exception that results from exiting a function without returning a value. It seems worthwhile to attempt extending this reasoning throughout Ada with the goal of making it a safer programming language.

[1030.3]
There may be the impression that warning and information-level diagnostics are inappropriate for an Ada compiler because they are not required by the standard. This impression, if it exists, should be corrected. In the interest of the early detection of software problems, compiler developers should be encouraged to provide as much information as is reasonably possible to the programmer about his/her program.

[1030.6]
In a similar vein, it should be clarified that compilers are allowed to have a special mode of operation that causes compilation to fail when warning diagnostics are generated. Since there appears to be strong desire for such a feature on the part of some, market pressure may cause vendors to provide this facility since it seems relatively easy to do. This approach is a general means of lessening the severity of the problems described above.

!appendix

%reference RR-0165 Allow parameter constraint violations to be compile-time error

This seems somewhat unclear. Appears to want compile-time detection of parameter constraint errors. Also a language change with a new mechanism permitting validation procedures to be attached to formal subprogram parameters.

%reference RR-0209 Require compiler to report certain-to-be-raised exceptions

Require a compiler to report known, certain-to-be-raised exceptions. It says "knowing this at compile-time will save debug time".

%reference RR-0211 Require compiler to report unrecognized or incorrect pragmas

A compiler is not required to report an unrecognized or incorrectly used (including parameter usage and placement) pragma.

%reference RR-0214 Require that a subprogram parameter be used within the body The Reference Manual does not require that the parameters of a subprogram be used within the subprogram body.

%reference RR-0216 Require that each task entry have at least one accept statement
The Reference Manual does not require that an "entry" point declared in a task specification have a corresponding "accept" with the same name.

%reference RR-0217 Require that a parameter of an entry be used within an accept
The Reference Manual does not require that the parameters of an "Accept" be used withing the "Accept" statement.

%reference RR-0242 Require compilation warnings for potential run-time errors

Require diagnostics (at least warnings) for compile-time recognizable errors.

%reference RR-0328 Require compilers to diagnose potentially bad situations

Add to 1.1.2 a list of questionable usages that a conforming compiler is required to diagnose, when requested by a compiler option.

%reference RR-0616 Require compilers to diagnose static-detectable constraint errors
Reword LRM 1.6.3(b) and 3.3.2(6..9) to indicate that compilers must perform reasonable analysis of potential constraint errors at compile time.

%reference RR-0692 Allow implementation-defined pragmas to affect legality of program
Allow implementation-defined pragmas to render programs illegal if they violate the assertions behind the pragmas.

%reference RR-0754 Require warnings for unrecognized pragmas

The Ada language does not require an implementation to warn the user about unrecognizable pragmas. Failure to warn the user about an unrecognizable pragmas can mislead the user. All pragmas which the implementation cannot recognize should produce a warning to the user.

%reference RR-0756 Require warnings when pragmas are ignored

Warnings should be required for ignored pragmas for two reasons. First, this allows a badly placed pragma to be ignored with no indication given to the programmer. Second, some pragmas (such as INLINE and PACK) may have important effects on the performance/size of the resulting code and it seems desirable to know whether or not the compiler intends to obey them.

%reference RR-0771 Require tasks to have an accept for each entry

Currently the standard does not require a task to have an accept statement for an entry declared in its specification. This seems inconsistent, since, e.g., functions MUST contain at least one return statement.

%reference WI-0104 Require impl's to detect and warn about unsound prog practices

Implementations should give warnings about or raise exceptions for erroneous conditions.

%reference AI-00031 Out-of-range argument to pragma PRIORITY

A bad argument to pragma PRIORITY simply means the pragma is ignored.

%reference AI-00242 Subprogram names allowed in pragma INLINE

Yes, the rules are cleared up as follows. Also, remember this simply clarifies when and when not pragma INLINE has "no effect".

%reference AI-00340 The model numbers for a fixed point type having a null range

This seems irrelevant to this issue. Why is this AI present here?

%reference AI-00417 Allowed range of index subtypes

An implementation cannot throw out
type EXAMPLE is array (LONG_INTEGER range 0 .. 100_000) of INTEGER;
even if that is bigger than the hardware will support. However, declarations of variables of this type are allowed to yield STORAGE_ERROR.

!rebuttal

!topic problems with erroneous execution and incorrect order dependence
!number RI-1031
!version 1.5
!tracking 4.2.2

!Issue presentation (1) important

1. ("Effect" of Execution) Ada9X shall clarify what is meant by the "effect" of the execution of a program (see, e.g., LRM 1.6(9), 6.2(7)).

!Issue revision (2) important,moderate impl,moderate compat,mostly consistent

2. (Review of Erroneous Execution & IOD) The Ada83 notions of erroneous execution and incorrect order dependence shall be given careful review. In particular, for each (existing and contemplated) condition causing erroneous execution or an incorrect order dependence, an attempt shall be made to:

1. make the condition illegal;
2. require a predefined exception to be raised for the condition;
3. bound the potential meanings of the execution (rather than defining the effect to be "unpredictable" for erroneous execution);
4. not allowing the effect of a particular instance of the condition to change dynamically during the execution of a program; or
5. make the condition legal with an unambiguously defined meaning (possibly by specifying various orderings or implementation options that are presently not defined in the language).

!reference RR-0042
!reference RR-0066
!reference RR-0329

!reference WI-0101

!reference AI-00832

!reference RI-1022

!problem

Safety-critical and trusted-system applications require predictable execution behavior in order to be susceptible to formal analysis. Two major contributing factors to unpredictable execution behavior in Ada83 are the notion of erroneous execution and the implementation-defined ordering for various operations. Experience with Ada83 has shown that these aspects in the definition of the language can greatly complicate the task of formal reasoning about the effects of programs.

!rationale

[1031.1] Important to the Ada83 definition of the correctness of a program is the concept of the "effect" of the program's execution. This concept should be defined in the LRM.

[1031.2] This is a rather high-level requirement with no specific compliance criteria. It is difficult to levy more concrete requirements lacking the context of the other changes that will be made to the language. The central idea behind the requirement is to remove from the language as much unpredictability as possible in the areas of erroneous execution and implementation-defined orderings. The motivation for the requirement is to make programs written in Ada more amenable to formal analysis.

!appendix

%reference RR-0042 Clarify the meaning of incorrect-order dependence and its effects

Clear up what "have a different effect" means in the definition of incorrect order dependence. Think about not defining programs with incorrect order dependencies to be in error (i.e., think about not allowing PROGRAM_ERROR, simply defining the multiple potential meanings). Think about defining an evaluation order in places where it is currently left open.

%reference RR-0066 Provide guidance for erroneous execution/incorrect ord. dependencies

Narrow the possible outcomes in certain circumstances rather than simply saying erroneous. Do not declare erroneous execution to be an error, i.e., it may be tolerable for a program to have different effects on different implementations. Force implementations to define implementation dependencies such as parameter passing mechanisms.

%reference RR-0329 Difficulties with deferred constants and erroneous execution

The example in this RR appears to be bad. Nevertheless, the issue here is: instead of erroneous try making these illegal or making them OK (and defining their meaning).

%reference WI-0101 Remove lang elements which permit unpredictable, imp dep behaviour

Unpredictable, nondeterministic, and implementation-dependent aspects of Ada must be identified and justified. ———

%reference AI-00832 Communication which is not allowed

If an Ada program calls a subprogram written in another language by means of pragma INTERFACE, is the program erroneous if communication is achieved other than via parameters and function results? It would seem reasonable to communicate via a file, perhaps in addition to parameters and function results.

!rebuttal

!topic strengthening subprogram specifications
!number RI-1032
!version 1.8
!tracking 4.2.3

!terminology

[Pure Subprograms] A subprogram in an Ada9X program is pure if and only if:

1. the states of the program just before and just after any execution of the subprogram are identical with the exception of changes due to the returned parameter values or returned function value, and;
2. the returned parameter values or returned function value are a function only of the input parameter values.

[Note: this definition is somewhat conservative in that a subprogram that modifies non-local data for the sole purpose of optimizing subsequent calls (e.g., the spell-checker example) is not pure.]

!Issue revision (1) desirable,upward compat,mostly consistent

1. (Clearly Pure Subprograms) Ada9X shall define compile-time testable rules sufficient to guarantee that a subprogram is pure. [Note: it is not required that the rules be both necessary and sufficient. E.g., rules that simply disallowed potential for inpurity such as machine-code insertions and reading (but perhaps not using) global variables are acceptable.] For purposes here, we call subprograms satisfying these rules clearly pure. It shall be possible for a clearly pure subprogram to explicitly invoke another clearly pure subprogram. It shall be possible for a clearly pure subprogram to declare ar. object whose implicit initialization requires the execution of a clearly pure subprogram.

!Issue revision (2) desirable,moderate impl,upward compat,mostly consistent

2. (Assertions for Clearly Pure Subprograms) Ada9X shall provide the programmer with a means of asserting that a subprogram is clearly pure. Such an assertion shall be visible to the caller of the subprogram (i.e., shall not be an aspect of the subprogram body definition). The assertion shall be checked when the subprogram body definition is compiled and the subprogram body shall be illegal if this check fails.

!Issue revision (3) desirable,moderate impl,upward compat,mostly consistent

3. (Assertions on Exceptions) Ada9X shall allow the programmer to assert that the execution of a subprogram does not raise any non-local, user-defined exceptions other than those in a particular set. Such an assertion shall be visible to the caller of the subprogram. The assertion shall be checked when the subprogram body definition is compiled and the subprogram body shall be illegal if this check fails.

!Issue revision (4) desirable,moderate impl,upward compat,mostly consistent

4. (Additional Constraints on Parameters) Ada9X shall allow additional constraints (beyond those provided by Ada83 subtypes) to be placed on the parameters that are inputs to and outputs from a subprogram. Such assertions shall be visible to the caller of the subprogram and shall be enforced by the implementation (in general, raising predefined exceptions for violations).

!reference RR-0030
!reference RR-0517
!reference RR-0518
!reference WI-0108
!reference WI-0108M

!reference RI-5080

!reference
Trusted System and Safety Critical Ada9X Workshop; Radisson Hotel, Falls Church, VA; January 25-26, 1990.

!problem

In general, subprogram calls within an Ada code segment pose major difficulties for the task of analyzing the code to derive properties of its meaning. Such analysis is often required, for example, in proving useful characteristics of programs and in determining legal optimizations a compiler may perform. The assumptions one can make concerning a subprogram call in Ada are limited by the information conveyed in the specification for the called subprogram (unless one chooses to require the subprogram body to be present at the time of analysis and make the result of the analysis dependent on the implementation of the subprogram).

Important characteristics of subprograms that cannot be expressed in Ada83 subprogram specifications include information about side-inputs and side-effects, possible exceptions raised, and input/output conditions (beyond those provided by Ada83 subtypes). Compiler optimization and formal analysis suffer because information items such as these cannot be expressed in the subprogram specification.

!rationale

Subprogram specifications contain the parameter and result-type information "needed to call the subprogram" [Rationale]. To facilitate verification and early optimization it seems beneficial to extend the expressive power of the subprogram specification to include additional important characteristics of the subprogram. This would strengthen the formal understanding between caller of a subprogram and its implementor and would provide additional opportunities for consistency checking between the specification of a subprogram and its implementation.

[1032.1 & 1032.2] It would be a major (and possibly undesirable) change to the language to allow complete specification of the inputs and outputs of a subprogram given their

open-scope nature. A modest but still-effective approach would be to allow specification of a subprogram as belonging to a restricted "class" with respect to its behavior. This is the central idea behind the above requirement. Some details for a proposal along these lines can be found in RR-0517. Note the proposal does not suggest altering subprogram specifications to allow for conveyance of this "class" information; rather this information is specified via a new language-defined pragma. However, this distinction is irrelevant in terms of its success in lessening the problems described above.

[1032.3] An equally important characteristic of a subprogram is the set of exceptions potentially raised by its execution. Adding specification of this information may be notationally painful unless a mechanism for arbitrarily grouping exceptions exists. To ensure that such a specification could be checked at compile time, it seems wise to restrict this specification to those exceptions that are explicitly raised.

[1032.4] Finally, it would be helpful if understandings on input and output parameter assumptions could somehow be expressed in a subprogram specification. Subtypes on parameters are of help here but (in Ada83) cannot be used to express, e.g., that an access-type parameter is non-null, that a particular relationship between two input parameters is assumed, or that a particular relationship is required between the initial and final values of an IN OUT parameter. See RR-0518 for a detailed proposal along these lines.

!appendix

RR-0517 points out that for a pure-subprogram idea to work it is probably necessary to be able make assertions about the pureness of default initialization and generic instantiation.

Examples of difficulties we are trying to lessen with the pure-subprogram idea: verifying that a subprogram call does not contain an incorrect order dependence when the actual parameters include function calls, verifying that passing a particular object as a parameter does not create an alias (additional access path) for that object within the subprogram, and the elimination of unnecessary function calls during optimization by a compiler.

%reference RR-0030 Require explicit importing of nonlocal objects

Allow the subprogram specification to say what non-locals are visible and how they may be used. This seems to be static/visibility sort of thing as opposed to what is suggested in RR-0517.

%reference RR-0517 Provide syntax to declare program units free from side-effects

Allow assertions that say a subprogram is free of side inputs as well as assertions about side outputs. Get compiler to verify these.

%reference RR-0518 Provide syntax to declare program unit pre/post conditions

Augment subprogram specification with limited pre and post conditions concerning parameters.

%reference WI-0108 Assertions in Ada are not required

To be of value, the assertion language would have to be much more powerful than Ada83 boolean expressions. Assertions for formal analysis and verification is of little value unless combined with a formal definition of dynamic semantics.

%reference WI-0108M Assertions in Ada are needed
Assertions can be useful with only minor extensions. An ANNA-like capability for assertions would be very nice.

!rebuttal

!topic fault tolerance
!number RI-1033
!version 1.5
!tracking 4.2.4

DRAFT DRAFT DRAFT DRAFT DRAFT DRAFT

!terminology

[Remote Operation] A remote operation in the execution of an Ada program is a function requiring synchronization or communication between the processor executing the function and an external agent. These points would include, e.g.,

1. an entry call to a task that executes on another processor;
2. a reference to a remotely-stored object;
3. the creation or termination of a task that executes on another processor;
4. a remote procedure call, and;
5. the completion of a rendezvous with a client task that is executing on another processor.

!terminology

[Failure of a Remote Operation] A failure of a remote operation in the execution of an Ada program is one where the underlying communication or synchronization that is required to implement the operation cannot take place. Possible reasons for failures might include, e.g.,

1. an addressed remote processor is not operational, and;
2. the communication required is not possible.

!Issue revision (1) compelling

1. (General Goal) Ada9X shall support to the fullest extent possible the construction of fault tolerant systems.

!Issue revision (2) important,severe impl,upward compat,inconsistent

2. (Failure Semantics) Ada9X shall define the effect of failures of remote operations on the executions of Ada9X programs. Detected failures of remote operations should therefore manifest themselves in the execution of a program in language-defined ways. Implementations should be encouraged to detect failures in remote operations, and, in particular, to minimize the situations where a thread of control in an Ada9X program can be "left hanging" indefinitely due to a failure of a remote operation.

!Issue implementation

[3 - Remote Operations for Real-Time Systems] The real-time implementation standards should define various remote operations of relevance to real-time systems and specify for which of these operations implementations are required to check for failures (and respond in accordance with [2]).

!Issue revision (4) important

4. (Late Binding) To help support dynamic reconfigurability, Ada9X shall allow late binding of Ada entities to memory locations to the fullest extent possible. Examples of late binding along this line include dynamic binding of entries to interrupt addresses and a "pointer to subprogram" capability (see RI-2031).

!Issue out-of-scope (research) (5) important,severe impl,upward compat,inconsistent

5. (Language Hooks for Recovery) Ada9X shall provide language mechanisms to allow recovery when failures of remote operations or other similar difficulties occur. Potential examples of such mechanisms might include, e.g.,

1. improved control over the elaboration (and finalization) of library units to re-initialize aspects of a system;
2. a facility for asynchronous notification of tasks that specific failures have occurred and a new mode of operation is required;
3. implementation-defined "task characteristics" for interaction with an underlying recovery system;
4. facilities for dynamic re-allocation of program segments;
5. support for transactions (checkpointing and rollback functions);
6. language support for "critical" distributed data (requiring, e.g., two-phase-commit protocol in the underlying implementation), and;
7. accessible unique identification of the threads of control in the program to be used for re-configuration and communication with run-time support software.

[Rationale] While the need here is genuine, the issue of effective support for recovery within a programming language is still a research area in computer science. It seems premature to make language changes along these lines until more experience with the development of fault-tolerant systems is gained.

!reference RR-0111
!reference WI-0406
!reference WI-0407
!reference WI-0407M
!reference WI-0408
!reference WI-0408M

!reference RI-2031

!reference

Knight, J., Urquhart, J., "On the Implementation and Use of Ada on Fault-Tolerant Distributed Systems", IEEE-TSE Vol. SE-13, No. 5, May 1987.

!problem

Ada83 does not adequately support the development of fault-tolerant systems. These deficiencies fall into two broad areas.

First, the language does not define semantics of Ada programs in the light of failures in the underlying hardware or support system. For example, consider the case of a distributed Ada program where the tasks of the program are partitioned among the available processors. Furthermore, consider the situation where one of the processors goes down during the execution of the program. The language does not define the effect of a task (on one of the functioning processors) attempting to rendezvous with a task that was executing on the disabled processor. This situation is therefore not an error condition that the implementation is encouraged to check for. If the implementation does check for the error condition, any semantics it provides for it are outside the language and therefore not.sp portable.

Second, the language does not adequately support the function of recovery in a fault-tolerant system. There are no facilities in the language for moving software (or data) from a disabled processor to another node in a distributed program; the language does not allow for "re-initializing" a subsystem in a program in order to bring it to a consistent state; no mechanism is provided in the language to asynchronously cause a task to assume a recovery mode of operation; and so forth. The language provides only the primitives of aborting tasks and (re-)starting tasks to program fault recovery with. These considerably limit the available options for recovery operations and encourage non-portable implementation-defined solutions to these problems.

!rationale

[RI-1033.1]

It is important that the language not "get in the way" of the development of fault-tolerant programs. This seems particular true for an objective such as fault tolerance since it is not clear how much direct language support can be provided to help build these systems. Its likely that implementation-dependent outside-the-language mechanisms will be relied on heavily in building fault-tolerant programs and it is important that the language accommodate these mechanisms in as graceful a manner as possible.

[RI-1033.2]

An unequivocal first step in building fault-tolerant programs in Ada is semantics associated with various failures in the underlying system. Without such semantics, there is no way to detect and recover from such failures within the language. Relatively simple and attractive approaches to this problem are possible (e.g., [Knight87] suggests that a

task executing on a disabled or unreachable processor have identical semantics to that of an aborted task.

It should be understood that defining the semantics of failures in the underlying system is not unprecedented in the language. For instance, there are failure semantics associated with malfunctions in the underlying system for many of the I/O operations described in LRM 14 (i.e., exception DEVICE_ERROR). What is being asked for here is extending this notion to other aspects of the language where faults in the underlying system can affect execution behavior and where it may be appropriate to program a response in an attempt to recover from the fault.

[RI-1033.3]

The real-time implementation section in the LRM is probably the place to define remote operations that might be applicable for fault-tolerant real-time systems.

[RI-1033.4]

The process of recovery would be simplified if the programming language had a large degree of support for the late binding of Ada entities to memory locations. For example, task access types can be important in building fault tolerant systems in Ada83; extending this kind of facility would allow easier and more flexible reconfiguration in response to a fault.

!appendix

%reference RR-0111 Provide explicit support for fault tolerance and recovery

The language shall contain a model for partial failure of an Ada program that includes at least: task operations, operators and subprogram calls, and data object accesses. The model shall define semantics sufficient to permit Ada applications to detect, confine, assess, report, and recover from these partial failures.

%reference WI-0406 Need failure semantics for abort, task attributes, task dep., etc.

Always want to be able to recover from failures, never want to be left "hanging".

%reference WI-0407 Need primitives to support construction of fault tol. systems
"Task characteristics" and elaboration control are said to be examples of what would be nice.

%reference WI-0407M Primitives for fault tol construction restricts flexibility

Its too early to add fault tolerance to Ada. Nothing's been proven. Not clear this can be added in a clean way.

%reference WI-0408 Provide exact def. semantics (inc fail sem) for all types of rndvs

Necessary to support fault tolerance, configurability, and transactions.

%reference WI-0408M Don't add def. semantics if cost of rendezvous increases

Rendezvous is already expensive enough.

!rebuttal

!topic miscellaneous reliability issues
!number RI-1034

!reference RR-0236
!reference RR-0386
!reference RR-0478
!reference RR-0729
!reference RR-0763
!reference AI-00585

!problem

There are a number of reliability problems covered herein. They include:

1. Lack of formal methods applicable to Ada83 programs;
2. Lack of a way in Ada83 to request that the compiler not attempt optimization;
3. Lack of features in Ada83 to protect against viruses and trojan horses;
4. The fact that the Ada83 pragma suppress can be dangerous toj use, and;
5. The fact that changing a discriminant in a record after evaluating a prefix in a name using the record type can lead to unpredictable results.

!appendix

%ref RR-0236 Need better support for formal analysis of Ada programs
%ref RR-0386 Need standard way of telling the compiler not to optimize
%ref RR-0478 Add language facilities for more safety in hostile environments
%ref RR-0729 Language should provide way to turn off optimization
%ref RR-0763 Allow nested scopes for effect of pragma suppress
%ref AI-00585 [BI,RE] discriminant change after prefix evaluation

4.3 Efficiency

RI-5040
RI-5080

!topic pre-elaboration
!number RI-5040
!version 1.7
!tracking 4.3.1

DRAFT DRAFT DRAFT DRAFT DRAFT DRAFT

!Issue revision (1) important,severe impl,upward compat,unknown compat

1. Ada9x shall provide a mechanism to achieve compile- and/or link-time elaboration of constructs whose characteristics are constant and known before execution.

!reference RR-0018 (related) static ragged arrays
!reference RR-0117 Provide pre-elaborable constructs
!reference RR-0243 Allow/require certain elaboration prior to run-time
!reference RR-0244 Ada elaboration rules adversely affect implementations
!reference RR-0245 Change Ada LRM to add encouragement for pre-elaboration
!reference RR-0246 Require constants to be static to encourage pre-elaboration
!reference RR-0247 Want no implicit code/action during object elaboration
!reference RR-0261 Need compile-time warnings of program_error
!reference RR-0285 (related) linkage optimization
!reference RR-0451 (related) deferred initialization of objects
!reference WI-0205 Don't constrain elaboration of library units

!problem

Execution-time elaboration for constructs that are constant and known at compile-time (or link-time) consumes processing time and memory that can impact embedded real-time applications. Other languages (e.g., assembler) allow programmers to specify constructs that are "pre- elaborated" or simply require no run-time elaboration. In addition, such constructs may often be allocated in read-only memory.

!rationale

Ada83 allows optimizations for compile- and/or link-time elaboration of constructs whose characteristics are constant and can be evaluated before execution. Programmers, however, have no control over this capability, if it is provided, and have no way to indicate that the "optimization" is essential to the correct operation of their programs.

!appendix

1. Execution-time elaboration can be a significant overhead for real- time applications.

2. Pre-elaboration is a reasonable optimization, although it may not be reasonable to require every compiler to implement it fully.

3. A proposed solution is to identify constructs for pre-elaboration by means of a pragma.

4. No language changes are necessary for vendors who wish to provide compile- or link-time elaboration as an optimization.

5. Below is a list of constructs identified in RR-0117 that could be pre-elaborated and should be considered for inclusion any language revision(s) that address this problem.

Static expressions
Scalar types – unconstrained or with static constraints
Subtypes – if their base types can be pre-elaborated and their elaboration cannot raise an exception
Array types – if their component and index subtypes can be pre- elaborated
Record types – if every component can be pre-elaborated
Access types – if the designated subtype can be pre-elaborated
Allocators – if their subtypes and initial values can be pre- elaborated
Aggregates – if their subtypes, index choices, and component values are can all be pre-elaborated
Type and subtype declarations – if the types or subtypes can be pre- elaborated
Constant declarations – if their subtypes and values can be pre- elaborated
Variable declarations – if they have no initial values and their subtypes can be pre-elaborated
Subprogram declarations
Generic instantiations – if their bodies and all generic parameters can be pre-elaborated and the instantiations cannot raise any exceptions
Packages – if all their components can be pre-elaborated and there is no sequence of statements to execute
Tasks and task types – if all their declarations can be pre- elaborated and they are not nested within other tasks or subprogram bodies
Generic declarations – if all expressions and types in their formal parts can be pre-elaborated and if their bodies are subprograms or packages that can be pre-elaborated

%reference RR-0018

Need ragged arrays of literal strings to pack text for application messages to avoid elaboration and heap allocation.

%reference RR-0117

Need quicker start-up than is possible if all elaboration is done at run time. Provide a pragma to pre-elaborate "allowable" constructs (listed above).

%reference RR-0243

Run-time elaboration overhead is too costly. Require elaboration to be done at the earliest possible time.

%reference RR-0244

Run-time elaboration is inefficient. Need support for pre-elaboration. Pre-elaboration of any declaration that would raise an exception should be illegal.

%reference RR-0245

Even though pre-elaboration is an allowed optimization, the LRM directs implementations to elaborate everything at run time and no vendors implement the optimizations.

%reference RR-0246

The definition of "constant" should be changed so that all constant values can be determined at compile time and stored in ROM.

%reference RR-0247

Code generated implicitly (i.e., without the programmer's direct action) for elaboration and initialization can be unsafe.

%reference RR-0261

A correct elaboration sequence should be assured at compile (link) time so that elaboration checks are not necessary and program_error is not raised.

%reference RR-0285

%reference RR-0451

Allow deferred definition of constant values and initial values for variables in package specifications, with complete definitions given in package bodies.

%reference WI-0205

The language should place no constraint on the elaboration of library units (or their bodies), other than the requirement that a given program unit must have been elaborated before the execution of any reference to it.

!rebuttal

!topic compile-time optimization
!number RI-5080
!version 1.6
!tracking 4.3.2

DRAFT DRAFT DRAFT DRAFT DRAFT DRAFT

!Issue revision (1) important,moderate impl,upward compat,consistent

1 – Canonical-order evaluation] Ada9x shall define and provide a mechanism (or mechanisms) to explicitly enforce deterministic canonical orders for expression and subprogram parameter evaluation, select- alternative guard evaluation, subprogram result copy-back, and statement execution. [Note: invoking this mechanism would defeat some of the optimizations allowed by the rules in requirement [3]. Semantically neutral optimizations would still be allowed.]

!Issue revision (2) important,moderate impl,upward compat,consistent

2 – Vector optimizations] Ada9x shall provide a mechanism to indicate that optimizations may alter the language's strict order of statement execution rules to utilize available machine vector and parallel operations, subject to the rules in requirement [3]. [Note: the language will not be able to specify which of any affected variables have been assigned new values when an exception is raised in vector-optimized code. Hence, any program that depends on the values of such variables when an exception is raised will be erroneous.]

!Issue revision (3) important,moderate impl,upward compat,consistent

3 – Optimization rules] Where canonical-order evaluation is not explicitly indicated and where vector optimizations are enabled, Ada9x shall allow implementations to change the order of operations and assignments to optimize code, subject to the following rules:

a. Operations affecting input values for subsequent operations in the canonical order must be completed prior to the values' use

b. Operations influencing neither control flow nor input values for subsequent operations need not be performed

c. Operations whose only *effect* is to implicitly raise a predefined exception need not be performed

!Issue revision (4) important,moderate impl,upward compat,consistent

4 – Side-effect free code] Ada9x shall define criteria that guarantee subprograms have no side inputs or side outputs, and shall provide a mechanism to indicate subprograms that must satisfy those criteria. Subprograms so marked that do not conform to the criteria shall be illegal. The no-side-input/no-side-output indication shall be part of the subprogram's specification.

!Issue out-of-scope (un-needed) (5) desirable,small impl,bad compat,inconsistent

5 – Short-circuit semantics] Change the semantics of Boolean expressions in Ada9x so that short-circuit evaluation is the normal semantics; remove short-circuit syntax.

[Rationale] The impact and implications of this proposal would be extensive; for example, would user-defined "and" and "or" operations also be short-circuited?

!reference AI-00280 pragma OPTIMIZE and package declarations
!reference AI-00284 definition of incorrect order dependence
!reference AI-00315 legal reorderings of operations
!reference AI-00380 reassociation and overloading resolution
!reference RR-0265 allow implementations to short-circuit in general
!reference RR-0387 remove canonical ordering of assignments
!reference RR-0517 identify side-effect free operations (related)
!reference RR-0683 clarify allowable operation substitutions (related)
!reference RR-0685 canonical execution order
!reference RR-0700 constant expressions
!reference RR-0718 predictability of optimizations
!reference RR-0738 vectorization, vector processing hardware
!reference RR-0739 relax canonical ordering of assignments
!reference RR-0740 scope of in-lined subprograms
!reference RR-0741 vector types and operations
!reference WI-0105 predictable effects of optimization
!reference
SEI Ada-9X Complex Issues Study: Exceptions and Optimization

!problem

Code optimization is essential to many Ada applications. Ada83's restrictions on the effects of optimization [LRM 11.6] allow little freedom for reordering instructions to improve performance or for using vector and parallel machine operations where available. On the other hand, Ada83 does not define the order in which subprogram parameters are evaluated or results are copyed back [LRM 6.4(6)], which provides more flexibility for optimization but can produce unpredictable results when exceptions are raised.

!rationale

Ada9x must allow for much more extensive optimization than is provided for by Ada83 [LRM 10.6 and 11.6]. At the same time, Ada9x must provide higher safety and more predictable program behavior than is provided by Ada83's non-definition of expression and parameter evaluation and parameter copy-back order. Requirements [1], [2], and [3] attempt to allow more flexibility for optimization, while also guaranteeing safety and predictable program behavior. Requirement [1] addresses the needs of the high assurance community who say they need to know the sequence of operations that will be executed. Requirement [2] addresses a problem in generating efficient code for vector and parallel hardware with Ada's current semantics. Requirement [3] attempts to provide the additional flexibility to apply a broader class of optimizations than Ada83

allows. Requirement [4] attempts to give compilers program "design knowledge" or "intent" that enables optimizations but is impossible to deduce automatically.

!appendix

The original statement of [1] was to make canonical-order evaluation the default. It was pointed out, however, that this would encourage programmers to write order-dependent code that could not be optimized. The intention is that order-dependent expressions and parameter lists will remain program errors – unless they are designated for canonical order evaluation, in which case they will be legal and produce predictable results. Removing canonical order designations, therefore, will have to be done carefully.

%reference AI-00280

Need a way to associate a pragma OPTIMIZE with a library unit package declaration; such a declaration has no declarative part.

%reference AI-00284

Questions the meaning, intent, and ramifications of the definition of incorrect order dependence in LRM 1.6(9).

%reference AI-00315

Extensive discussion of legal reorderings of operations, interactions between exceptions and optimization, and proposals for interpretations (and elimination) of LRM 11.6.

%reference AI-00380

How does associativity of operations interact with overloading?

%reference RR-0265

Short-circuiting should become the normal semantics for Boolean operations and optimizing compilers should handle all cases correctly.

%reference RR-0387

Revise LRM 11.6(3) to allow vectorization of loops, code motion, and optimal instruction scheduling.

%reference RR-0517

Need ways to declare subprograms, instantiations, and initializations to be side-effect free. Need way to restrict visibility of implicit inputs.

%reference RR-0683

Need clarification of operation substitutions allowed in optimization; e.g., can (-X+Y) be evaluated as (Y-X)?

%reference RR-0685

Need more freedom for parallelizing and reordering instructions to achieve optimal code on vector and pipelined machines. Base rules on data dependencies.

%reference RR-0700

Compilers need to be able to recognize that expressions such as SINE(10.0) are constant.

%reference RR-0718

Need more predictability in numerical results in the face of optimization.

%reference RR-0738 Need more effective optimizations for vector machines – perhaps pragma OPTIMIZE(VECTORIZE).

%reference RR-0739

Relax canonical ordering rules to allow reordering assignment statements.

%reference RR-0740

Allow scope of inlined subprograms (local declarations and exception handlers) to be combined with their enclosing scopes to facilitate optimization.

%reference RR-0741

Create a class of vector expressions rather than expecting compilers to vectorize loop statements as an optimization. (Ada does not have FORTRAN's dusty deck problems.)

!reference WI-0105

A canonical order for expression and parameter evaluation shall be established. Allowable reorderings and code motion for optimization shall be limited so that program functionality remains predictable.

!rebuttal

4.4 Simplicity/Consistency

RI-5100
RI-2022
RI-5061
RI-5062
RI-1050
RI-5110

!topic Special Case Rules
!number RI-5100
!version 1.5
!tracking 4.4.1

DRAFT DRAFT DRAFT DRAFT DRAFT DRAFT

!Issue revision (1) important,moderate impl,moderate compat,consistent

1 – Remove special case rules] An attempt shall be made to reduce the number of special case rules, inconsistencies across language features, and obscure interactions between features in Ada9x.

!reference AI-00119 prefix of an expanded name
!reference AI-00161 index constraints with mixed bounds
!reference AI-00193 'FIRST's argument in overload resolution
!reference AI-00280 pragma OPTIMIZE and package declarations
!reference AI-00284 incorrect order dependence
!reference AI-00394 is a numeric type a derived type?
!reference AI-00470 undefined function result subcomponents
!reference AI-00490 erroneous execution for private types
!reference AI-00496 negative STORAGE_SIZE value for tasks
!reference AI-00606 implicit conversions in overload resolution
!reference RR-0319 arbitrary language restrictions, orthogonality
!reference RR-0363 Allow 'value and 'image to apply to reals
!reference RR-0426 special cases
!reference RR-0427 complexity
!reference RR-0612 Should allow both delay and terminate alternatives
!reference RR-0664 Need 'IMAGE and 'VALUE for floating-point

!problem

Special case rules, inconsistencies across language features, and obscure interactions between features make learning and using Ada more difficult than necessary. Examples range from semantic interpretations to syntax issues, to reference manual wording:

a. Special handling of renamed constructs in deciding whether a component selection is ambiguous. (AI-00119)

b. Disallowing mixed values of universal and named integer types for array bounds. (AI-00161)

c. Disallowing, in the visible part of a package, use of a declared derived type as the parent type in another derived type declaration. (LRM 3.4(15), AI-00394)

d. Restricting the location of pragma OPTIMIZE to a declarative part. (It should be allowed to appear anywhere a basic declaration may appear.) (AI-00280)

e. Disallowing the combination of a terminate and a delay alternative in the same select statement. (RR-0612)

1. Not providing attributes 'image and 'value for fixed point and floating point types.

1. Not allowing generics and packages to be defined by just a body (rather than distinct specifications and bodies).

1. The execution of a program is erroneous if it attempts to evaluate a scalar variable [or result of a function call] with an undefined value. (LRM 3.2.1(18), AI-00479, AI-00490)

!rationale

Special case rules, inconsistencies across language features, and obscure interactions between features increase the number of programming errors made and extend both software development and training time. Language changes that unify rules and features are expected to reduce overall Ada software development costs.

!appendix

%reference AI-00119

Since the prefix of an expanded name cannot be a renaming, renamings are not considered when deciding whether a selected component name is ambiguous.

%reference AI-00161

Combinations of universal_integer and named integer types are not allowed in bounds of discrete ranges. Can this be fixed?

%reference AI-00193

Overloaded identifiers as arguments for 'FIRST, 'LAST, and 'RANGE must be resolved independently of the context in which the attributes are used.

%reference AI-00280

There appears to be no way to associate pragma OPTIMIZE with a library package since the pragma may only appear in a declarative part and a package declaration does not contain a declarative part.

%reference AI-00284

Need clearer definition of incorrect order dependence. How are exceptions and storage allocation to be interpreted in terms of their "effect" on execution?

%reference AI-00394

Is a numeric type a derived type? LRM sections 3.5.5(5), 3.5.7(11), and 3.5.9(9) describe them as "equivalent" to derived types.

%reference AI-00470

Attempts to evaluate undefined components of a function result or to apply predefined operations to them are erroneous.

%reference AI-00490

Evaluation of an undefined variable of a private type whose full type is a scalar type is erroneous.

%reference AI-00496

What happens if a negative STORAGE_SIZE value is given for a task? CONSTRAINT_ERROR? STORAGE_ERROR when the task is elaborated?

%reference AI-00606

The number of implicit conversions is not considered in resolving overloaded expressions.

%reference RR-0319

Many Ada constructs are allowed in specific contexts but not in other, similar contexts. This violates the "law of least astonishment."

!reference RR-0363

Provide 'value and 'image for floating point and fixed point types.

%reference RR-0426

Many of the ad hoc special cases in Ada are intended to help the programmer, but many of them have actually made life harder for them.

%reference RR-0427

There are many awkward interactions between Ada features that cause compilation problems yet are of little benefit to programmers.

!reference RR-0664

Provide 'value and 'image for floating point types.

!rebuttal

!topic Finalization
!number RI-2022
!version 1.4
!tracking 4.4.2.1 4.4.2.2

DRAFT DRAFT DRAFT DRAFT DRAFT DRAFT

!Issue revision (1) important,moderate impl,upward compat,mostly consistent

1. (Finalization) An attempt shall be made to add finalization as a uniform concept in Ada9X, e.g. finalization operations would be added for all entities for which it makes sense to do so. Possible candidates for finalization include

1. packages;
2. subprogram and block activations;
3. tasks;
4. objects of specified types.

When finalization operations are invoked and the ordering among finalization operations shall be defined by Ada9X. [Note: the order could be defined to be implementation-dependent.]

!reference RR-0003
!reference RR-0092
!reference RR-0676
!reference RR-0203
!reference RR-0385
!reference WI-0507
!reference RR-0019
!reference RR-0168
!reference RR-0466
!reference RR-0475
!reference RR-0523

!problem

Often, a program invokes operations that need to be "undone" or "reversed" as the program exits, e.g. locking databases or seizing physical non-sharable resources. This is equally true for program components; here the operations are more likely releasing record locks or releasing allocated storage. The issue of releasing allocated storage is very important because of the phenomenon of "memory leakage" that occurs when a client uses a dynamically allocated structure and forgets to call the appropriate deallocation routine. The problem is that while it is relatively easy to specify what must be done it is quite difficult to set up the control structure to invoke the "undo" or "deallocate" operation at the correct time.

!rationale

The most important problems that relate to the exit of scope are those of releasing locks and reclaiming storage. In each case, the situation is exasperated in an Ada program trying to cope with a second process model, such as a DBMS. A uniform finalization mechanism should be added to the language to solve these problems.

!appendix

%reference RR-0003 Provide a compiler-independent finalization mechanism

RR-0003 suggests that finalization is needed for library packages. Mention is made of the problem of tying finalization order to elaboration order.

%reference RR-0092 Allow user-specified finalization
%reference RR-0676 Repair asymmetry; add finalization for packages, types

RR-0092 and RR-0676 suggest that capabilities are needed to address three different types of finalization: for packages, for declared objects, and for component objects.

%reference RR-0203 Allow termination code for packages and tasks

RR-0203 wants termination code for packages and tasks. An interesting solution is suggested for tasks--allow code to be placed after a terminate alternative.

%reference RR-0385 Need finalization code for packages

RR-0385 suggests the need for package finalization; a possible solution following UCSD Pascal is presented.

%reference WI-0507 Provide initialization/finalization mechanism for all Ada entities

%reference RR-0019 Allow types to specify finalization procedures

RR-0019 suggests that a capability is needed to supply finalization procedures for types, especially limited private types.

%reference RR-0168 Allow implicitly-invoked finalization code associated with an ADT

RR-0168 wants to associate a finalization procedure with a(n AD)type that is called for each element of the type on block exit.

%reference RR-0466 Allow user-defined finalization for objects of a type

RR-0466 wants to associate finalization procedures with a type. Solutions are presented for a reserved procedure name and a pragma.

%reference RR-0475 Need auto-invoked user-defined routines to reclaim storage

RR-0475 suggest the need for finalization of objects on block-exit to reclaim storage.

%reference RR-0523 Allow user-defined finalization for objects of a type

RR-0523 wants user-defined, automatic finalization of objects of a type; the author could apparently live with limited private types only.

!rebuttal

!topic constraints in renaming declarations
!number RI-5061
!version 1.3
!tracking 4.4.3

!Issue revision (1) important,small impl,moderate compat,consistent

1. (Subtype constraints on renamed objects)
Ada9x shall not allow renaming declarations for objects to specify different subtype constraints than those specified for the original object.

!Issue revision (2) important,small impl,moderate compat,consistent

2. (Subtype constraints on renamed subprograms)
Ada9x shall not allow renamed subprograms to specify different subtype constraints for parameters or function results than those specified for the original subprogram.

!Issue out-of-scope (un-needed) (3) desirable,moderate impl,upward compat,mostly consistent

3. (Constraint checking for results of renamed subprograms)
Ada9x shall provide a mechanism similar to renaming that will "derive" a new subprogram with appropriate constraint checks when the new subprogram specifies different subtype constraints for parameter and function result types than those of the original subprogram.

!reference LRM 8.5 (para. 4 and 8)
!reference RR-0275 There are problems with RENAMEs in the language
!reference RR-0510 (related) Allow renames/subtypes to alter index bounds

!problem

There are several cases where constraint information in renaming declarations is ignored:

1. declarations for objects require that objects be of the base type of the type mark used for the new name. Any constraints implied by the type mark for the new name, however, are ignored (LRM 8.5(4)).
2. Renaming declarations for subprograms similarly ignore subtype constraints for parameters and function results (LRM 8.5(8)).

These rules violate users' expectations that declared constraints are always meaningful.

!rationale

Ada's constraint checking is a program safety feature that should not be easily defeated by other facilities in the language. Ignoring constraints implied by type marks in object and subprogram renaming declarations only serves to make the language less safe. If subtypes are required in object and subprogram renaming declarations, programmers have reason to expect that the subtype's constraints will be enforced on assignment and subprogram calls.

The renaming capability is necessary for resolving name visibility problems and, for this purpose, the ability to specify different subtype constraints is not needed. Since there is a straightforward workaround using explicit conversions, an additional capability to "derive" new subprograms with different subtype constraints (and corresponding checks) from existing subprograms does not seem to be warranted.

Issue revisions [1] and [2] may break some existing programs. One must wonder, however, why these programs should be considered correct.

!appendix

There appear to be three possible solutions to this problem:

1. require declared constraints to be checked
2. eliminate constraint information from the declaration
3. require constraints to match

The first seemed to be asking for too much considering the purpose of the renaming capability. This would provide a way to change index constraints on renamed arrays, however. The second solution would break perfectly good code. The third seems to be the best compromise.

%reference RR-0275
Constraints supplied in renaming declarations should not be ignored.

%reference RR-0510
Renaming array objects should enable one to change the bounds of the index. This would make vector and matrix operations simpler and more efficient.

!rebuttal

The intentions behind [1] and [2] are good but the concern here is that there may be lots of Ada83 code where the constraints do not match in renaming declarations. This is a non-upward-compatible change addressing a relatively minor language problem and should therefore be out of scope.

The checks suggested in [1] and [2] must (in general) be made at the time the renaming declaration is elaborated; hence Ada83 code may compile successfully and only fail at run-time. This aggravates the problem with upward compatibility.

For renaming declarations of objects, the suggestions here would make more sense if a

subtype indication (as opposed to a type mark) were allowed in the declarations. The proposed language change will require not only altering existing renaming declarations but (in general) inserting additional subtype declarations as well.

No better solution than [1] and [2] is proposed here; the difficulty concerning the constraints in renaming declarations is a problem Ada9X should probably live with.

!topic inconsistencies in renaming declarations
!number RI-5062
!version 1.4
!tracking 4.4.3

DRAFT DRAFT DRAFT DRAFT DRAFT DRAFT

!Issue out-of-scope (un-needed) (1) desirable,moderate impl,upward compat,consistent

1 – Renamed construct as prefix] Ada9x shall remove the restriction of LRM 4.1.3(18) that says "A name declared by a renaming declaration is not allowed as the prefix."

[Rationale] This restriction was imposed in Ada83 because of ambiguities that could arise between components of renamed constructs and components of record values returned by parameterless functions. The restriction is perhaps stronger than it needs to be, but does not appear to cause severe problems in practice. Relaxing the restriction should be considered in light of any changes made to renaming declarations; otherwise, no special attention is warranted. The remainder of the problem appears to be a presentation issue.

!Issue revision (2) desirable,small impl,upward compat,mostly consistent

2 – Renaming enumeration literals] Ada9x shall provide a mechanism to declare overloadable, static synonyms for enumeration literals. (Constants declared as synonyms for enumeration literals are not overloadable, and parameterless functions that rename enumeration literals cannot be used in static expressions.)

!Issue revision (3) desirable,severe impl,upward compat,mostly consistent

3 – Uniform and consistent renaming] An attempt shall be made in Ada9x to provide uniform and consistent mechanisms for creating synonyms for arbitrary named language constructs.

!reference AI-00016 Using a renamed package prefix inside a package
!reference AI-00119 The prefix of an expanded name
!reference AI-00412 Expanded names for generic formal parameters
!reference AI-00504 Expanded names with a renamed prefix in generics
!reference RI-5010 Renaming declarations for types
!reference RI-5020 Subprogram body renaming
!reference RR-0096 Remove unnecessary restrictions on RENAMES clause
!reference RR-0275 There are problems with RENAMEs in the language
!reference RR-0570 Allow prefix of a name to denote a renaming
!reference RR-0601 Allow library-level declarations to be defined by RENAME

!problem

1. The rules in LRM 4.1.3 (14, 15, and 18) and the referenced AIs about where a renamed entity can be used as a prefix of a component selection are over-restrictive and confusing.

2. Declaring a constant as a synonym for an enumeration literal creates a value that is static but not overloadable. Renaming an enumeration literal as a parameterless function creates an operation that is overloadable but not static. There is no way to get the combination of an overloadable, static synonym for an enumeration literal.

3. Renaming declarations are not completely systematic and consistent, making language rules and compilers more complicated than they need to be.

!rationale

Providing overloadable, static synonyms for enumeration literals should be relatively simple to implement and would solve a minor problem that cannot be easily worked around.

The principal requirements for extension of Ada83's renaming declarations are covered in this and two other revision issues:

> Requirement [2], above, captures a desirable requirement for renaming enumeration literals.
>
> RI-5010 captures a compelling requirement for allowing renaming declarations for types.
>
> RI-5020 captures an important requirement for allowing subprogram bodies to be implemented by mechanisms such as renaming and generic instantiation.

The intent of requirement [3] is to integrate the renaming of these, and additional constructs where feasible, in a uniform and consistent manner.

!appendix

Renaming declarations for enumeration literals is one obvious solution to requirement [2]. Another, which would have broader implications, is to allow constants to be overloaded. What advantages would the general ability to overload constants have?

Some of the examples of unnecessary restrictions on renaming given in the referenced revision requests, other than enumeration literal, type, and subprogram body renaming, do not provide sufficient basis for a language revision. These include:

> Renaming enumeration literals as character literals (in RR-0096)
> Library-level renames (in RR-0601)

It seems renaming enumeration literals as character literals would have to allow

declarations like:

```
'X': CHARACTER renames 'Y';
```

It is not clear what is meant by library-level renaming. Perhaps something like the following is desired as a compilation unit:

```
with OLD_LIB_PKG;
package NEW_LIB_PKG renames OLD_LIB_PKG;
```

%reference AI-00016

An expanded name is legal if the prefix denotes a package and the selector is a simple name declared within the visible part of the package, regardless of whether the prefix is a name declared by a renaming declaration.

%reference AI-00119

Since the prefix of an expanded name cannot be a name declared by a renaming declaration, names declared by renaming declarations are not considered when deciding whether a selected component is an unambiguous expanded name.

%reference AI-00412

A formal parameter of a generic unit can be denoted by an expanded name.

%reference AI-00504

The prefix of an expanded name occurring within a generic package can be a name declared by a renaming declaration if the selector is a simple name, character literal, or operator symbol declared immediately within the visible part of the generic package.

%reference RR-0096

Renaming has some unnecessary limitations; e.g., an enumeration literal cannot be renamed as a character literal.

%reference RR-0275

It should be possible to rename any construct to provide a shorthand name for a previously defined object.

%reference RR-0570

The restriction of LRM 4.1.3(18) that "A name declared by a renaming declaration is not allowed as the prefix" should be removed.

%reference RR-0601

Library-level renames are not allowed by the standard, even though there appear to be no drawbacks to them, and there are cases in which they would be useful.

!rebuttal

!topic complexity/surprises in overloading
!number RI-1050
!version 1.5
!tracking 4.4.5

DRAFT DRAFT DRAFT DRAFT DRAFT DRAFT

!Issue presentation

[1 - Clarify Definition] Ada9X shall clarify the language rules regarding overload resolution and implicit type conversions.

!Issue revision (2) important

2. (Attempt Simpler Rules) An attempt shall be to simplify and make more natural the Ada9X rules concerning overload resolution and implicit type conversions.

!Issue revision (3) desirable,small impl,upward compat,mostly consistent

3. (Fix Specific Inconsistency) Ada9X shall treat the legality of "-1..10" as a discrete range in an index constraint, iteration scheme, or entry declaration as it does "1..10" in the same circumstances.

!reference RR-0156
!reference RR-0519
!reference RR-0724
!reference AI-00140
!reference AI-00148
!reference AI-00240
!reference AI-00457
!reference AI-00136
!reference AI-00606

!problem

[RI-1050.1]
The Ada83 LRM rules concerning overload resolution and implicit conversions are unclear to many users and implementors alike. As an example, RR-0724 observes that five validated compilers give three different interpretations of the following:

```
function "<" (L,R: INTEGER) return INTEGER;
procedure P(L: BOOLEAN; R: INTEGER);
procedure P(L: INTEGER; R: BOOLEAN);
...
P((1<2)<(3<4), 5<6);
```

[RI-1051.2]
One of the main reasons that the rules concerning overload resolution and implicit

conversions are so poorly understood is because they are extremely complex. Furthermore, some users view the rules as unnatural and counterintuitive. As an example, some users may feel that the Ada in the above example should be illegal since "(1<2)<(3<4)" should always be FALSE (and of type BOOLEAN), "5<6" should always be TRUE (also of type BOOLEAN), and there is no procedure P in the example that takes two BOOLEAN parameters.

[RI-1050.3]
From th ›int of view of the run-of-the-mill Ada programmer, it is baffling that the loop iteration s eme

```
for I in 1..10
```

is legal while at the same time

```
for I in -1..10
```

is illegal. This inconsistency is irritating, and, more importantly, makes the language more difficult to teach. Of course, it should be mentioned that the workaround is quite straightforward:

```
for I in INTEGER range -1..10
```

!rationale

[RI-1051.2]
This is a high-level requirement with no specific compliance criteria. This is because it is not clear precisely what improvements are possible along these lines. What is clear is that a serious attempt must be made to make these rules more understandable and intuitive.

[RI-1051-3]
Although the workaround shown above is trivial to accomplish, this problem is an embarrassment to the language. Furthermore, it seems reasonably easy to solve this problem given the current definition (e.g., by adding, if necessary, a special case to the definition of a convertible operand (LRM 4.6(15)).

!appendix

%reference RR-0156 A negative literal should be allowed wherever a literal is allowed

Allowing "for i in -5..5" would make the language more consistent and easier to learn.

%reference RR-0519 Simplify overload rules for ambiguous/universal expressions

Simplify these rules for the sake of implementors and users. Ideas: (1) don't require implicit conversions to be at the leaves of an expression tree, (2) for things like "null" and string literals say nothing about the assumptions on the type and require the type to be fully resolved from context, and (3) consider every literal as belonging to some "universal type" with an appropriate set of operators.

%reference RR-0724 Need clearer/simpler overload res. rules, esp. w/ implicit conversn

These rules are *very* poorly understood by everyone. Validated compilers are treating expressions differently. The whole matter badly needs clarification. Clarifying rules are proposed.

%reference AI-00140 Allow -1..10 as a discrete range in loops

This is a study AI that suggests fixing this problem.

%reference AI-00148 Legality of -1..10 in loops

This is an approved ramification that confirms that indeed this should not compile.

%reference AI-00240 Integer type definitions cannot contain a RANGE attribute

A ramification confirming this. Not clear why this is present in this RI.

%reference AI-00457 Real type definitions cannot contain a RANGE attribute

A "received" AI. Not clear why this hasn't been filled in and made official. Not clear why it is present in this RI.

%reference AI-00136 Implicit conversion rules

An approved ramification that deals with when implicit conversions are applied.

%reference AI-00606 The number of implicit conversions in overloading resolution

An unapproved ramification that argues that the number of implicit conversions is not considered when resolving an overloaded expression.

!rebuttal

!topic miscellaneous consistency and complexity issues
!number RI-5110
!version 1.5
!tracking 4.4.6

DRAFT DRAFT DRAFT DRAFT DRAFT DRAFT

!Issue revision (1) desirable,moderate impl,moderate compat,mostly consistent

1 – Implicit conversion in overloaded operations] Ada9x should reconsider the rules for implicit conversion of universal numeric values where operations defined over universal types are overloaded. [For further discussion see RI-1050.]

!Issue revision (2) desirable,moderate impl,upward compat,consistent

2 – "OTHERS" in default aggregate values] Ada9x shall allow named associations for default array aggregates to include an OTHERS choice where no subtype conversion is applied to the aggregate; i.e., no "sliding" of index values will be allowed. [Note: This requirement reflects the intent of LRM 4.3.2(6) as described in AI-00473.]

!Issue revision (3) desirable,moderate impl,moderate compat,mostly consistent

3 – Consistent semantic rules] Ada9x shall provide consistent semantic rules for syntactic constructs that appear in different parts of the language. In particular, the conformance rules for identifier lists in subprogram specifications [LRM 6.3.1] should be similar to the equivalence rules for single and multiple object declarations [LRM 3.2(10)], rather than being based on lexical structure.

!Issue out-of-scope (un-needed) (4) desirable,moderate impl,moderate compat,consistent

4 – Default constant values] Ada9x should provide for defining constants with default values obtained from their type definitions.

[Rationale] Constants require explicit initialization clauses and, hence, cannot easily assume default values provided in their type declarations. This is a language anomaly but it does not appear to cause sufficient programming difficulties to warrant a language change.

!Issue out-of-scope (un-needed) (5) desirable,moderate impl,moderate compat,consistent

5 – I/O operations for characters] Ada9x should provide GET_LINE and PUT_LINE operations for CHARACTERs in TEXT_IO.

[Rationale] Declarations of GET_LINE and PUT_LINE are not consistent with GET and PUT, which are defined for both CHARACTER and STRING arguments. This requirement would imply that INTEGER_IO, FLOAT_IO, FIXED_IO, and ENUMERATION_IO should also provide GET_LINE and PUT_LINE operations. Another way achieve consistency would be to eliminate GET_LINE and PUT_LINE

for STRINGs, but this would not be upward compatible. The problem does not appear to cause sufficient difficulties to warrant a language change.

!Issue revision (6) desirable,small impl,upward compat,consistent

6 -- 'RANGE for Scalar Types] Ada9X shall define the RANGE attribute for scalar types consonant with the definition of 'RANGE for array types and objects.

!reference AI-00136 implicit conversion rules
!reference AI-00473 named associations for default array aggregates
!reference AI-00606 implicit conversion in overload resolution
!reference RR-0094 complete and consistent declaration rules
!reference RR-0100 default constants
!reference RR-0132 (related)
!reference RR-0155 Define RANGE attribute for scalar types
!reference RR-0193 (related)
!reference RR-0295 text_io.put_line for types other than string
!reference RR-0304 Define RANGE attribute for scalar types
!reference RR-0321 (related)
!reference RR-0344 simplify/relax the conformance rules
!reference RR-0623 Define RANGE attribute for discrete ranges
!reference RR-0631 relax and make consistent conformance rules
!reference RR-0728 (related)
!reference WI-0219 consistency of generics

!problem

Special case rules, inconsistencies across language features, and obscure interactions between features make learning and using Ada more difficult and error-prone than necessary. Specific examples covered by these requirements include:

1. Overload resolution rules for operations defined on universal types
2. Allowable default values for arrays in subprogram declarations
3. Consistent semantic rules for similar syntactic structures that appear in different parts of the language
4. Consistent combinations of attributes

!rationale

Special case rules, inconsistencies across language features, and obscure interactions between features increase the number of programming errors made and extend both software development and training time. Language changes that simplify and unify rules and features are expected to reduce overall Ada software development costs.

!appendix

There is every reason to believe that the mapping/revision team should visit each predefined attribute and determine the most consistent/ orthogonal definition.

%reference AI-00136

What are the rules for implicit conversion of universal operands when predefined operations on universal types are overloaded?

%reference AI-00473

Named associations for default array aggregates should be allowed but are currently forbidden by LRM wording.

%reference AI-00606

The number of implicit conversions is not considered in resolving overloaded expressions.

%reference RR-0094

The concise syntactic form of multiple object declarations and their equivalence to sequences of single declarations should be applied consistently to other declarations.

%reference RR-0100

Constant declarations cannot automatically pick up the default value defined for the type.

%reference RR-0132 (related)

Allow a "when" clause on "raise" statements similar to that for "exit" statements.

%reference RR-0155

All of RR-0155, RR-0304, and RR-623 want to have a RANGE attribute for scalar and/or discrete types; the argument is for consistency and readability.

%reference RR-0193 (related)

Require consistent implementation of tasking priority for allocating processing resources.

%reference RR-0295

The definitions of PUT_LINE and GET_LINE in package TEXT_IO are not consistent.

%reference RR-0304 (see RI-0155)

%reference RR-0321 (related)

Allow anonymous array and record types as components of array and record declarations.

%reference RR-0344

Rules for conforming declarations should be stated more simply and optional syntax, such as "in" for subprogram input parameters, should be ignored.

%reference RR-0623 (see RI-0155)

%reference RR-0631

Some rules for conforming declarations require "equivalence" while others require identical lexical elements. Make the rules consistent.

%reference RR-0728 (related)

%reference WI-0219 consistency of generics

Non-uniformities in the treatment of generic units should be removed: in particular, all language constructs should have consistent rules whether used inside non-generic program units or inside generic units.

!rebuttal

4.5 Information Hiding

Section 4.5 includes three RIs:

RI-3000
RI-5020

!topic Information Hiding.
!number RI-3000
!version 1.11
!tracking 4.5.1/4.5.1.2

DRAFT DRAFT DRAFT DRAFT DRAFT DRAFT

!Issue revision (1) important,small impl,upward compat,consistent

1. (Defer Implementation Details in Body) Ada9X shall allow the implementation of a subprogram body by the renaming of a conforming subprogram body or generic instantiation.

!Issue revision (2) desirable,severe impl,upward compat,consistent

2. (Private Part Separation) Ada9X shall reduce the dependency that compilation units have upon information given in the private part of a package in order to reduce compilation dependencies and recompilation overheads.

!Issue revision (3) desirable,moderate impl,upward compat,consistent

3. Ada9X shall allow the declaration of visible non-constant objects of a private type or of a record type containing a private type.

!reference RAT-9.3.3
!reference RI-1016
!reference RI-2500 (STRIPES)
!reference RI-5020
!reference RR-0043
!reference RR-0055
!reference RR-0093
!reference RR-0098
!reference RR-0153
!reference RR-0157
!reference RR-0231
!reference RR-0268
!reference RR-0307
!reference RR-0313
!reference RR-0364
!reference RR-0423
!reference RR-0451
!reference RR-0470
!reference RR-0542
!reference RR-0550
!reference RR-0666
!reference RR-0667
!reference RR-0725
!reference RR-0764

!reference AI-00270
!reference AI-00327
!reference AI-00404
!reference WI-0207
!reference

Parnas, David L. : On the Criteria to be used in Decomposing Systems Into Modules, CACM, Dec. 1972.

!problem

Ada83 is perceived to not fully support the principle of information hiding. The textual separation of information between package specification, private and body parts causes information flow evidenced in recompilation requirements. Additionally, some constructs are allowed in package specifications that evidence subprogram implementations while the same constructs are precluded in package bodies (e.g., renames). Finally, the limitation on use of objects of, and private types prior to complete declaration appears to be limited for similar reasons and fixed by similar requirements.

!rationale

The principle of information hiding is an accepted principle of modern software engineering since [Parnas]. This principle holds that a module is

> "characterized by its knowledge of a design decision [whether data structures or algorithms] which it hides from all others. Its interface or definition was chosen to reveal as little as possible about its inner workings."

Ada83 generally supports the principle of information hiding. However, there exist a number of features that appear to be contrary to this principle in the language. The most common example cited is the recompilation requirements that derive from implementation information that is placed in a package specification, particularly the private portion of such a package. Although the principle of information hiding may have been originally applied to the behavioral aspect of a module, an extension can be applied to the textual representation aspects of a module, particularly as implemented with the Ada private concept. This is clearly a case of information flow.

In the original Parnas example, the discussion centered on the hiding of information to limit the number of source modules that had to be modified when a change was made. A parallel can be made with respect to the number of modules that have to be recompiled when a change is made to a package private part in Ada. Although no actual source changes need to be made to the corresponding body modules, the impact of the change in the private part nonetheless propagates in a similar manner.

A tradeoff occurs with respect to the presentation of information in the private part of a package specification that is claimed to be necessary for proper (and/or efficient) code generation of users of packages. [Rationale, in Appendix] It is not clear that this information is so much required by the language as by the current paradigm and

mechanisms for program compilation. If such information is deemed to remain necessary, then full support of information hiding as defined here will have to continue to be accepted as an unattainable goal. A certain consideration in this respect will be the possible implementation impact upon existing compilation systems.

Finally, to fully adhere to the principle of information hiding would require that the current capabilities (such as allowing renames in the package specification) in the language be removed. Since this would be clearly non-upwardly-compatible, it must be rejected from consideration.

Requirement 1 is designed to remove the current dependence upon the specification of a package when renames are used in the specification. They are currently prohibited from being performed in the body.

Requirement 2 requests relief from the massive recompilations that arise when only the private parts of a package are changed. One approach may be to allow the private part of a package to be separated from the package specification, either as a separate compilation unit, or in the body of the package. Implementation should be discouraged from creating compilation order dependencies upon such private parts, except where optimizations force such dependencies. It is anticipated that, as compilers mature, (as happened) with separate specifications and bodies of generics, (RI-1016)) they will reduce this dependence.

Requirement 3 also requires the use of thunks, but cannot be alleviated by a user-specifiable option. It, and #3 may be possibly solved in a different manner by the STRIPES proposal (RI-2500).

!appendix

Requirements [2-4] are really !sub-requirements, or even supporting requirements, possibly even CONSTRAINTS, v. the GOAL of [1]. Unfortunately, we have no way of expressing this within the current template.

%reference RI-1016 true separation of generic specifications and bodies.

%reference RI-2500 STRIPES

This is supposed to be written in the future.

%reference RI-5020 subprogram body implementation

This RI posits the issue as a clean-up item specifically for renaming, rather than in the more global context of information hiding.

%reference RR-0043 Make easier (and more portable) to use assembler with Ada

This RR makes the point that the use of pragma interface in a package specification violates information hiding, and that it should be allowed in the package body, rather than required in the package specification.

%reference RR-0055 Allow a subprogram body to be defined as "another" subprogram body

This revision request wants to be able to do renames in a package body. The two issues are (1) deferring the rename and (2) the syntax for the rename. Only issue (1) is considered covered here. It is anticipated that issue (2) is covered elsewhere.

%reference RR-0082 Allow declaration of private objects in visible package spec

The limitations on the use of incomplete private types appears to be overly restrictive. (See also RR-0542, RR-0153 for possible rationale.)

%reference RR-0093 Allow constants to be deferred to package body

This revision request wants to have deferred constants in package bodies, a la RR-0313.

%reference RR-0098 Generalize incomplete typing for types other than access or private

This RR wants to be able to use incomplete types in more areas than are currently allowed. It is similar in come senses to RR-0082.

%reference RR-0153 Private part foils separation of specification and implementation

This revision request describes to the use of private parts in package specifications as antithetical to the principle of information hiding. A change in the private part, which is a part of the implementation, requires recompilation not only of bodies, but also of clients, which is claimed to violate the principle of information hiding.

Unfortunately, this is a case where the principle of information hiding must be traded off against the needs of the separate compilation facility. In the Rationale, section 9.3.3:

> This extra information is needed by compilers for the treatment of variables that are declared in one compilation unit but whose type is a private type declared in a different compilation unit. The difference essentially concerns storage allocation: knowledge of the amount of storage needed for such variables is necessary for selecting the machine instructions used for operations on the variables; this code selection is not a decision that could be postponed until the program is complete (that is, until linkage editing time).

%reference RR-0157 Allow renaming when defining a subprogram body

This revision request notes that the inconsistency between the ability to rename procedures in package specifications and not in package bodies violates the principle of information hiding. Deferring should be covered here, while the syntax for renaming should be covered elsewhere.

%reference RR-0231 Allow a rename definition of a subprogram body

This revision request wants renames that are currently possible in a package specification to be allowed in a package body. This would support information hiding.

%reference RR-0268 Separation of spec and body is not worth it

This revision request desires (among other things) that the concept of specification and body separation be removed. Such a request is contrary to the principle of information hiding.

%reference RR-0307 Allow private ???? to be detailed in the package body

This revision request wants the private part of a package specification to be physically removed from the package specification because it includes implementation details. The practical result of the current situation is that changes to the package private parts, which only affect the implementation and not the visible interface, require recompilation of not only all bodies in the package, but also all users of the package.

Unfortunately, as mentioned earlier (RR-0153 analysis) there is information in the the package private part that is deemed necessary for code generation. However, it may be only an artifact of current compiler technology that code generation is performed at the same time that syntactic and semantic analysis is performed. If it were possible to defer code generation to just prior to the above-mentioned "linkage" time, then this separation might be effected by some mechanism.

%reference RR-0313 Allow deferred constants of arbitrary (i.e., non-private) types

This revision request desires to have deferred constants of arbitrary types. That the constant must be fully defined in the private part of the package specification is a perceived violation of the principle of information hiding. Allowing the full declaration of such constants to be deferred to the package body would alleviate this situation.

%reference RR-0364 Allow subprogram body to be defined by generic instantiation

This revision request seeks to allow instances of generic subprograms as subprogram bodies in a package body. It is currently possible to do so in the package specification. (This is possibly also a matter of regularity of the syntax of renames.)

%reference RR-0423 Remove restrictions on full declarations of private types

This revision request desires the relaxation of the rule that prevents the full declaration of a private type from containing discriminants. This is in concert with maintaining the consistency with package instantiation on types with discriminants that was remedied in AI-00037/12. This allows the programmer to implement the private type in a appropriate manner without information flow to the specification.

%reference RR-0451 Separate declaration and initialization of all data objects

This revision request probably describes the ultimate in deferral for constants - to the start of execution of the program. Possible workarounds are to read them in from some startup file, set of command line arguments, prompting, or obtaining the value from some location in memory. Pragma interface might cover the case at link time, but not at run time. This issue has a significant overlap with program building, RI-????

%reference RR-0470 Allow renaming or generic instantiation to define subpgm body

This revision request wants the ability to defer the rename of a subprogram body to a generic instantiation in the package body.

%reference RR-0542 Allow usage of private type before its comp. decl

The usage of private types prior to complete declaration appears to be overly restricted. (See also RR-0082, RR-0153 for possible rationale.)

%reference RR-0550 Allow subpgm bodies to be defined by RENAME or gen. instantiation

Allow a subprogram body to be supplied by a generic instantiation with the proper parameter profile in a package body.

%reference RR-0577 Allow deferred composite of non-prvt. type where element is private

Another limitation on the use of private types with incomplete declarations. (See also RR-0082, RR-0153 for possible rationale.)

%reference RR-0666 Allow subpgm body to be given by generic instantiation

Wants bodies to be provided by an instance of an appropriate generic in the body.

%reference RR-0667 Allow subpgm body to be given by RENAME

Wants bodies to be provided by a rename in a package body.

%reference RR-0725 Need rename in pkg body for routine in pkg spec

Allow renames in a package body.

%reference RR-0730 The private part of a package should have its own context clause

Requiring a context clause on the package specification simply to support the private type implementation introduces an unnecessary dependence, just to implement the type, rather than to specify it.

%reference RR-0764 Allow subprogram bodies to be defined by RENAME

Allow renames in a package body.

%reference AI-00270 The properties of an object declared to have a private type

The properties of a private type are different depending upon whether its declaration is completed or not. (Sounds like a stripes issue also.)

%reference AI-00327 Instantiating with an incomplete private type

Uses of an incomplete private type are too constrained. They preclude doing things that might be useful.

%reference AI-00404 Incomplete types as formal object parameters

Allow greater use of a type with an incomplete declaration.

%reference WI-0206 Provide better user control of elaboration

Allow renames in a package body.

!rebuttal

!topic subprogram body implementation
!number RI-5020
!version 1.6
!tracking 4.5.1.2

!Issue revision (1) important,small impl,upward compat,consistent

1. Ada9x shall allow subprogram bodies to be specified by identifying another (conforming) subprogram that is to serve as the implementation or by instantiating a (conforming) generic subprogram.

!reference RI-3000 (related) Information hiding (requirement ???)
!reference RI-5060 (related) Inconsistencies in renaming declarations
!reference RR-0055 Allow one subprogram body to be defined by another
!reference RR-0096 (related) Remove unnecessary restrictions on renames
!reference RR-0157 Allow renaming when defining a subprogram body
!reference RR-0231 Allow a rename definition of a subprogram body
!reference RR-0364 Allow subprogram body to be defined by instantiation
!reference RR-0470 Allow renaming or instantiation to define subpgm body
!reference RR-0550 Allow subpgm body definition by RENAME or instantiation
!reference RR-0666 Allow subpgm body to be given by generic instantiation
!reference RR-0667 Allow subpgm body to be given by RENAME
!reference RR-0725 Need rename in pkg body for routine in pkg spec
!reference RR-0764 Allow subprogram bodies to be defined by RENAME
!reference WI-0207 Allow subpgm body definition by RENAME or instantiation

!problem

Ada83 allows subprogram bodies to be defined only by a code block. Where a subprogram is to be implemented by renaming another subprogram or by a generic instantiation, there are two possible approaches:

1. A renaming declaration or generic instantiation can be used for the subprogram specification, but this requires revealing the subprogram's implementation and, possibly, context information as well, which violates separation of specification and implementation details; or

2. The subprogram body can call the actual implementation subprogram.

A less verbose and more efficient solution is to allow a subprogram body to be implemented directly by an existing subprogram or by an instantiation of a generic subprogram.

!rationale

Improving separation of specifications from implementation details should be part of the Ada9x "clean-up" revisions.

!appendix

1. Ada83's single allowable form of subprogram body definition does violate the language's general principle of separating specification and implementation details.

2. Renaming declarations are alternative forms of subprogram definition that are allowed in other contexts and, hence, programmers could reasonably expect them to work for body definitions.

3. Ada83's subprogram renaming mechanism is not a good solution to this problem, because it allows tighter subtype constraints to be specified for returned results in the "new" subprogram but it does not check to ensure that they are satisfied. The solution should allow tighter subtype constraints and "derive" a new subprogram that includes the checks.

4. Following the line of reasoning in paragraph 2, above, package and task bodies should also be definable by this new mechanism. No revision requests appear to ask for these broader changes.

5. The work-around of defining a "dummy" body that calls the actual implementation subprogram is not particularly elegant and the extra level of subprogram call is a concern for real-time applications. A legal implementation technique, however, is to make the extra call anyway, not to create a synonym.

6. Implementing this revision should be relatively simple and should not impact compilers significantly.

%reference RI-3000

RI-3000 covers a broader scope of issues related to information hiding. Hiding subprogram implementation details is just one example.

%reference RR-0055

The body of a subprogram cannot be produced, independently of the specification, by a generic instantiation or by renaming another subprogram.

%reference RR-0096 (related) Remove unnecessary restrictions on renames

%reference RR-0157

it is desirable to implement subprograms in package bodies by renaming other subprograms.

%reference RR-0231

It is not possible to separate a subprogram specification from its body where the implementation is to be by a renaming declaration.

%reference RR-0364

Renaming declarations and generic instantiations should be allowed where subprogram bodies are required.

%reference RR-0470

It is desirable to hide subprogram implementation details by providing the specification and the body separately. The body, however, cannot be provided by renaming another subprogram.

%reference RR-0550

Subprogram bodies cannot be supplied by renaming or instantiation, and requires duplicate declarations violating the factorization principle.

%reference RR-0666

Allow subpgm body to be given by generic instantiation It is customary practice to design subprogram specifications and bodies separately. Bodies, however, cannot be provided by instantiating generic subprograms.

%reference RR-0667

It is customary practice to design subprogram specifications and bodies separately. Bodies, however, cannot be provided by renaming other subprograms.

%reference RR-0725

Subprogram renaming in a package specification requires recompilation of the entire package when the subprogram body is modified.

%reference RR-0764

Ada83 treats renaming declarations as subprogram specifications but not also as subprogram body definitions; this can be inconvenient for structuring programs.

%reference WI-0207

When declaring a subprogram body, it should be possible to indicate that that subprogram is only an alternate name for an existing one with the same parameter modes and types.

!rebuttal

4.6 Reuse

Please refer to RI-1017 in Section 5.13, Generics.

4.7 Portability/Interoperability

RI-3986

!topic Portability.
!number RI-3986
!version 1.9
!tracking 4.7.1, 4.7.4

!Issue revision (1) compelling,small impl,upward compat,consistent
[1 - Overarching Goal] Ada9x shall attempt to maximally support the principle of portability.

!Issue administrative (2) compelling

2. (Dependency Documentation) An implementation shall provide as much information on implementation dependent characteristics as necessary to support portability.

!reference AI-00584
!reference LRM 1.1
!reference RI-3429
!reference RR-0043
!reference RR-0252
!reference RR-0253
!reference RR-0254
!reference RR-0333
!reference RR-0346
!reference RR-0365
!reference RR-0371
!reference RR-0432
!reference RR-0698
!reference Rationale 5.4
!reference WI-0218
!reference WI-0218M

!reference

Kernel Ada Programming Support Environment (KAPSE) Interface Team Public Report, Volume 1. Patricia A. Oberndorf (NOSC), Chairman. 1 April 1982, NOSC Technical Document 509.

!reference

Portability and style in Ada. Edited by John Nissen and Peter Wallis. The Ada Companion Series, Cambridge University Press. 1984.

!reference

Ada Tool Transportability Guide, KAPSE Interface Team, Guidelines and Conventions Working Group, Draft. October, 1987. Additional, apparently later release, undated. (Available from ajpo.sei.cmu.edu.)

!problem

The purpose of the Ada standard is "to promote the portability of Ada programs to a variety of data processing systems." (LRM 1.1) However, a number of locations in the standard allow compilers to implement various aspects of the language in the most appropriate manner for the particular target environment. These locations are evidenced by phrases like "in a manner that is not specified", or "is implementation-dependent", or "may be either ... . Programmers are often unable to determine what manner has been chosen for many of these alternative without resorting to writing test programs to attempt to discern the behavior.

Examples of some of these freedoms include:

1. extraction of mantissa and exponent from a floating point number without recourse to extensive run time execution.
2. use of 'DIGITS rather than 'BITS for numeric computations
3. meaning of 'ADDRESS, its relationship to values of access types

!rationale

Ada83 allows compiler vendors to make numerous decisions throughout the implementation that are either "not defined by the language", or "implementation dependent", etc. The only manner in which the effect of some of these decisions can be determined is through trial and error. Portability of software would be improved if these implementation dependencies could at least be documented. Should it be found that some of the decisions are universally implemented, such implementation freedoms should be considered for standardization. Documenting these decisions may not actively improve portability, but it will at least provide programmers the opportunity to get information on an implementation's characteristics.

!appendix

This problem is the "blind men and the elephant" syndrome.

I have subsumed many of the RRs from 4.7.4 (Miscellaneous) into this RI as well. That may make the entire thing too abstract (as might be levied against ri-3000), but it appears that the approach that lists the problems in the !problem statment is an appropriate abstraction. (RI-3429 addresses the portability problem specifically with respect to rounding as an example of how these individual issues might appear.) This is to test drive the level of approach for these RIs in comparison to that taken in RI-3000.

Portability and interoperability are recurring themes in the KIT Public Report.

%reference Rationale 5.4

[The above example illustrates] the essential dilemma between efficiency and portability that intrudes into certain sensitive areas of numerical computation; there are occasions where the demands for a very efficient implementation outweigh those for compiete portability. The facilities in Ada enable the non-portable parts to be readily identified and encapsulated so that a proper balance between the conflicting aims can be obtained.

%reference RR-0043 Make easier (and more portable) to use assembler with Ada

This RR indicates that the general approach to the use of machine code.sp insertions is flawed with respect to portability. In particular, it argues that the constraints placed on the implementation are insufficient to allow portable software to be developed because individual vendors and implementations might provide different interfaces to the equivalent services.

Perhaps this should be considered as a secondary standards issue.

%reference RR-0252 Math model leads to inconvenience, inefficiency, non-portability

The entire math model in Ada is insufficient to p' 'de an effective, efficient (both in terms of development and execution performance), and portable approach to doing software development for use with numeric algorithms.

%reference RR-0253 Digits and DELTA approach leads to inefficiency, non-portability

DIGITS specification of floating point precision is insufficient to support portable numeric applications in a straightforward manner.

%reference RR-0254 Too much freedom allowed wrt exceptions and intermed. expr results

Lack of semantics for intermediate expressions leads to non-portability of programs.

%reference RR-0333 Unpredictable behaviour of TEXT_IO

This RR wants a tighter definition of TEXT_IO in order to make the behavior of programs more predictable. No specific constructs are mentioned.

%reference RR-0346 Need portable way to extract mantissa/exponent from fp number

The only method currently available within Ada to extract mantissa and exponent from a floating point number is to execute a program. Adding such functions to the language would not necessarily be any more portable, but could be made more efficient (unless such programs are currently implemented by implementation dependent solutions). However, the ease of providing these features from within the language (and hence the compiler implementation) is relatively simple and straightforward.

%reference RR-0365 Implementation options lead to non-portability and non-reusability

This RR wants (at least) better documentation on the implementation dependent choices taken throughout the language implementation.

%reference RR-0371 Need more usable and portable machine code insertions

%reference RR-0432 Implementation options lead to non-portability and non-reusability

This RR is substantially equivalent to RR-0365.

%reference RR-0698 Need pragma to identify machine-dependent pieces of program

This RR suggests a pragma based solution to being able to provide conditional compilation of units based upon the compilation environment. This same effect (which is insufficient for the conditional compilation needs, particularly wrt declarations, etc.) can be handled by the following program sample.

```
with system;
with text_io; use text_io;
procedure test_system is
  type systems is (sun_unix, vax_unix);
  x : constant systems
      := systems'value (system.name'image(system.system_name));
begin
  case x is
    when vax_unix =>     put_line ("vax_unix");
    when sun_unix =>     put_line ("sun_unix");
    when others =>   put_line ("erro  :");
  end case;
  if x = vax_unix then
    put_line ("vax_unix");
  elsif x = sun_unix then
    put_line ("sun_unix");
  else
    put_line ("error 2");
  end if;
end test_system;
```

Note that neither solution (the proposed one, or the one indicated above) solves the so-called "pre-processor problem", of being able to conditionally compile declarations or procedures, do macro expansions, etc.

The following three references are examples of items that result from the indiscriminant use of features specific to an implementation (e.g., INTEGER, INTEGER'SIZE). The obvious solution to the writing of bad Ada in these situations is to NOT use implementation specific characteristics and expect them to be portable.

%reference WI-0218 Implicit use of type INTEGER in loop vars, etc. is undesirable

situation is to be more specific on the type wanted, which is perfectly allowable in Ada83.

%reference WI-0218M Define the set of predefined numerics, and characteristics

The minority position is that there should be a set of predefined numerics, (e.g., 32 bits for INTEGER) that should be imposed upon all implementations. This would have adverse effects on low-end implementations.

%reference AI-00584 Restrict argument of RANGE attribute in Ada 9x

This AI is really about the use of INTEGER'SIZE in an implementation specific manner.

!rebuttal

5. SPECIFIC ASPECTS OF THE LANGUAGE

5.1 Syntactic and Lexical Properties

RI-2003
RI-1081
RI-1082
RI-1083
RI-1084

!topic Character Set Issues
!number RI-2003
!version 1.9
!tracking 5.1.1 5.2.2.2

DRAFT DRAFT DRAFT DRAFT DRAFT DRAFT

!Issue revision (1) essential,small impl,moderate compat,consistent

1. (Base Character Set) The base character set of Ada9X shall be based on the ISO8859-1 LATIN-1 character set in the same way as ASCII was the basis of the character set in Ada83.

!Issue revision (2) essential,small impl,upward compat,consistent

2. (Extended Graphics Literals) Ada9X shall define a mechanism by which an implementation can extend the set of graphic symbols that may be appear in character and string literals. The mechanism shall specifically address how an implementation may provide literals for graphic symbols from other parts of ISO 8859 (i.e. other than LATIN-1) and from international character sets with more than 256 graphic symbols.

!Issue revision (3) essential,moderate impl,upward compat,consistent

3. (Extended Character Set Support) Ada9X shall provide input/output facilities for extended character sets (specifically, for ISO 8859 and for international character sets with more than 256 graphic symbols) comparable with what is provided for ISO 8859-1.

!Issue revision (4) important,small impl,upward compat,consistent

4. (International Graphics in Identifiers) Ada9X shall extend the set of graphic symbols used in identifiers to include ones not in ASCII.

!Issue presentation (5) compelling

5. (Unrestricted Comment Graphics) The revised standard shall clearly specify that graphics not in base character set may appear in comments. [Note: it may be argued that the current standard already does so.]

!reference RR-0034
!reference RR-0330
!reference RR-0367
!reference RR-0619
!reference RR-0746
!reference RR-0438
!reference WI-0201
!reference WI-0202
!reference WI-0201M
!reference WI-0202M

!reference ai-00420
!reference RR-0050
!reference RR-0148
!reference RR-0311
!reference RR-0331
!reference RR-0339
!reference RR-0390
!reference RR-0736

!reference

[Brender 1989] Ronald F. Brender Character Set Issues for Ada9X Software Engineering Institute Special Report SEI-89-SR-17 October 1989

!problem

It is difficult to create Ada83 programs that deal with characters and strings in languages other than English in a portable way. Additionally, creation of programs that are easily maintained by non-English-speaking programmers is difficult.

!rationale

It is very unlikely that Ada9X will pass the next standardization muster unless solutions are found for problems faced by non-English-speaking programmers trying to use Ada. These requirements address the concerns raised by the referenced revision requests. I/O support for ASCII-with-parity and other 256-way sets should be considered when revising the I/O consonant with RI-2003.3.

!appendix

%reference RR-0034 Ada should use ISO 8859/1-9 (8-bit) character set

RR-0034 recommends adopting ISO standard for 8-bit ASCII instead of ISO standard for 7-bit ASCII.

%reference RR-0330 Allow national characters in literals, comments and identifiers

RR-0330 explains the problems rather fully. Possible solutions include pragmas to tell which character set is being used, or extension to the LATIN-1 set, or standardizing the lower 128 characters and leaving the top 128 as implementation defined.

%reference RR-0367 Need support for national lang. char. sets; string comparison too

RR-0367 explains that Ada is not so good for French; ISO8859 is recommended as a adequate solution.

%reference RR-0619 Eliminate three replacement characters, stick to normal ASCII

%reference RR-0746Allow pictures/graphics as comments in source code

%reference RR-0438Allow use of multi-octet character set

RR-0438 suggests that the use of multiple-octet character sets would be an adequate solution to the problem.

%reference WI-0201Support use of int char sets, comments & char literals minimum

%reference WI-0202Support for int char sets in keywords and identifiers is optional

%reference WI-0201MPermit other char sets, inc multi-byte in literals, strings, IO

%reference WI-0202MSupport int char sets for comments, literals, identifiers

%reference ai-00420Allow 256 values for type CHARACTER

%reference RR-0050 Provide multinational and multibyte characters

RR-0050 suggests that all that need be done is to allow 0..255 representation with the bottom half ASCII and the top half implementation-defined. The author is concerned with early standardization.

%reference RR-0148 Provide support for extended and graphic characters (256 ASCII set)

RR-0148 suggests that STANDARD.CHARACTER be extended to a 256-long enumeration and the names be provided for the top half. Portability if the primary concern.

%reference RR-0311 Generalize standard.character, decouple ada from character set

RR-0311 is concerned that the definition of STANDARD.CHARACTER and other character facilities is TOO detailed. The suggestion is that Ada not prohibit larger (i.e. more than 128) enumerations for STANDARD.CHARACTER and also not proscribe additional graphic symbols.

%reference RR-0331 Need pre-defined long_character (16 bits) & long_long_character (32)

RR-0331 explains that 16- and 32-bit representations are needed to represent the graphics in some languages. A suggested solution is that new predefined types be added to the language.

%reference RR-0339 Provide support for string comparison base

RR-0339 explains that different languages imply different sorting algorithms; suggestions are made as to the appropriate location of the desired facilities.

%reference RR-0390 Need 8-bit unsigned CHARACTER for Greek and graphics symbols

RR-0390 discusses a problem with communications software that must deal with incoming parity bits; the suggestion is that an 8-bit character type would be helpful.

%reference RR-0736Need 8-bit ASCII in Ada

!rebuttal

!topic consistency of syntax
!number RI-1081

%reference RR-0028
%reference RR-0132
%reference RR-0141
%reference RR-0200
%reference RR-0362
%reference RR-0499
%reference RR-0625
%reference RR-0751
%reference RR-0753

!problem

There are potential changes to the syntax of the language that might improve the internal consistency of the syntactic rules. Refer to the above-referenced revision requests for details.

!appendix

Note no judgement is being passed on these suggestions.

%ref RR-0028 Add a semicolon terminator to SEPARATE statement syntax
%ref RR-0132 Allow "when condition" on raise for consistency w/exit stmt
%ref RR-0141 Allow "when <condition>" on ""aise" statements
%ref RR-0200 Allow optional when_clause on raise and return statements
%ref RR-0362 Allow optional when_clause on the raise statement
%ref RR-0499 Like other "blocks", allow exception handlers in accept stmts.
%ref RR-0625 Change EXIT/WHEN to WHEN/EXIT to parallel Ada IF and English
%ref RR-0751 Add when/raise construct to the language
%ref RR-0753 Make syntax for task type decls more consistent

!topic self-documenting syntax
!number RI-1082

!reference RR-0199
!reference RR-0205
!reference RR-0340
!reference RR-0596
!reference RR-0673

!problem

There are potential improvements to the syntax of the language that might make Ada9X programs more self-documenting. Refer to the above-referenced revision requests for details.

!appendix

Note no judgement is being passed on these suggestions.

%ref RR-0199 Allow IF, CASE, and SELECT constructs to be named
%ref RR-0205 Allow program unit name on "private", "begin", and "exception"
%ref RR-0340 Allow optional simple name on case, if, and select statements
%ref RR-0596 Allow "END type_name" to substitute for "END RECORD"
%ref RR-0673 Allow "END RECORD type_name" to substitute for "END RECORD"

!topic clarity of syntax
!number RI-1083

!reference RR-0126
!reference RR-0250
!reference RR-0251
!reference RR-0264
!reference RR-0397
!reference RR-0491
!reference RR-0534
!reference RR-0556
!reference RR-0632
!reference RR-0695
!reference RR-0708
!reference RR-0755

!problem

There are potential improvements to the syntax of the language that might make Ada9X programs easier to read. Refer to the above-referenced revision requests for details.

!appendix

Note no judgement is being passed on these suggestions.

%ref RR-0126 Allow underscore before "E" in exponents
%ref RR-0250 Define clearer notation for expressing null ranges
%ref RR-0251 Syntax for names and expressions is confusing
%ref RR-0264 Discriminants need to stand out more
%ref RR-0397 Replace keyword "pragma" with something capturing meaning better
%ref RR-0491 Code would be clearer if one could "exit" from a block statement
%ref RR-0534 Allow brackets other than "(", ")" in aggregates, etc.
%ref RR-0556 Parentheses are used for too many purposes in the language
%ref RR-0632 Allow EXIT from a block statement for consistency
%ref RR-0695 Allow EXIT from block for legibility
%ref RR-0708 Allow infix function calls
%ref RR-0755 Allow '[' instead of '(' for indexed components

!topic convenience of syntax for programmer
!number RI-1084

!ref RR-0049
!ref RR-0152
!ref RR-0221
!ref RR-0391
!ref RR-0504
!ref RR-0548
!ref RR-0614
!ref RR-0615

!problem

There are potential changes to the syntax of the language that might make programming more convenient for the user. Refer to the above-referenced revision requests for details.

!appendix

Note no judgement is being passed on these suggestions.

%ref RR-0049 Allow special notation when same name on both sides of :=
%ref RR-0152 Allow e.g., a < b < c which would mean a < b AND b < c
%ref RR-0221 Need to write common code for group of exception handlers
%ref RR-0391 Clumsy syntax for based numbers, esp. in aggregates
%ref RR-0504 Add an exchange operator
%ref RR-0548 Allow convenient syntax for instantiating nested generic unit
%ref RR-0614 Allow WHEN/RETURN in functions similar to EXIT/WHEN for loops
%ref RR-0615 Define LOOP/UNTIL control structure as in Pascal

5.2 Specific Kinds of Types

RI-5120
RI-5130
RI-5141
RI-5142
Ri-5150
RI-5160
RI-2032
RI-2033
RI-5170
RI-0901
RI-0902
RI-0904
RI-0905
RI-0906

!topic aggregates
!number RI-5120

!reference AI-00473 Named associations for default array aggregates
!reference AI-00681 Can't declare a constant of a 'null' record type
!reference RR-0029 Allow a more flexible use of others clause in aggregates
!reference RR-0053 Allow aggregates for null records and arrays
!reference RR-0198 Allow positional aggregate for single-component composite
!reference RR-0240 Make aggregate matching more like assignment matching
!reference RR-0571 Improve the rules for OTHERS choices in aggregates
!reference RR-0605 Rules for OTHERS in aggregates are confusing

!problem

LRM 4.3.2(6) unnecessarily restricts the use of an OTHERS clause in array aggregates with named components to (nongeneric) actual parameters and function results, making aggregates difficult to use and complicating implementations. Other anomalies with array aggregates include:

1. The inability to declare constant values of 'null' arrays
2. The need to name the component for single element array values
3. ???

!appendix

%reference AI-00473
The intention of the rules in LRM 4.3.2(6) was to allow named associations with an OTHERS choice only when no subtype conversion is applied to the aggregate; i.e., to allow such a form only when no "sliding" of the bounds occurs.

%reference AI-00681
There is no way to declare a constant value of a 'null' record type. (There is but it is really awkward.)

%reference RR-0029
LRM 4.3.2(6) unnecessarily restricts the use of named associations in array aggregates when an OTHERS clause is used.

%reference RR-0053
There is no way to specify aggregate values that correspond to null records and arrays.

%reference RR-0198
The required use of named notation for single-component aggregates seems an unnecessary exception to the interchangeable use of positional notation.

%reference RR-0240
The restrictive rules for aggregate component associations force the use of multiple component assignments, which are more difficult to maintain than a single assignment of an aggregate value.

%reference RR-0571
Remove the restrictions on array aggregates with an OTHERS clause given in LRM 4.3.2(6).

%reference RR-0605
The rules for when an OTHERS clause is permitted in array aggregates seem arbitrary and are difficult to understand and/or remember.

!topic array slices
!number RI-5130
!version 1.4
!tracking 5.2.1.2.1

DRAFT DRAFT DRAFT DRAFT DRAFT DRAFT

!Issue out-of-scope (un-needed) (1) desirable,moderate impl,upward compat,mostly consistent

1 – Two-dimensional array slicing] Ada9x shall support slicing of two-dimensional arrays. In particular, this support should include:

a. Selecting a single row or column of a two-dimensional array as a one-dimensional array

b. Selecting an arbitrary two-dimensional subarray as a slice of a larger two-dimensional array

[Rationale] Ada's one-dimensional array slicing is elegant and efficient. Similar elegance could be extended to matrix operations with moderate implementation impact. However, this capability would essentially require dope-vector implementations for "sliceable" matrices, which could carry significant performance penalties for all arrays. Creating a new class of "sliceable" arrays to limit the extent of the performance penalties would add another wart to the language. In addition, there is the difficulty of defining the type of a row or column sliced as a one-dimensional array.

!Issue out-of-scope (un-needed) (2) desirable,severe impl,upward compat,inconsistent

2 – General array slicing] Ada9x shall support slicing of arrays of any dimension. In particular, it shall support selecting single (hyper)planes and subarrays of arrays of any dimension.

[Rationale] While a slicing notation for higher dimension arrays can yield elegant algorithms, it is not clear that generated code for indexing for the completely general case would be significantly more efficient than hand-written code. (In fact, hand-written code may be better.) The complexity of implementing completely general slice indexing would increase the size of compilers and reduce compilation speed with no guarantee of improved run-time performance.

!reference RR-0323 Generalize slice for multi-dimensional arrays
!reference RR-0494 Allow slices for any dimension in multi-dimensional arrays
!reference RR-0508 Allow slices for any dimension in multi-dimensional arrays

!problem

Ada83's one-dimensional array slicing facility enables many vector operations to be implemented elegantly and efficiently. The fact that this capability does not extend to higher dimension arrays is a language inconsistency and results in inelegant and inefficient implementation of many matrix and higher dimension array operations.

!rationale

!appendix

Performance improvements for many algorithms depend heavily on the storage representation used for two-dimensional arrays. There is some sentiment in the community for a facility such as "pragma ROW_MAJOR;" to handle this.

%reference RR-0323

Slices are allowed only for single-dimensional arrays. Slicing on multiple indices is important. Slicing on the last index is essential.

%reference RR-0494

Slices are available only for one-dimensional arrays. It would be very useful to have a similar facility for higher dimensional arrays.

%reference RR-0508

Ada83 allows slicing of one-dimensional arrays, but not of higher dimensional arrays. This non-uniformity impacts the efficiency of many numerical algorithms.

!rebuttal

!topic variable-length strings
!number RI-5141
!version 1.4
!tracking 5.2.1.2.2

[Note. This is one of several RIs that address or reflect a need for more general facilities for defining abstract data types in Ada. They will be collected and coalesced into a higher-level ADT requirement.]

!Issue secondary standard (1) desirable

1. An associated standard should be created to define the declarations, operations, and semantics for variable-length strings.

!reference RR-0054 Do not add variable length strings to the language
!reference RR-0163 Need support for variable-length strings
!reference RR-0324 Add more flexible support for string manipulation
!reference RR-0327 Add varying strings to the language
!reference RR-0419 Add support for varying length strings to the language

!problem

Ada83 does not provide variable-length strings and does not provide any way for users to define them in a safe and efficient way. Workarounds are inefficient and have potential storage leaks and awkward syntax. It would be nice to have complete and uniform support for variable-length strings for portability.

!rationale

A standard variable-length string package would satisfy the need for uniform solutions to string manipulation problems without building the solution into the language. This will require data abstraction facilities that are not presently in the language, however. Without such facilities there would be cause to consider adding variable- length strings to Ada9x as a predefined type.

!appendix

Additional data abstraction facilities are needed in Ada9x for solving much more than the variable-length string problem. This is merely an example where adding fundamental capabilities would avoid a collection of language bells and whistles.

Finalization of abstract data objects on exiting their scope (or garbage collection) will be required to avoid storage leakage, assuming that dynamic storage structures will be used in implementations. Users cannot be relied upon to do this explicitly.

Overloading of assignment and equality operations would add considerably to the look and feel of abstract data objects as first class objects. It might also be nice to have the

syntax for the variable-length string LENGTH function be that of an attribute. These extras are not necessary but they would make it hard to distinguish an ordinary library package implementation from built-in data types.

%reference RR-0054
Curing Ada's variable-length string deficiency would be worse than the disease. If variable strings are added, put them in TEXT_IO.

%reference RR-0163
Ada's support for strings in inadequate. Workarounds such as defining your own variable-length string package are error-prone and generally unsatisfactory.

%reference RR-0324
Ada currently has only primitive facilities for text manipulation. Many applications require better, standard string manipulation facilities, including varying strings.

%reference RR-0327
Ada has no mechanisms for defining and manipulating varying strings. Unconstrained strings are expensive and have the wrong semantics.

%reference RR-0419
Support for varying length strings is needed to avoid the use of existing incompatible and non-portable implementations.

!rebuttal

!topic miscellaneous string issues
!number RI-5142

!reference RR-0047 (related)
!reference RR-0257 Allow BOOLEAN and BYTE strings as well as CHARACTER strings
!reference RR-0310 Need convenient way to pad with blanks in string assignment
!reference RR-0324 Add more flexible support for string manipulation
!reference RR-0552 (related)
!reference RR-0597 (related)

!problem

Ada83 lacks a number of convenient and useful string manipulation facilities, including:

1. Boolean and byte strings as well as character strings
2. Padding strings with blanks on assignment
3. Input and output of unconstrained strings
4. ???

!appendix

%reference RR-0047 (related)

Need varying strings or TEXT_IO GET and PUT *functions* that return unconstrained string values to avoid fixed-size buffers.

%reference RR-0257

Allow BOOLEAN and BYTE strings as well as CHARACTER strings.

%reference RR-0310

Ada does not provide any convenient way to pad strings with blanks on assignment.

%reference RR-0324

Ada currently has only primitive facilities for text manipulation. Many applications require better, standard string manipulation facilities.

%reference RR-0552 (related)

Add procedures for padding strings to TEXT_IO.

%reference RR-0597 (related)

Add a GET_LINE *function* to TEXT_IO that returns an unconstrained string value to avoid fixed-size buffers.

!topic implicit subtype conversion
!number RI-5150
!version 1.1
!tracking 5.2.1.2.3

DRAFT DRAFT DRAFT DRAFT DRAFT DRAFT

!Issue revision (1) important,small impl,upward compat,mostly consistent

1 – Implicit Conversion of Array Subtypes] An attempt shall be made in Ada9x to extend the circumstances under which implicit conversion of array subtypes is provided.

!reference RI-5061 constraints in renaming declarations
!reference RR-0510 allow renames/subtypes to alter array index bounds
!reference RR-0520 distinguish "sequence" and "mapping" arrays
!reference RR-0573 apply implicit subtype conversions consistently
!reference RR-0734 generalize implicit subtype conversions
!reference RR-0749 apply implicit subtype conversions consistently

!problem

Implicit subtype conversion for arrays is restricted to array assignment statements (and certain other operations) [LRM 3.6.1(4)]. The "matching" of components that occurs in relational operations on arrays [LRM 4.5.2(7)] has a similar effect. There are several situations, however, where the absence of implicit conversions causes programming difficulties, including:

a. assignment of record aggregate values that contain array components [LRM 5.2.1(5)]

b. subprogram parameter associations [LRM 6.3.1(9)]

c. returning array values from functions (as for universal numeric values [LRM 5.8(8)])

d. renaming declarations [LRM 6.3.1(10)]

!rationale

Implicit conversion of array subtypes on assignment in Ada83 is a convenient feature that avoids frequent use of awkward explicit conversions. It would be useful to have this convenience extended to include assignment of records that contain array components, subprogram parameter associations, and function return statements.

!appendix

A previous RI (RI-5061.1) recommended requiring that the subtype information specified in a renaming declaration always match that of the object being renamed.

%reference RR-0510

To efficiently implement array operations it would be useful to be able to shift array index bounds so that constant index offsets do not have to be written or computed for each array element reference. Honoring the subtype specification in array renaming declarations offers one possible solution.

%reference RR-0520

It should be possible to distinguish between arrays used as "sequences" and arrays used as "mappings". Sliding should not apply to mapping arrays. The lower bound for all sequence arrays should be 1.

%reference RR-0573

Implicit array subtype conversions should be performed for default initialization of array-valued record components and for evaluation of array-valued components of aggregates.

%reference RR-0734

When a record contains an array component, assigning an aggregate value to a record is not equivalent to assigning values to components individually. The implicit subtype conversion rules for array assignment should apply to larger aggregates, function return statements, and subprogram parameter associations.

%reference RR-0749

Ditto.

!rebuttal

!topic miscellaneous array issues
!number RI-5160
!version 1.2
!tracking 5.2.1.2.4

DRAFT DRAFT DRAFT DRAFT DRAFT DRAFT

!Issue out-of-scope (un-needed) (1) desirable,moderate impl,upward compat,mostly consistent

1 – Vector and array operations] Ada9x shall provide efficient built-in vector and array operations for inner product, outer product, element-wise relational operations, reductions, etc.

[Rationale] Ada9x should not unduly restrict vendors from supplying vector and array operations in library packages and taking advantage of available machine vector and array operations. Such features should not have to be embedded in the language to achieve either convenience or performance.

!Issue out-of-scope (un-needed) (2) desirable,small impl,bad compat,unknown compat

2 – Anonymous array types] Ada9x shall eliminate anonymous array types.

[Rationale] Implicitly defined anonymous array types are a consequence of being able to define simple array (sub)types by simple, direct declarations. Such a non-upward-compatible language change would invalidate too much existing code without adding significant benefit.

!Issue revision (3) desirable,small impl,upward compat,mostly consistent

3 – Partially constrained array types] Ada9x shall support the declaration of array types with combinations of constrained and unconstrained index types. [Note: the solution should be consistent with that for partially constrained record types. (See RI-2032.4)]

!reference RI-0106
!reference RI-2032 requirement 4, partially constrained record types
!reference RR-0139 (related)
!reference RR-0308 array processing facilities
!reference RR-0473 (related) allow "partially" constrained records
!reference RR-0617 ban constrained arrays with anonymous base types
!reference RR-0713 unify arrays, fix generics

!problem

1. Ada83 does not provide efficient built-in operations for manipulating vectors or arrays.
2. Ada83's anonymous array types encourage bad programming habits.
3. Ada83 does not allow mixing constrained and unconstrained index types in array declarations.

!rationale

Mixing constrained and unconstrained index types should be relatively straightforward to implement and would allow the declaration of a class of arrays that cannot now be defined.

!appendix

%reference RR-0139 (related)

Ada83 does not provide shift and rotate operations for packed Boolean arrays. (See RI-0106)

%reference RR-0308

Ada9x should provide efficient built-in vector and array operations for inner product, outer product, element-wise relational operations, reductions, etc.

%reference RR-0473

Allow "partially" constrained subtypes of discriminated records. The alternative of unconstrained types wastes space and does not give appropriate type control.

%reference RR-0617

Ada9x should eliminate anonymous array types which encourage bad programming habits – non-upward-compatible but only badly-written code will fail to compile.

%reference RR-0713 unify arrays, fix generics

Ada83 does not allow mixing constrained and unconstrained array indexes, which makes it impossible to correctly declare many useful array types.

!rebuttal

!topic Record discriminants
!number RI-2032
!version 1.5
!tracking 5.2.1.3.1

DRAFT DRAFT DRAFT DRAFT DRAFT DRAFT

!Issue out-of-scope (un-needed) (1) desirable,severe impl,upward compat,inconsistent

1. (Nonstatic Discriminants in Aggregates) Ada9X shall not require that the value specified for a discriminant that governs a variant part in an record aggregate be given by a static expression.

[Rationale] RI-2032.1 might be used when one encounters a discriminant with a large span of values that is semantically partitioned into a few cases. This would allow the code to represent the semantic partitioning instead of representing the Ada83 rules on staticness. However, the idea of using an "auxiliary discriminant" to represent the partition is seen as a not too onerous workaround.

!Issue revision (2) desirable,small impl,upward compat,mostly consistent

2. (Relax Restrictions on Discriminant Types) Lacking a more direct facility for supplying parameters to component initialization expressions, Ada9X shall not arbitrarily restrict the type of record discriminant that is not used to provide size or shape information for a record type.

!Issue out-of-scope (un-needed) (3) desirable,small impl,upward compat,inconsistent

3. (Setting Discriminants w/o Complete Assignment) Ada9X shall allow assignment to discriminants without requiring assignment to the whole record.

!Issue revision (5) important,moderate impl,upward compat,inconsistent

5. (Partial Constraining of Record Types) Ada9X shall allow record subtypes in which one or more of the discriminant types of the corresponding base record type are refined, that is, the discriminant would be constrained to a more restrictive subtype. [Note: this might also apply to array types.]

!Issue revision (5) important,moderate impl,upward compat,inconsistent

5. (Suppiying a Subset of the Discriminants) Ada9X shall allow record subtypes in which some but not all of the discriminants of the corresponding base type are supplied. [Note: this might also apply to array types.]

!Issue revision (6) desirable,moderate impl,upward compat,mostly consistent

6. (Discriminants Defaults on Allocators) Ada9X shall define a form of dynamic allocation so that allocated objects of unconstrained record types are unconstrained even when discriminant defaults are supplied.

!Issue out-of-scope (un-needed) (7) desirable,moderate impl,upward compat,inconsistent

7. (Late Constraining of Components) Ada9X shall allow a record type where the types of record components are unconstrained at the declaration of the record type but which become constrained at allocation for any object of the record type.

[Rationale] The basic problems here are (1) that discriminants of the components have to be promoted causing a modularity problem in specifying aggregates and (2) that the discriminants would be stored twice. Providing defaults seems an adequate workaround for the first problem; a good compiler would not have to store two copies of the discriminant since it could simply map the higher level discriminants onto the lower level ones.

!reference RR-0212
!reference RR-0248
!reference RR-0341
!reference RR-0473
!reference RR-0497
!reference RR-0522
!reference RR-0530
!reference RR-0531
!reference RI-2012
!reference RI-5110

!problem

There are three problems here. First, record discriminants are essentially the only mechanism available in the language for initialization parameterization. That being the case, arbitrary restrictions on what a discriminant type can be should be removed.

The other problem is that Ada83 forces premature constraining of record types. This interferes with the modularity of programs because a program cannot export a partially discriminated type whose discrimination is to be completed (or further constrained) by a later constraint. One important example of the use of such a mechanism would be in defining a convenient form of variable-length string type, meeting a common need of many users. Following is an example of how such a type might be defined and used:

```
type VAR_STRING ( LENGTH: NATURAL := 0 ) is
  record
        VALUE: STRING( 1 .. LENGTH );
  end record;

subtype TEXT_STRING( LENGTH => 1 .. 80 );
```

```
subtype COMMAND_STRING is TEXT_STRING( 1 .. 10 );

function VS ( S: STRING ) return VAR_STRING;

LINE : TEXT_STRING;

CS : COMMAND_STRING := VS( "list" );

T1 := VS( "This is a string of text" );

T1 := CS; -- assignment of a short string to a long string
```

The advantage of this approach over the conventional approach of using a discriminant to express the maximum size of a given object is the compatibility of objects with different subtypes (and hence possibly different maximum sizes). This also can provide significant space savings by not requiring the definition of a single maximum size subtype that must be used for objects of widely varying sizes.

The last problem addressed in this RI is that an object of an unconstrained record type is constrained if allocated by an allocator if it has defaults for its discriminants. The creates some difficulty in creating a unconstrained allocated object of the type, clearly a useful facility. Also, it is a relatively difficult concept to grasp. For example, by 4.8(5), if one writes

```
type T (D: Boolean := False) is
  record
    ...
  end record;

type T_Ptr is access R;

R : T;
RP : T_Ptr := new T;
```

R is unconstrained but RP.all is subject to the constraint (D => False). This a surprising nonuniformity; it can be particularly troubling in a generic to allocate an object only to find that it cannot hold any value of the base type..

!rationale

Requirement RI-2032.2 deals with the problem of using discriminants for initialization parameters and for constant components. The PUN value is low because the two concepts need to be addressed more directly. In the original language, discriminants could only be used for giving sizing information; in 1983 the role of discriminants was opened up so that discriminants could be used as initialization parameters. In this role, it makes sense that any type could be used; there may be some economy gained by restricting to types whose size is known at compile time. It is probably preferable to solve the initialization problem directly as in RI-2012.

Requirement RI-2032.4 deals with the issue that many users feel that Ada83 forces premature constraining of record types. RI-2032.4 would allow a module to export a type that was partially constrained along with operations that make sense for all "subtypes" meeting the partial constraint. A later subtype declaration would then be able to more fully constrain the type without affecting the previously exported operations. RI-2032.5 deals with a special case of this idea that would at least allow a subset of the discriminants to be supplied rather than all of them.

Requirement 2032.6 addresses the problem of allocating an unconstrained record object when the record type has defaults for discriminants. If the form of allocation used was the one built into Ada83 (i.e. new) then this would not be upwards compatible.

!appendix

RI-2032.7 deals with the modularity problem that occurs when a single type is to represent a large class of types with some common operations. Consider the following:

```
type ct1(d11:t1,d12:t2,d13:t3) is new somewhere_else.ct1;
type ct2(d21:t4,d22:t5,d23:t6) is new somewhere_else.ct2;
type ct3(d31:t7,d32:t8,d33:t9) is new somewhere_else.ct3;

type r_kinds is (k1,k2,k3);

type r(d:r_kinds) is record
  case d is
          when k1 => c1: ct1;
   when k2 => c2: ct2;
          when k3 => c3: ct3;
  end case;
end record;
```

This is, of course, illegal since c1, c2 and c3 have unconstrained types. The solution to this problem is straightforward: gather up all the discriminants and "post them" to the highest level as in

```
type r_kinds is (k1,k2,k3);

type r(d:r_kinds;
           d11:t1,d12:t2,d13:t3;
           d21:t4,d22:t5,d23:t6;
           d31:t7,d32:t8,d33:t9
    ) is record
  case d is
          when k1 => c1: ct1(d11,d12,d13);
   when k2 => c2: ct2(d21,d22,d23);
          when k3 => c3: ct3(d31,d32,d33);
  end case;
end record;
```

Not only is this wasteful of space (for the straightforward implementation), but it makes things less modular in the following way. One can imagine in the first definition that an assignment such as

```
my_r:= r'(d=> k1, c1=> ct1(d11,d12,d13));
```

instead of

```
my_r:= r'(k1,d11,d12,d13,d21,d22,d23,d31,d32,d33
      ,c1=> ct1(d11,d12,d13));
```

An example of this class hierarchy is the numeric types supported by some Lisp implementations where the operations are charged with the responsibility for coercing among the types. In such a case, Ada83 requires all of the discriminants of all of the subcomponents to be promoted to the top-level. Thus, a user-specifying a "LISPNUM" that was a rational might also have to include constraints for LISPNUMs represented as strings or as packed decimal. This is antithetical with good modularity. However, the modularity problem is solved by simplying giving defaults for all of the discriminants; thus, we are left only with the problem of having the discriminants stored twice. A good optimizing compiler could solve this problem by simply not storing a copy of the discriminants in the outer structure.

I was not able to find any rationale for the current prohibition against partially unconstrained record subtypes and unconstrained component types. I think that the idea was to enable efficient implementation by the straightforward approach.

%reference RR-0212 Allow assignment to record discriminant like other components

RR-0212 notes that it is very inconvenient to do a complete record assignment just to change the value of a discriminant; wants to assign to discriminants just like components.

%reference RR-0248 Allow user control over where discriminants are stored (and ?)

RR-0248 seems to present two issues: (1) that the user cannot control the placement of discriminants in a record [actually, repspecs seem to do exactly this], and (2) that a user may be confused by syntactically similarity of discriminants and other language entities. I did not understand the part about how discriminants may change the calling/receiving sequence of a subprogram.

%reference RR-0341 Allow discriminant value in record aggregate to be non-static

RR-0341 points out that the staticness requirement for the discriminant values in record aggregates greatly complicates working with discriminanted records. The essential rub comes about when a discriminant type has a large number of possible values but the space is partitioned w.r.t. meaning. A solution is proposed: remove the staticness requirement.

%reference RR-0473 Allow "partially" constrained subtypes of discrimted. records

RR-0473 wants partially constrained discriminated records. The alternative of unconstrained wastes space and does not give appropriate type control. The RR presents a very general solution based on refining the range of a discriminant instead of being required to supply a value.

%reference RR-0497 Discrmt defaults give init. val for obj decls, constraint for NEW

RR-0497 gives a messy example of the problem of treating unconstrained types with discriminant with defaults as constrained; the suggestion is that the language not do so.

%reference RR-0522 Allow non-discrete record discriminants

RR-0522 brings up several ideas relating to discriminants, specifically that discriminants are restricted in type. First, discriminants are the only parameterization available; thus, parameters are type restricted. Second, discriminants are the only way to get constant components; again, this means that constant components are restricted.

%reference RR-0530 Allow assignment to record discriminant like other components

RR-530 suggests that not allowing assignment to record discriminants is too restrictive on the one hand, and (on the other) does not address completely the problem of undefined values. The suggestion is that the problem of undefined values be handled more generally and that the restriction on discriminant assignment be removed.

%reference RR-0531 Nested records with discriminants are awkward and un-modular

RR-531 details the difficulties of having nested variant records. The essential rub is that one wants to be able to construct records with unconstrained variants then then to constrain them in a subsequent usage.

!rebuttal

!topic Enhancing the Model of Record Types
!number RI-2033
!version 1.3
!tracking 5.2.1.3.2

DRAFT DRAFT DRAFT DRAFT DRAFT DRAFT

!Issue out-of-scope (un-needed) (1) desirable,severe impl,upward compat,inconsistent

1. (Generic Record Attributes) Ada9X shall define attributes that would allow a generic unit to decompose a record type by iterating over its components.

[Rationale] It makes no sense in a strongly typed language to iterate over a heterogeneous structure.

!Issue out-of-scope (un-needed) (2) desirable,severe impl,upward compat,inconsistent

2. (Anonymous Component Types) Ada9X shall allow as component types anonymous array types whose array component size is dependent on the value of a discriminant. [Note: the determination that this item is out of scope does not dictate that all forms of anonymous types are out of scope.]

!Issue out-of-scope (un-needed) (3) desirable,small impl,upward compat,mostly consistent

3. (Defining by Component-Wise Application) Ada9X shall provide a mechanism by which a function can be defined by specifying (1) a function designator specifying a function to be applied to each component of a record type and (2) a function to reduce the values returned by (1) to a single value. The mechanism shall not require that each component of the record type be mentioned explicitly.

!Issue revision (4) desirable,moderate impl,upward compat,inconsistent

4. (Unrestricted Component Naming) Ada9X shall allow the same component of a record (i.e. the same name and the same subtype) to be visible in more than one variant of a record.

!Issue revision (5) desirable,small impl,upward compat,mostly consistent

5. (Multiple Variant Parts) Ada9X shall allow multiple variant parts where Ada83 only allows one.

!reference RR-0027
!reference RR-0381
!reference RR-0532
!reference RR-0568
!reference RR-0707
!reference WI-0215

!reference AI-00429
!reference RI-2034

!problem

The problem is that Ada83's record types are not adequate for modeling data coming into the program. In some cases, this is because the data does not contain explicit discriminants. This issue is not dealt with here but in RI-2034 on data interoperability. The issue here is that record types would be a much better match if certain naming restrictions were removed and if a slightly more general structure were allowed.

!rationale

The main argument for these items is that it should be possible to model incoming data in terms of (rep.spec'ed) Ada records. Multiple variants or components of equal name in different variants are seen as a way to overcome the strictly hierarchical variant structure of Ada-83 records. As to the overhead of multiple names, under the assumption that equally named components are rep.spec'ed into the same place (and are of equal type; see overloading issues otherwise), no run-time overhead accrues.

!appendix

%reference RR-0027 Need additional record type attributes

RR-27 suggests a capability by which a generic could decompose a record type using attributes; specifically, attributes would be defined to provide the number of fields, the type of the i-th field and the component selector name for the i-th field.

%reference RR-0381 Records should have composed operations wrt components

RR-381 suggests that the language support automatic composability over records. The idea is that if some function f is defined for each component of a record, the f should be defined for the record as well by the obvious composition. Each component of the record should not need to be mentioned.

%reference RR-0532 Allow same-type record components in different variants to share name

RR-532 wants the ability for components in different variants of a record to be able to share the same name (1) if they share they same type and (2) are defined in the same variant part.

%reference RR-0568 Allow multiple non-nested va.iants in record types

RR-568 show that a structure frequen'ly arises where it is natural to have multiple unnested variants; the workaround is easy but somewhat messy and slightly less memory efficient.

%reference RR-0707 Need same-name component identifiers in different variants

RR-707 is a mixture of RR-532 and RR-568. What is asked for is the solution of RR-532; lacking that RR-568 would be better than nothing.

%reference WI-0215 Records should have composed operations wrt components

If equality is defined on all components of a composite type, whether predefined or redefined, then equality should also be defined on the composite type, as the conjunction of the component-wise comparisons. (If, as a result of other requirements, it is also possible to redefine assignment, then the same rule should apply to component-wise assignment).

%reference AI-00429 Allow array type definition for record component

AI-429 wants to be able to write

```
type PERMUTATION (N : NATURAL) is
 record
        PERM : array (1..N) OF INTEGER range 1..N;
 end record;
```

instead of

```
type PERM_ARRAY is array (INTEGER range <>) OF INTEGER;

type PERMUTATION (N : NATURAL) is
 record
        PERM : PERM_ARRAY (1..N);
 end record;
```

The argument is that the former allows the size of the array element to be adjustable whereas the latter does not.

%reference AI-00274 Proposed extension of the USE clause - Record component visibility

The idea of AI-274 is to have a Pascal-style with statement; the questioner does not mention that an essentially identical capability is provided by renaming. Of course, one still has to say "z.compname" instead of just "compname" but this does not seem unreasonable.

!rebuttal

!topic Uniformity of integers
!number RI-5170
!version 1.6
!tracking 5.2.3.1

DRAFT DRAFT DRAFT DRAFT DRAFT DRAFT

!Issue revision (1) important,small impl,upward compat,consistent

1. An attempt shall be made to improve the uniformity of the treatment of integer types and associated operations in Ada9x.

!reference RR-0122 Integer use for indexing may be too large
!reference RR-0315 Enhance rules for optional pre-defined integer types
!reference RR-0366 Subtype natural should not include 0
!reference RR-0495 Remove leading space in 'IMAGE for integers
!reference RR-0572 Need operations for all pre-defined integer types
!reference RR-0680 Need ** whose right operand is any integer type

!problem

A number of anomalies in the treatment of integer types and their associated operations in Ada83 have been reported, including:

a. Operations defined for type INTEGER, but not for other integer types (i.e., exponentiation and fixed point multiplication and division).

b. The leading space in the 'IMAGE attribute of non-negative integers.

c. The value of STANDARD.NATURAL'FIRST.

!rationale

Users expect operations defined for one arithmetic type to be available and behave similarly on similar types. To the extent that this can be achieved (upward-compatibly) in Ada9x, it will be an improvement over Ada83.

!appendix

%reference RR-0122

Allowing any integer type as an array index, case selector, or discriminant is impractical.

%reference RR-0315

Support arbitrary length (in bits) integers as predefined integer types rather than as subtypes with representation specifications.

%reference RR-0366

The definition of subtype NATURAL does not correspond with the usual mathematical definition; drop the predefined types NATURAL and POSITIVE.

%reference RR-0495

Remove the leading space in the result of the 'IMAGE attribute for integer types.

%reference RR-0572

Operations defined for integers should be uniformly defined for all integer types, including universal integers.

%reference RR-0680

The exponentiation operation should be defined with a right operand of any integer type.

!rebuttal

RI-0901

!topic fixed point range end points
!number RI-0901
!version 1.4
!tracking 5.2.3.3.1, 5.2.3.3.2, 5.2.3.4.2
DRAFT DRAFT DRAFT DRAFT DRAFT DRAFT

!Issue out-of-scope (un-needed) (1) desirable,small impl,bad compat,consistent

1. Ranges for fixed point types should include end points. One possible approach to meeting this requirement would be to change the rule in 3.5.9(6) replacing the text "at most" by "less than".

[Rationale] Although the points raised by the RR's are certainly valid in the sense that it might have been better to choose another definition in the first place, it is clearly inappropriate to introduce an incompatibility of this magnitude in the revised language. Changing the language in the recommended manner would result in existing programs being rejected at compile time, or giving different results at execution time with no rejection.

It would be desirable for the reference manual to contain some examples to make the rules with regard to the choice of the mantissa size absolutely clear.

To see why the original rule was chosen, consider the following example:

+5 type X is delta 2**(-31) range -1.0 .. 1.0;

With the rule as currently stated, this fits in 32 bits, with full use of these 32 bits. The modified rule would require 33 bits for this declaration, and result in a situation in which the 33 bits were only actually needed for the one extra model number, 1.0, at the end of the range.

!Issue presentation (2) desirable

2. The RM should contain additional examples clarifying the effect of the rules on fixed point range end points.

!reference RR-0191
!reference RR-0566
!reference RR-0425
!reference RR-0252

!problem

The fact that the end points of a fixed-point range as given in the declaration are not necessarily included in the range of the resulting fixed-point number has been a source of continuing confusion to many users of Ada, and is often interpreted as a problem in the Ada definition, or as a bug in implementations. Nevertheless, the RM is quite explicit in allowing the end points of a fixed point range to be excluded from the implemented range (this is a direct result of the "at most" clause in 3.5.9(6)). The decision in the language design was deliberate. Implementations are not merely allowed to exclude the end points, but are required to do so in some cases. The ACVC suite contains tests to ensure that implementations conform to this requirement, so all existing validated implementations behave as specified in the RM.

!appendix

%reference RR-0191 Mantissa of Fixed-Point Types Unreasonably Small

In this RR, it is noted that given the declaration:

```
type F_TYPE is delta 0.3 range 0 .. 1.1;
```

that "two different Ada compilers come up with" LARGE = 0.75 and MANTISSA = 2. Hopefully ALL ada compilers come up with this, since it is the required choice of the reference manual. The RR argues that 1.0 should be in the range.

%reference RR-0566 Fixed-Point Model Numbers

This is essentially identical in content to RR-0191. As with RR-0191, the writer is under the impression that the compiler is free to choose a more appropriate range, when actually the situation is worse than the writer thinks (from his point of view) -- the compiler is forced to choose what he regards as an inappropriately small mantissa size.

%reference RR-0425 Open Ranges for Real Types

This RR argues in favour of a syntactic extension to allow open ranges for real types, which is another approach to this problem. However, it seems rather heavy to add new syntax for this purpose, when the result can always be achieved by rewriting the bounds carefully.

%reference RR-0252 Doing Math in Ada

This RR is a general complaint about the mathematical model in Ada, it suggests getting rid of "the verbage having to do with model and safe numbers". At the same time it wants more control over such areas of rounding. It also argues in the list of solutions that the range for real types should include the end-points, and that non-binary representations for delta and digits are not "helpful in the embedded community".

!rebuttal

Ada9x should attempt to resolve the problems caused by Ada83's rules for determining values and ranges of fixed-point types. These rules are considerably more complex than they first appear to be in the reference manual and have consequences that are not at all intuitive.

Part of the problem is the presentation of the rules in the reference manual. Additional examples, of course, are always helpful. It would be more helpful if the rules themselves were more clearly explained.

An understandable explanation of the rules, however, does not imply intuitive results. "Intuitive" for fixed-point types can mean several things. A first intuition, for the uninitiated, for the declaration

```
type FIX1 is delta 0.3 range 0.0 .. 1.2;
```

is that it captures the values 0.0, 0.3, 0.6, 0.9, and 1.2. A second intuition, after learning that only binary fractions are supported for ALL,isthat

```
type FIX2 is delta 0.25 range 0.0 .. 1.0;
```

captures the values 0.0, 0.25, 0.5, 0.75, and 1.0. An expert reading of the reference manual says that FIX2'LARGE = 0.75, however, which implies that 1.0 is not a value of this type. Running a test program on the nearest compiler at hand yielded FIX2'LARGE = 0.75 and FIX2'LAST = 1.0, and accepted the declaration

```
X: FIX2 := 1.0;
```

This indicates that either I still don't understand the rules, that compiler implementers don't understand the rules, or that they bend the rules to produce results users expect.

The example used in the !rationale to justify the current rules implies that programmers do not understand that at most 2**N distinct values can be represented in N bits. Another opinion is that programmers understand this simple fact, whereas understanding Ada83's rules for fixed-point types is altogether another story!

An example of an upward compatible extension for Ada9x would be to provide a slightly different syntax such as

```
type FIX_a_la_9X is SMALL 0.3 range 0.0 .. 1.2;
```

with semantics that captures the values 0.0, 0.3, 0.6, 0.9, and 1.2. This would yield an intuitive set of values based on integral multiples of the specified SMALL, including both end points. The semantics of Ada83 fixed-point types declared using DELTA would not have to be affected.

!topic Implementation Freedom in Fixed Point
!number RI-0902
!version 1.4
!tracking 5.2.3.4.1, 5.2.3.4.2, 4.7.2
DRAFT DRAFT DRAFT DRAFT DRAFT DRAFT

!Issue revision (1) important,moderate impl,moderate compat,consistent

1. Implementation dependence in results of fixed-point calculations should be minimized.

!Issue revision (2) important,moderate impl,moderate compat,consistent

2. The representation of fixed-point types should be specified by the standard.

!Issue revision (3) important,moderate impl,moderate compat,consistent

3. Subtypes of fixed-point types should not be stored with reduced accuracy.

!reference RR-0256
!reference RR-0144
!reference RR-0409
!reference RR-0733
!reference WI-0313

!reference
Dewar, R., "The Fixed-Point Facility in Ada", SEI Special Report
SEI-90-SR-2, February, 1990.

!problem

Fixed-point arithmetic in Ada provides far too much implementation freedom, much of it derived from analogy with floating-point where such freedom is justified by hardware considerations.

!rationale

The results of fixed-point calculations are currently not defined by the language, except in terms of model number intervals. Such implementation dependence is justified for floating-point, where the hardware models vary. However, for fixed-point, the underlying operations are integer operations and there is thus no reason not to specify their semantics exactly.

One issue here are whether fixed-point values should be left or right justified in the word. For example, given the declaration:

```
type X is delta 0.25 range -1.0 .. +1.0;
```

If an implementation intends to use 32 bits in any case for this type, it may choose to provide extra accuracy (as though the specified delta had been 2.0**(-31)). This may be desirable, but gives quite different results from an implementation which takes the approach of right justifying (i.e. storing integers in units of the declared delta). The two approaches can both be justified in appropriate circumstances, but it is undesirable that the choice is made by the implementor rather than the programmer.

Another issue is whether results of fixed-point operations should be rounded or not. Again, this is left to the implementation, and since virtually all implementations use integer arithmetic for implementing fixed-point, this freedom is not appropriate. If rounding is desirable, then it should be under control of the programmer, or be the required default. If rounding is not desirable, it should not be permitted.

Requirement [2] suggests that the RM should be more explicit in describing how fixed-point types are represented. The RM currently strongly hints that fixed-point values are held as integers, but does not make this explicit.

Finally, requirement [3] addresses the issue of whether subtypes of a fixed-point type can have a representation with less accuracy than the base type. This is similar to the issue raised in AI-0407 for floating-point types, and extensive analysis seems to indicate the conclusion that it is undesirable for accuracy to be lost in conversion to a subtype (such loss of accuracy occurs implicitly in situations, like assignment statements, where programmers do not expect the values to be modified silently).

!appendix

Requirements 1 is similar to requirement 11 (page 19) of "The Fixed Point Facility in Ada", and requirements 2 and 3 are derived from requirements 1 (page 4), 2 (page 7) and 3 (page 8) of this report.

%reference RR-0256 Fixed-Point Scaling and Precision

This RR is a general complaint that fixed-point is error prone, but it is hard to get any specific information. The possible solutions are unclear. Probably the basic complaint is that implementations (a) have too much freedom in how they deal with fixed-point and (b) do not fully implement the SMALL representation clause which would provide the precise control that the submitter is looking for.

%reference RR-0144 Floating-Point Coprocessors

This RR requests that compilers provide fixed-point support even if a floating-point coprocessor is not present. This is clearly not a language issue, but rather deals with characteristics of implementations. Valid Ada compilers must in any case support floating-point. Most likely the RR arises from a situation where a vendor requires a coprocessor for support of floating-point, and the compiler is being used on a system with no coprocessor. The user expects that fixed-point arithmetic will still be available on this configuration. This may be desirable, but is neither a language issue nor a validation issue (since the compiler could not in any case be validated on such a system, since floating-point would not be supported).

%reference RR-0409 Rounding of Numeric Conversions

This RR discusses the issue of rounding to integer, but its reasoning is equally applicable to rounding of fixed-point.

%reference RR-0733 Uniform Representation of Fixed Point Precision for All Ranges

This RR is apparently generated by the incorrect view that the delta for a fixed-point number cannot be smaller than SYSTEM.FINE_DELTA. This is just wrong, so the concerns of the RR are not clear. One point that can be gleaned is that the proposer would like to have biased fixed point representations. Thus this is really a chapter 13 issue, and is nothing to do with fixed-point semantics.

%reference WI-0313 Need fixed point model numbers which are exact

!rebuttal

Item [2] above should be out of scope. The standard should not overly constrain implementations by specifying the internal representation of fixed-point data. Tightening the semantics of fixed-point operations and data values is a more appropriate way to correct problems with unnecessary implementation freedom in fixed-point.

!topic Fixed-Point Support for Commercial Applications
!number RI-0903
!version 1.4
!tracking 5.2.3.6

DRAFT DRAFT DRAFT DRAFT DRAFT DRAFT

!Issue revision (1) important,moderate impl,upward compat,consistent

1. Decimal SMALL values must be fully supported.

!Issue revision (2) important,moderate impl,moderate compat,mostly consistent

2. It must be possible for implementations to support packed decimal and other explicit decimal formats where the range is not binary.

!Issue revision (3) important,moderate impl,upward compat,consistent

3. Implementations must provide sufficient precision for commercial applications.

!reference RR-357

!reference
Dewar, R., "The Fixed-Point Facility in Ada", SEI Special Report SEI-90-SR-2, February, 1990.

!problem

The fixed-point facility in Ada appears to address the needs for support of decimal calculations in commercial applications, but there are problems both in the level of support actually provided implementations and in the language definition.

It has been argued that [1] is already required by the RM as it stands, but practice has been to reject decimal SMALL clauses in many implementations, as indicated by the experience reflected in RR-357. The ACVC suite, even in its latest version, 1.11, has declined to confirm that implementation of decimal SMALL is required, and indeed, by labeling the few tests that appear in this regard as DEP tests, encourages the view that decimal SMALL is optional.

It is clear that decimal SMALL is required for commercial applications, as indicated by the wide use of decimally scaled fixed-point types in COBOL and other commercial languages. This should be made a clear requirement in Ada 9X. However, the situation is not completely straight-forward, since some operations, in particular, mixed multiplications and conversions, have semantics which may imply difficult implementation burdens. These points are further addressed in RI-0904.

For commercial applications, the range of facilities needed is fairly limited. In particular, it is probably not required to be able to mix binary and decimal scaled data. A

reasonable minimal functionality is to match that provided by the current (1983) COBOL standard.

The problem with respect to [2] is that the definition in the RM requires that the range of the base type from which the fixed-point type is derived be essentially a binary range, as though the representation is binary. This requirement makes it harder to support decimal types and should be reviewed. In addition, the revision should address the question of whether a standard form should be provided (e.g. an optional pragma) for specifying decimal representation.

Finally the provision of decimal SMALL is only useful in commercial applications if the precision is sufficient to represent fiscal quantities. COBOL requires 18 decimal digits, corresponding to 64 binary bits, which is a reasonable guide for a minimum required precision. Note that there are also some real-time situations in which extended precision for fixed-point is desirable. In particular, if the clock resolution is very fine (e.g. one microsecond), then 32 bits is insufficient for the representation of DURATION values for delay.

!rationale

Although Ada was originally designed primarily for embedded applications, it is clear that it is potentially usable for a much wider range of applications. In particular, commercial applications can be supported given a more complete support for decimal calculations as described in this RI. Such support does not affect the efficiency of generated code for applications not using this facility, so providing for commercial applications in the manner suggested by this RI does not compromise real-time and embedded applications.

!appendix

Requirement 1 is similar to requirements 5 and 6 (page 10) of "The Fixed-Point Facility in Ada." See also requirement 12 (interoperability with COBOL).

!reference RR-357 Decimal

This RR consists of two complaints. The first is that the compiler in use does not allow decimal small. The writer recommends that

> "The language should be amended to allow the clause: for MONEY'SMALL use 0.01"

It is an interesting comment on the state of affairs that the writer should be unaware that the current language requires acceptance of this clause!

The second point is that reasonable precision (at least 63 bits) be provided.

!rebuttal

!topic Restrictions in Fixed-Point
!number RI-0904
!version 1.4
!tracking 5.2.3.6, 5.2.3.4.2
DRAFT DRAFT DRAFT DRAFT DRAFT DRAFT

!Issue revision (1) desirable,moderate impl,moderate compat,mostly consistent

1. Remove the requirement that universal real operand be qualified in fixed-point multiplication and division operations.

!Issue revision (2) desirable,moderate impl,moderate compat,mostly consistent

2. Remove the requirement that the result of fixed-point multiplication and division operations be explicitly converted.

!Issue revision (3) desirable,moderate impl,moderate compat,mostly consistent

3. Allow mixed types in addition and subtraction of fixed-point operands.

!Issue revision (4) desirable,moderate impl,moderate compat,mostly consistent

4. Review required SMALL values. Binary SMALL values are clearly required, and decimal SMALL values are required for fiscal applications. Are more general SMALLs required? Ada should be precise in specifying what SMALL values must be supported.

!Issue revision (5) desirable,moderate impl,moderate compat,mostly consistent

5. Specify accuracy requirements for all fixed-point operations, including particularly input-output, multiplication and division requirements. Ensure that all accuracy requirements are efficiently implementable.

!reference RR-0357
!reference RR-0400
!reference RR-0401
!reference RR-0591
!reference RR-0592

!reference
Dewar, R., "The Fixed-Point Facility in Ada", SEI Special Report SEI-90-SR-2, February, 1990.

!problem

The fixed-point facility in Ada has suffered from lack of complete implementation and a concern that complete implementation involves infeasible accuracy requirements. Both these problems must be addressed in Ada 9X. In addition there are cases of restrictions which seem unnecessary and inconvenient to users of fixed-point.

Requirement [1] addresses one such restriction. The RM requires that universal real operand be qualified in fixed-point multiplication and division operations (this is a consequence of the rule in RM 4.5.5 (10), since the use of a universal real operand is ambiguous. This requirement is present because of concerns that there may be implementation difficulties in allowing the use of literals and other universal real operands.

Similarly requirement [2] addresses the fact that we can write:

```
F1 := F2 + F3;
```

but not

```
F1 := F2 * F3;
```

and must instead write:

```
F1 := F1_TYPE (F2 * F3);
```

The requirement of specifying the intermediate result is understandable in the middle of a complex expression (although COBOL does not have a requirement of this type), but is certainly unexpected in this simple case where the type of the result is obvious.

Similarly, requirement [3] addresses the fact that fixed-point types can be mixed in multiplication and division, but not in addition and subtraction. This leads to inconvenient conversions that cannot always be written in a simple manner (since introducing the conversions may result in constraint errors).

Requirement [4] asks that Ada 9X be explicit in what values of SMALL must be supported. At the current time, this decision is in practice the choice of individual implementations.

Finally requirement [5] addresses the concerns that the current accuracy rules in Ada for fixed-point operations (particularly in the case of multiplication and division) are impractical to meet if SMALL values other than binary SMALLs are implemented. Difficulties in handling marginal cases correctly have contributed to the failure of implementations to support SMALL values other than binary, despite the fact that typical operations which are likely to be used do NOT raise these accuracy issues.

!rationale

Requirements [1], [2] and [3] really come under the general aim of removing restrictions where possible. In at least some of these cases, the restrictions imposed by the current RM may not be required, and Ada 9X should minimize these restrictions to the greatest extent possible.

Requirements [4] and [5] reflect the fact that much greater uniformity is both possible and desirable with respect to fixed-point implementation. The RM allows considerable freedom to implementations in the choice of how to implement fixed-point and what values of SMALL to allow, under the impression that this may be hardware dependent.

Since fixed point operations are simply integer operations at the underlying implementation level, and all machines support integer operations, there are really no implementation dependent considerations, or considerations having to do with the semantics of underlying hardware. It is thus practical and desirable that the RM exactly specify the level of required support for fixed-point.

Requirement [5] addresses the current situation in which it is not hardware considerations which dominate implementation decisions, but rather algorithmic concerns as to what can or cannot be implemented. There is still some controversy over what requirements of the RM can be met in an efficient manner. These concerns must be resolved in Ada 9X, and it must be the case that the accuracy requirements reflected in Ada 9X correspond to operations which can be efficiently implemented given typical integer operations available on a wide range of hardware.

!appendix

See requirements on pages 9, 11, 18, 21, 22 of "The Fixed-Point Facility in Ada".

%reference RR-0357 Decimal

This RR consists of two complaints. The first is that the compiler in use does not allow decimal small. The writer recommends that

"The language should be amended to allow the clause:

for MONEY'SMALL use 0.01"

As noted in RI-0903, the RM certainly allows this clause, and in RI-0903, it is made clear that this representation clause should be required.

%reference RR-0400 Fixed Multiplication & Division with Universal Real Operands.

This RR suggests that it should not be necessary to qualify universal real operands in fixed-point multiplication and division, and claims that the problems in allowing this extended usage are tractable.

%reference RR-0401 Accuracy Required of Composite Fixed-Point Operations

This RR argues in detail that the accuracy requirements for fixed-point are too severe. It is perhaps somewhat too pessimistic, for example it does not take into account Paul Hilfinger's work, which shows that at least some of the cases labeled as impractical or unknown difficulty are in fact tractable. Nevertheless, the basic concerns in this RR are consonant with the concerns of this RI.

%reference RR-0591 Fixed Multiplication/Division with Universal Real Operands

This RR argues for allowing the omission of explicit types for real literals and other universal real operands in fixed-point multiplication and division.

%reference RR-0592 Accuracy Required of Composite Fixed-Point Operations

This RR argues that the accuracy requirements of composite operations are excessively severe.

!rebuttal

This RI must be considered incomplete until it documents why the designers of Ada83 chose to distinguish multiplication and division from addition and subtraction in defining the operations on fixed-point types.

From safe-programming point of view, there is much to be said for requiring the programmer to carefully specify his intentions when doing something out of the ordinary that may well be a mistake. From this point of it seems unwise to allow addition between different fixed-point types without explicit conversions. If it is true that these conversions "cannot always be written in a simple manner (since introducing the conversions may result in constraint errors)" then a more appropriate solution might be to repair the type conversion mechanism.

!topic Simplify Floating-Point Model
!number RI-0905
!version 1.4
!tracking 5.2.3.3.1, 5.2.3.3.2, 5.2.3.5.2, 5.2.3.5, 3.2.1
DRAFT DRAFT DRAFT DRAFT DRAFT DRAFT

!Issue revision (1) important,small impl,moderate compat,mostly consistent

1. Remove all discussion of model numbers and model intervals from Ada.

!Issue revision (2) compelling,small impl,moderate compat,mostly consistent

2. Require that Ada floating-point semantics be defined by reference to the ISO standard for language-independent floating-point (LCAS, the Language Compatible Arithmetic Standard).

!reference RR-0225
!reference RR-0252
!reference RR-0253
!reference RR-0255
!reference RR-0346
!reference RR-0348
!reference RR-0425
!reference RR-0454
!reference RR-0492
!reference RR-0564
!reference RR-0636
!reference RR-0637
!reference RR-0664
!reference RR-0720
!reference RR-0731

!problem

The current floating-point model in Ada is at the same time complex and unsatisfactory.

It has been criticized by users from two points of view. First of all, it is complex and leads to confusion. Second, the fact that Ada attempts to be more explicit than any other language in the required semantics for floating-point confuses users. In practice, implementations simply map to the underlying floating-point hardware, but users are worried that implementations may take advantage of the freedom stated in the RM (which is there to accomodate various hardware models of floating-point) to do undesirable strange things. For example RR-0253 is concerned that a declaration of DIGITS 6 in an IEEE implementation will result in something other than the natural choice of IEEE short form.

It has also been criticized by numerical analysts of the IEEE school, who find that the requirements in Ada are not in fact strong enough to allow useful analysis of programs in an implementation-independent manner. They would prefer NO statement at all with regards to floating-point formats and accuracy, as is the custom in other languages.

Finally, it is criticized by those who DO support the general approach as not being a successful implementation of this approach.

!rationale

Ada attempts to go much further than any other language in defining floating-point semantics. As noted above, the attempt is generally perceived as unsuccessful, and adds a great deal of complexity to the reference manual.

The LCAS effort is an attempt to rework the basic Ada approach to floating-point semantics in a more successful manner. By simply referencing this standard, we can achieve a significant simplification of the RM, while at the same time improving the floating-point semantics. This approach will not satisfy the IEEE school numerical analysts, who prefer nothing to be said, and are not favourably disposed towards the LCAS effort, but it is certainly an improvement from their point of view, and importantly it will simplify the language from a users point of view with no loss of functionality.

In practice, implementations simply map floating-point operations in Ada to the underlying hardware. This is done even in cases where the hardware does NOT satisfy the required RM semantics (e.g. division on the Cray), so the proposed change, although having a large effect on the formal semantics of Ada, and in particular on the presentation in the RM, will not have any significant effect on either Ada implementations or existing Ada code.

!appendix

%reference RR-0225 Floating-Point Precision

This RR complains that DIGITS is not an appropriate approach to specifying precision in Ada. It argues that the digits approach gives too much freedom to compilers, since the length of the binary mantissa cannot be exactly specified.

%reference RR-0252 Doing Math in Ada

This RR is a general complaint about the mathematical model in Ada, it suggests getting rid of "the verbage having to do with model and safe numbers". At the same time it wants more control over such areas of rounding. It also argues in the list of solutions that the range for real types should include the end-points, and that non-binary representations for delta and digits are not "helpful in the embedded community".

%reference RR-0253 Digits to Specify Real Number Accuracy and Precision and the Associated Transportability/Efficiency Problems (Similarly for Delta)

This RR complains that the DIGITS specification is not what is needed to specify the desired accuracy. It also claims that non-binary representations are not useful in the embedded community, and complains that IEEE arithmetic features such as infinity are not supported or supportable.

%reference RR-0255 T'EPSILON is Inadequate for Real, Floating-Point Numbers

This RR complains that T'EPSILON is not uniform through the floating-point range, which is of course true for all floating-point models. It seems completely confused, but presumably expresses a general confusion at the complexity of the current numeric model.

%reference RR-0346 Determination of Mantissas and Exponents for Real Numbers

This RR wants a procedure for splitting reals into mantissa and exponent values. Note that such an operation is provided in the proposed Generic Primitive Functions secondary standard.

%reference RR-0348 Operations on Real Numbers

This is a request for the inclusion of standard math library functions such as trigonometric functions. Note that these operations are provided in the proposed Generic Elementary Functions secondary standard.

%reference RR-0425 Open Ranges for Real Types

This RR argues in favour of a syntactic extension to allow open ranges for real types.

%reference RR-0454 Entier Functions of Real Types

This RR wants entier (truncate to integer) functions for floating-point types (the heading says real, but the body is for floating-point only, though presumably the same need exists for fixed-point). Note that this operation is provided in the proposed Generic Primitive Functions secondary standard.

%reference RR-0492 Suppress the Binding Between Mantissa and Exponent Size in

Floating-Point Declarations)

This RR argues that the link between precision and the exponent range results in inappropriate type selection in some cases.

%reference RR-0564 Safe Numbers for Floating-Point Types

This RR argues that safe numbers of a floating-point base type should not necessarily be defined in terms of the model numbers of the base type, on the grounds that this would give desirable additional implementation freedom.

%reference RR-0636 Floating-Point Non-Numeric Values (NaN's)

This argues for NOT permitting the admission of NaN's into the language, and admits that this is incompatible with the IEEE standard. It also contains a number of proposed floating-point axioms, some of which are potentially problematical with regards to negative zero.

%reference RR-0637 The Status of Floating-Point "Minus Zero"

This RR complains that minus zero is an "algebraic abomination" and should be eliminated from Ada. Current Ada actually is in full agreement with the RR as written, but, as the writer notes, this is incompatible with the IEEE standard.

%reference RR-0664 Adding Attributes 'IMAGE and 'VALUE to Floating-Point Types

This RR wants IMAGE and VALUE extended to floating-point, it does not mention fixed-point, although presumably the same argument applies.

%reference RR-0720 The Floating-Point Model Needs to be Improved

This RR argues that the Brown model as implemented now is unhelpful and inadequate.

%reference RR-0731 Simplification of Numerics, Particularly Floating-Point

This RR argues for taking the LCAS (Language Compatible Arithmetic Standard) into account as a tool for simplifying the description of numerics.

!rebuttal

!topic Provide for Compatibility with IEEE Floating-Point
!number RI-0906
!version 1.3
!tracking 5.2.3.3.1, 5.2.3.5, 5.2.3.3.2, 5.2.3.5.2
DRAFT DRAFT DRAFT DRAFT DRAFT DRAFT

!Issue revision (1) important,moderate impl,upward compat,mostly consistent

1. Ada should allow implementations to conform to the IEEE 754 floating-point standard.

!Issue secondary standard (2) compelling

2. A standard binding for IEEE-754 Floating-Point should be provided.

!reference RR-0011
!reference RR-0253
!reference RR-0346
!reference RR-0454
!reference RR-0492
!reference RR-0636
!reference RR-0637

!reference

"The Importance of Nothing", Kahan

!reference

Proposed IEEE binding for Ada, Robert B. K. Dewar

!problem

The current Ada standard is mostly consistent with the implementation of a full IEEE_754 binding. However, there are a number of small respects in which there is a clash between IEEE semantics and Ada semantics as follows:

— Machine overflows is static which is not appropriate to the IEEE model of dynamic control over the handling of overflow.

— Infinite and NaN values should be permitted. This is not to say that Ada should require the handling of such values, but it should not preclude them. At the moment, infinite values can be introduced if MACHINE_OVERFLOWS is FALSE, but the introduction of NaN's is more dubious.

— TEXT_IO must allow for the possibility of infinite and NaN values on both input and output, and also when 0.0 is output, the sign should be preserved, since negative zero is importantly different from positive zero

in IEEE implementations.

— A full implementation of P754 must take advantage of the 11.6 freedom to keep intermediate results in higher precision than the result type. This is not merely an optimization, but is fundamental to IEEE semantics, and thus should be addressed in the context of floating-point operations, not simply as a possible optimization.

Even on machines supporting IEEE arithmetic, Ada implementations do not implement the full IEEE-754 standard. Requirement [2] suggests that such an implementation requirement would be appropriate for IEEE machines.

!rationale

The importance of the IEEE-754 standard is clearly established. Virtually all high-performance microprocessors implement this standard, and it is likely that new super-computer hardware will also follow in this direction.

While it is not practical to require that Ada follow IEEE semantics, it is important that Ada accommodate IEEE implementations, and indeed it is highly desirable that a standard IEEE binding be accorded secondary standard status.

The semantic adjustments required to Ada for smooth support are minor and Ada 9X should take this requirement for IEEE compatibility into account.

Addressing RR-0637, it certainly should be the case for normal Ada semantics, as indeed it is in P754, that -0.0 should be treated the same as +0.0, but it is not at all the case that -0.0 is an algebraic abomination. On the contrary, as discussed in the Kahan paper, -0.0 stands for a very small negative number that has underflowed, and is thus quite different from +0.0, which stands for a very small positive number that has underflowed. Ada should not preclude treating signed zeroes appropriately.

RR-0636 argues directly against the support of the IEEE standard, but does not give a convincing rationale for requiring Ada to take a hostile approach. It certainly seems inappropriate to move in the direction of being even less friendly to IEEE than the current Ada standard.

Finally, addressing RR-0011, giving a value of 1 to 0**0 is algebraically defensible and generally consistent with IEEE semantics. If we regard 0.0 as a stand-in for a small value which has underflowed, then 1.0 is a much more appropriate value for 0.0 ** 0.0 than raising an exception. Consider the expression:

f(n) ** g(n)

where f(n) and g(n) are non-zero analytic functions whose limits as n approaches 0 themselves approach 0. In this situation the value of the expression ALWAYS approaches 1.0 as n approaches 0.

!appendix

!reference RR-0011 0**0 should be indeterminate

This RR argues that 0**0 should raise an exception rather than return 1.0. Although the IEEE standard does not address this issue explicitly, since exponentiation is not included in the standard, it is general practice in IEEE implementations including exponentiation to return 1 for such operations as in Ada.

!reference RR-0253 Digits to Specify Real Number Accuracy and Precision and the Associated sp This RR complains that the DIGITS specification is not what is needed to specify the desired accuracy. It also claims that non-binary representations are not useful in the embedded community, and complains that IEEE arithmetic features such as infinity are not supported or supportable.

!reference RR-0346 Determination of Mantissas and Exponents for Real Numbers

This RR wants a procedure for splitting reals into mantissa and exponent values. Note that such an operation is provided in the proposed Generic Primitive Functions secondary standard. This operation is also recommended by the IEEE standard.

!reference RR-0454 Entier Functions of Real Types

This RR wants entier (truncate to integer) functions for floating-point types (the heading says real, but the body is for floating-point only, though presumably the same need exists for fixed-point). Note that this operation is provided in the proposed Generic Primitive Functions secondary standard. This operation is also recommended by the IEEE standard.

!reference RR-0492 Suppress the Binding Between Mantissa and Exponent Size in Floating-Point Declarations

This RR argues that the link between precision and the exponent range results in inappropriate type selection in some cases. In particular, the relation as currently required leads to a requirement for understating the precision of the IEEE standard types.

!reference RR-0636 Floating-Point Non-Numeric Values (NaN's)

This argues for NOT permitting the admission of NaN's into the language, and admits that this is incompatible with the IEEE standard. It also contains a number of proposed floating-point axioms, some of which are potentially problematical with regards to negative zero.

!reference RR-0637 The Status of Floating-Point "Minus Zero"

This RR complains that minus zero is an "algebraic abomination" and should be eliminated from Ada. Current Ada actually is in full agreement with the RR as written, but, as the writer notes, this is incompatible with the IEEE standard.

!reference

"The importance of Nothing", Kahan [This reference needs to be stated accurately!]

This paper describes in detail the importance of negative zero in floating- point calculations.

!reference

Proposed IEEE binding for Ada, Robert B. K. Dewar

This is a complete proposal for an IEEE binding for the current definition of Ada. It is under active consideration by the NUMWG working group of SIGAda, and by ISO WG9/NRG (Numerics Rapporteur Group). This document includes a discussion and indication of problems in interfacting P754 to the current definition of Ada.

!rebuttal

5.3 Types/Operations/Expressions

RI-5010
RI-5190
RI-5200
RI-0004
RI-0110
RI-2011
RI-2034
RI-2035
RI-1010
RI-1011
RI-5210
RI-0200
RI-3749

!topic renaming declarations for types
!number RI-5010
!version 1.6
!tracking 5.3.1.1

!Issue revision (1) compelling,moderate impl,upward compat,consistent

1. Ada9X shall provide a means to declare synonyms for types, which shall allow packages to import types and then re-export them with the original operations and literals (under the new type name), along with any newly defined operations. Subtype and derived type declarations in Ada83 do not achieve the desired type renaming capability.

!reference AI-00378 Subtype declarations as renamings
!reference AI-00390 Visibility of character literals
!reference RR-0069 Visibility probs when creating new types from old
!reference RR-0172 Need type renaming for exporting imported types
!reference RR-0239 Need true type renaming
!reference RR-0455 The import and export mechanisms of Ada are too limited
!reference RR-0467 Need way to rename type and get its operations
!reference RR-0610 Why not allow RENAME for types and subtypes?

!reference
Bardin, B. and C. Thompson, "Composable Ada Software Components and the Re-Export Paradigm," Ada Letters, Vol. VIII, No. 1, January 1988, pp. 58-79.

!problem

Library packages that import types from other packages to provide additional operations, cannot easily re-export the enhanced type without revealing the source of the original imported type. Subtype and derived type declarations do not achieve exactly the same effects as renaming. When both the original and enhanced types are visible, extensive use of type conversions and/or selected component names is required.

!rationale

Ada9X must allow programmers to hide program implementation details and adequately control visibility. Ada83's facilities require programmers to reveal sources of imported types, or to re-declare operations and literal values that will conflict if original types are visible.

!appendix

1. Using a subtype declaration to rename a type for export does not make the original operations visible. To avoid a context clause referencing the source of the original type in an application, new operations must be explicitly defined for the subtype.

2. Using a subtype declaration to rename an enumeration type for export does not make the original enumeration literals visible. Two workarounds include: declaring constants of the renamed subtype (assigning them corresponding values from the original type), and renaming the enumeration literals as parameterless functions. The renamed functions are not static, however, and cannot be used as case statement choices.

3. If both the original and an exported subtype with redefined literal values and operations are visible, extensive use of selected component names is required to distinguish redefined literals and operations.

4. Derived scalar types come closer to the desired capability. A derived type, however, is a new type, not simply a pseudonym for the original. For derived composite types, component and index types are not accessible. A workaround for index types and scalar components is to export them as subtypes. There are no workarounds for exporting the types of composite components of composite types.

5. If both the original and an exported derived type are visible, extensive use of type conversions is required to distinguish the two types. This can't be helped even if the new type was intended only to enrich the original.

6. The referenced revision requests ask for a renaming declaration that would provide a simple pseudonym for a type. Operations declared for any pseudonym would (presumably) be applicable to all values/objects of the type.

7. Potential solutions should take into account possible interactions with Object-Oriented Programming (OOP) changes to Ada9X's type system (e.g., visibility rules for packages declaring "parent" types) and passing packages as parameters to generics.

%reference AI-00378
Using a subtype to rename an enumeration type does not make the enumeration literals visible. This capability should be considered for a possible future language revision.

%reference AI-00390
Refers to a subtype declaration as a type renaming and discusses the unexpected consequences.

%reference RR-0069
Describes the difficulty of importing a type, adding some new functionality, and then exporting the extended type without revealing the source of the original type.

%reference RR-0172
Ditto.

%reference RR-0239
Describes the shortcomings of subtype declarations (i.e., losing the enumeration literals) as a mechanism for renaming enumerated types.

%reference RR-0455
Describes the difficulty of importing a type, adding some new functionality, and then exporting the extended type without revealing the source of the original type.

%reference RR-0467
Describes the shortcomings of subtype declarations (losing the the operations and literals) and derived types (requires numerous type conversions) as mechanisms for renaming types.

%reference RR-0610
Points out inconsistency in not being able to rename types.

!rebuttal

!topic multiple related derived types
!number RI-5190
!version 1.2
!tracking 5.3.1.2

!Issue out-of-scope (un-needed) (1) desirable,moderate impl,upward compat,mostly consistent

1. Ada9x shall provide a mechanism by which two or more types can be derived at the same time, yielding subprograms with the corresponding combinations of parameter and result types.

[Rationale] The examples presented in the referenced revision requests make it clear that generics satisfy the stated need. This is a case where an Ada83 language feature almost provides a desired capability but, since Ada83 provides a workable solution, extending the language is not warranted.

!reference RR-0052 Multiple derived types from same package is very awkward
!reference RR-0482 Multiple derived types from same pkg don't do what you want

!problem

When a subprogram has multiple parameter/result types from which new types are derived, the subprograms derived are those that differ from the original in only one type. Often, what is needed are subprograms defined on the combination of new types, but these are not produced.

!rationale

!appendix

Workarounds include using lots of type conversions, and defining the desired subprograms as generics with the derived types passed in as instantiation parameters.

Suggested solutions include multiple derived type declarations (RR-0052), and derived subprogram declarations (RR-0482).

%reference RR-0052
When a subprogram has multiple parameter/result types from which new types are derived, the subprograms derived are those that differ from the original in only one type. Often, what is needed are subprograms defined on the combination of new types, but these are not produced.

%reference RR-0482
Ditto.

!rebuttal

!topic Type Declarations
!number RI-5200
!version 1.4
!tracking 5.3.1.3

!Issue revision (1) desirable,moderate impl,moderate compat,mostly consistent

1 – Unify type declarations] An attempt shall be made, in considering other changes to Ada's type system, to remove unnecessary restrictions and inconsistencies in type declarations that complicate programming.

!Issue out-of-scope (un-needed) (2) desirable,severe impl,bad compat,inconsistent

2 – Mutable types] Ada∋x shall provide mutable types to support knowledge representation for artificial intelligence applications.

[Rationale] This would be a radical change to Ada's type model.

!reference RI-3000 information hiding, private types
!reference RR-0010 allow private/incomplete types to be derived
!reference RR-0012 non-orthogonality makes type derivation hard
!reference RR-0080 derived types are clumsy
!reference RR-0437 "supertype" capability for merging enumeration types
!reference RR-0455 (related) re-export mechanism
!reference RR-0558 derived types should identify subset of operators
!reference RR-0560 derived private types are awkward/inconvenient
!reference RR-0690 complete incomplete/private types by subtype declaration

!problem

A number of Ada83's type declaration rules make programming awkward and inconvenient. User's complaints include:

a. Not being able to use a derived type to complete the declaration of a private or incomplete type with discriminants

b. Not being able to use a subtype to complete the declaration of a private or incomplete type

c. Not having a way to hide selected operations for derived types

d. Not having a way to combine enumeration types to form supertypes

e. Not being able to define subtypes with non-contiguous ranges

!rationale

Removing unnecessary restrictions and making type declaration rules more uniform would simplify program development, reduce programming errors, and simplify software maintenance.

!appendix

%reference RR-0010
The full definition of a private or incomplete type with discriminants should be allowed to be a derived type, provided the discriminants of parent type conform in some reasonable way.

%reference RR-0012
Ada should provide mutable types to support knowledge representation for artificial intelligence applications. The author recognizes that this would be a radical change to Ada's type model but would like to see something done to improve Ada's abstraction capabilities.

%reference RR-0080
Current syntax requires renaming of derived types and operations, which is clumsy, verbose, and limits the utility of derived types.

%reference RR-0437
Provide the converse of Ada's subtyping mechanism to allow the creation of supertypes by merging or extending "compatible" enumerated types.

%reference RR-0455 (related) re-export mechanism
Describes the difficulty of importing a type, adding some new functionality, and then exporting the extended type without revealing the source of the original type.

%reference RR-0558
There should be a way to selectively hide specified operations for derived types.

%reference RR-0560
When defining two closely related packages, there should be a way to derive a type in one package from a private type declared in another and also have access to the type's representation.

%reference RR-0690
Incomplete and private types should be allowed to be completed by subtype declarations rather than full (derived) type declarations.

!rebuttal

!topic Syntax of relational expressions
!number RI-0004

!reference RR-0031
!reference RR-0046

!problem

Users would like a convenient method for specifying non-continuous elements of a discrete type in a "membership" expression. For example,

```
if X in 3 | 4 | 6 | 9 then ...
```

!topic Non-contiguous subtypes of discrete types
!number RI-0110

!reference RR-0058
!reference RR-0603

!problem

Subtypes of discrete types currently must consist of contiguous ranges of elements. Unfortunately, there are many cases in which a subtype of non-contiguous elements is desired. When interfacing to external environments, an order for the elements may be implied by a representation clause. As a result, a subtype cannot be used to represent the desired data type.

!appendix

%reference RR-0058

%reference RR-0603

!topic Restrictive Pointers
!number RI-2011
!version 1.8
!tracking 5.3.3

DRAFT DRAFT DRAFT DRAFT DRAFT DRAFT

!Issue revision (1) compelling,moderate impl,upward compat,mostly consistent

1. (General References) Ada9X shall provide a sort of type whose values are references to objects of a given base type. The referenced objects may include both those created by an allocator and those created by a declaration. Ada9X shall provide operations for obtaining a reference for an object in any appropriate reference type; these operations shall maintain strong typing. Ada9X shall restrict the arguments and placement of these operations based on safety considerations; for example, Ada9X may restrict obtaining references to stack-allocated objects to eliminate the potential for dangling pointers.

!Issue revision (2) important,small impl,upward compat,mostly consistent

2. (Address Conversion) Ada9X shall provide operations to convert machine addresses into values of some reference type. An attempt to obtain a reference for an object in a type that is not appropriate shall be an error. [Note: a type might not be appropriate because it has restricted range or requires an alignment different from the machine address being converted.] An implementation shall be required to document those situations where the operations for obtaining access values and for address conversion will cause an exception to be raised.

!reference RR-0018
!reference RR-0258
!reference RR-0293
!reference RR-0338
!reference RR-0524
!reference RR-0726
!reference RR-0773

!problem

There are many problems that are caused by the lack of general addressing in Ada83. Among them are the ability to create statically allocated data structures and to force the semantics of call-by-reference. Another problem occurs when the objects of a type are forced to particular memory locations but the number and placement of these objects is not known until execution time; a mechanism is needed to allow access to the objects from a set of address values.

!rationale

[2011.1]
The ability to have a flexible, general mechanism for the late binding of objects to names is fundamental to programming; the real question is how far a language can go and still maintain a high degree of safety. The solution described in RR-524 gives reasonable confidence that safety rules can be constructed that allow the use of general pointing without attendant dangling reference problems for a significant number of cases. Certainly, all objects that are statically allocated or allocated by an allocator could be covered.

[2011.2]
Address conversion should either work or (at least) cause an exception. The "Documentation" allows users with general addressing requirements to determine relatively quickly whether an implementation will suit their needs. The issue here is mainly one of portability since 'ADDRESS/UNCHECKED_CONVERSION frequently works; however, there are a number of cases where the values of reference types might not be the same (bit-wise) as SYSTEM.ADDRESSES, e.g. for an architecture with tagged references.

!appendix

%reference RR-0018 Need arrays w/variable-size elements and ptrs to objs via obj decls

RR-0018 wants to get access values for statically allocated objects; the generic supplied in the solution is erroneous.

%reference RR-0258 Need good way for access objects to point to object decl objects

RR-258 wants access values for all objects whether created by declaration or by an allocator. Also desired is conversion between type ADDRESS and access values.

%reference RR-0293 Improve usefulness of type ADDRESS, make more like access types

RR-0293 wants to have subtypes of ADDRESS that are like access types to facilitate type checking. By providing explicit conversion among these types and regular access types, an "untyped pointer capability" can be obtained.

%reference RR-0338 Provide ptrs to static objs, conversion between ADDRESS and access

RR-0338 wants access values for at least static objects but possibly for all objects. It goes on to cite a JIAWG "requirement" for arbitrary convertibility between ADDRESSes and access values; it then questions the need for this second capability.

%reference RR-0524 Support explicit references to an object

RR-0524 asks for support for explicit references to objects. Such references are divided into two types: passed (as in subprograms) and stored. The issue raised is that stored references have a lifetime problem (i.e. a dangling reference problem). Several useful concepts easily implemented with "explicit references" are mentioned.

%reference RR-0726 Need non-contiguous arrays, static pointers

%reference RR-0773 Problems building variable-length records from messages

RR-0773 wants to be able to obtain ADDRESSes for individual components of a record and then export them using access types. In addition, RR-0773 wants to be able to change the "last" discriminate of a record type to grow the record as more and more data is added.

!rebuttal

There is some sentiment that any facility for creating pointers should be coupled with a mechanism by which the declarer of an object or type should be able to restrict the creation of pointers. This mechanism could be employed so that the subunits of a compilation unit may be restricted from creating access values to objects local to the parent unit and exporting them in an uncontrolled fashion. In such a situation, a programmer might have to assume that even local variables were being changed by other program units (specifically, other tasks) in parallel with his own execution. Programming under such an assumption would be very difficult. One may argue that such a situation is already erroneous; the inclination here would be to have it possible to be illegal.

!topic Typing at a Program Interface / Interoperability on Data
!number RI-2034
!version 1.5
!tracking 5.3.4

DRAFT DRAFT DRAFT DRAFT DRAFT DRAFT

!terminology

In the following, a "external interface block" (EIB) is an area of memory that is used to communicate with a system element that is external to an Ada program. In the following, a "field" is a contiguous set of bits in an EIB.

!Issue revision (1) compelling,moderate impl,upward compat,consistent

1. (Layout of External Interface Blocks) Ada9X shall provide operations

   1. to allocate an external interface block of a given size;
   2. to specify the bit and storage unit ordering associated with an external interface block [Note: this includes some resolution of the bit- and byte-sequence or "endian" problem];
   3. to define a field in an external interface block;
   4. to fetch and store from any field in a external interface block;
   5. to convert between the data stored in a suitably-sized field and an Ada object of a discrete type specifying the same logical data.

[Note: this requirement should not be read as requiring a new entity to be introduced into the language. Tightening the allowed interpretations of UNCHECKED_CONVERSION and introducing repspecs for the the "endian" problem" could form the basis of a solution.]

!Issue revision (2) desirable,moderate impl,upward compat,mostly consistent

2. (Data Validation) Ada9X shall provide a mechanism to validate data that enters a program via an untyped interface. For this reason, Ada9X shall prohibit an implementation from removing uses of this mechanism during optimization.

!Issue revision (3) important,small impl,upward compat,mostly consistent

3. (References to Statically-Allocated External Interface Blocks) Ada9X shall provide a mechanism to statically allocate an external interface block and to obtain a reference for the block. The basic operations for such references shall include selection of a field from the external interface block designated by the reference, the operations involved in assignment, and the operation of converting the address of an external interface block into a reference for that external interface block. Ada9X shall allow the declaration of objects whose values are references to external interface blocks and aggregations of such objects. For language safety reasons, Ada9X shall restrict where a referenceable

external interface block may be declared, where such references may be obtained and stored, or both. [Note: the mechanism for selecting a field from a reference to an EIB does not necessarily follow the syntax of record component selection.]

!Issue revision (4) desirable,small impl,upward compat,mostly consistent

4. (Knowledge of Contiguous Allocation) Ada9X shall provide mechanisms by which an executing program may determine if a particular object is contiguously allocated without hidden fields. For such objects, Ada9X shall provide a portable way to treat the object as an array of fixed-size objects related to the fundamental memory width of the underlying machine.

!Issue out-of-scope (un-needed) (5) desirable,moderate impl,moderate compat,consistent

5. (Canonical UNCHECKED_CONVERSION) Ada9X shall define a UNCHECKED_CONVERSION so that no implementation-dependent fields are returned. Specifically, hidden fields are to be excluded and access values are not permitted to substitute for the data that would be accessed by the pointer.

!Issue out-of-scope (un-needed) (6) desirable,moderate impl,upward compat,inconsistent

6. (Weakening Typing at Subprogram Boundaries) Ada9X shall provide a mechanism to specify the weakening of typing at a subprogram boundary so that structural equivalence rather than name equivalence would be used to determine type conformance.

!reference RR-0017
!reference RR-0018
!reference RR-0103
!reference RR-0289
!reference RR-0353
!reference RR-0450
!reference RR-0458
!reference RR-0459
!reference RR-0554
!reference AI-00345

!problem

The problem that is being addressed here is that users need to specify the layout of memory in order to interface with external components in a system in a type-safe way and the current facilities are not adequate to do so. According to RR-459, the following considerations apply when considering the use of Ada objects as external interface blocks.

1. the effect of representation clauses is not fully specified (currently being addressed by the ARG) - unsigned and biased representations are not addressed by the standard
2. the effect of pragma Pack on arrays and records is not fully specified (also being addressed by the ARG)

3. the (minimal) required effect of Unchecked_Conversion is not well-defined
4. too much freedom is allowed in ordering of bits and in the extent of bit-numbering in record representation clauses
5. the permissible optimizations which remove redundant constraint checking are not well-defined and may be implemented in such a fashion as to prevent

In addition, the precise location of these external interface blocks should be a parameter of a program for portability reasons. What is needed is the ability to access an external interface block by some kind of reference and to have such references be eligible for being variables and being aggregated as can other Ada types.

!rationale

RI-2034.1 addresses the fact that a user must be able to explicitly control every aspect of the memory layout for a memory area that is to be used to communicate with an external component. It is not necessary to have the same control over the layout of objects once they have entered the program. An important point is that "memory layouts" are inherently more weakly typed than Ada objects. As an example of this, it is entirely reasonable for Ada objects to ALWAYS carry internal discriminants, but this is an impossible constraint for "memory layouts". Although not a requirement, programs using these facilities could be made more type-safe if the language supported some kind of referencing capability for fields; this is different from the referencing capability for EIBs in RI-2034.3.

RI-2034.2 addresses the issue that data coming in through an external interface will not always obey the data constraint rules of Ada objects. Some consideration must be given in the language to how such data can be valida d for safe use throughout the program. This facility could be built into the fetch-field operation of RI-2034.1.

RI-2034.3 addresses the fact that the semantics of the underlying execution unit may dictate that routines that are parameterized by a memory area used for communications may not use by-copy parameter transmission. In such cases, Ada9X should provide a mechanism so that a reference to the memory area can be passed as the parameter instead. Also, the addresses of external interface blocks are frequently set by the systems design process and not by the program itself. Nevertheless, the program must be able to reference such external interface blocks and to manipulate lists of such references. Consider the "redundant data scheme" where several processor are mapping redundant data into the address of another processor. In such a case, the processor will want to have a list of the locations where the data can be found; this, of course, depends on the number of processors in the system– a parameter that can vary during the execution of the program and is almost certainly a parameter to the program. During "normal" operation, the program will want to access the data through a reference without incurring any overhead; but, in case of a fault, the program will want to be able to change its reference to a different external interface block by consulting its list of "good locations". Finally, certain data may be preloaded into a external interface block (a read-only memory, say) and the program will wish to reference this data directly rather than incur the memory and time penalty of copying the data into dynamically-allocated objects.

RI-2034.4 addresses the fact that the external communications and I/O interfaces of execution environments are frequently characterized in terms of "memory objects" instead of "Ada objects". Frequently, a user would like to view an Ada object as a string of memory objects but he has no portable indication of when it is reasonable to do so and no portable mechanism to use when it is reasonable. By having the ability to query this characteristic of objects, a program could test during elaboration to see if assumptions on certain objects were fulfilled and raise an exception before execution actually begins.

!appendix

%reference RR-0017 Allow conversion of complex types to a simple array of storage units

RR-17 addresses the problems that arise when constructing low-level I/O interfaces. Such interfaces are usually untyped and are defined in terms of a concept of physical continuity or (at best) byte/word vectors/streams. Significant maintenance problems arise because the is not sufficient language manifestations of the underlying implementation. Specific requests are: (1) for references to statically allocated objects, (2) the ability to treat objects as a byte vector, (3) the ability to query in the code (better if at compile time) to determine whether by-reference parameter transmission is used, whether an object is contiguously allocated, whether it is fixed-size.

%reference RR-0103 Unchecked_Conversion facilities are too limited

RR-103 describes a problem where interfacing is substantially complicated because UNCHECKED_CONVERSION is formally a function call rather than a type cast. In the best case, this may cause wasteful copying. However, if the structure is big enough then copying may not be possible. This situation would be less constraining (according to the RR) if one could get a reference to a substructure in a portable way.

%reference RR-0289 Need multiple views of a record structure but want no discriminant

RR-289 addresses the issue where the layout of a memory structure is determined by information not within the structure itself. This may occur for communications protocols where the interpreter state determines the correct view of the block. Ada's implementation of variant records is a bad match here.

%reference RR-0353 Unchecked conversion should not expose compiler-dependent fields

RR-353 discusses the problem that unchecked conversion on variant records is not at all portable depending on various hidden fields and allocation strategies employed by the compiler. The request suggests that unchecked conversion be defined to return a canonical representation.

%reference RR-0450 Need fast, powerful, and safe mechanism to break strong typing

RR-450 describes a significant problem where the format of storage is determined by several fields of a structure (not as discriminants) and/or by the program state. After discussing the problems with the available techniques to handle this problem, the request suggests that a local referencing capability be added to solve the problem. %reference RR-0458 Need convenient way to escape into weakly typed subpgm call

There is a significant maintenance problem that obtains when interfacing strongly-typed code to weakly-typed code such as database management systems, graphics subsystems, or communications subsystems. The goal is to avoid having the programmer respecify the type weakening at each call to a weakly-typed interface routine. The suggestion made is that a form of structural equivalence be instituted for parameter matching of specified weakly-typed subprogram parameters. Note that Modula-2 uses this approach in allowing arbitrary pointer types to be automatically cast into machine addresses at subprogram boundaries.

%reference RR-0459 (4.7.1)

RR-0459 is concerning with the interoperability between Ada programs; this is particular true in how a compiler is allowed to interpret certain specification w.r.t the layout of objects in memory. after a lengthy discussion, the following are recommended:

a. length clause given for a scalar type must be obeyed exactly (or rejected for AI-325 supportable reasons) and must apply equally to all subtypes of that type so that the logical size of objects of the type (the size of the type) is determined,

b. record representation clauses must be obeyed exactly (or rejected for AI-325 supportable reasons) and must be able to exclude hidden components from simple objects of the record type,

c. pragma Pack for arrays must result in no gaps between components whose size is commensurate with the hardware architecture and for arrays of Booleans must force a 1-bit representation on its components,

d. the combination of pragma Pack and a length clause for its component type must completely determine the top-level layout of unconstrainted array types,

e. length clauses applied to constrained array (sub)types must not include descriptors in the size,

f. at least signed and unsigned representations of discrete types must be supported for scalar objects and for scalar components of arrays and records,

g. unchecked conversion must be possible between the canonical representations of any two objects with the same logical size (as determined by length clauses, but excluding descriptors and hidden components),

h. explicitly written membership tests on values must never be optimized away.

%reference RR-0554 Need constraint checks for target of unchecked_conv. & i/o input

RR-554 notes the difficulty of validating objects after they are obtained via unchecked conversion and suggests a remedy of a pragma that forces checks in certain situations.

%reference AI-00345 Record type with variant having no discriminants

AI-345 wants pascal-style undiscriminated unions; since the AI mentions I/O explicitly, we can assume that the need is for specification of memory blocks.

%reference RR-0018 Need arrays w/variable-size elements and ptrs to objs via obj decls

RR-0018 wants to get dereferenceable references for statically allocated objects; that is, the references should have the same operations as access values. This is because there are applications where the objects actually are allocated by the systems design process and because untoward copying of constants is to be avoided.

%reference RR-0726 Need non-contiguous arrays, static pointers

RR-0726 wants to be sure that the static references of RR-0018 can be placed into an array and that there are objects whose values are such references.

!rebuttal

Although not explicitly required, the concept of an EIB could be realized by an addition to the language. It would be greatly preferable if the concept were mapped into some existing concept in Ada such as an array of storage units or (even better) a record with a fully specified representation. As such, requirements stating a need for improvements to the current facilities for representation specifications would be a better way to get at the user problems covered by this RI.

!topic Equality - redefinition
!number RI-2035
!version 1.4
!tracking 5.3.5

DRAFT DRAFT DRAFT DRAFT DRAFT DRAFT

!Issue revision (1) important,small impl,upward compat,consistent

1. ("="(x,y:same_type) return STANDARD.Boolean) Ada9X shall allow the predefined operation "=" on any type to be redefined and overloaded with a user-supplied definition.

!Issue revision (2) desirable,small impl,upward compat,consistent

2. ("="(x:a_type;y:any_type) return STANDARD.Boolean) Ada9X shall allow "=" to be overloaded with any user-supplied definition returning STANDARD.Boolean.

!Issue out-of-scope (un-needed) (3) desirable,small impl,upward compat,inconsistent

3. ("="(x:a_type;y:any_type) return whatever_type) Ada9X shall allow "=" to be overloaded with any user-supplied definition.

[Rationale] This proposal breaks the current model in that it is not clear how "/=" would be the opposite of "=" in this case.

!reference RR-0008
!reference RR-0025
!reference RR-0412
!reference RR-0513
!reference REFER ALSO TO RR-0609
!reference WI-0214

!problem

Users would like to be able to define their own equality function and to use this function as the normal infix definition to improve readability.

!rationale

The two requirements here propose different degrees of relaxation of the current rule prohibiting (direct) overloading/redefinition of "=". The ability of a user to overload such a designator must be carefully balanced against a potential loss of readability that results when the infix operators are redefined in nonobvious ways. The issue here is simply the syntactic one; the requirements do not try to address what may be thought of as various implicit uses of equality throughout language (such as in case statements and membership tests).

!appendix

%reference RR-0008 Allow overloading of equality operator for all types

RR-8 wants to be able to overload "=" for any type and returning STANDARD.BOOLEAN. The request would still allow the restriction to having the operands be of the same type.

%reference RR-0025 Allow overloading of equality operator with different type operands
%reference RR-0412 Allow overloaded "=" for all types, not just limited types

RR-25 and RR-412 want to be able to overload "=" for any typeS and returning STANDARD.BOOLEAN. The request would disallow the restriction to having the operands be of the same type.

%reference RR-0513 Allow "=", "/" overloading for any type, returning any result type

RR-513 wants to be able to overload "=" for any typeS and returning any type. The request would provide for APL-style "=".

%reference REFER ALSO TO RR-0609 (4.1.1)

RR-609 wants to be able to overload "=" for any type and returning STANDARD.BOOLEAN. The request would still allow the restriction to having the operands be of the same type.

%reference WI-0214 All cases of operator overloading should obey the same rules

All cases of operator overloading should obey exactly the same rules.

!rebuttal

!topic definition of static expressions (not wrt generic formals)
!number RI-1010
!version 1.8
!tracking 5.3.6

DRAFT DRAFT DRAFT DRAFT DRAFT DRAFT

!Issue revision (1) desirable,moderate impl,upward compat,consistent

1. The definition of a static expression in Ada9X should more closely correspond to the notion of "reasonable to evaluate at compile time". In particular, certain uses of type conversions, membership tests, and short-circuit control forms shall be allowed in static expressions. Finally, an attempt shall be made to include certain references to array attributes and the 'SIZE attribute in static expressions.

!reference RR-0009
!reference RR-0099
!reference RR-0452
!reference RR-0455
!reference RR-0639
!reference RR-0705
!reference RR-0712
!reference AI-00128
!reference AI-00539
!reference AI-00812

!problem

The definition of what constitutes a static expression in Ada83 appears to be unnecessarily and arbitrarily restricted. As there are a number of places in the language where static expressions are required, these restrictions increase the complexity and difficulty of using the language.

Furthermore, generalizing the definition of a static expression is seen by some as a way to force more expressions to be evaluated at compile time and thus encourage a form of compiler optimization.

!rationale

Static expressions in Ada must be restricted enough so that it is reasonable to require an implementation to evaluate them at compile time and yet general enough to not be a burden to the programmer in the places where the language requires static expressions.

It seems Ada83 is too cautious in the way this line is drawn. There is no reason why type conversions (where the expression is static and the target type is discrete and static), membership tests (where the simple expression and range or type mark are static), and short-circuit control forms (where both operands are static) should not be allowed in static expressions. Although there are relatively easy workarounds if these forms are not

included in static expressions, making these changes seems to make the language more consistent and also seems relatively easy to do with regard to the current language definition. Finally, it also appears to be desirable to extend the class of attributes that are allowable in static expressions if this does not add significant complexity to the definition of the language.

!appendix

%reference RR-0009 Some type conversions should be static

The 'POS workaround is circuitous.

%reference RR-0099 Explicit type conversions should not disturb static expressions

Also, some array attributes should be static.

%reference RR-0452 Need functions in static exprs (or overloadable constants)

Constants and functions returning a constant value are similar but constants can be in static expressions but may not have overloaded names and functions returning a constant value cannot be in static expressions but may have overloaded names.

%reference RR-0455 The import and export mechanisms of Ada are too limited

Re-exporting generic formal types and objects causes loss of staticness.

%reference RR-0639 Need compile-time interpretation of procedural init. code

Compile-time interpretation of procedural code that builds constant tables would be helpful.

%reference RR-0705 For better performance, remove restrictions on static expressions

More expressions should be considered static to encourage compilers to evaluate them at compile time and not generate code to evaluate them. These include such things as trig functions, type conversions, attributes of arrays/records, concatenations, and array operations.

%reference RR-0712 Generalize defn. of static exprs, esp. within generic units

Static expressions are too conservative. Include more expressions, attributes, and array bounds.

%reference AI-00128 Membership tests and short-circuit not allowed in static exprs

This is a binding interpretation.

%reference AI-00539 Allow static array and record subtypes

This is a study AI. Submitter wants 'SIZE on an array or record type to be static for certain array or record types.

%reference AI-00812 SAFE_SMALL and SAFE_LARGE should be static

This is a study AI. The value returned is a characteristic of the base type, not the subtype. Hence this is knowable at compile time.

!rebuttal

!topic staticness inside generic units with respect to generic formals
!number RI-1011
!version 1.7
!tracking 5.3.6

DRAFT DRAFT DRAFT DRAFT DRAFT DRAFT

!Issue out-of-scope (un-needed) (1) important,severe impl,upward compat,inconsistent

1. Ada83 restrictions based on the "non-staticness" of generic formal parameters shall not be present in Ada9X. For example, it shall be possible to parameterize a generic unit with respect to a compile-time known quantity, it shall be possible for a static expression to be of a generic formal type, attributes of generic formal types shall be allowed within static expressions, and restrictions on case statements and discriminants with respect to generic formal types shall be removed.

[Rationale] These suggestions are contrary to two Ada principles: (1) that generic units be compiled separately from their instantiations and (2) that static expressions be evaluated at compile time. It has been suggested that the ideas described above could be provided in the language by improving the "contract" for generics to the point of defining new kinds of generic formals that match only static actuals. This is insufficient since it would still be impossible to evaluate, at compilation time for a generic unit body, expressions involving these generic formals. As an illustration of this difficulty, consider the expression A'FIRST(E) appearing in a generic unit where E is a "static" expression involving generic formals. In compiling the generic unit it is clearly necessary to determine the type of the expression, but this in turn depends (in general) on the value of E (which cannot be determined independent of the instantiations of the generic). What appears to be needed to solve the problems here is a form of generic unit in which the unit is not compiled on its own; rather it is compiled (and in particular, legality analysis is performed) at the point of each instantiation. This seems like a major shift in philosophy concerning generic units in is therefore felt to be unwarranted in light of the degree of urgency that seems to be associated with this problem.

!reference RR-0048
!reference RR-0190
!reference RR-0227
!reference RR-0342
!reference RR-0445
!reference RR-0455
!reference RR-0511
!reference RR-0705
!reference RR-0712
!reference AI-00190
!reference WI-0219

!reference RI-1017

!problem

The usefulness of generic units is limited due to the fact that there are language contexts that require static expressions and static expressions may not depend in any way on generic formal parameters. Similarly, there are restrictions concerning case statements and discriminants in generic units due to the fact that details on a generic formal type are not known when the generic unit is compiled. This makes it difficult to develop generic software. It also gives rise to frustrations in converting non-generic software to generic software.

!appendix

%reference RR-0048 Extend definition of static subtypes to generic formal types

This may be a red herring. Problem in his example appears to be due to 'SIZE on composites. Possible a RI-1010 item.

%reference RR-0190 Provide mechanisms for accessing the base type of a constrained type

Not clear this belongs here. He feels his problem would be lessened if he could have a type declaration in a generic unit where the range depended on generic parameters.

%reference RR-0227 Allow generic parameterization with static numeric quantities

His device handler argument appears to be incorrect.

%reference RR-0342 Don't implement requests which will break generic code sharing

He is concerned about staticness, contract model. Do not allow generic parameters to be static.

%reference RR-0445 Non-staticness of generic formals poses problems

Case statements, discriminant types, MANTISSA/DIGITS attributes all pose problems.

%reference RR-0455 The import and export mechanisms of Ada are too limited

Would like named numbers as generic formals. May want staticness only for re-exported named numbers.

%reference RR-0511 Provide mechanisms for accessing the base type of a constrained type

Would like attributes of generic formal types to be static to be used in type declarations.

%reference RR-0705 For better performance, remove restrictions on static expressions

The concern here is forcing the compiler to do more expression evaluation.

%reference RR-0712 Generalize defn. of static exprs, esp. within generic units

Wants to allow more expressions to be considered static, particularly those involving generic formals

%reference AI-00190 Static expressions cannot be of a generic formal type

Illegal aggregates in generic units must be detectable at compile time.

%reference WI-0219 Generic and non-generic constructs should be uniform (DW 3.III.F.1)

Restrictions on generic formal parameters wrt staticness are painful.

!rebuttal

!topic null ranges
!number RI-5210
!version 1.7
!tracking 5.3.8

DRAFT DRAFT DRAFT DRAFT DRAFT DRAFT

!Issue revision (1) desirable,small impl,moderate compat,unknown compat

1 -- Null ranges] Ada9x should redefine 'LAST to be 'FIRST-1 for ranges where the specified value for 'LAST is less than 'FIRST-1.

!reference RR-0234 restrict null ranges
!reference RR-0249 'FIRST and 'LAST for null ranges are defined oddly

!problem

In Ada83, null ranges where 'LAST is less than 'FIRST-1 are awkward because 'LAST-'FIRST+1 does not equal 'LENGTH. In addition, they exact a significant run-time performance penalty, especially on machines that provide hardware index checking based on a lower bound and (non-negative) length.

!rationale

Null ranges where the specified 'LAST is less than 'FIRST-1 do not appear to be of any particular value. Such ranges can be handled much more efficiently by defining the "effective" 'LAST to be 'FIRST-1. An upward compatibility problem would exist if a program depended on the difference between 'FIRST and 'LAST of a null range, but such programs are hopefully extremely rare.

!appendix

%reference RR-0234

Null ranges are anomalous and cause significant overheads on machines with hardware indexing based on a lower bound and (non-negative) length.

%reference RR-0249

The definitions of 'FIRST, 'LAST, and 'LENGTH are inconsistent for null ranges, which leads to programming mistakes.

!rebuttal

The recommended language change is not upward compatible and there is no easy way to determine the number of existing programs that would be affected. Furthermore, there would be no warning except that at some point these programs could behave differently than they do now. Changing the language so that correctly functioning Ada83 programs behave in unpredictable ways without warning the programmer should not be tolerated. A safer revision would be to raise an exception (e.g., CONSTRAINT_ERROR) when elaborating a declaration where 'LAST < where the bounds are constants known at compile time), the program should be illegal. This would affect more programs but the effects would be more visible.

!topic Anonymous Types Definitions
!number RI-0200

!reference RR-0672
!reference RR-0617
!reference RR-0321
!reference RR-0336
!reference RR-0443
!reference AI-00538
!reference LSN.222
!reference NOTE-054
!reference NOTE-055
!reference NOTE-076

!problem

There are several inconsistencies in the use of anonymous types. For example, array and record constructs are not allowed in an object declaration nor as a component of an array or record. The fields of a record are not the same as object declarations since they cannot be declared as constrained array definitions. Anonymous types would eliminate the need to introduce extra type names when defining data structures.

!appendix

%reference RR-0672

Anonymous types would eliminate the need to introduce extra type names when defining data structures.

%reference RR-0321

There are several inconsistencies in the use of anonymous types. For example, array and record constructs are not allowed in an object declaration nor as a component of an array or record. The fields of a record are not the same as object declarations since they cannot be declared as constrained array definitions.

%reference RR-0617

Eliminate anonymous array types from the language; they encourage bad programming habits.

%reference RR-0336

Allow general type definitions in records that are variable in more than one dimensions:

```
type T (n: integer) is record
   D: array(1..n) of string(1..n);
end record;
```

%reference RR-0443

Allowing anonymous arrays in records would reduce the number of extra names that must be introduced. As a result of the current rules, there are several useful cases that cannot be expressed, and several anomalies that arise with respect to discriminated records and arrays.

%reference AI-00538

Unconstrained array definitions are not allowed in the syntax of constant array definitions; however, they are allowed if an anonymous type is not used for the definitions.

%reference LSN.222

Discusses the motivations for the 1981 definitions of Ada an anonymous types.

%reference NOTE-054

Implementing double dependency on discriminants is an expensive operations and is justification for eliminating double dependencies in record definitions (the Emilion proposal) such as:

```
    D: array(1..n) of string(1..n);
end record;
```

%reference NOTE-055

More justification for eliminating multiple dependencies in declarations (the Emilion proposal).

%reference NOTE-076

The terminology used in NOTE-054 and 055 is not correct; these are not double dependencies – what is being discussed is the use of discriminant in the component subtype of an (anonymous) array within a record. For example,

```
type text_array(i,j: integer) is
   record
          v:array(1..i) of text(max_len=>j);
       end record;
```

The only use is as a shorthand for giving a type declaration, and support type composition. If type composition is eliminated, then anonymous array types should also be eliminated.

!topic Dimensional mathematics.
!number RI-3749
!version 1.4
!tracking 5.3.11

DRAFT DRAFT DRAFT DRAFT DRAFT DRAFT

!Issue revision important,moderate impl,upward compat,somewhat consistent

[1] Ada9x should provide mechanisms that make it convenient for programmers to specify the dimensional units of quantities in their program in such a way that compilers will check them for consistency.

!reference RR-0354
!reference RR-0745
!reference

[1] PIWG ...

[2] Hilfinger, P. N. "An Ada package for dimensional analysis." TOPLAS 10, 2 (April, 1988), pp. 189-203.

[3] Gehani, N. "Ada's derived types and units of measure." Software Practice and Experience 15, 6 (June, 1985), pp. 555-569.

!problem

Ada83's type mechanism does not directly support the attribution of dimensional information to numerical quantities in a program. That is, there is no way to specify the units of measure for any given object or value, to check that expressions or assignments are dimensionally well-formed, or to scale quantities automatically, as dictated by their units of measure. Known methods for achieving at least some of these effects all suffer from various drawbacks.

Even before Ada83 was adopted, it was well-known that derived types provide no semblance of the desired functionality. The problem is that multiplication, division, and exponentiation of dimensioned quantities generally produce quantities with new dimensions. This does not fit well with the results of applying type derivation to numeric types.

One could define, once and for all, a package that contained types representing all combinations of dimensions programmers are likely to need, but the number of operations needed is very large, and the approach runs into considerable difficulty when applied to fixed-point types.

Hilfinger [2] has shown how to define a type QUANT that is abstractly a floating-point value whose discriminants encode its units of measure. His package provides for scaling and dimensional consistency checks. However, he has also pointed out several difficulties with his solution. First, there are problems in applying the approach to fixed-point types.

Second, the definition of subtypes of QUANT that represent new units is awkward. Third, the checks are, of necessity, written as execution-time checks. While they are of a kind that a compiler could easily check, this is not guaranteed.

!rationale

There have been numerous proposals for extending Ada to allow the addition of dimensional information. Gehani's [3] is a reasonable indication of how this might be done.

The moral of Hilfinger's work, however, is that Ada83 comes surprisingly close to supporting the feature already. This makes it difficult to justify a radical extension of the language.

!appendix

%reference RR-0354 Introduce dimensional mathematics into the language

This RR would like to be able to define physical data types in a manner that provides some level of type safety with respect to the operations on a type. His only complaint is that the only way in which to ensure that operations are limited to those which make sense (addition and subtraction) is to redefine the predefined operations inherited when the definition is provided by a derived type mechanism. He would like to be able to have this type security enforced by the compiler, rather than depend upon the programmer.

He would like to also have the ability to perform dimensionless scaling of these types.

A proposed solution would include "...a standard package defining all international (SI) physical data types and the allowed set of operations between them".

%reference RR-0745 Add facilities for dimensional mathematics to the language

This RR presents a language solution to providing protection from the limited amount of type security for derived types and the use of *, /, and ** operators.

%reference PIWG

The physics package in the PIWG suite contains a first approximation to a collection of routines intended to allow the development of programs with dimensional mathematics.

!rebuttal

5.4 Packages

RI-2105

!topic Miscellaneous
!number RI-2105
!version 1.1
!tracking 5.4.3

DRAFT DRAFT DRAFT DRAFT DRAFT DRAFT

!Issue out-of-scope (un-needed) (1) not defensible,small impl,bad compat,inconsistent

1. (Require extra specification for own variables) Ada9X shall require extra specification for the objects declared in library packages indicating that the objects are not deallocated during the execution of a program.

[Rationale] This idea is wholly upwards incompatible. It is likely to beak essentially every existing Ada program that uses a library package.

!problem

For some users, it is nonintuitive that the objects declared in library packages live for (essentially) the entire program execution but this is not so for library procedures and functions.

!rationale

!appendix

%reference RR-0271 Own variables in packages are inappropriate

RR-0271 wants wants is stated in RI-2105.1.

!rebuttal

For some users, it is nonintuitive that the objects declared in library packages live for (essentially) the entire program execution but this is not so for library procedures and functions.

5.5 Subprograms

RI-1000
RI-1022
RI-1021
RI-1023

!topic reading non-input parameters
!number RI-1000
!version 1.7
!tracking 5.5.1

!Issue revision (1) desirable,small impl,upward compat,consistent

1. Ada9X shall provide a mode of formal parameter that (1) allows reading and updating of the formal parameter and (2) does not imply that the actual parameter is an input to the subprogram.

!Issue out-of-scope (un-needed) (2) desirable,small impl,moderate compat,consistent

2. Ada9X shall require OUT-mode parameters of an access type to be initialized to NULL rather than being initialized with the value of the corresponding actual parameter.

[Rationale] This is primarily motivated by the view that [1] should be satisfied by allowing reading of OUT-mode parameters and that, for safety, OUT-mode parameters should be initialized like other objects. RR-0574 also points out this language change would eliminate a certain need for constraint checking that occurs after a subprogram call. In any event, the recommended language change is non-upward compatible. The benefit gained by such a change does not seem to outweigh this disadvantage.

!reference AI-00478
!reference AI-00479

!reference RR-0002
!reference RR-0303
!reference RR-0539
!reference RR-0559
!reference RR-0574

!problem

It is common in programming to want use a result produced by a subprogram inside that subprogram. Accomplishing this in Ada is unnecessarily error-prone or confusing for parameter results that are not also inputs to the subprogram.

!rationale

There are two ways in Ada83 to use (i.e., read) a subprogram parameter that serves as a result inside the subprogram. One is to use an OUT-mode parameter together with a local copy of the result that is assigned to the OUT-mode parameter before returning. This solution is error-prone and unnecessarily inefficient. The second is to use a parameter of mode IN OUT. This solution is confusing to the program reader in that it implies flow of useful data coming in to the subprogram.

A relatively simple and upward-compatible solution to this problem is to allow reading of OUT-mode parameters. Refer to RR-0002 for an analysis of such a language change. An unmentioned disadvantage of this approach is that it makes the wanted-it-IN-OUT-but-by-mistake-I-made-it-OUT error not detectable at compile time.

Of course, an alternate solution is a new parameter mode that carries the desired characteristics.

A parameter mode as described in the requirement would make the language no more "unsafe" in that the opportunities for reading uninitialized data are the same as for objects declared by a variable declaration. Similarly, the opportunities for erroneous execution by depending on a particular parameter passing mechanism are no greater than those associated with IN OUT parameters.

!appendix

Its been suggested the !problem beats around the bush and that the second sentence should read:

> Ada83's restriction on reading an OUT parameter after it has been assigned a value is more severe than necessary and workarounds are error-prone.

This change has not yet been made because it was felt to be too solution-oriented.

%reference AI-00478 Referring to out-mode formal parameters to be allowed
Allow reading of OUT-mode parameters.

%reference AI-00479
Access type out-variables should be null before call If allow reading of OUT-mode parameters, they should be initialized like other objects.

%reference RR-0002 Allow reading OUT parameters Allow reading of OUT-mode parameters.

%reference RR-0303 Allow reading OUT parameters Allow reading of OUT-mode parameters.

%reference RR-0539 Allow reading OUT parameters Allow reading of OUT-mode parameters.

%reference RR-0559 If allow reading of OUT parameters, initialize OUT access to NULL If allow reading of OUT-mode parameters, they should be initialized like other objects.

%reference RR-0574 Constraint-check elim. change requires change for OUT access params Some constraint checking code could be removed.

!rebuttal

Requirement [1] above should be dropped. Lets face it, adding a new parameter mode is not practical because it is too big of a change. Altering the language to allow reading OUT-mode parameters has one of three significant problems. Either (1) the rules concerning copy-in for OUT-mode access type parameters are changed, (2) it is defined to be erroneous to read an OUT-mode access type parameter before it is assigned a value in the subprogram, or (3) there is no difference between IN OUT and OUT parameters of an access type.

Possibility (1) is not upward compatible. Possibility (2) is silly because the result of reading the OUT-mode access-type parameter could be perfectly well defined (since it has a known value). Possibility (3) is an alarm indicating something is badly wrong.

All this for something with a "perceived user need" of 1!

!topic user control over parameter passing mechanism
!number RI-1022
!version 1.11
!tracking 5.5.1

DRAFT DRAFT DRAFT DRAFT DRAFT DRAFT

!Issue out-of-scope (un-needed) (1) desirable,moderate impl,upward compat,mostly consistent

1. Ada9X shall provide a facility for specifying the parameter-passing mechanism to be selected in situations where multiple such mechanisms are possible.

[Rationale] It seems dangerous to allow programmer control over the parameter-passing mechanism. There can only be two motivations for exerting such control: to make the logic of the program dependent on the parameter-passing mechanism or to improve performance.

Allowing logical dependence on the parameter-passing mechanism in a program is antithetical to the interests of the trusted-systems and safety-critical community, since the parameter-passing mechanism can make a difference only in programs where aliasing occurs or where OUT and IN OUT actual parameters are examined when an exception is raised within a subprogram. In such cases, the dependence on the parameter mechanism is subtle. Providing this control is likely to lead to less reliable, less understandable, more fragile Ada programs. For programs where the level of assurance of a formal proof is required, it seems reasonable to require that no aliasing occurs and that there is no dependence on subprogram OUT or IN OUT parameter values when an exception is raised. The proof can verify these characteristics and then proceed using proof rules that assume copy-in/copy-out for parameters.

Performance was clearly an issue when trying to develop real-time applications with early Ada compilers. Selecting the most efficient parameter-passing mechanism is a relatively easy calculation given knowledge of internal compiler mechanisms and the target machine architecture. Without this information it amounts to trial and error. It seems likely that the maturity of Ada compilers has solved (or will solve shortly) this problem.

Although it may be possible to satisfy this request with a relatively small language change, the logical dependence problem it would introduce seems to out-weigh any advantage it would provide in controlling performance.

!Issue out-of-scope (un-needed) (2) desirable,small impl,upward compat,inconsistent

2. Where Ada9X allows implementations a choice among multiple parameter-passing mechanisms, implementations shall be required to document which mechanisms they provide and under what circumstances these mechanisms are used.

[Rationale] This can only encourage the construction of non-portable Ada software. It is counterproductive to require documentation of those aspects of implementation behavior upon which the standard forbids the programmer to depend.

!Issue revision (3) important,severe impl,upward compat,consistent

3. The defined meaning of parameter passing in Ada9X shall be bounded in all cases by the individual meanings associated with the allowed parameter-passing mechanisms. In particular, an execution of an Ada9X program shall not be totally unpredictable because the program behaves differently for different choices of parameter-passing mechanisms.

!reference WI-0103
!reference WI-0308

!reference AI-00349

!reference

Trusted System and Safety Critical Ada9X Workshop; Radisson Hotel, Falls Church, VA; January 25-26, 1990.

!problem

There are three aspects to this problem. First, for a parameter of a composite type, the parameter-passing mechanism used is left to the implementation and therefore there is no way for the programmer to control which of these mechanisms is used in situations where this is of concern to him. Secondly, where implementations are allowed choices in parameter passing mechanisms, there is no requirement on the vendor to document its strategy on passing parameters. Finally, certain executions of Ada83 programs are technically undefined due to their dependence on an implementation- selected parameter passing mechanism, when, in practice, the set of meanings of these programs is quite bounded.

!rationale

That reliance (for effect) on a particular parameter-passing mechanism for composites makes program execution erroneous seems unnecessarily harsh. It appears that in practice, rather than being completely "unpredictable", such programs have a well-bounded number of potential meanings.

!appendix

%reference WI-0103 Allow user control of subprog parameter passing techniques

"Nondeterminism becomes a problem when the number of possible different results ... grows too large to reason about". "This change would allow programmers to document, in a formal way, program dependencies on particular implementation techniques."

%reference WI-0308 Allow user control of subprog parameter passing techniques

Need more control over parameter passing to, e.g., interface to foreign code, control parameter results when subprogram is abandoned due to exception, operations on memory-mapped objects, prohibit aliasing, interface to atomic update operations such as test-and-set.

%reference AI-00349 Delete copy-in/copy-back for scalar and access parameters

Get rid of language requirement that says scalars and access type parameters must be passed by copy.

!rebuttal

It is important for the user to have control over the parameter- passing mechanism used when a choice is left to the compiler.

The "application problem" for embedded system integrators is to get a working system out the door with a given set of hardware which usually by definition is barely adequate to do the job. The implication that such a user's problem domain is independent from the language designer's, and the compiler writer's, is wrong, as anyone who has implemented Ada programs on a distributed real-time system can relate.

Quite the contrary to to the feeling expressed in the body of this RI, giving the user control over his environment makes tremendous sense. In the problem addressed herein, for instance, I am unable to make what is really a system design decision for all the Ada community on how parameters should be passed. However, I do think I am capable of making a system design decision for my own system, and would welcome the ability to specify that a large object should be passed by reference, instead of by copy, in a time-critical section of code.

In practice, of course, the embedded systems integrator does what is necessary to get his project to work; he perforce must do it in a non-portable way either with his compiler vendor or on his own, as things stand now.

I agree with the assumption that not all, indeed few, users should exercise this kind of control. However, I do not agree that the solution is to deny all users language support for control over their environment. A consequence of this attitude is to cause those who must

do such things to do them outside of the language; portability and reuse suffer, and life cycle costs inevitably rise.

It is also important that where choices for things such as parameter passing mechanisms are left to the compiler, the compiler vendor should document his approach.

It is perfectly legitimate for a user to try to maximize performance if this is his primary concern. There is an implicit judgment here that enhancing reuse takes precedence over performance.

There is enough variation between ACS vendors anyway that performance of reused code is likely to be very non-portable. At least let the code perform well in its original setting!

I prefer to think of Item [2], above, as allowing programs whose primary concern is performance as opposed to reuse/portability to know what is happening to them without the agony of surprises in their maintenance or development.

Maybe this is a candidate for the real-time appendix/section/whatever.

!rebuttal

The rationale for out-of-scope item [1] above misses a third reason for the programmer wanting control over the parameter transmission mechanism. That is, when passing a shared (among tasks) object as a parameter where it is semantically relevant whether the subprogram operates directly on the object or an a copy. In such cases it can be necessary (in terms of the correctness of the program) to force the implementation to pass the parameter by reference. Also, strictly speaking, it may be necessary to improve the functionality of pragma SHARED for this to work.

!topic pragma inline inflexibility
!number RI-1021
!version 1.6
!tracking 5.5.3

DRAFT DRAFT DRAFT DRAFT DRAFT DRAFT

!Issue revision (1) desirable,moderate impl,upward compat,consistent

1. (Inline On Subprogram Basis) Ada9X shall allow request for inline implementation on a subprogram basis, not simply on the basis of a subprogram designator which could apply to multiple subprograms.

!Issue revision (2) desirable,moderate impl,upward compat,consistent

2. (Inline On Subprogram Call Basis) Ada9X shall allow selective control over which calls to a particular subprogram get inlined. [Note: it is acceptable if inlined calls must make use of a different designator for the subprogram than non-inlined calls.]

!reference RR-0060
!reference RR-0398
!reference RR-0575
!reference RR-0687

!problem

Two problems exist with respect to pragma INLINE in Ada83. One is the lack of control for selective inlining among a set of subprograms with the same designator declared in the same declarative part (or package specification). The other is lack of control for selective inlining among a set of calls of the same subprogram.

!rationale

It is pointed out in RR-0687 that the present pragma INLINE mechanism can lead to unexpected and undesirable results when a new (perhaps large) subprogram is added that gets unintentionally inlined due to a previously existing pragma INLINE. This problem can be solved by requiring some sort of tight coupling (physically) between the specification of a subprogram and a pragma INLINE that applies to it. This approach is in some sense not upward compatible (the set of subprograms to which a pragma INLINE applies would not be the same). An alternative is to add (perhaps optional) parameter and result type profile information to pragma INLINE.

Concerning the second problem discussed above, it is true that "where" a subprogram call takes place is often more important than "which" subprogram is called in determining space/time tradeoffs. It might be perfectly reasonable to trade space for time inside, e.g., several nested loops or inside an interrupt entry, while not being willing to make the same trade elsewhere. Potential language solutions to this problem range from allowing a new form of pragma in the calling code to allowing an Ada83 pragma INLINE to apply to

a RENAME of a subprogram (without affecting calls to the original).

!appendix

A related issue here is that an implementation is allowed to ignore a perfectly valid pragma INLINE and not give a warning or error to the user. This issue is scheduled for a separate RI in the 1K series.

%reference RR-0060 Allow selective inlining of subprograms

Need to decide whether to inline on a per-call basis

%reference RR-0398 Need clearer/more selective rules for pragma inline applicability

For overloaded subprograms, need to allow specification of inline based on parameter-result type profile (or similar)

%reference RR-0575 Need better (more selective) control over inlining

Pragma INLINE applied to a subprogram should be able to work the problem of controlling which calls to the subprogram get inlined, e.g., procedure Inlined_Foo (X : Integer) renames Foo;

%reference RR-0687 Pragma inline should not apply to all overloads; only closest

Inserting a large subprogram with an overloaded name can be bad news because it may be mistakenly inlined.

!rebuttal

!topic Subprogram Issues
!number RI-1023

!reference RR-0269
!reference RR-0598

!problem

The problems covered herein consist of

1. Typically, an embedded system does not make use of subprogram recursion. Recursion is seen by many embedded systems developers to be inappropriate for these applications due to the possibility of STORAGE_ERROR from stack overflow. These programmers accordingly use programming standards that prohibit the use of recursion. Perhaps this prohibition should be included in the Ada itself, specifically disallowing recursion unless explicitly asked for by the programmer.

and

2. It is often desirable to return from a function two results, e.g., a data item as well as status information. Functions are needed here because of their ability to return objects of unconstrained types. All known workarounds for this kind of problem are inconvenient and/or error prone.

!appendix

%ref RR-0269Make subprograms not recursive by default

%ref RR-0598Need functions which return more than one parameter, eg. status

5.6 Input/Output

RI-3700
RI-3600

!topic I/O and External Environments
!number RI-3700
!version 1.3
!tracking 5.6.1.1, 5.6.1.2, 5.6.1.3, 5.6.3, 5.6.5

[Note: See also the companion RI-3600 on The I/O Abstraction.]

!Issue revision (1) important,small impl,upward compat,consistent

1. (Incomplete Functionality) Ada 9X shall provide additional functionality in the interface to the external environment. In particular, the areas to be considered for addition include:

a. A mechanism for appending to an existing file.

b. A mechanism for determining the existence of a file.

c. A mechanism for renaming an existing file (regardless of contents).

[Note: Item (c) may be unachievable in some environments in which all other aspects of a file system can be accomplished.]

!Issue secondary standard (2) important

2. (Terminal I/O) The standard shall provide an interface for the manipulation of data appropriate to a terminal interface. The functionality of such a standard shall include: a mechanism to begin/end interactive I/O, terminal functions such as cursor addressing and keystroke identification, etc.

!Issue secondary standard (3) important

3. (Business Application I/O) The standard shall specify additional functionality to support business applications. Some of the functionality of such a standard shall include:

a. indexed sequential I/O.
b. file and record locking capabilities.

!Issue administrative (4) compelling

4. (Validation of Compilation Systems Without I/O) The procedures for validating systems that do not provide an underlying operating system and I/O should be rewritten to take this into account. [Note: this may require the GET & PUT functions that currently operate between strings and the various numeric and enumerated types be considered for inclusion outside of the I/O packages.]

!Issue out-of-scope (un-needed) (5) desirable,unknown impl,upward compat,unknown compat

5. (Stream I/O) Ada 9X shall provide primitives for manipulation of stream-based I/O.

[Rationale: The issues here seem to be with providing a language mechanism to support the synchronization and buffering of (particularly) input from an external device. Yet the implementation appears to be so specific to the particular application and data type in the stream that the application would need more control than a simple abstraction would be able to provide.]

!Issue out-of-scope (un-needed) (6) desirable,unknown impl,upward compat,unknown compat

6. (Mandate Variant Record I/O) Ada 9X shall mandate the implementation of variant record I/O in DIRECT_IO and SEQUENTIAL_IO.

!Issue out-of-scope (un-needed) (7) desirable,small impl,very bad comp,inconsistent

7. (I/O as a Secondary Standard) All I/O shall be moved to a secondary standard in Ada 9X.

!reference AI-00329 look-ahead operation for TEXT_IO
!reference AI-00544 File "append" capability proposed
!reference AI-00545 Procedure to find if a file exists
!reference RI-0005 Asynchronous Communication
!reference RI-3600 The I/O Abstraction
!reference RR-0077 Provide stream I/O
!reference RR-0089 Provide facilities for interactive I/O
!reference RR-0146 Support for file/record locking
!reference RR-0147 Support for Index-sequential access
!reference RR-0149 Provide a keyboard input/output package
!reference RR-0159 Add pre-defined package of general file system functions
!reference RR-0207 Add Text_IO support with Exists function and Append procedure
!reference RR-0235 Need support for interactive terminal input/output
!reference RR-0294 I/o as presently defined is inappropriate for embedded systems
!reference RR-0297 Low_LEVEL_IO was a bad idea, remove this package
!reference RR-0382 Need to be able to rename and append to a file in standard Ada
!reference RR-0404 Need convenient way to find out if a particular file exits
!reference RR-0405 Need convenient way to append to a file
!reference RR-0420 Need file "extend" or "append" capability
!reference RR-0593 Mandate impl. of variant record i/o in Direct_IO/Sequential_IO
!reference

Nyberg, Karl A., A Study of Ada's Input/Output Packages, SEI Complex Issues Study SEI-SR-90-XXX, to appear.

!problem

Input and Output facilities in Ada83 have a number of missing capabilities with respect to external environments. These should be cleaned up in the revision.

In addition, a number of features could easily be provided via secondary standards to make Ada more palatable to some particular application. communities.

Finally some policy issues need to be addressed. These include how to deal in validation with systems, particularly embedded ones, that don't support I/O.

!rationale

[3700.1]

These are some missing features that are annoying to programmers. Providing these additions during the revision phase seems the appropriate approach.

!appendix

%reference AI-00544 File "append" capability proposed

This study AI indicates a desire to be able to specify a mode of OPEN that will allow append. Current thinking within the ARG appears to allow a band-aid solution that will allow an implementation to provide this functionality via FORM parameters. This should probably be provided in the language with 9X.

%reference AI-00545 Procedure to find if a file exists

This study requests a function to indicate whether a file with the given name exists in the file system. The current workaround almost accomplishes the needed functionality through the use of exceptions raised upon attempting to open the file, but can possibly change access dates, etc. as a result, and may give an incorrect message when the file exists, but may not be readable.

%reference RI-0005 Asynchronous Communication

This RI deals with the specific issues of Asynchronous communication.

%reference RI-3600 The I/O Abstraction.

This RI deals with the I/O abstraction.

%reference RR-0077 Provide stream I/O

Embedded applications often have to deal with streams of input and output, particularly in signal processing applications. Ada does not currently provide a mechanism for synchronizing such entities. (This might be covered under LOW_LEVEL_IO?)

%reference RR-0089 Provide facilities for interactive I/O

This RR wants the ability to provide a terminal interface with various screen operations. Character-at-a-time I/O is necessary to accomplish this result. A package of interface functions could otherwise be written to provide the necessary functions for screen manipulation.

%reference RR-0146 Support for file/record locking

This RR is asking for a specific language feature. It does not provide any applications for which the feature is required and the lack of which causes Ada to be unusable. It requests that the file/record locking be performed on a language basis, but the implementation will have to be done on a system wide basis. This seems an appropriate issue for an implementation-defined FORM parameter, possibly standardized in one manner or another.

%reference RR-0147 Support for Index-sequential access

Business programmers want to be able to use ISAM (Indexed Sequential Access Method). This looks likes a reasonable item for a secondary standard.

%reference RR-0149 Provide a keyboard input/output package

This RR asks for something called keyboard I/O without describing WHAT the functionality of such a package is. It could be that what is represented as necessary is a package to support terminal interactions based upon different terminal configurations. If it's really that big of an issue, it could be made into a secondary standard. A number of companies in the marketplace are already marketing packages to support these capabilities using curses, X windows, etc.

%reference RR-0159 Add pre-defined package of general file system functions

Consider other potential aspects of a file besides open, close, etc., possibly including existence, permission, and name.

%reference RR-0207 Add Text_IO support with Exists function and Append procedure

Same as AI-00544 & AI-00545.

%reference RR-0235 Need support for interactive terminal input/output

Identical to rr-0089.

%reference RR-0294 I/o as presently defined is inappropriate for embedded systems

The current I/O definition is geared to the more general operating system environment and is insufficient to meet the needs of embedded systems developers. One complaint is that there is too much unnecessary functionality and another is that it can't meet the needs for high speed devices, memory mapped IO, etc.

%reference RR-0297 Low_LEVEL_IO was a bad idea, remove this package

This RR wants LOW_LEVEL_IO to be removed because of its lack of utility and potential for precluding other solutions.

%reference RR-0382 Need to be able to rename and append to a file in

The functionality of rename and append are desired for files. Note that rename doesn't care what type of data is contained in the file. It is really an operation on the NAME of the file.

%reference RR-0404 Need convenient way to find out if a particular file exits

Same as AI-00544.

%reference RR-0405 Need convenient way to append to a file

Same as AI-00545.

%reference RR-0420 Need file "extend" or "append" capability

Need a file extension capability.

%reference RR-0593 Mandate impl. of variant record i/o in Direct/Sequential_IO

Two issues here. interoperability of data and uniformity of file operations across implementations. Apparently some implementations require additional form parameters for variant record I/O, which leads to non-portable code. Certainly the format for storing variant records in a file may be different between vendors.

%reference WI-0504 Improve/extend I/O to include required capabilities

The required capabilities include stream I/O, indexed I/O, and terminal I/O. A recommendation to move I/O to a secondary standard.

!rebuttal

!topic The I/O Abstraction.
!number RI-3600
!version 1.3
!tracking 5.6.2.1, 5.6.2.2, 5.6.4

[Note: See also the companion RI-3700 on I/O and External Environments.]

!Issue revision (1) important,small impl,moderate compat,consistent

1. (Remove I/O Anomalies) Ada 9X shall remove the anomalies of I/O. Some particular items include:

a. The semantics of end_of_page and end_of_line, and their relationship with skip_line, get_line and get. [Note: This area should be made more intuitively understandable and consistent for both interactive and non-interactive situations.]

b. Provide a mechanism for saving and restoring the standard input and output.

c. Remove the restriction on independence of standard input and output to allow better control over formatting.

d. Allowing per-instantiation DEFAULT values to be formal parameters (e.g., FORE, AFT, WIDTH).

e. Allow real number output in non-decimal bases.

f. Provide a function to determine line length.

g. Allow data of mode IN for SEND_CONTROL.

!Issue revision (2) important,severe impl,upward compat,consistent

2. (I/O is Multi-tasking) Ada9x shall disallow task starvation to occur in instances where one task executes an I/O instruction.

!Issue secondary standard (3) desirable

3. (Business Numeric Formatting Capabilities) The standard shall provide facilities for converting numeric values to strings with formats similar to those commonly found in other business processing languages. [Note: this may actually be a portability issue at heart.]

!reference AI-00003 Allow DATA of mode "in" in SEND_CONTROL (in Low_Level_IO)
!reference AI-00329 look-ahead operation for TEXT_IO
!reference AI-00485 Must standard input and output files be independent?
!reference AI-00487 The TEXT_IO procedures end_of_page and end_of_file
!reference AI-00488 Skipping of leading line terminators in get routines
!reference RI-3700 I/O and External Environments.
!reference RR-0047 Make put and get functions instead of procedures
!reference RR-0127 Allow real number output in non-decimal bases

!reference RR-0130 Replace DEFAULT_xy variables with functions
!reference RR-0164 Provide multitasking i/o in text_io
!reference RR-0359 Allow "mixed case" output for enumeration literals
!reference RR-0360 Add "picture-formatting" capabilities to text_io
!reference RR-0361 Improve formatting capabilities for integer i/o
!reference RR-0447 Need to be able to preserve/restore the default file
!reference RR-0484 Add DEFAULT_xy functionality as parameters to generic textio
!reference RR-0485 Provide means to get the line length of an I/O device
!reference RR-0551 Need assignment capability for Text_IO.File_Type
!reference RR-0552 Need "padded" line input with truncation and pad-fill
!reference RR-0553 Text_IO.Get_Line is difficult due to sometimes use of Skip_Line
!reference RR-0597 Need functional version of Get_Line instead of procedural
!reference RR-0711 I/O by a task in multi-task should not block whole pgm
!reference WI-0504 Improve/extend I/O to include required capabilities
!reference

Nyberg, Karl A., A Study of Ada's Input/Output Packages, SEI Complex Issues Study SEI-SR-90-XXX, to appear.

!problem

Input/Output as defined in Ada lacks a number of features and has a few whose use is annoying. These need to be cleaned up to make the language more usable.

!rationale

Some of these issues are currently perceived as impediments to the use of Ada in certain application environments. To the extent that a secondary standard can meet that need, such an approach is preferable to a language change.

[3600.1.a]

The semantics of the various line, page and file markers is one area that causes implementations the greatest amount of difficulty and confuses users of the language because of the non-intuitive meanings. This issue is included as a requirement rather than as a presentation issue because it may require more than just additional textual clarifications to resolve the matter.

[3600.1.b]

The Rationale for the Ada Programming Language says that being able to do this is one reason why the current text_io definition approach was provided, but it actually precludes such usage.

[3600.1.c]

This restriction seems antithetical to the use of even simple minded modern-day interactive terminals.

[3600.1.d]

This is a simple improvement that would allow default values to be based upon the type being instantiated rather than completely implementation dependent.

[3600.1.e]

A minor inconvenience. Relatively easily remedied.

[3600.1.f]

The workaround is available through run-time execution of a program, while the value should reasonably be expected to be availbe from a function.

[3600.1.g]

The current use of IN OUT violates the semantics of a SEND_CONTROL and also forces constants to be wrapped in a dummy variable.

[3600.2]

[Note: It was felt that this particular item is sufficiently different from the other ones, in scope and implementability (there may be some systems for which it is impossible to meet this requirement) that it should be separated from the others to get additional input.]

[3600.3]

The use of Ada in business/commercial applications is currently limited in part due to the perceived inability to provide the kind of formatting capabilities present in other languages with respect to display of currency values. There is no reason to expect that such a standard could not be defined specifically for such applications outside of the language.

!appendix

%reference AI-00003 Allow DATA of mode "in" in SEND_CONTROL (in Low_Level_IO)

This study AI indicates that the second argument of SEND_CONTROL is often a constant, when a variable is required. A current trivial workaround exists by creating a dummy variable to hold the value. Given the semantics of a SEND operation, the parameter should be of mode IN.

%reference AI-00329 look-ahead operation for TEXT_IO

This study AI indicates that the user would like to be able to do look-ahead in order to write his own I/O get routines. There seem to be no reasons why look-ahead is necessary to accomplish this.

%reference AI-00485 Must standard input and output files be independent?

This study AI contends that the validation suite tests to ensure that standard input and output are independent of one another are unnatural to the way in which it might be desired in some systems.

%reference AI-00487 The TEXT_IO procedures end_of_page and end_of_file

This study AI notes that end_of_page and end_of_file return true even if there is still an empty line to be read on the page or in the file.

%reference AI-00488 Skipping of leading line terminators in get routines

This study AI notes that the skipping of leading line terminators in the definition of the get routines precludes the use of these routines to obtain default values, rather requiring the use of strings and conversions.

%reference RI-3700 I/O and External Environments.

Primarily issues of the I/O interface with underlying system.

%reference RR-0047 Make put and get functions instead of procedures

This RR wants to have GET and PUT as functions instead of procedures in order to not have to waste space in the converting string or pre-calculate the correct size for the string.

%reference RR-0127 Allow real number output in non-decimal bases

This is a minor inconsistency where input in non-decimal bases is allowed for real numbers, but not output.

%reference RR-0130 Replace DEFAULT_xy variables with functions

The values used for DEFAULT in the generic IO packages cannot be provided at instantiation time. This means that all instantiations must use the same set of defaults and then change them via assignments to the default variables. Promoting them to generic formal parameters can alleviate this situation by allowing them to be a part of the instantiation.

%reference RR-0164 Provide multitasking i/o in text_io

The need is to not have a task that performs I/O cause the rest of the runtime system to block.

%reference RR-0359 Allow "mixed case" output for enumeration literals

Only lower_case and UPPER_CASE are allowed as elements of the enumerated type for which enumerated types may be displayed when an instance of enumeration_io is made. Initial capitalization might be another choice.

%reference RR-0360 Add "picture-formatting" capabilities to text_io

Text_io does not provide any mechanism to specify input and output conversions a la PL/1 & COBOL.

%reference RR-0361 Improve formatting capabilities for integer i/o

Output formats for numeric values are limited in functionality.

%reference RR-0447 Need to be able to preserve/restore the default file

<<This is the limited private issue.>>

%reference RR-0484 Add DEFAULT_xy functionality as parameters to generic textio

Same issue as RR-0130.

%reference RR-0485 Provide means to get the line length of an I/O device

The only way to currently get the line length of a device is to increase the argument to set_line_length until an error is raised.

%reference RR-0551 Need assignment capability for Text_IO.File_Type

<<This is the limited private issue.>>

%reference RR-0552 Need "padded" line input with truncation and pad-fill

This RR wants get to pad unfilled characters.

%reference RR-0553 Text_IO.Get_Line is difficult due to sometimes use of Skip_Line

The semantics of get_line and skip_line are inconsistent and confusing.

%reference RR-0597 Need functional version of Get_Line instead of procedural

Need get_line as a function because of the added inefficiencies of creating a buffer of a sufficiently large size and the necessity to maintain a length pointer in the current approach.

%reference RR-0711 I/O by a task in multi-task should not block whole pgm

Equivalent to RR-0164.

%reference WI-0504 Improve/extend I/O to include required capabilities

APPEND, FILE_EXISTS, etc. were to be added.

!rebuttal

5.7 Visibility

RI-2023
RI-2024
RI-2025
RI-2027
RI-2500

!topic Visibility of operators in a WITHed package
!number RI-2023
!version 1.8
!tracking 5.7.1.1

DRAFT DRAFT DRAFT DRAFT DRAFT DRAFT

!Issue revision (1) important,moderate impl,upward compat,consistent

1. (Automatic Visibility of Derivables) For Ada9X, an attempt shall be made to provide automatic visibility of certain derivable operations (specifically to include (1) the predefined infix operations and (2) the enumeration/character literals) of a type in any scope where an object of the type can be declared. The language shall define which of the derivable operations of the type are made visible in this way and how such operations are used in overload resolution.

!Issue revision (2) important,small impl,upward compat,consistent

2. (Visibility for Derivables Only) Lacking automatic visibility of derivables, Ada9X shall provide a mechanism to establish direct visibility of certain derivable operations (specifically to include (1) the predefined infix operations and (2) the enumeration/character literals) of a type without importing all declarations in the scope where the type is declared and without explicitly mentioning each imported operation.

!reference RR-0022
!reference RR-0057
!reference RR-0232
!reference RR-0393
!reference RR-0555
!reference RR-0727
!reference RR-0429
!reference RR-0474
!reference RR-0624
!reference RR-0694
!reference AI-00480

!problem

In Ada83, it is frequently inconvenient to get the operators for a type even if the type is visible. Employing a USE-clause frequently works but also frequently sacrifices maintainability as it makes more visible than is desired. Also, many programmers expect that the infix operators and enumeration literals will be visible wherever the type is visible when, of course, they are not. The method of renaming often works but also sacrifices maintainability as the maintenance of the lists of renaming declarations can become a significant headache. In addition, the renaming method does not always work because there is no way to rename a character literal. A workaround is to put each type in its own "envelope" package; the type would be imported by USEing the envelope package. Essentially this allows/requires the programmer to set up his own restrictions on USE.

!rationale

Lacking appropriate mechanisms to selectively import operations from a package, users are essentially encouraged to insert use clauses when a narrower import would be more appropriate. The best situation would be that achieving visibility of those operations that are very closely coupled with a type could be done explicitly importing them at all. Failing this, a mechanism should be provided to get visibility of just these and no others.

!appendix

%reference RR-0022 Visibility of basic operators in another package is needed
%reference RR-0057 Need direct visibility of infix operators in another package
%reference RR-0232 Need better control over visibility (esp. operators) from packages
%reference RR-0393 Visibility of predefined fixed-point "/" hard (can't RENAME)
%reference RR-0555 Need "selective" use clause to just get operators, etc.
%reference RR-0727 Need component-specific USE clauses

RR-0022, RR-0057, RR-0232, RR-393, RR-555, and RR-727 suggest that a capability if needed to get the operations associated with a type imported by another package; use clause is said to make too much visible. Prevalently mentioned is a scheme by which a limited use clause would make d.visible all operations associated with a specific type. RR-0232 suggests a template-matching scheme. RR-0393 points out a quirk with fixed-point "/" because it cannot be renamed. RR-0555 and RR-0727 like the Modula-2 style of importing.

%reference RR-0429 Need USE-like scheme to make only overloadables visible

RR-0429 suggests the capability to get the operations associated with an important type. Also suggested is a capability for importing all of the overloadable declarations from another package.

%reference RR-0474 Need more selective USE-clause to get just certain operators, etc.

RR-0474 suggests the need for more selective USE-clauses; the idea of importation-by-renaming is mentioned.

%reference RR-0624 Need more selective USE capability

RR-0624 want more selective USEing; an important concept introduced here is that the effect of multiple such clauses should be additive, i.e. bringing in more stuff, not covering up the same declaration previously made visible.

%reference RR-0694 Need easy visibility over "=" from pkg; should it be a basic op.?

RR-0694 wonders is "=" shouldn't be visible whenever a type is visible; the renaming solution is considered ugly.

%reference AI-00480 Visibility of predefined operators with derived types see RR-0022 above.

!rebuttal

!topic Visibility and Safety
!number RI-2024

!problem

The problem here is mainly dealing with the maintenance consequences of the visibility rules of Ada83 for overloading, hiding, and open scoping. First, it should not be required that a client who needs read access to a certain object also have write access; indeed, really safe programming demands that write access be generally restricted. Second, maintenance problems are often created when a maintenance programmer adds declarations in a "middle" scope that change the imported declarations for an "inner" scope.

!appendix

%reference RR-0270 Allow specification of read-only package usage
RR-0270 suggests that read-only constraints should be supported for package access.

%reference RR-0588 USE clause is confusing and leads to maintenance problems
RR-0588 suggests the maintenance problem that occurs when a declaration USEd by an enclosed unit is overridden by a directly declared declaration during maintenance. This actually occurs for any overridden declaration whether overridden or not. The suggestion is for a use clause that overrides direct declarations.

%reference RR-0589 Need stronger kind of USE for less dependence on containing scope
RR-0589 discusses the same problem as RR-0588 except that the solution is an the form of an enveloping use-clause at the context level that would override direct declarations.

%reference RR-0266 Operator overloading is dangerous
RR-0266 suggests that overloading and hiding can be very dangerous and suggests that capabilities be added to restrict overloading and hiding within inner scope. Suggests that more infix symbols might reduce the desire for overloading.

!rebuttal

RI-2025

!topic Less Restrictions on Overloading; Enhanced Resolution
!number RI-2025
!version 1.4
!tracking 5.7.2.1 5.7.2.2 5.7.5

DRAFT DRAFT DRAFT DRAFT DRAFT DRAFT

!Issue out-of-scope (un-needed) (1) desirable,severe impl,bad compat,mostly consistent

1. (Parameter Names for Overloading) Ada9X shall allow subprogram names to overload one another if the names of the formal parameters are different, even if the parameter/result profiles are the same.

[Rationale] While this is a desirable idea, it is not upward compatible and relatively difficult to implement given the ripple effect of overload resolution in the language. Further, under this scenario all positional calls to such a subprogram would always be ambiguous so more text would have to be provided to disambiguate the call.

!Issue out-of-scope (un-needed) (2) desirable,severe impl,upward compat,mostly consistent

2. (Generic Formal Names for Overloading) Ada9X shall allow generic unit names to overload one another whenever the profile of the generic formal parameters is different or the generic formal parameter names are different. Also, Ada9X shall perform overload resolution on generic unit names using both the profile of the generic formal parameters and also any generic formal parameters names mentioned in a generic_actual_part to be used for overload resolution.

[Rationale] The rules for overload resolution are quite complex and the interaction with the rest of the language is very subtle. The need for the feature does not justify making the rules more complex.

!Issue out-of-scope (un-needed) (3) desirable,severe impl,upward compat,mostly consistent

3. (Trailing Attributes for Overloading) Ada9X shall use an attribute to assist in the resolution of the prefix to which the attribute is applied.

[Rationale] The rules for overload resolution are quite complex and the interaction with the rest of the language is very subtle. The need for the feature does not justify making the rules more complex.

!Issue out-of-scope (un-needed) (4) desirable,small impl,upward compat,consistent

4. (Additional Infix Operators) Ada9X shall use introduce a number of additional infix operators that would be eligible for overloading.

!Issue out-of-scope (un-needed) (5) desirable,moderate impl,upward compat,mostly consistent

5. (Overloading of Indexing) Ada9X shall use allow an indexing operation for a limited, private type to be overloaded on "()".

!reference RR-0035
!reference RR-0606
!reference REFER ALSO TO RR-0041
!reference REFER ALSO TO RR-0607
!reference RR-0201
!reference RR-0266
!reference RR-0395
!reference RR-0600
!reference RR-0663
!reference RR-0682
!reference RR-0131
!reference AI-00529

!problem

The problem is that users would like for overload resolution to do more.

!rationale

!appendix

%reference RR-0035 Allow generic units to be overloaded

RR-0035 wants to allow generic units to be overloaded whenever the generic formal profile is different.

%reference RR-0606 Allow generic subprogram names to be overloaded

RR-0606 wants generic subprogram names to be overloadable if the generic formal profiles are different.

%reference REFER ALSO TO RR-0041 (5.12.2)

RR-0041 wants to remove the restriction that names of overloadables must be different is they share their library package ancestor.

%reference REFER ALSO TO RR-0607 (5.12.2)

RR-0607 wants to be able to have overloading for library-level entities.

%reference RR-0201 Liberalize overloading of operators to other character sequences

RR-0201 wants two items: (1) to define := for limited private types, and (2) to introduce more infix operators into the language that would then be eligible for overloading. Only (2) is covered here.

%reference RR-0266 Operator overloading is dangerous

RR-0266 suggests that overloading and hiding can be very dangerous and suggests that capabilities be added to restrict overloading and hiding within inner scope. Suggests that more infix symbols might reduce the desire for overloading.

%reference RR-0395 Include formal parameter names in parameter/result-type profile

%reference RR-0600 Allow formal parameter names & other stuff to resolve overloading

RR-395 and RR-0600 wants to be able to overload procedure names with different parameter names even if the parameter/result profile matches, and to be able to disambiguate based on supplying named associations.

%reference RR-0663 Allow certain overloading of ":=" and "()"

RR-0663 wants to allow overloading of := and (), i.e. indexing, for limited private types. ":=" is not covered here.

%reference RR-0682 Allow user-defined overloaded operators such as "?", ":-", etc.

RR-0682 wants extra infix operators that can be overloaded.

%reference RR-0131 Alter the visibility inside a qualified expression based on prefix

RR-0131 wants the resolution rules to be changed so that the prefix of a qualified expression can be used to aid in the resolution of the expression.

%reference AI-00529 Resolving the meaning of an attribute name

AI-529 points out a situation where only one resolution of a name is meaningful to the user but the language rules do not pick up enough context (i.e. from an attribute) to succeed in resolution.

!rebuttal

!topic Unexpected Consequences of Visibility Rules
!number RI-2027
!version 1.4
!tracking 5.7.2.1 5.7.4 5.7.5

!Issue revision (1) desirable,moderate impl,upward compat,consistent

1. (Unexpected Results of Visibility Rules) In Ada9X, an attempt shall be made to eliminate unexpected restrictions in visibility rules.

!Issue presentation (2) compelling

2. (Visibility in Accept Statements) The standard shall describe precisely name visibility within accept statements.

!reference RR-0476
!reference RR-0462
!reference RR-0579
!reference RR-0675
!reference RR-0483
!reference AI-00593

!problem

For large projects, the overloading and visibility aspects of Ada are among its greatest strengths. However, certain consequences of the visibility rules are difficult or cumbersome to work around. For example, Ada83 specifies (ALRM 8.3(16)) that, within the specification of a subprogram or a generic subprogram, all declarations of the same designator as the subprogram be hidden. (This can usually be gotten around by renaming.) In addition, Ada83 specifies that a function and a type conversion cannot overload one another in the same scope; this gives conversion functions for user-defined types a second-class status within the language.

Of course, the rules for overload resolution are quite complex and the interaction with the rest of the language is very subtle. The need for these particular features does not justify making the rules more complex; however, the rules might be simplified.

!rationale

The rules for visibility and overload resolution should be examined for possible simplification.

!appendix

%reference RR-0476 Allow user-written type-conv. fns. with same name as target type

RR-0476 shows another problem with hiding all visible declarations of the same designator in a subprogram specification.

%reference RR-0462 Eliminate weird visibility rule in 8.3 (16) on subpgm formal parts

%reference RR-0579 Fix disallowance of ref to subpgm identifier in its specification

%reference RR-0675 Fix disallowance of ref to subpgm identifier in its specification

%reference RR-0483 Allow instantiated subpgm to have same id as generic unit (as pkgs)

RR-0462, RR-0579, RR-0675 want the rule to be reworked that requires ALL declarations to an designator to be hidden in a specification of a subprogram with the same designator. RR-0483 shows the same problem with generics.

%reference AI-00593 Visibility of accept statements

AI-593 points out that the language used to discuss the (extended) scope of the name of an entry (specifically, an accept part) in the LRM is technically imprecise.

!rebuttal

!topic Visibility/Private/Hidden/Protected
!number RI-2500
!version 1.2
!tracking 5.7.1.3

DRAFT DRAFT DRAFT DRAFT DRAFT DRAFT

!Issue revision (1) compelling,moderate impl,upward compat,consistent

1. (Increased Flexibility of Privating) An attempt shall be made to increase the ability of an Ada9X programmer to provide declarations that are fully usable within the package containing the declaration but not visible outside the package. [Note: some examples of the kinds of declarations that should be "private-able" include component and entry declarations. An example of being able to use a private declaration more fully is the ability to declare an externally-visible variable object of a private type.]

!Issue out-of-scope (un-needed) (2) desirable,small impl,upward compat,consistent

2. (Specification of Constant Components) Ada9X shall provide a mechanism to specify that certain components of an object of a record type cannot be overwritten without completely rewriting the object. Ada9X shall not arbitrarily restrict the type of these components nor require user-intervention in initializing them.

!Issue revision (3) desirable,small impl,upward compat,consistent

3. (No Aliasing by Subunits) Ada9X shall provide a mechanism to declare objects in a body that are fully accessible by bodies in the same compilation unit but not accessible by bodies provided in subunits.

!problem

The Ada83 implementation of limited visibility suffers two problems. First, not all declarations can be easily hidden. Also, not all declarations that are hidden are fully available for use by the declarer. Two related problems are that a record component cannot be made constant without allowing the user to initialize it and that there is no way to simply way to ensure that a variable that is shared by multiple procedures is not corrupted by subunits.

The model that many programmers seem to want (and one that is easy to understand) is that of being able to write the declaration of a task, package, or record type in the "normal" way and then to designate that some of the declarations are not to be visible outside the package or task. One could think of the declarations being highlighted by a transparent colored marker giving a "striped" effect.

!rationale

[2500.1] This requirement would cause an attempt to address the lack of completeness of the private facilities in Ada, within the constraints of other revision goals. Increased visibility control is needed to provide full access to information where access is necessary and to increase language security where access should be restricted.

[2500.3] It is often the case that a compiler is forced to generate inferior code if the code is generated before the bodies of subunits are seen because the compiler cannot know what the referencing needs of certain "own" variables are with respect to the subunits. This requirement would provide a mechanism to provide this information in the most important case without further constraining the compilation structure of the program.

!appendix

An alternative to 2500.1 which is much more solution oriented would be like the following:

%requirement 3 2 3 3

[1] To some reasonable extent, Ada9X shall allow a package specification to include interleaved visible and private declarations. In particular, it shall be possible to declare in a package specification:

1. a private component declaration within a visible record type definition;
2. a visible variable declaration whose type is given by a preceding private type declaration, and;
3. a private entry declaration within a visible task specification.

%reference RR-0229 Need better support for private scalar types

RR-229 wants to be able to have finer control over what is exported with a private type. Specifically, the proposal is to have private scalar types where the importer would know it was scalar (and could therefore use it in a discrete range, say) but not know other details (such as its endpoints).

%reference WI-0216 Give finer granularity of visibility of private type's properties

WI-216 seems to be a restatement of RR-229.

RR-560 also refers to the problem that some users of a package should be granted more access than others.

RR-0679 is really all about the idea that selection should be definable. It's an interesting idea.

AI-327 is about the fundamental mismatch that occurs because of linear elaboration order on the one hand and gathering up privates at the bottom on the other.

!rebuttal

5.8 Parallel Processing and Concurrency

RI-5241
RI-5220
RI-5230
RI-7005
RI-7007
RI-7010
RI-7001
RI-7020
RI-2101
RI-1060
RI-2106
RI-2012
RI-7003
RI-7030
RI-7040
RI-1040
RI-0003
RI-0005
RI-0001
RI-0002
RI-0009
RI-0201
RI-2107
RI-2108
RI-2001

!topic Complexity and Efficiency of Tasking Model
!number RI-5241
!version 1.2
!tracking 5.8.1, 5.8.1.3
DRAFT DRAFT DRAFT DRAFT DRAFT DRAFT

!Issue revision (1) compelling,small impl,upward compat,mostly consistent

1 – Improved tasking performance] An attempt shall be made to provide facilities in Ada9x that will improve the performance of tasking operations over that experienced in Ada83.

!reference RI-5220 fast mutual exclusion
!reference RI-5230 low-level tasking primitives
!reference RR-0078 Ada tasking is too complex, inflexible and inefficient
!reference RR-0084 Provide restricted tasking syntax for efficiency
!reference RR-0151 Improve Ada tasking model, support priority interrupts
!reference RR-0185 rendezvous is slow, semaphores would be better (related)
!reference RR-0241 Need efficient support for mutual exclusion (related)
!reference RR-0278 Tasking model is too complex, requires too much overhead
!reference RR-0747 Use Linda tuples for Ada tasking (related)
!reference WI-0415 Need efficient mutual exclusion (related)

!problem

The general theme of these revision requests is the poor performance of implementations of Ada's rendezvous tasking mechanism. The problem is that, because of the poor performance of many implementations, Ada's tasking facilities are of little practical use for the language's primary target area of embedded real-time applications.

!rationale

The perceived (and often real) poor performance of implementations of Ada's tasking facility has forced many application developers to use non-standard and non-portable facilities. Examples include low-level tasking library packages and custom-developed run-time systems. Thus, one of the primary objectives in supporting tasking within the language (i.e., common facilities for concurrency) has not been achieved. Every attempt should be made to rectify this in Ada9x.

RI-5220 requires creation of a supplemental standard for a fast mutual exclusion mechanism. RI-5230 requires creation of a supplemental standard for low-level tasking primitives. These two revisions should go a long way toward satisfying this requirement.

!appendix

Performance becomes a language issue when no implementations seem to be able to provide adequate performance for important classes of machines.

Solutions proposed in the revision requests range from specific low-level tasking primitives, to language subsets, to higher-level concurrency mechanisms.

%reference RR-0078

Ada's tasking semantics incur high overhead. Vendor's low-level tasking facilities and customized run-time executives are non-portable.

%reference RR-0084

Ada's tasking involves too much run-time overhead for high-performance embedded systems applications. Real-time programmers are forced to circumvent Ada's tasking facilities, often by calling non-portable run-time services or by rewriting parts of the underlying run-time system. A possible solution is to establish conventions for restricted use of tasking features that would enable the allowed features to execute substantially faster than they would in the general case.

%reference RR-0151

Ada's tasking model is awkward and arbitrary. Avionics software currently must avoid tasking. One feature that is needed is a non-maskable interrupt type task, which gives that task immediate access to the CPU.

%reference RR-0185

Use of the Ada rendezvous for inter-task communication or for protected access to shared data leads to unacceptable performance in some applications compared with simpler facilities such as semaphores.

%reference RR-0241

Ada83 does not support highly efficient mutual exclusion and current compilers are unable to recognize all the tasking idioms used for simple mutual exclusion that could be optimized.

%reference RR-0278

Ada's tasking model is too complex and has too much overhead for embedded systems. Well-known basic scheduling disciplines are not supported.

%reference RR-0747

Ada's tasking mechanism is too expensive in inflexible to serve as a basis for parallel and distributed real-time applications. The rigid synchronization and naming constraints are both inefficient and hard to use. (Includes proposal to substitute Linda's tuple space for Ada's concurrency paradigm.)

%reference WI-0415

The language shall provide efficient mutual exclusion, and consistent semantics for such exclusion, in a program (including a program distributed throughout a distributed or parallel system).

!rebuttal

!topic provide construct for super-fast mutual exclusion
!number RI-5220
!version 1.2
!tracking 5.8.1.1

DRAFT DRAFT DRAFT DRAFT DRAFT DRAFT

!Issue secondary standard (1) important,small impl,upward compat,consistent

1 – Fast mutual exclusion] An Ada9x supplemental standard shall specify a mechanism to support efficient exclusive access to data or operations.

!reference RR-0185 (related)
!reference RR-0241 (related)
!reference RR-0461 provide standard package of semaphore operations
!reference WI-0415 Need efficient mutual exclusion

!problem

Ada83 provides no highly-efficient, portable mechanism for mutual exclusion. Many applications cannot afford the overhead of a general rendezvous to achieve simple mutual exclusion and are forced to adopt non-portable solutions.

!rationale

Mutual exclusion can be implemented (several ways) using rendezvous. However, it is not reasonable to expect compilers to recognize all of the tasking idioms and provide satisfactory performance for all possible implementations of user-defined mutual exclusion. Providing a standard interface for a highly efficient mutual exclusion mechanism should satisfy the need.

!appendix

The intent here is to standardize the many variants of semaphore packages currently provided by vendors.

%reference RR-0185

Use of the Ada rendezvous for inter-task communication or for protected access to shared data leads to unacceptable performance in some applications compared with simpler facilities such as semaphores.

%reference RR-0241

Ada83 does not support highly efficient mutual exclusion and current compilers are unable to recognize all the tasking idioms used for simple mutual exclusion that could be optimized.

%reference RR-0461

Ada83 provides no standard form for efficient mutual exclusion. Many applications cannot afford the overhead of a general rendezvous to achieve simple mutual exclusion. Semaphores would provide the needed capability.

%reference WI-0415 Need efficient mutual exclusion

The language shall provide efficient mutual exclusion, and consistent semantics for such exclusion, in a program (including a program distributed throughout a distributed or parallel system).

!rebuttal

!topic need low-level primitives for flexible, arbitrary scheduling
!number RI-5230
!version 1.4
!tracking 5.8.1.2

DRAFT DRAFT DRAFT DRAFT DRAFT DRAFT

!Issue secondary standard (1) desirable,moderate impl,upward compat,mostly consistent

1 – Low-level tasking facilities] An Ada9x supplemental standard shall specify low-level tasking facilities suitable for building efficient alternative tasking or task scheduling mechanisms.

!reference RI-7005 Priorities and Ada's scheduling paradigm
!reference RI-7007 Priorities and Ada's scheduling paradigm
!reference RI-7010 Provide more flexibility in Ada's rendezvous
!reference RR-0016 (related)
!reference RR-0186 task control mechanisms to write OS
!reference RR-0379 (related)
!reference RR-0656 (related)
!reference RR-0748 standard low-level task scheduling utilities
!reference
ARTEWG, "A Model Runtime System Interface for Ada", version 2.3, Ada Runtime Environment Working Group, ACM SIGAda, October 1988.

!problem

Ada's high-level tasking facilities do not allow alternative tasking or task scheduling mechanisms to be implemented efficiently. This means that where efficient, alternate tasking or task scheduling mechanisms are needed to satisfy application requirements, the only available solutions are implementation specific and non-portable. Examples of tasking facilities that are needed but currently prohibited or unsupported include:

a. dynamic priorities
b. priority entry queuing
c. priority inheritance

Examples of alternate task scheduling mechanisms that are needed but not currently supported include:

a. rate-monotonic
b. earliest-deadline-first

!rationale

It is reasonable for Ada's general-purpose high-level tasking facility to be rather rigidly defined and not allow substituting semantics. It is not reasonable, however, to expect that this high-level facility will always meet the performance needs of special-purpose real-time applications. To avoid solutions that go outside the language, Ada9x should define a set of lower-level tasking primitives from which alternative higher-level mechanisms with satisfactory performance can be built. Considerable work in this direction has already been done by the ACM SIGAda Ada Runtime Environment Working Group (ARTEWG).

!appendix

This RI presents an alternate approach to rectifying the limitations of Ada's tasking scheme than that reflected in RI-7005, -7007, and -7010. It is NOT necessarily intended that Ada9x do both! If this RI can be interpreted as a valid solution to the -7000 series requirements, then that is what is meant and this RI should be a !mapping. Otherwise, we should find some way to present the two approaches as possible alternative solutions, identify and state the problem accurately, and state the higher-level requirement(s) for Ada9x language changes.

%reference RR-0016

Embedded real-time systems need alternate task scheduling disciplines, selectable at run-time, to maximize resource utilization and to handle mode changes.

%reference RR-0186

It is difficult to write an operating system using Ada's tasking facilities and remain independent of the compiler supplier's run-time system.

%reference RR-0379

We would like to have a choice of scheduling algorithms and be able to control the choice at run time to fit the needs of embedded system applications.

%reference RR-0656

In order to support deadline scheduling, it is desirable to set a timer that raises an exception in a task if the time runs out before the timer is cleared or reset.

%reference RR-0748

Ada83's rendezvous is not sufficient for practical implementation of highly constrained real-time systems including distributed systems. Ada9x should define a standard set of asynchronous real-time primitives that can be implemented directly in the run-time executive.

!rebuttal

!topic Priorities and Ada's Scheduling Paradigm
!number RI-7005
!version 1.12
!tracking 5.8.2.1 5.8.2.3 5.8.2.4 5.8.2.5 5.8.3.1.1 5.8.3.1.2 5.8.3.1.3

!Issue revision (1) compelling,moderate impl,upward compat,mostly consistent

1. (Consistent with Priority) Ada9X shall provide a run-time model which is provides maximal support for a priority-based tasking model. At a minimum, the following facilities are needed:

a. Interprocess communication must follow priority. Specifically, queuing order of tasks on entry queues, selection decisions from open select alternatives, task elaboration and creation, must use priority as the main criteria for choosing the next candidate when more than one choice is possible.

b. Priority Inheritance, where an executing task inherits the priority of the highest priority task queued on any of its entry queues, is required.

c. Dynamic Priorities, where an executing task can modify its own, priority or the priority of other visible tasks, is required.

d. Other task interactions which may be defined as part of Ada9X must be made consistent with the priority model.

This model may be in addition to the current model, in which case mechanisms shall be provided to permit specification of the paradigm selected.

!Issue out-of-scope (un-needed) (2) important,moderate impl,bad compat,inconsistent

2. Ada9X shall do all entry queuing and selection for rendezvous based solely upon priority.

[Rationale] This need, as expressed by the real time community, would radically alter the behaviour of existing code which uses the FIFO queuing and implementation-dependent selection paradigms. It can be met by letting the developer chose a paradigm for his program, as expressed in the preceding requirement.

!reference RR-0013
!reference RR-0015
!reference RR-0020
!reference RR-0021
!reference RR-0072
!reference RR-0076
!reference RR-0116
!reference RR-0121
!reference RR-0124
!reference RR-0192

!reference RR-0193
!reference RR-0337
!reference RR-0347
!reference RR-0415
!reference RR-0654
!reference RR-0657
!reference AI-00033
!reference AI-00233
!reference AI-00288
!reference WI-0412
!reference

(1) Cornhill, D., Sha, L., Lehoczky, J., Rajkumar, R., and Tokuda, H., "Limitations of Ada for Real-Time Scheduling," Proceedings of the First International Workshop on Real Time Ada Issues, Moretonhampstead, Devon, U.K., 1987.

!reference

(2) General Accounting Office, "Status, Costs, and Issues Associated With Defense's Implementation of Ada", report commissioned by the House Appropriations Subcommittee on Defense (1989).

!reference

(3) L. Sha, R. Rajkumar, and J.P. Lehoczky, "Priority Inheritance Protocols, An Approach to Real-Time Synchronization", technical report CMU-CS-87-181, Carnegie Mellon University (November 1987).

!problem

Ada83 does not provide a run-time model consistent with priority scheduling. It also does not provide a complete run-time model based upon a general concept of priority. The priority model chosen by Ada83 only provides for the scheduling of tasks ready for execution on the cpu. Specific deficiencies in the present model are that:

1. Task queuing on an entry queue does not take priority into account
2. Entry queue selection for the next rendezvous is arbitrary, not based upon priority
3. The priority of an executing (or ready-to-execute) server does not inherit the priority of the highest priority task on its entry queues
4. Task priority is defined statically at compile time, and cannot be altered except by implementation-specific calls.
5. Unacceptable race conditions may occur by Ada83 because the order of task placement on entry queues, and in the selection of entry queues for the next rendezvous.

This forces developers of real-time software away from the Ada model and into non-validatable runtime executives, non-Ada runtime executives with customized interfaces, or into multiprogram solutions with customized RTEs.

!rationale

The Ada programming model has known deficiencies in its current model of task synchronization and selection. Consistent treatment of priorities in the scheduling of tasks, and in all of their interactions is imperative to support the needs of the real-time community. The changes in the current Ada runtime model will change the behaviour of tasking in existing code: for this reason the mechanism needs to be selectable in the source code. It would be acceptable to include a new mechanism or if such behaviour could be triggered by the recognition of specific situations, such as the use of multiple priorities.

The need for dynamic priorities, as expressed in [7005.1(c)] does not extend to the general case of altering priorities in general. The two most compelling needs are to permit a task to adjust its own priority in recognition of new conditions, and to provide support for mode changes.

!appendix

Discussion of restrictions on Dynamic Priority Explicity Dynamic priorities, and Priority Inheritance both create Dynamic priority situations. Dynamic priorities bring their own set of problems, such as the reevaluation of queues, or inconsistencies in the selection of open alternatives, when the priority of a queued task is changed. Priority adjustment of the active task precludes these difficulties, because the task making the call cannot be rendezvousing at the time of the call. It is also the intent here that priority inheritance would not be transitive, as this would similarly create race conditions in queueing and selection.

%reference RR-0013 Allow dynamic task priorities

Activation of a task suspends parent. Activating a low priority task adversely impact a high priority parent. Need concept of activation priority and execution priority or dynamic priority.

%reference RR-0020 Dynamic task priorities are needed

Relative importance of tasks may change during different program phases. Need ability to adjust priority of a task to reflect this behaviour.

%reference RR-0021 Need priority inheritance for server tasks

When high priority tasks call entries in low priority tasks, it is unacceptable if the server does not execute at at least that priority.

%reference RR-0072 Task priority mechanism is inadequate for real-time processing

Ada priority system is inadequate for hard realtime systems. More precise definition of priority is needed. Queue FIFO model at odds with priority mechanism. Entry accepts must be priority based. Priority inheritance is needed. Entry families should be removed from the language.

%reference RR-0116 Provide dynamic task priorities

Allow the definition of priority for any task object. Allow the later modification of such priority dynamically. Support for mode changes and load balancing.

%reference RR-0121 Provide more control over "scheduling" decision.

The full power of Ada tasking model can not be used for real-time applications because of lack of control over scheduling decisions. Use priority for all choices in execution, also give user ability to control race conditions by using priority as condition for entry queue ordering and select mechanism.

%reference RR-0124 Reduce risk of priority inversion by revoking AI-000502.

AI-00288 Effect of priorities during activation

%reference RR-0192 Allow dynamic priorities for tasks

Allow a task to read its own priority via an attribute and to dynamically modify it to support mode changes and system degradation.

%reference RR-0193 Ada task priorities cannot be used consistently in synchronization

FIFO queues conflict with priority mechanisms.
Priority inheritance is needed.

%reference RR-0337 Provide some form of dynamic priorities

Allow a task to dynamically modify its own priority to support mode changes and fault recovery.

%reference RR-0347

Need better support for priority levels

%reference RR-0415

Allow priority inheritance & prioritized entry-queues & sel. wait

%reference RR-0654 Need dynamic priorities

Provide a priority-setting primitive to permit dynamic priority setting.

%reference WI-0412 Don't prohibit scheduling by context, incl. dynamic

!section 5.8.3.1.1

%reference RR-0015 Allow task priorities to control ALL queuing/select decisions

This RR makes a very strong case, that whenever a choice is to be made in an Ada Run time executive about which task is eligible for execution (or eligible for service), that priority must be taken into account. Workarounds are discounted because of the high overhead, or the difficulty maintaining the code.

%reference RR-0072 Task priority mechanism is inadequate for real-time processing

%reference RR-0121 Provide more control over "scheduling" decisions

%reference RR-0193 Ada task priorities cannot be used consistently in synchronization

%reference RR-0415 Allow priority inheritance & prioritized entry-queues & sel. wait

This RR points out the priority inversion problems inherent in Ada83 because the priority model conflicts with the FIFO queuing model. It asks that all such decisions be based upon priority.

5.8.3.1.2 Select from multiple open accepts based on priority

%reference RR-0076 Allow open alternatives selection based on priorities

This RR asks that the Ada select mechanism for selection of an open alternative be based upon priority of the calling tasks at the queue heads.

%reference AI-00233 The effect of priorities in a selective wait statement
%reference RR-0015
%reference RR-0072

%reference RR-0121
%reference RR-0415

5.8.3.1.3 Order entry queues based on priority

%reference RR-0657 Order entry queues based on priority

This RR asks that a higher priority task be served before a lower priority task which called the same entry, but is not yet served.

%reference AI-00033 Effect of priorities on calls queued for an entry
%reference RR-0075
%reference RR-0015
%reference RR-0072
%reference RR-0121
%reference RR-0193
%reference RR-0415

!rebuttal

!topic Support for Alternate Run Time Paradigms
!number RI-7007
!version 1.5
!tracking 5.8.2.6

DRAFT DRAFT DRAFT DRAFT DRAFT DRAFT

!Issue revision (1) important

1. (Do not preclude alternate RT models) Ada9X shall not preclude the use of alternate run-time scheduling paradigms such as priority-based, rate-monotonic and earliest-deadline-first in conjunction with the existing concurrency and inter-task communications mechanisms.

!Issue revision (2) compelling,small impl,upward compat,mostly consistent

2. (Remove RTE Restrictions) Ada9X shall remove restrictions which prohibit or have the effect of prohibiting alternate run-time paradigms. Ada9X shall not specify new constructs or restrictions which have the effect of prohibiting the use of alternate run-time paradigms.

!Issue out-of-scope (un-needed) (3) important,severe impl,upward compat,inconsistent

3. (Alternate run time paradigms) Ada 9X shall provide a mechanism to permit specification of alternate run-time scheduling paradigms from within the language, including rate-monotonic and earliest-deadline-first. The mechanisms anticipated here include the ability to specify a scheduling behaviour based upon high performance (non-Ada83) scheduling/rendezvous paradigm, and also includes mechanisms to give implementations information about task behaviour, such as iteration rate, processing deadline, or promised execution time.

!Issue secondary standard (4) compelling

4. () The standard shall provide an interface specification to alternate run-time executives which support paradigms such as Rate-Monotonic or Earliest-Deadline-First.

!reference RR-0016
!reference RR-0020
!reference RR-0021
!reference RR-0037
!reference RR-0170
!reference RR-0379
!reference RR-0656
!reference WI-0413
!reference WI-0413M
!reference RR-0124

!reference RI-7010

!reference
[1] J. Goodenough and L. Sha, "The Priority Ceiling Protocol: A Method for Minimizing the Blocking of High Priority Ada Tasks", Proceedings of the Second International Workshop on Real-Time Ada Issues, *Ada Letters* Volume VIII, Number 7 (Fall 1988), pp. 20-31.

!reference
[2] L. Sha, R. Rajkumar, and J.P. Lehoczky, "Priority Inheritance Protocols, An Approach to Real-Time Synchronization", technical report CMU-CS-87-181, Carnegie Mellon University (November 1987).

!reference
[3] J. Goodenough, L. Sha "Real Time Scheduling in Ada" Computer, April, 1990

!problem

Alternate run-time models, such as deadline-driven scheduling or rate-monotonic based scheduling are required in the real time community. These cannot be straightforwardly dealt with in Ada83, and in fact are virtually prohibited by the FIFO queuing of task entries, and the fixed nature of task priorities. In order to get the scheduling behaviour that they require, many members of the real-time community abandon the Ada tasking model and use cumbersome, multi-programming techniques, or use non-Ada tasking with customized library calls. This has an adverse impact on development time, code reusability, and maintenance of software systems developed for these application areas.

The following areas are perceived as being obstacles to the use of real-time tasking paradigms in conjunction with Ada83's tasking:

a. Ada83's specification of static priorities
b. FIFO task entry queues
c. Lack of priority inheritance

!rationale

The embedded real-time community uses a variety of different concurrency paradigms to match their processing environment to the physical environment with which they must function. In developing systems, they choose a run time model which will give them the performance behaviour that they require.

It is accepted by the real-time community at large that a complete solution to each of the paradigms needed cannot be provided using language mechanisms designed for a more general-purpose community. At the same time, many of the abstractions provided by Ada (or contemplated by Ada9X), such as tasks, entries, and the select mechanism, would be useful if they were less restrictive.

Language mechanisms which prohibit the introduction of such paradigms must be removed, made optional, or made user-selectable. Furthermore, the language must not introduce new mechanisms which further impair the use of alternate run-time paradigms in the language.

It is felt that the mechanisms needed for each paradigm are distinct and evolving, and that a common set of language constructs (even pragmas) is unachievable. An interface package to the Run Time Executive with a standardized set of subroutines would let a task specify its behaviour characteristics (such as rate & duty cycle or rate & deadline). Other language supporting mechanisms are addressed in separate RIs (such as RI-7010 for user-specified queuing and select).

!appendix

%Reference RR-0016

This RR makes a very strong case for allowing developers to control in source code the RTE scheduling paradigms needed to support their program. It details the effects on high-profile projects of the current lack of support for such control.

%reference RR-0020 Dynamic task priorities are needed

Relative importance of tasks may change during different program phases. Need ability to adjust priority of a task to reflect this behaviour.

%reference RR-0021 Need priority inheritance for server tasks

When high priority tasks call entries in low priority tasks, it is unacceptable if the server does not execute at at least that priority.

%reference RR-0037

This RR requests more support to do simulations. Specific requests are for ways of connecting to user-written executives and clock support.

%reference RR-0170 Permit or provide alternate scheduling algorithms

This RR specifically states a need to be able to specify alternate scheduling protocols, such as rate-monotonic.

%reference RR-0379 Application should select the specific scheduling algorithm

This RR states the need to be able specify a run-time paradigm consistent with the problem that a program is attempting to solve. It asks for standard scheduling algorithms, Hard deadline algorithms, and more determinism in the language specification for tasking.

%reference RR-0656 Need timed exceptions for deadline scheduling

This RR wants more support for hard-deadline scheduling in Ada.

%reference WI-0413 Need mechanisms for scheduling by mult (&user def) characteristics

This Destin Workshop requirement states a need to permit user-specified scheduling.

!rebuttal

!topic Flexibility in selection for rendezvous
!number RI-7010
!version 1.8
!tracking 5.8.3.1.1 5.8.3.1.2 5.8.3.2 5.8.3.3

!Issue revision (1) important,moderate impl,bad compat,mostly consistent

1. (Explicit user Control over Accepts) Ada9X shall provide explicit user control over the mechanisms that a task uses to select the next candidate for a rendezvous, when more than one choice is possible. The choices shall include which candidate to select from a single entry queue, from multiple entry queues in a SELECT statement, or from one or more entry families. It must be possible to choose priority as the criteria of the choice, but the choice mechanism should include other criteria, such as parameters of the call, attribute(s) of the callers, or state of the acceptor. The choice mechanism must be notationally compact.

!reference RR-0072
!reference RR-0121
!reference RR-0737
!reference RI-7005
!reference WI-0413
!reference WI-0413M

!reference
[1] T. Ellrad
Comprehensive Race Controls: A Versatile Scheduling Mechanism for Real Time Applications
Proceedings of the Ada Europe Conference, June 1989
Cambridge University Press, 1989

[2] Cornhill, D., Sha, L., Lehoczky, J., Rajkumar, R., and Tokuda, H., "Limitations of Ada for Real-Time Scheduling," Proceedings of the First International Workshop on Real Time Ada Issues, Moretonhampstead, Devon, U.K., 1987.

!problem

Ada83 does not provide a complete run-time model based upon a general enough concept of priority to meet the requirements of system developers targeting to distributed/multiprocessing environments. The priority-based model requested in RI-7005, while making large gains in eliminating priority inversion and improving throughput, are incomplete when attempting to design and implement systems for this environment. Much of the request for dynamic priorities which occur in the RRs for RI-7005 occur because the relative importance of a task changes when interacting with clients and servers, and that change cannot be modelled by a static priority model. It can be solved by providing more mechanisms at the task synchronisation level, without resorting to the overhead and non-determinism of a complete dynamic priority executive.

Specific problems are that

1. Task queuing on an entry queue cannot be specified by the designer to be other than FIFO, i.e. to be based upon some parameter, including priority
2. Entry queue selection for the next rendezvous is arbitrary
3. Entry queue selection for the next rendezvous cannot be specified by the designer, i.e. to be based upon a parameter, including priority
4. Unacceptable race conditions are caused by Ada83 in the order of placement on entry queues, and in the selection of entry queues for the next rendezvous.

Entry families provide a partial alternative in Ada83 to entry queue ordering and selection control. Entry families are very limited, however. The entry index must be enumeratable, and can be based only upon a single parameter. The selection mechanism to achieve the required order is cumbersome and error-prone, as shown in the following example:

```
select
  accept service(100)(...) do ... end;
or
 when service(100)'count  = 0 =>
  accept service(99)(...) ...
or
 when  service(100)'count = 0
                and service(99)'count = 0 =>
  accept service(98)(...) do ... end;
...
or
 when  service(100)'count = 0
                and service(99)'count = 0
                ...
                and service(2)'count = 0 =>
  accept service(1)(...) do ... end;
end select;
```

The technique shown above in Ada83 is deficient for the following reasons:

1. It works for small numbers of entries in a family, but deteriorates as the number rises, for what is basically a simple concept, select the next accept statement from the highest valued entry queue.
2. It is error-prone as it forces coders to retype (or cut-paste) massive code segments, increasing chances of typographical errors. This code example results in 5,500 lines of code simply to evaluate which entry to accept, before executing any "active" code.
3. The basis for choice is limited. Internal task state(s) and entry queue counts are about all that can be tested. No information from the caller, such as attributes or parameters, is available for evaluation.

4. The evaluation of which entry to accept occurs only at select evaluation time, not once the acceptor is waiting at open alternatives. It is possible, especially in multicpu situations, for multiple entry calls to be placed at different entries simultaneously, and the mechanisms in Ada83 do not provide guidance as to how to choose one for rendezvous.

5. The evaluation of the select happens in the context of the acceptor, not the run time executive, providing little or no opportunity to optimize the choice mechanism.

The difficulties discussed above force developers to use non-portable implementations, or to use unvalidatable RTEs, or to use multiprogramming solutions with customized RTEs and interprogram communication service packages. This has an adverse impact on code portability and reuse, and on maintainability.

!rationale

The concept of priority used in Ada83 was included only to solve scheduling contention on a single cpu. RI-7005 asks specifically for Ada9X to include this priority in all task selection for rendezvous decision points. The choices made in RI-7005 are not sufficient to solve scheduling in a distributed environment or in a complex single cpu environment, however.

1. In a multiple cpu system, priority refers only to the relative urgency between tasks on the same processor. Tasks on different processors also have relative urgencies in the order in which they are served by a common server. This order is distinct from processor priority in general. The language could attempt to widen its concepts of priority to create priority tuples, but any such attempt predetermines usage, and will fail to map into the user-need space. The only solution which is general enough is to use task parameters assigned at task creation, or parameter(s) of each entry call. One of the selection criteria available to application designers must be task priority as defined in Ada83.

2. The Ada83 notion of priority is defined by the implementation. In many applications more granularity is required for the selection mechanism and order of queuing than the implementation's priority scheme can provide. In addition, portability is impaired because the new implementation's priority scheme may be more restrictive and break or change the execution behaviour of task interactions. User-defined entry and selection criteria lessens the dependence on an implementation's priority scheme and enhances portability.

Nonstandard uses of priority, such as attempting to simulate a multi-processor multi-tasking system on a single-cpu, are also achievable with user-specified queuing and selection criteria, without breaking the priority-based run-time model or resorting to non-standard run time executives, as would be required if Ada priority alone were used.

Entry families are a way of segregating calls by a parameter. The parameter of the entry

family is ordered, but the order of selection from such a collection, in Ada83 must be explicitly written into the selection criteria, as shown in the example in !problem. Designers must be able to direct implementations to select from open members of an entry family based upon increasing or decreasing entry index value, and upon other parameters, such as Ada priority of callers, and other parameters of the callers entry calls. The mechanism chosen should be expressible in a constant number of Ada statements.

!appendix

%reference RI-7005
RI-7005 addresses requests which specifically want entry queuing and selection based upon priority. The need for dynamic priority is a need for a mechanism more flexible than static priority to solve the complete range of problems faced in developing real time systems.

%reference RR-0072
This RR states the need for more control of task interactions. It explicitly asks for strict adherence, then asks for more flexibility and control of these interactions be provided, suggesting that dynamic priorities would be a mechanism.

%reference RR-0121
This RR asks for better facilities to control race conditions in Ada. It asks for all choices to be based upon priority, or upon user-selectable criteria.

%reference RR-0737 Allow preference control for entries in a select statement
This RR requests that Ada9X provide a mechanism to specify a selection order from a number of open alternatives. The mechanism should be compact, expressive, and complete.

%reference WI-0413 Need mechanisms for scheduling by mult (&user def) characteristics
This Destin Workshop requirement states a need to permit user-specified scheduling (including queuing and selection) in an Ada program. Priority is one of the likely-to-be-chosen criteria, but in a distributed environment, other parameters are often needed. The members of this group were unanimous that the use of priority alone would not solve their problems.

%reference WI-0413M Provide spec language to direct scheduling outside "Ada env"
This was the minority view to WI-413. It was concerned that techniques are still in research and that Ada may do it wrong. It stated that such mechanisms should be outside the language.

%reference [5] T. Ellrad Comprehensive Race Controls: A Versatile Scheduling Mechanism for Real Time Applications Proceedings of the Ada Europe Conference, June 1989 Cambridge University Press, 1989

An example of where Ada priority is insufficient, even for a single cpu:

Consider a disk handler. The mechanical transit time of a disk head, and the rotation rate, are such that, if you were to order queues by priority, you would obey the priority model, but your throughput would be intolerable. If the sector being requested by two tasks of different priorities are on the same track, but the head will position over the lower priorities sector first, the high priority task gets no better service if you service it second, but the lower priority task's service (and overall throughput) degrade's drastically. The same discussion holds for track=>track head movement (almost).

In order to write a proper disk driver, I need to be able to queue requests in an order that depends upon increasing or decreasing track count, and always increasing sector count (unless, of course they interleaved sections; or, I realize that as the head moves tracks, sector 1 is 1/2 turn away from the head. Then I need to queue based upon a function of the sector number).

The relevant issue is that priority helps us arbitrate a scarce resource, the cpu, between competing users. Usually the time frame of other events in the system are in the same relative granularity as cpu cycles, so the mapping to cpu-based priority is isomorphic. When the time frame, or number of states, of other events in the system do not map easily onto the rigid priority model, other paradigms of the sequencing of task interactions are needed. It is at this point that user-specifiable criteria (such as parameters of a call or attributes of a task) are essential, and mapping to priority is insufficient.

!rebuttal

All 3 of these requirements, as worded, are out of scope for Ada9X. These are very big changes to the language. They will slow the availability of Ada9X implementations. They will make the Ada run-time support software larger, slower, and more complicated. Queuing/selection based on priority combined with dynamic priorities is a MUCH simpler approach to addressing the problem of lack of flexibility in Ada rendezvous and is nearly as effective.

!rebuttal

The problem with the requirements of RI-7010 is that they cause a significant language change while not solving a sufficiently general class of the concurrent programming problems faced by Ada programmers. It must be said that RI-7010 solves some problems that are not easy to solve using dynamic priority a la RI-7005. What is perhaps more important is that it solves some problems that are solved by dynamic priority but inappropriately. One of the submitters in AI-594 laments that many programs implemented on uniprocessors will not be able to be transported easily to a multiprocessor because priority is being used (inappropriately) as a synchronisation mechanism.

The real issue is that RI-7010 does not adequately address the issue of dynamic determination of the next client to be scheduled. Adding some ability to specify a function to sequence over an entry family will certainly make it easier to specify the correct selection but at the cost of having the underlying system poll across a large number of entries looking for the correct one. If the search is to be parameterized by a user-defined policy, one must then wonder why the user did not do the search himself using the intended policy. (The answer is that it is not easy to parameterize functions and procedure with respect to entries in Ada83; perhaps this is the problem that should be addressed.)

For [1], the problem is that one wants to determine selection order at the time of service, not at queuing time. In the general case, one would not want to search the queue for the best client--rather, a special structure would be built that was optimized for this search. The best approach, it seems, would be to let the user build the appropriate data structure himself; but in order to do so he must currently rely on multiple synchronizations between the server and the client. An optimizing compiler can sometimes remove the extra synchronizations; it seems far better to allow the result of the optimization to be expressed directly in the language than to rely on complex, interprocedural optimization.

The bottom line is that the acceptance of RI-7010 decreases the chances that the M/R team will look for and find a solution to the general problem that is modest in implementation impact.

!topic Programmer control of the real time clock
!number RI-7001
!version 1.1
!tracking 5.8.4

DRAFT DRAFT DRAFT DRAFT DRAFT DRAFT

!Issue revision (1) compelling,small impl,upward compat,consistent

1. (resume execution after delay) Ada9X shall require that the completion of a delay cause the affected task to immediately resume execution.

!Issue secondary standard (2) compelling,severe impl,upward compat,consistent

2. The standard shall provide a method of defining implementation-specific behaviour of all clocks and timers used by a run-time executive to support an Ada9X program.

!Issue secondary standard (3) compelling,small impl,upward compat,consistent

3. The standard shall provide a complete interface for an application to set or manipulate the calendar clock.

!Issue revision (4) compelling,severe impl,upward compat,mostly consistent

4. (Improve Clock Specifications) Ada9X shall provide a more complete and general specification of time for delay, calendar, and for systems where no time is needed or provided.

An implementation of Ada9X which does not need to support any concept of time (e.g. delay statement, delay alternative, or package calendar) shall not be required to support these structures as part of the base language.

????? Something about better specification of Ada clocks ??? (AIs 223, 201, 366)

!reference AI-00223
!reference AI-00201
!reference AI-00366
!reference AI-00442
!reference RR-0037
!reference RR-0105
!reference RR-0107
!reference RR-0276
!reference RR-0280
!reference RR-0286
!reference RR-0352
!reference WI-0417

!problem

Ada83 has a very fixed model of time. There is effectively no way within the language to connect the run time executive to an implementation of the timer services which support it. Users must drop into extralingual solutions, usually customization of the executive, to solve their problems, which include customizing the interface to clock and timer services provided by hardware or the operating system, setting or adjusting the clock for discontiguous changes (such as Daylight Saving time or time zones) and clock drift adjustments.

An additional problem is that to validate, current compilers must demonstrate correct usage of time. On embedded systems that do not have support for this concept, Ada83 cannot be currently supported.

!rationale

[RI-7001.1] The current language is perceived as being too loose in its specification of what occurs when a delay completes. It is not possible to specify an absolute resumption time criteria, hence the Ada concept of "immediate" is used.

[RI-7001.2] Package calendar is a very rudimentary package for providing date and time information to an Ada program. It does not support time concepts before 1900; it does not support concepts such as clock setting or adjustment, and it does not (necessarily) support the same time resolution as SYSTEM.TICK. A deeper issue also exists in that setting a clock is not a permitted activity on most systems, for legal and safety reasons. The Ada9X process should attempt to define a more comprehensive calendar.clock package which would provide the additional services needed, and also provide protection for situations where setting or adjusting the clock is a controlled operation.

[RI-7001.3]
A run time executive often maintains multiple concepts of time, even on a single cpu system. An executive can provide a combination of hardware calendar clocks, software calendar clocks, hardware timers and software timers. These often have different functionality, resolution, and drift. This imposes a diversity upon the language which is judged to be outside the LRM scope to solve in general.

In order to assist compiler writers and implementers who need access to low-level time interfaces, it should be possible to define an interface package between the run-time executive and the lower level (OS or hardware). This package could provide details about the number of time services available, their resolution, addresses and interrupt locations, etc. Such a standard should be a controlled standard within Ada9X.

[RI-7001.4]
Ada9X should examine the time management situation in Ada to see if there is anything which can be done to give better specifications. AI-00201 points out that there is no relation between system.tick and duration'small, or the execution of delay statements. There is also often no relation between SYSTEM.TICK and the time increment provided by the hardware or underlying system.

The fact that Ada is not validatable for hardware which does not support a concept of time is unacceptable to many organizations that need high order language support but don't have the hardware to support this mechanism. Just as most features of text_io can be waived for embedded systems lacking a filing system, so also could time-oriented language features (such as delay statements or alternatives, and package calendar) be waived for implementations not using time.

!appendix

Clock often provided in hardware - no control, can't conform to LRM Can't adjust behaviour in a common way (eg - run @ 1/2 speed) Have to drop into RTE customizing to connect RTE with actual clock(s) (RR-107) Maybe need clearer definition of what the RTE actually controls (RR-0276) What if the hardware doesn't support time? - don't do any delays

%reference AI-00223 Resolution for the function CLOCK.

This AI points out that current LRM wording on time allows arbitrary (even unreasonable) implementations of time in Ada programs, such as clocks with a time resolution of 1 second or greater.

%reference AI-00442 Time zone information in package calendar

The concepts of time zone and Daylight Savings need to be added to the language. Doing so may require gradual clock readjustment, not just step functions.

%reference AI-00201

This AI points out there there is no relation between

SYSTEM.TICK and DURATION'SMALL
SYSTEM.TICK and delay statements

Author's note: there is no guarantee that SYSTEM.TICK is related to an implementation's clock tick rate, since an implementation may customize an RTE after package system has been compiled.

%reference AI-00366 The value of SYSTEM.TICK for different execution environments

SYSTEM.TICK should have a value that reflects the precision of the clock in the main program's execution environment. If SYSTEM.TICK does not have an appropriate value, the effect of executing the program is not defined.

%reference RR-0037 Allow user-controllable real-time system clock

This RR needs to be able to to provide pseudo-real time for simulations. Presently patches the RTE or provides own customized RTE for this. This reviewer doesn't understand why the following doesn't work?

delay pause_time * adjustment;

where adjustment is a constant (say 0.5, 0.1, etc.) for the run defined in a highest-level types pkg?

%reference RR-0105 Provide a user-setting/adjusting of CALENDAR.CLOCK

This ARTEWG RR states the need to be able to adjust the calendar clock. The contention is that Calendar. Clock should provide the services. The need is to be able to make incremental or large changes, smoothly or in a single step.

%reference RR-0107 Provide low-level CALENDAR.CLOCK interface (ARTEWG)

This ARTEWG RR states the need for standard way for application builders to provide the configuration-dependent information upon which the implementation of CALENDAR.CLOCK depends. A developer needs to be able to specify target dependant information, such as tick rate, to the language.

%reference RR-0276 The timing/clock aspects of the language should not be required

This RR wants the language decouple the RTE from timer considerations, letting the application developer add her own if needed. For systems with no on-board timers. the language and ACVCs make it impossible to use Ada.

%reference RR-0280 There are problems with Calendar, delays, timing in Ada

- — The syntax/semantics of delay are too loosely defined.
- — Ada needs two (2) concepts of delay
  - — 1 in the microsecond, millisecond range,
  - — the other as is defined presently.
- — Package calendar should use hours instead of years/days,etc.

%reference RR-0286 Embedded systems don't want ada runtime to muck with clock/interrupts

This RR wants the Ada language to give complete control of clocks and hardware interrupts to users. The main reasons stated are for safety, and to permit users to write self-test routines, etc.

%reference WI-0417 Need different perception of time a diff parts of 1 Ada program

Ada tasks on a distributed system will have different timers with drifting perceptions of time. Ada9X must account for this and provide mechanisms to let the application developer manage such a system.

!rebuttal

!topic Interrupts
!number RI-7020
!version 1.3
!tracking 5.8.5 5.8.2.2

DRAFT DRAFT DRAFT DRAFT DRAFT DRAFT

!Issue revision (1) compelling,small impl,moderate compat,consistent

1. (Correct Interrupt Handling) Ada9X shall provide mechanisms for manipulating hardware and delivering hardware interrupts into a program in a manner which accounts for a wide diversity of computer architectures. At a minimum, these mechanisms shall be flexible enough to:

a. map a chosen interrupt to a selected entry of a specified task object at execution time,

b. disassociate a task entry from a previously associated interrupt,

c. map a chosen interrupt to a specified procedure at execution time,

d. disassociate a procedure from a previously associated interrupt, and

e. permit a code segment to declare and manipulate hardware registers which may be memory mapped or which may use other mechanisms for accessing the device.

!Issue revision (2) compelling,moderate impl,upward compat,mostly consistent

2. (Non-blocking Inter-Process Communication) Ada9X shall permit a task to pass parameters to another visible task, or to unblock another visible task, without having to accept an entry call from that task, or having to call an entry on that task.

!Issue revision (3) compelling,moderate impl,upward compat,consistent

3. (Priorities can Exceed Hardware Interrupt Priorities) Ada9X shall allow task priorities to match or exceed hardware task priorities outside of accept blocks, except where prevented by an underlying operating system. For safety reasons, Ada9X shall provide a mechanism which is distinct from the pragma priority method used today.

!Issue out-of-scope (un-needed)

[4 - No loss of interrupts] Ada9X shall specify that interrupts connected to a task entry where the task is not in a state to receive them shall be queued for later processing.

[Rationale] If a task entry remains the only mechanism to intercept interrupts, designers must recognize that such a requirement is impossible to meet in many systems. Many embedded systems require explicit resetting of hardware before another interrupt can occur, or connect interrupts from many devices to the same hardware vector. In such cases, the interrupt must be immediately processed. It is anticipated the the explicit inclusion of procedures as the target of interrupts will solve the problems encountered

here.

!Issue out-of-scope (un-needed) (5) desirable,small impl,upward compat,inconsistent

5. (Prohibit Software invocation of Interrupt Code) Ada9X shall provide a mechanism to allow a program to specify that an entry or procedure selected as the target of an interrupt shall not be callable from software.

[Rationale]
This statement is desired for safety and efficiency reasons. There are potentially many other times when an implementation may want the ability to invoke these code segments from software. Implementation-specific pragmas which provide direction to an implementation are an appropriate method of solving this need. Such an implementation-defined pragma is non-portable. Such a pragma should be non-portable, as long as compiler errors are generated in compilers which do not support it.

!reference RR-0087
!reference RR-0115
!reference RR-0151
!reference RR-0179
!reference RR-0316
!reference RR-0349
!reference RR-0421
!reference RR-0686
!reference RR-0735
!reference WI-0309
!reference WI-0310
!reference WI-0310M

!problem

Manipulating hardware associated with an embedded system, and connecting/processing hardware interrupts is perceived to be one of the major weaknesses in Ada. Specific complaints are:

a. Interrupts are specifiable only in task declaration blocks, meaning that arrays of tasks, etc. to manage hardware is virtually unachievable.

b. LRM 13.5.1(2) states that interrupts act like tasks whose priority is higher than the priority of the main program, or of any user-defined task. Ada83 tasks, acting as device handlers, must often interact with associated hardware in a state that precludes interruption by the associated hardware, or any hardware of a lower priority. Such direct manipulation is likely to cause an uncontrollable interrupt in Ada83 because of this rule.

c. Tasks may lose interrupts because the task entry/rendezvous mechanism is often not supported by hardware systems. If the interrupt occurs before the task arrives at the accept, the interrupt is not required to be queued, and in fact many hardware systems cannot queue such an interrupt.

d. The mechanism for the specification of hardware registers and of interrupt vector locations is different from the mechanism for memory specifications on many architectures. The uniform Ada83 specification does not map easily into these architectures, leading to significant programming "challenges".

e. Task entries specified as being interrupt entries can be called by other tasks as a normal rendezvous. This may lead to Erroneous programming situations. It also slows down interrupt code because it must take the general entry call into account.

f. Entries cannot be dynamically connect to an interrupt. For reasons of reconfigurability and fault tolerance, the configuration of an interrupt/entry pair often must change dynamically.

g. Tasks cannot explicitly schedule other other tasks without doing a rendezvous with them.

h. Tasks cannot communicate with other tasks without a full rendezvous. Non-blocking Inter-process communication is required so that a time-critical task can transmit parameters to other tasks without explicit rendezvous.

The difficulties encountered in this area are severe. In order to solve them, implementations are resorting to implementation-dependant tricks and pragmas, using procedures for interrupts, or not supporting interrupts in Ada. Users are using specialized run-time executives which promote semaphores, monitors, etc., as well as interrupt support, are using assembly-language level kludges, or are writing complex, multiple-task(nested) structures to get the effects they need.

!rationale

[7020.1]
The interrupt mechanisms currently available in Ada83 are insufficient as documented in the problem statement. The user community is sharply divided on what fixes are needed, but agree that this area must be improved. "Continued confusion over how to program interrupt handling in Ada will cause either the avoidance of Ada interrupt entries, incorrect usage of tasks as interrupts handlers, or the proliferation of nonstandard approaches for handling interrupts." (ARTEWG RR-0115)

Ada9X must provide the expressive power to enable developers to match the programs they are developing to the hardware/software they are using. Developers working on target hardware which does not have a memory-mapped view of devices and hardware should not have an artificial structure imposed upon them. Developers not using tasking should not be forced into the use of tasking to handle interrupts, the way they are presently with task entry constructs as their only mechanism for connecting to an interrupt. Similarly, a developer needing arrays of tasks, each dynamically connected to a different interrupt must be given the language constructs to reasonably develop the algorithms he/she needs.

[7020.2]
The ability to schedule another task, or to pass parameters to another task without being

blocked is fundamental to the community communicating with hardware through interrupts. The lack of such a facility in Ada83 has contributed to the continued success of non-standard run-time executives and extra-lingual solutions.

[7020.3]
There is a fundamental need for the highest priority task to execute before those of lower priority. There are times when it is a mistake to have a software task suspended so that an interrupt can execute. This requirement rescinds the Ada83 position that interrupts always take priority over user-written tasks.

!appendix

%reference RR-0087 Allow software priorities to match/exceed hardware priorities

Modeling an interrupt as an entry call issued by a hardware task whose priority is higher than the priority of the main program or tasks, is too restrictive.

%reference RR-0115 Provide better interrupt handling model

This (ARTEWG) RR states that the interrupt handling mechanism defined by the Ada Reference Manual (ARM) should model real interrupt handlers. Current implementations provide inconsistent and often inadequate support (and documentation) for the mechanisms used to handle interrupts in Ada; causing user confusion over how interrupt handling should be correctly programmed in Ada applications.

%reference RR-0151

This RR states that the Ada tasking model is awkward and arbitrary. A specific complaint wrt interrupts is that tasks cannot take advantage of non-maskable interrupts.

%reference RR-0179 Problems with interrupt handling

The LRM does not fully consider the need for, and implications of, executing an accept on hardware interrupt level.

%reference RR-0316 Improve interrupt handling, e.g., with interrupt procedures

The current Ada mechanism for handling interrupt is ambiguous, awkward and inefficient. It is conducive to undetected loss of interrupts and to timing errors

%reference RR-0349 Definition of hardware interrupt handling has two flaws

This RR notes that the Ada83 mechanism for specifying an interrupt address is erroneously mapped over Ada's concept of a memory space, i.e. the current definition uses the same type for memory addresses and interrupt addresses. It also complains that Ada does not make it illegal for multiple entries to have the same interrupt location.

%reference RR-0421 Interrupt handling and interrupt entry association have problems

This RR discusses a number of problems in interrupt handling in Ada. Specific complaints are:

— The connection of an interrupt with an entry

— The homograph between memory addresses and interrupt entries

— Interrupt entries cannot be applied to instances of a task type

— Operating System behaviour can affect spec and use of interrupts

%reference RR-0686 Priority of interrupts higher than normal tasks is ill-conceived

Whether a hardware interrupt interrupts an Ada task is application dependent

%reference RR-0735 Correct a number of aspects relating to interrupt handling in Ada

This RR discusses a number of problems in interrupt handling in Ada. Specific complaints are:

— Flexible/dynamic connection between interrupts and task entries needed

— Backlogging of interrupts should not be entertained

— Interrupt entries should not be callable from other tasks

— Direct scheduling of other tasks (sans rendezvous) is needed

— Asynchronous rendezvous wanted by some

— The uniform memory/interrupt-vector approach doesn't fit many machines

— Binding to interrupts may be application-specific, not doable in RTE

%reference WI-0309 Provide safe way to dynamically connect/change int and task entry

This Destin Workshop Requirement followed a long discussion about difficulties with the current Ada83 interrupt specification and use model.

%reference WI-0310 Ada runtime should not lose interrupts tied to task entry

Specific concern was that task behaviour could cause it to miss an interrupt, with very serious consequences.

%reference WI-0310M Backlogging of interrupts unsupportable in some hardware

Several group members were concerned about forcing all implementations to support an interrupt backlogging mechanism. It was suggested that this mechanism should be an option supported by the vendor.

Many hardware systems require specific action from the software to repost or reenable an interrupt. It is unreasonable to expect the run-time executive to know how to do this for an arbitrary device being directly manipulated by the application.

The Case For Procedures as a Target for Interrupts.

Using the axiom that hardware interrupts are manifestations of hardware tasks needing to interact with software tasks, it is clear that at some point the interrupt needs to be delivered to a task entry for proper synchronization. The difficulty arises that there is often work to be done before this synchronization can occur. Consider a system where multiple hardware devices interrupt through the same vector. A distinct task is written to manage each device. When the interrupt occurs, a check of all hardware registers is needed to determine which raised the interrupt, then the appropriate task entry can be called. The contention is that this appropriate for a procedure, not a task which would have to ensure an additional rendezvous (even with optimizations).

```
Package Low_Level_Stuff is

Task Device_1 is
 entry Device_Needs_Service;
 – for Device_Needs_service use at some_location;
end Device_1;

Task Device_2 is
 entry Device_Needs_Service;
 – for Device_Needs_service use at some_location;
end Device_2;

Task Get_Real_Interrupt is
 entry Device_Needs_Service;
 for Device_Needs_service use at some_location;
end Device_1;

end Low_Level_Stuff;

Package body Low_Level_Stuff is

 Task Device_1 is separate;

 Task Device_2 is separate;

 Task Get_Real_Interrupt is
  – declarations
 begin
  – setup stuff
  loop
   accept Device_Needs_Service do
                  – determine what device it was that generated the interrupt
                  case device_called is
                   when Device_1 => Device_1. Device_Needs_Service;
                     – Although this section of code executes at the priority
                     – of the hardware, the true interrupt service code likely
                     – does not as it is a secondary call.
```

```
                        -- It is unlikely that optimizers would be able to "pacify"
                        --   this task according to rules known today, since
                        -- 1. It does work outside the rendezvous as well as inside
                        -- 2. It calls other tasks inside the rendezvous
                      when Device_2 => Device_2. Device_Needs_Service;
                    end case;
  end Device_Needs_Service;
 end loop;

end Get_Real_Interrupts;

end Low_Level_Stuff;
```

!rebuttal

!topic Distributed systems
!number RI-2101
!version 1.2
!tracking 5.8.6

!terminology
The requirements in RI-2101 deal with a type of parallel computer architecture which has more than one CPU, i.e. more than one instruction stream. Such parallel architectures are sometimes classified by whether communications between CPUs are via shared-memory or by I/O. The former are called "shared-memory architectures"; the latter are called "distributed architectures". A parallel architecture is homogeneous if all of the CPUs are identical; it is heterogeneous otherwise. In the following, a cluster of CPUs to which a partition of a program may be allocated is called an "execution site".

!Issue revision (1) compelling,severe impl,moderate compat,inconsistent

1. An attempt shall be made in the design of Ada9X to reduce the number impediments in Ada83 that make it difficult to distribute a single Ada program across a distributed or shared-memory architecture. Examples of such impediments include:

    1. the exact semantics (behaviour and timing, including failure modes) of timed and conditional entry calls are not well defined in Ada83 for a single program distributed across a distributed environment.
    2. the exact semantics and available recovery from hardware failure is not well-defined in Ada83 for a single program distributed across a distributed environment;
    3. there is a single package SYSTEM for the whole program and this implies that all those items in SYSTEM are common to all parts of the hardware running the program. This includes STORAGE_UNIT, the number of bits in a storage unit.
    4. There is an implication that there is a single simultaneous CLOCK across the whole system and a fixed granularity to DURATION. Note that package CALENDAR may not be directly available throughout the system. The problem of simultaneity is related to the meaning of timed and conditional calls.

!Issue revision (2) compelling,moderate impl,upward compat,mostly consistent

2. Ada9X shall not preclude the distribution of a single program across a homogeneous distributed or shared-memory architecture.

!Issue out-of-scope (research) (3) important,severe impl,unknown compat,inconsistent

3. Ada9X shall not preclude the distribution of a single program across a heterogeneous distributed or shared-memory architecture.

[Rationale] There is no evidence that solutions to the various problems of dealing with different underlying representations of types in a heterogeneous environment are well-enough understood to design them into the language.

!Issue revision (4) compelling,small impl,upward compat,mostly consistent

4. Ada9X shall not preclude partitioning of a single program for execution on a distributed or shared-memory architecture. [Note: existing approaches based on link-time partitioning of program units shall remain viable.]

!Issue secondary standard (5) compelling

5. An associated standard should be established to specify how information regarding partitioning and allocation is to be made available to an Ada translation system targetting a distributed or shared-memory architecture. An Ada translation system not targetting a distributed or parallel architecture shall not be required to use this information.

!Issue out-of-scope (research) (6) compelling,moderate impl,moderate compat,mostly consistent

6. Ada9X shall support the explicit management of the partitioning of a single program for execution on a distributed or shared-memory architecture.

!Issue out-of-scope (research) (7) compelling,severe impl,moderate compat,mostly consistent

7. For a single Ada9X program partitioned for execution on a distributed or shared-memory architecture, Ada9X shall support the static allocation of the partitions to execution sites.

!Issue out-of-scope (research) (8) compelling,severe impl,moderate compat,mostly consistent

8. For a single Ada9X program partitioned for execution on a distributed or shared-memory architecture, Ada9X shall support the dynamic allocation of the partitions to execution sites.

!Issue out-of-scope (research) (9) compelling,severe impl,moderate compat,inconsistent

9. To support execution of a single Ada program on a distributed and shared-memory architecture, Ada9X shall support different scheduling paradigms in different parts of the program.

!Issue out-of-scope (un-needed) (10) desirable,severe impl,upward compat,mostly consistent

10. For the purposes of error reporting, load balancing, and load shedding, Ada9X shall provide accessible unique identification of threads of control in a single Ada program executing on a distributed or shared-memory architecture.

[Rationale] Why is this out-of-scope? First, because it does not work the problem of load balancing or load shedding. For either of these, it seems that you need the appropriate capabilities provided by the underlying run-time system; in such a case, it seems more logical that the run-time system would define how to identify tasks instead of the language. As for error reporting, the requirement fails to provide the correct capability--rather, it provides an abstract solution that might be used to solve the problem. One can imagine implementations satisfying the requirement which would not be useful for error reporting.

!Issue revision (11) important,moderate impl,upward compat,mostly consistent

11. Ada9X shall provide some manifestation testable in the language of whether a call to a particular subprogram of entry is remote, e.g. is the code being invoked resident in the same site of execution as the caller. Ada9X shall specify a model of the failure modes for remote calls in a distributed architecture and shall specify the semantics for each failure mode in the model.

!Issue out-of-scope (un-needed) (12) desirable,severe impl,moderate compat,inconsistent

12. Ada9X shall incorporate a model of virtual memory and physical memory protection into its execution model.

!Issue out-of-scope (un-needed) (13) important,severe impl,moderate compat,inconsistent

13. Ada9X shall incorporate a model of autonomous "producer-consumer-like" intertask communications between tasks executing in different execution sites.

!Issue secondary standard (14) compelling

14. (External Interface for Main Programs) An associated standard should be established to define the means for an Ada9X program to specify an external interface for the purposes of interprogram communications and synchronization. Interprogram communications shall defined in such a way as to promote type safety--that is, the sender and receiver should have the same interpretation of the data as Ada objects in so far as that is technically possible and feasible in implementation. The standard shall define the means for an Ada9X program to change its communications interconnections dynamically.

The standard shall define the means for an Ada9X program (the "invoking program") to cause another Ada9X program (the "invoked program") to begin execution in a locus of execution for which this execution is permitted. The standard shall allow the invoking program to specify some initial communications connections for the invoking program. The standard shall specify a means for one Ada9X program to signal another Ada9X program that the latter program is requested to terminate.

!Issue revision (15) important,small impl,upward compat,consistent

15. Ada9X shall support mechanisms to restrict the declarations in a package whose text is shared between two programs to only those declarations that do not imply the sharing of state between the two programs.

!reference

[Gargaro etal. 1989]
A.B. Gargaro, S.J. Goldsack, R.A. Volz, and A.J. Wellings
Supporting Reliable Distributed Systems in Ada9X
Proceeding of Distributed Ada 1989 held December, 1989
at the University of SouthHampton, pp. 301-330.

!reference RR-0071
!reference RR-0109
!reference RR-0182
!reference RR-0224
!reference RR-0372
!reference REFER ALSO TO RR-0374
!reference REFER ALSO TO RR-0375
!reference RR-0376
!reference RR-0377
!reference RR-0378
!reference RR-0661
!reference RR-0665
!reference RR-0723
!reference RR-0747
!reference WI-0401
!reference WI-0402

!reference WI-0402M
!reference WI-0403
!reference WI-0404
!reference WI-0404M
!reference WI-0405
!reference WI-0411
!reference WI-0414
!reference WI-0419
!reference WI-0419M
!reference WI-0420
!reference AI-00594
!reference REFER ALSO TO WI-0417

!problem

The problem is that users expect that Ada is a viable language for distributed processing and it is not. The specific reason why it is not depends on which of two "modes" is chosen to represent a distributed processing system. In one mode, a distributed processing system is represented as a collection of Ada programs. There are two subproblems here. The first is that a standard has not emerged for representing a system in this way; thus, the interface code for the individual components is not portable. The second problem is that the language does not provide any facilities to specify where Ada programs must be data-interoperable. Thus, extralingual mechanisms must be used so that portability is lost. Many of the problems for this mode are the same whether the multiprogramming occurs on a single host or on multiple hosts.

The second mode for representing a distributed system in Ada is as a single program. Here the problems are a bit different. First, the language provides no mechanism to bundle parts of the code into "allocatable units" (the partitioning problem), no mechanism to name the loci of execution, and no mechanisms to cause an allocatable unit to be allocated to a locus and started (the allocation problem). Various research groups are working on techniques for annotating a user's program with this information; as usual, each group does it differently. A second problem is that one might desire compiler support for enforcing restrictions in the interactions among allocatable units. A third problem is that there are certain "physical dependencies" that essentially prohibit distribution in various ways. For example, if there is a single package STANDARD defining a single STANDARD.INTEGER then it is difficult to understand how STANDARD.INTEGER should be defined in a distributed system in which the "natural" integer is thirty-two bits for some processors (like most micros) versus thirty-six bits for some processors (like some mainframes) versus twenty-five bits for still others (like the TI-CLM). This type of consideration is why heterogeneous architectures are perceived to be more difficult to support than homogeneous. Similarly, having only one STANDARD.CALENDAR in a program implies a single concept of clocktime throughout a distributed system; this is very difficult to achieve in practice.

Both the single program mode and the multiple program mode are actively being pursued by various groups as reported at Distributed Ada 1989. Unless the language wants to take a stand on the "correct" view of a distributed system, the problems of each mode need to be somehow addressed.

!rationale

Items RI-2101.1 through RI-2101.13 deal with the single Ada program approach to distributed programming. The concept of distributed programming in single-program mode has been well-studied for Ada, RI-2101.1 presents some of the specific points in the language that are at odds with distributed programming and asks for as many as possible to be solved. RI-2101.2 calls for the revision to take specific cognizance of the need to consider the effect of language design decisions on distributed programming so that the revision will not introduce any further anomalies for distributed programming in a homogeneous environment. RI-2101.4 is intended to protect the approach to distributed, single-program Ada that is being used now– link-time partitioning of the object files; it would be very serious if this method were rendered ineffective and nothing given to replace it.

One of the main problems in the single-program approach is that a different specification technique is being used to specify the partitioning and allocation information, even when a very similar model is involved. For portability reasons, the notation used should be unified across a number of projects; RI-2101.5 calls for the development of such a uniform notation. However, it seems premature to incorporate the notation directly into the language and require support from all compilers, even those not supporting a distributed or shared-memory architecture.

One significant difference between "normal" programming and "distributed" programming is that a distributed program may have different failure modes than a nondistributed one. In the current language, there is no manifestation of whether a call is local or remote even though local and remote calls may have different failure modes from the language point of view. RI-2101.11 would require the development of a model of the failure modes for remote calls and a specification of the semantics for possible failures. A program could test during elaboration to determine if the assumed configuration with respect to local/remote calls was correct. For example, if a program were written assuming that certain calls were local and therefore not including handlers for remote call failure modes, then it could check during elaboration to ensure that this assumption was valid.

RI-2101.14 and RI-2101.15 address the major issues when using the multiple program approach to distribution. RI-2101.14 addresses the portability problem that users are encountering when using the multiple Ada program approach. Candidates for a portable model include tuple-space (see RR-747) and typed channels. RI-2101.15 allows a programmer using the multiple program approach to share packages among programs without sharing state. This is important in large, multicontractor developments where the interface packages might be developed first so as to eliminate some downstream integration problems based on different contractor views on the representation and meaning of certain data types.

!appendix

RI-2101.15 would make an excellent addition to other package constraints as well as user-defined type and object constraints if there were such an RI.

The fault-tolerance aspects of WI-419 are referred to that RI, RI-1033. The task parameterization aspects of WI-419 are referred to RI-2012. The exception propagation and fault-tolerance aspects of RR-665 are referred to RI-1033. The same is true of RR-376.

%reference RR-0071 Improve support for multiprocessing

RR-0071 sees the issue of distributing in a monoprogram mode as full of unresolved questions. Nevertheless, two (allegedly) small impediments should be removed: single manifestation of SYSTEM characteristics throughout the program and single manifestation of time through CALENDAR throughout the program.

%reference RR-0109 Provide support for distributed processing of a single Ada program

RR-0109 sees three possible small signal changes for distributed monoprogrammed Ada: (1) detailed definition of the semantics of timed and conditional rendezvous for distributed environments, (2) having multiple manifestations of the underlying CPUs via STANDARD and SYSTEM, and (3) detailed definition of the semantics of hardware failure.

%reference RR-0182 Provide better support for distributed Ada programs

RR-0182 points out that most designs for distributed systems take the underlying distributed environment into account in that certain kinds of task interactions are methodically avoided for tasks to be assigned to different processors. The suggestion is to for the language to specify permissible implementation-defined limits on visibility between parts of a program running on different processors.

%reference RR-0224 Add communication support required for distributed systems

RR-0224 discusses the multiple program approach to distribution and notes that the industry is moving towards vendor-specific models and specification of the required interprogram communications and synchronization. The suggestion is for a portable standard solution.

%reference RR-0372 Solve prob where heterogeneous processors view memory differently

Ada does not provide a point mechanism to specify whether an underlying architecture is big-endian or little-endian. Rather, the specification is spread in rep-specs throughout the code. In order to solve this problem among heterogeneous (with respect to -endian-ness) machines a special protocol must be defined--the use of such protocols at the applications level can degrade performance. The Mach operating system has tools for dealing with this sort of problem (Matchmaker). This RR might be better categorized under RI-2034 on data interoperability.

%reference REFER ALSO TO RR-0374 (5.11.3)
%reference REFER ALSO TO RR-0375 (5.11.3)
%reference RR-0376 Distributed systems need propagation/identification of exceptions
%reference RR-0377 Ada does not allow partitioning of programs for distributed env.
%reference RR-0378 Need standard means of communication in distributed system

RR-037[45678] discuss various aspects of distributed processing as they pertain to
4) virtual memory.
5) memory protection and security.
6) propagation of exceptions across machine boundaries.
7) specification of partition and allocation
8) underlying communications support.
No specific solutions are offered.

%reference RR-0661 Need language features for assigning tasks to nodes

RR-0661 is concerned that there is no interlingual, standard way to specify the allocation of Ada-entities to execution nodes. The solution is to formally identify the nodes and utilize a new declaration ("PLACE entity IN node").

%reference RR-0665 Provide language support for needed Ada program distribution
RR-0665 provides an excellent discussion of essentially every aspect of Ada with distributed systems in a nonprogram mode. The four key points are (1) the need for partitioning/allocating support in the language, (2) the need for autonomous "producer-consumer-like" inter-task communications, (3) propagation of the concept of time in a distributed system, (3) propagation of exception conditions in a distributed system, (5) asynchronous events for fault-tolerance.
%reference RR-0723 Need PEARL-like approach to distributed Ada programs

RR-0723 suggests that Ada's tasking facilities promote a transparent implementation that leads to inefficiencies. The suggested solution is to replace Ada's tasking model with the facilities provided by Pearl-- a distributed programming language.

%reference RR-0747 Provide better support for "light-weight" parallelism (ala Linda)

RR-0747 points out that the Linda paradigm for distributed/shared-memory programming is much more elegant and efficient than what Ada has today. The Linda tuple space concept seems to be an excellent mechanism for integrating multiple main programs on multiple hosts for communications and synchronization. RR-0747 proceeds to suggest that the Ada tasking model be REPLACED by the Linda facilities--this suggestion does not distinguish between what may be a very sound idea (i.e. Linda for parallel/distributed processing) and what looks like not as good an idea (Linda for concurrent programming).

%reference WI-0401 Do not preclude dist of 1 Ada program over homogeneous nodes

The language shall not preclude the distribution of a single Ada program across a homogeneous distributed or parallel architecture.

%reference WI-0402 Do not preclude dist of 1 Ada program over heterogeneous nodes

The language shall not preclude the distribution of a single Ada program across a heterogeneous parallel or distributed architecture.

%reference WI-0402 Should not preclude dist of 1 Ada pgm over heter. nodes

%reference WI-0403 Do not preclude partitioning of 1 Ada program over dist system

The language shall not preclude partitioning of a single Ada program in a distributed or parallel system.

%reference WI-0404 Support the explicit management of a partitioned Ada program

The language shall support the explicit management of the partitioning of a single Ada program.

%reference WI-0404MDirect support is premature; Primitives ok

%reference WI-0405 Support explicit allocation of 1 partitioned Ada program

The language shall support the allocation of a partitioned single Ada program. The intent is to support both dynamic and static allocation.

%reference WI-0411 Provide remote procedure syntax/semantics (incl. failure sem)

The language shall explicitly support Remote procedure calls (with failure semantics).

%reference WI-0414 Need support for different scheduling paradigms in diff prog parts

The language, to support distributed and parallel systems, shall support different scheduling paradigms in different parts of the system.

%reference WI-0419 Provide accessible unique identification of threads of control

The language shall provide accessible unique identification of threads of control for a distributed or parallel system.

%reference WI-0419M Unique id of threads-of-control not appropriate in lang.

%reference WI-0420 Multiprogramming support of some type is needed in Ada

%reference AI-00594 Sensible preemption by high priority tasks

AI-594 is a very long AI discussing a proposed return to the idea of "sensible preemption" instead of immediate pre-emption as specified by AI-32. There is not much here about distributed processing except (1) the continued realization that priority cannot easily be propagated across machine boundaries (in many cases) and (2) a discussion about the desire to simulate the scheduling of a multiprocessor system on a single processor. The real problem is that there are a number of programs that rely on preemption as a synchronization mechanism. These programs may work on a uniprocessor but fail when ported to a multiprocessor.

%reference REFER ALSO TO WI-0417 (5.8.4)

In a distributed system, the language shall not require the same perception of time at all points in the system.

!rebuttal
This rebuttal is directed at the research classifications of [6 .. 8]

It is agreed that the general problem of partitioning a general program across a distributed target is a research problem. The Ada community today is attempting to do this partitioning. They need all the help that the language can give them today.

There are currently 3 ways being used to distribute a system using Ada. One can use tasks as the unit of distribution, packages, or multiple Ada programs. Tasks are potentially suitable for distribution across processors of a tightly coupled homogeneous system. Packages are potentially suitable for distribution across homogeneous systems sharing a distributed run time executive (usually tightly coupled, but could be loosely coupled with the proper executive). Multiple Ada programs are potentially suitable for homogeneous or heterogeneous systems, tightly or loosely coupled.

The multiple program approach is currently the most flexible and most widely used approach. It suffers drawbacks which many users wish to avoid by building their system as a single Ada program. (This is not a diatribe against the multiple program approach. Problems encountered here include less flexibility in the face of hardware changes, more

expensive configuration control, and loss of the benefits of the strong type checking present in a single Ada program)

The task-based and package-based distribution approachs do not give enough flexibility to provide distribution over hardware with any real differences. Perception of time (different tick rate or different calendar time), hardware differences (such as FPU, memory management, or processor differences), and inability to specify representation specifications which include processor information all make putting a single Ada program onto a distributed environment.

An achievable middle ground appears to be the concept of a partition mechanism different than a package. A single partition would include its own run-time environment, permitting differences in system.tick, system.name, FPU presence, and possibly processor differences (eg. 80386 and 8086 - many compilers today can generate code for both). The partition would be different from a package in that it could not export types, variables, or tasks, and that all units "belonging" to a partition could not be used by other units (not also belonging to the same partition). The interfaces would be strictly procedural. It is felt that enough is known about distribution, and partitioning a program that a useful specification of such a mechanism could be done. Such an approach, put into Ada today, would help those projects doing distribution.

!topic reference to a task outside its master
!number RI-1060
!version 1.5
!tracking 5.8.7

DRAFT DRAFT DRAFT DRAFT DRAFT DRAFT

!Issue revision () desirable,small impl,moderate compat,mostly consistent

No Task Reference Outside Master] It shall be an error in Ada9X to reference a task outside the lifetime of its master. Furthermore, an attempt shall be made to make this condition illegal (i.e., detected at compile time).

!reference RR-0104
!reference RR-0194
!reference AI-00167/04

!reference
Lomet, D.B., "Making Pointers Safe in System Programming Languages" IEEE Transactions on Software Engineering, Vol. SE-11, No. 1 (January 1985), pp. 87-96.

!problem

There is a special case in Ada83 in which a task can be accessed from outside the lifetime of its master. This one anomaly causes a run-time penalty in terms of either execution or storage overhead.

Typically, implementations represent task values by a pointer to a data structure (i.e., a task control block). This data structure is deallocated when exiting the master of the task. Consider, however, the following code:

```
...
task type T;

function F return T is
        T1 : T;
begin
        return T1;
end F;
...

if F'TERMINATED then ...
```

In this example, the return value of function F is a task declared local to the function; consequently the returned task depends on the function as its master, and the function may not exit until the dependent task T1 has terminated. If an implementation is to reclaim the storage for a task control block upon exiting the task's master scope, such reclamation will render the pointer to the result's task control block invalid prior to return from the function. It is the special handling required for cases such as these that

complicates implementations and causes run-time performance to suffer in general. For more details, please refer to RR-0104.

!rationale

This language difficulty does indeed upset very reasonable storage allocation strategies and is generally agreed by implementors to be a significant irritation.

If this problem is going to be solved, situations such as those shown above must not be permitted in Ada9X. If Ada9X rules can be such that these situations are always illegal without making other more-reasonable uses of tasks also illegal, this is clearly preferable to a new erroneous condition in the language (as suggested in the RRs) or a new situation of a predefined exception being raised at run time.

The subject of this RI can be viewed as an instance of the general problem of dangling references in programming languages that allow pointers to stack-allocated objects. Work in [Lomet85] indicates that compile-time rules are possible that avoid such dangling references and that are not overly restrictive. If similar rules can be integrated into Ada without adding significant complexity to the language and without imposing undo hardship on the programmer, this is clearly the preferred solution.

!appendix

There was a good thread of DR mail on this topic marked as:

> Subject: Re: RI-2011 Refs = dynamic renames
> !topic Refs; task outside of its master
> !reference RI-2011(1.6)

%reference AI-00167/04

This is an approved confirmation AI, that says, yes indeed, a task can be referenced outside the lifetime of its master.

%reference RR-0104 Prohibit access to a task outside its master

%reference RR-0194 Disallow referencing a task from outside its master

These two RRs are very similar. This one problem in the language has a negative (execution-time or storage) impact on performance, even if it is never taken advantage of by the programmer.

!rebuttal

!topic Library unit task termination
!number RI-2106
!version 1.1
!tracking 5.8.8

DRAFT DRAFT DRAFT DRAFT DRAFT DRAFT

!Issue presentation (1) important

1. (Termination of Library Tasks) The standard shall clearly specify that an implementation MAY abort library tasks if the main program is abandoned due to an exception but NOT if it completes normally.

!Issue out-of-scope (un-needed) (2) desirable,small impl,bad compat,inconsistent

2. (Forced Termination of Library Tasks) Ada9X shall be defined so that a library task is terminated whenever positioned at a terminate alternative.

!Issue out-of-scope (un-needed) (3) desirable,small impl,bad compat,inconsistent

3. (Forced Termination of Library Tasks with Main) Ada9X shall be defined so that a library task is terminated whenever positioned at a terminate alternative after the completion of the main program.

!Issue out-of-scope (un-needed) (4) desirable,severe impl,bad compat,inconsistent

4. (Changing Masters and Termination) Ada9X shall be defined so that whatever unit WITHs a library package becomes a master of the tasks in that package instead of the "environment" task.

!problem

In constructing a system, it is frequently convenient to have tasks be contained in library packages with the intent that these tasks terminate when the main program completes. Ada does not support this particular tasking paradigm.

!rationale

A tasking paradigm where library tasks terminate automatically when the main completes is fundamentally at odds with the paradigm of having a "vacated" main program and doing all of the work in library tasks. This latter paradigm is one frequently found in real-time systems since it precisely realizes the set-of-realtime-processes model of a realtime system. The maximum that can be done here is to explain in the standard that this is the semantics that has been chosen.

!appendix

%reference RR-0023 Require terminate alternative to terminate library tasks

RR-23 wants the language to specify that a library task positioned at a terminate alternative is in fact terminated.

%reference RR-0215 Clarify termination of tasks dependent on library packages

RR-215 wants to make sure that "library tasks" are not required to terminate when the main (sub)program completes.

%reference RR-0496 Clarify termination of tasks whose masters are lib. units

RR-496 wants the language to specify clearly the semantics of task termination of "library tasks".

%reference AI-00399Status of library tasks when the main program terminates

AI-399 says basically that an implementation MAY abort library tasks if the main program is abandoned due to an exception but NOT if it completes normally.

%reference REFER ALSO TO RR-0370 (5.12.6.2)

RR-370 wants to separate the compilation-dependency aspect of WITH from the elaboration control aspect of WITH. The idea here seems to be that whoever WITHs a package becomes a master of any tasks declared in the package, as opposed to the "external task" idea of AI-399.

!rebuttal

!topic Initialization/Parameterization of Types
!number RI-2012
!version 1.7
!tracking 5.3.7 5.8.9

DRAFT DRAFT DRAFT DRAFT DRAFT DRAFT

!Issue revision (1) compelling,small impl,upward compat,consistent

1. (Default Expressions) Ada9X shall provide a mechanism to incorporate a default expression into a type (or subtype) declaration so that objects of the type (or subtype) are automatically initialized whenever created whether by declaration or allocator, or as a component of another object.

!Issue revision (2) desirable,moderate impl,upward compat,consistent

2. (Self-Referential Initialization) Ada9X shall provide a mechanism whereby a reference to the object itself is available to the initialization operation for an object.

!Issue revision (3) compelling,moderate impl,upward compat,mostly consistent

3. (Parameterization of Task Elaboration) Ada9X shall provide a mechanism to pass parameters to a task that are available during the elaboration of the task; the types of these parameters should not be arbitrarily restricted. Ada9x shall not preclude the parallel invocation of the initialization operations for task objects that are components of an array, with parameters corresponding to the position of an individual task in the array.

!reference RR-0086
!reference RR-0129
!reference RR-0161
!reference RR-0230
!reference RR-0456
!reference RR-0506
!reference RR-0595
!reference RR-0649
!reference RR-0677

!reference RR-0123
!reference RR-0133
!reference RR-0334

!reference

[Knuth 1969] Knuth, Donald E., 1969, The Art of Computer Programming, Volume 1: Fundamental Algorithms, Second Printing, Addison-Wesley Publishing Co., Menlo Park, California.

!problem

There are two problems here: both have to do with initialization. The first is that there is no way for a provider of a (abstract data) type can ensure that the type is initialized to a correct initial state without relying on either user intervention or dynamic allocation. A serious subproblem of this exists when the structure needing initialization is self-referential (as for a circular, doubly-linked list [Knuth, 1969]) or when the initialization structure of components depends on the components position within an array. The second problem, obviously related to the first, it that it is difficult to parameterize tasks whose environment and behaviour are dependent on external parameters; again, user intervention or dynamic allocation or both is required to pass in the correct parameters. Frequently, an additional entry and rendezvous are required.

!rationale

The requirements follow directly from the need to solve the initialization problem for arbitrary types and the parameterization problem for tasks. It is an important methodological consideration that the providers of types be able to force them into a correct initial condition without client intervention. Passing initialization parameters to tasks via an initial rendezvous is error-prone since it requires that the initialization of a task be arbitrarily separated from its declaration.

!appendix

%reference RR-0086 Need to initialize record field to address of allocated record

RR-0086 wants to allow self-referential initialization as for circular doubly-linked lists.

%reference RR-0129 Allow initialization for all non-limited types

RR-0129 wants to be able to specify initialization as part of a type declaration for all nonlimited types.

%reference RR-0161 Allow initialization with any non-limited type

RR-0161 also wants to specify initialization as part of a type declaration; it further notes that this is essentially the same as LI58 of the SIGAda ALIWG.

%reference RR-0230 Allow initialization to be associated with any type definition

RR-0230 wants initialization in the type definition. It brings out the important aspect that default initialization should require no intervention by the declarer of the type.

%reference RR-0456 Allow initialization to be associated with a type definition

RR-0456 wants initialization in the type definition. It brings out the important aspect that increased distance in the source code between the type definition and the object declaration increases the chance of errors. (So too does the distance from declaration to initialization, if initialization is by assignment).

%reference RR-0506 Allow initialization to be associated with a type definition

RR-0456 wants initialization in the type definition, particularly scalar types.

%reference RR-0595 Allow default initialization for all types, attribute to know

RR-0595 wants initialization in the type definition. Interestingly, it also wants an attribute that tells if automatic initialization is specified for a type.

%reference RR-0649 Allow default initialization for all types (not just records)

RR-0649 wants initialization in the type definition.

%reference RR-0677 Allow initialization clauses on scalar type declarations

RR-0677 wants initialization clauses to be added to scalar type declarations.

%reference AI-00681 Can't declare a constant of a 'null' record type.

%reference RR-0123 Provide initialization values to tasks at startup

RR-0123 wants the concept of record discriminants to be extended to apply to tasks. This information would be used to uniquely identify tasks within a array during elaboration. Interestingly, the main concern expressed is for the inefficiency of the initial rendezvous, but the example trades this in for the inefficiency of dynamic allocation.

%reference RR-0133 Allow array task element to get its index

RR-0133 notes that there are some algorithms that could be represented concurrently by arrays of tasks but that the implementation of such tasks depends on knowing the tasks position in the array. An attribute-based solution is proposed.

%reference RR-0334 Need to supply initialization for a task object

RR-0334 wants to get parameters into a type at elaboration time. It mentions that having parameters to a task is more general than discriminants.

!rebuttal

!topic Representation Clauses Belong on Task Objects, not Task Types
!number RI-7003
!version 1.3
DRAFT DRAFT DRAFT DRAFT DRAFT DRAFT
!tracking 5.8.10

!Issue revision (1) important,small impl,upward compat,consistent

1. (Support for Ada Object attributes/Rep Specs) Ada9X shall provide mechanisms for the specification of representation clauses and storage allocation on a task object basis rather than a task type basis. Ada9X added features, such as (possibly) task attributes must also be applicable to an instance of a task type, not only to the type.

!Issue revision (2) desirable,small impl,upward compat,consistent

2. (User-defined defaults for tasking) Ada9X shall provide a mechanism to permit specification of default task attributes, such as storage size and priority to be applied on a program-wide basis.

!reference AI-00533
!reference AI-00596
!reference RR-0114
!reference RR-0195
!reference RR-0464
!reference RR-0648
!reference RR-0703
!reference RR-0171
!reference RR-0421
!reference WI-0309

!problem

Ada does not give the user sufficient control over his program when it comes to defining and creating tasks, and when connecting them to external hardware and interrupts. Specific problems are as follows.

a. Task entries used for interrupts cannot be specified for an individual task object. Since an interrupt almost always applies to a single task entry, this paradigm mismatch prevents the ready specification of arrays (etc.) of tasks to manipulate arrays (etc.) of devices.

b. Attributes, such as 'STORAGE_SIZE, and priority are applicable only to the task type, not the task object of a task type. Such specifications are often required to be applicable on a task object basis, but this is unachievable in Ada83 without unduly complicated workarounds.

c. Default tasking conditions, such as the default priority, and the default 'STORAGE_SIZE are defined by an implementation and cannot be modified without modifying the run-time sources, or putting explicit priority

and STORAGE_SIZE specifications on every task declaration.

Because of these difficulties, users are forced into unnatural and less general programming techniques. This often causes unacceptable code expansion due to replicated code and extended workarounds, or causes multiple layers of tasks to get the effect needed, resulting in more waste of memory and more difficult inter-task communications.

!rationale

[7003.1]
An equivalent situation to the "attribute applying to task type" problem existing for variables (such as pragma pack and representation specifications) was solved for Ada83 by permitting them to apply to subtypes, and to permit subtypes to be used in place of the parent. The rationale for fixing this problem is virtually identical. It must be possible to create arrays of tasks or other structures containing tasks which have different priorities, different storage allocated, and manage different hardware, but which execute the same code.

[7003.2]
Programs developed with one implementation in mind usually determine what the defaults for that implementation are, then work around the defaults. It isn't until porting to a new target or implementation/target, or until attempting to combine components built on different implementations, that the seriousness of uncontrollable defaults becomes evident.

!appendix

An example of the required functionality using interrupt entry representation specifications is an array of device drivers. In this case, each task algorithm is identical, but attaches to different physical devices. A minimally required approach is to permit a specification of the task object's entry representation specification clause to override the type-level specification. The other is to dynamically associate an interrupt with an entry for all tasks. The second approach gives less opportunity for compile-time checking for conflicts.

The representation of priority specification is achieved in Ada83 by a pragma which must occur within the task specification. There is no opportunity to declare a priority for a task object. If a more general priority management mechanism is created for Ada which takes into account individual task priorities, even for those sharing a common base type, such a mechanism should satisfy this requirement.

Storage Size
A workaround for this difficulty exists by creating a generic package with the task type inside it. This solution does not permit such tasks to exist as common objects in an array, or using common allocators. Should packages become first class types, then this solution may be viable.

Address Clauses

Current workarounds for entry address clause rep specs are to

1. create a unique type for each hardware device,
2. create a generic package with 1 task, or
3. create a parent task which creates a child task with the proper attributes.

Workaround 1 and 2 will not permit arrays of such tasks. Workarounds 1 and 3 (and 2 for most compilers) cause unnecessary code replication, and potential for divergence in maintenance.

Task Priority

The previous workarounds using generics or embedded structures are ineffective for task priorities because priority is a static expression.

%reference AI-00453 STORAGE_SIZE for Tasks

This AI points a ramification of the current storage_size representation clause rules which can cause major code shuffling when the default storage size for tasks changes (between releases or when porting programs).

%reference AI-00596 'ADDRESS for derived task types

This AI points out that for a derived task type, the 'ADDRESS attribute cannot be altered from that defined for the defining task type.

%reference RR-0114 Allow an address clause for each task instance not just the type

Many applications are required to handle interrupts from multiple identical devices. An interrupt handler in Ada is written as a task. The most natural way to write interrupt handlers for more than one identical device is to declare multiple objects of the same task type. This cannot be done as the address clause of an entry is associated with the type of the task rather than each object of the task type. Hence the address clause is evaluated once, when the type comes into scope.

%reference RR-0171 Separate program build-info from source

Move items such as storage_size into files separate from the source code.

%reference RR-0195 Need interrupt address per task, not task type

It should be possible to create multiple task objects of the same task type for multiple interrupting devices of similar kind. The number of device driver tasks and their binding to interrupts should be determinable at run time.

%reference RR-0464 Storage_Size should be applicable for task objects as well as types

The restriction of STORAGE_SIZE representation specifications places serious constraints on developers.

%reference RR-0648 Need 'SIZE on tasks, not task types

The restriction of STORAGE_SIZE representation specifications prevents developers from tailoring specific instances of a task type.

%reference RR-0703 Need storage_size on task object, not task type

The restriction on storage_size specification for task types only causes inappropriate type definitions.

%reference RR-0421 Interrupt handling and interrupt entry association have problems

This RR itemizes some of the problems in using interrupts and Ada task entries. A heretofore unmentioned problem is that, on some architectures, the address used to specify interrupts may not be compatible with system.address.

%reference WI-0309 Provide safe way to dynamically connect/change int and task entry

!rebuttal

!topic Support for periodic tasks
!number RI-7030
!version 1.6
!tracking 5.8.11

!Issue revision (1) compelling,small impl,upward compat,consistent

1. (Provide Periodic iteration) Ada9X shall provide a mechanism to permit a code segment to be executed periodically without drift or flutter.

!Issue out-of-scope (un-needed) (2) desirable,severe impl,upward compat,inconsistent

2. (specification of a periodic task) Ada 9X shall provide a mechanism to permit specification that a task is periodic, and to give the periodicity.

[Rationale] A generalized specification of "periodic" would generate a complete new task class within Ada. Its behaviour in rendezvous, priority, and possible restrictions make this language change expensive. Since all of the required functionality can be provided with requirement 1, it is judged out-of-scope.

!reference RR-0108 Provide a DELAY UNTIL (absolute time) mechanism
!reference RR-0306 Need to be able to delay until an absolute time
!reference RR-0410 Need better support for periodic tasks, delay stmt doesn't hack it

!problem

At present, Ada does not give any direct support for the concept of periodic tasks. Periodic tasks are ones which must begin an execution cycle at a predetermined rate; this rate must not drift or flutter. The only existing language construct is the delay <relative_time> construct. The calculation of <relative_time> is not atomic with respect to the delay statement, allowing the potential the significant time may pass between the evaluation of <relative_time> and the call to delay. The result is that developers requiring periodic tasking paradigms must reject Ada, or resort to customized services and non-standard implementations.

!rationale

The provision of such a facility is very important to the real-time community which needs to implement periodic tasking paradigms. The situation in Ada83 where no portable atomic means exists of calculating and setting the next iteration needs correction.

!appendix

The simplist means of providing such a facility is a "Delay Until <time>" statement, where the time is an absolute time. If such a mechanism is provided, it is felt that enough variety should be provided to specify an absolute time of day, down to the smallest resolution of the calendar clock.

%reference RR-0108 Provide a DELAY UNTIL (absolute time) mechanism

This ARTEWG submission makes a strong case for providing an absolute argument for a delay statement. It includes the discussion on the non-atomicity of the current delay statement, as well as concerns about the correctness relative delays when clock adjustments occur.

%reference RR-0306 Need to be able to delay until an absolute time

This RR requests that the problems with Ada83's delay <relative_time> be corrected for the real-time community.

%reference RR-0410 Need better support for periodic tasks, delay stmt doesn't hack it

It states reasons such as lack of atomic time-evaluation-delay-commencement, but also points out logic errors make a "delay until" construct more error-prone.

!rebuttal

!topic Need for Selective Accept Mechanism
!number RI-7040
!version 1.4
!tracking 5.8.12

!Issue out-of-scope (un-needed) (1) desirable,moderate impl,bad compat,mostly consistent

1. (Eliminate terminate alternative) The terminate alternative shall be removed from Ada9x.

[Rationale] Discussions with compiler vendors indicate that the relative penalty in the presence of the terminate alternative is very low for tasks and scopes enclosing tasks which do not use terminate, and should become nil as compiler/run time technology improves. In addition, there are formal cases in the language where a reference to a task is impossible, and the terminate alternative is the only way to make it completed.

!Issue revision (2) desirable,small impl,upward compat,consistent

2. (Coexistence of terminate and delay) Ada9x shall not prohibit the existence of the terminate alternative and the delay alternative in the same select statement.

(This requirement should percolate up to RI-5100, inconsistencies.)

!reference RR-0079 Terminate alternative adds little value and is rarely used
!reference RR-0431 Terminate alternative is not really usable
!reference RR-0612 Should allow both delay and terminate alternatives in selective wait
!reference LSN-005
!reference LSN-269

!problem

There is a perception in the user community that the terminate alternative is not useful, and that a great deal of the synchronization-point overhead comes from attempts to evaluate termination rules.

!rationale

Although the terminate is potentially the only way of stopping some tasks in a scope, its use is impaired because it cannot be used together with the delay statement, even if the delay statement has a guard which makes the presence of both mechanisms simultaneously impossible.

!appendix

%reference RR-0079 Terminate alternative adds little value and is rarely used

This RR states that the terminate alternative adds complexity to the Ada runtime, but affords very little in program control; construct not frequently used.

%reference RR-0431 Terminate alternative is not really usable

This RR states that the terminate alternative adds complexity to the Ada run time. It also gives a demonstration of the difficulty in use of tasking where both a terminate and a delay are used.

%reference RR-0612 Should allow both delay and terminate alternatives in selective wait

This RR asks for the ability to include terminate alternatives and delays in the same select statement.

%reference LSN-005

This language study note explains why the terminate alternative was felt essential for Ada. The basic discussion is that active tasks may be inaccessible through other means, meaning that the scope can never close unless a language-defined terminate rule exists.

%reference LSN-269

This study note discusses among other things, the inclusion of the terminate alternative at a DR meeting.

!rebuttal

!topic shared variables
!number RI-1040
!version 1.8
!tracking 5.8.13

DRAFT DRAFT DRAFT DRAFT DRAFT DRAFT

!Issue presentation (1) important

1. (Pragma SHARED Means Always Load/Store) Ada9X shall clarify that every reference to a pragma SHARED variable must always correspond to a single load or store of the variable. If the LRM 11.6 restrictions on allowed optimizations and reorderings are relaxed, it should be made clear that the relaxed rules do not apply to pragma SHARED variables.

!Issue revision (2) desirable,small impl,upward compat,mostly consistent

2. (Nonatomic Version of Pragma SHARED) Ada9X shall provide a mechanism for declaring that local copies should not be used for a particular shared variable (i.e., every reference must correspond to reading or writing the true variable) without implying that atomic access/update to the variable (or its subcomponents) is required.

!Issue revision (3) important,moderate impl,upward compat,consistent

3. (Pragma SHARED Effect for Components) In Ada9X it shall be possible to obtain the effect of pragma SHARED (as clarified by RI-1040.1, above) for each scalar or access type subcomponent in a composite object.

!Issue revision (4) important,moderate impl,moderate compat,mostly consistent

4. (Sharing Packed Composites) The circumstances under which different components of a shared composite object may be accessed independently by different tasks shall be defined by Ada9X in a manner that does not require explicit synchronization code for correct implementation on common processors. [Note: Ada83 does not meet this requirement for "packed" composite objects. This statement is based on the present unapproved status of AI-00004.]

!Issue revision (5) desirable,moderate impl,moderate compat,consistent

5. (Means to Distinguish Shared Variables) Ada9X shall provide a mechanism that the programmer can use to distinguish variables that are accessed by more than one task from variables that are only accessed by a single task.

!reference RR-0119
!reference RR-0434
!reference RR-0515
!reference RR-0521
!reference RR-0544

!reference RR-0678
!reference WI-0312
!reference AI-00142
!reference AI-00004

!reference
Dewar, R., "Shared Variables and Ada 9X Issues"; SEI Special Report SEI-90-SR-1, January 1990.

!problem

There are a variety of problems in Ada83 on the topic of variables shared among tasks. They are identified individually as follows.

[RI-1040.1]
The exact effect of applying pragma SHARED to a variable is not clear from the wording of LRM 9.11.

[RI-1040.2]
In some circumstances, the access/update must-be-indivisible aspects of pragma SHARED are not required (and might not be possible). These situations arise when the logic in the program is such that access/update to the shared data is always done on an exclusive basis. This is typically the case when polling on other shared variables is used to control access to the shared data. Refer to [Dewar90] for an example of this.

[RI-1040.3]
Pragma SHARED cannot be applied to a component of a shared record or array object. For a shared memory buffer in real-time systems, e.g., it is frequently necessary to be able to get the effect of pragma SHARED for each of the components in the buffer. This problem is described in detail in RR-0119.

[RI-1040.4]
Ada83 allows different tasks to have independent access to distinct components of a shared composite object. For "packed" composite objects where distinct components could lie within the same word of memory, implementing this independent access would require (for many processors) the insertion of synchronization code by the implementation. This is clearly not the intention behind shared variables in Ada. Furthermore, since it is very often not clear to a compiler which packed composite objects are being shared between tasks, synchronization code would be inserted by an implementation in many cases when it was actually unnecessary.

[RI-1040.5]
That the language does not provide a mechanism for marking shared variables is unfortunate since it can sometimes force compilers to make the pessimistic assumption that a variable is shared when in fact it is not shared. On certain distributed architectures, this can lead to inappropriate allocation of the memory for a variable, unnecessary cache flushes, and so forth. To work around these difficulties, implementations of Ada for distributed architectures have provided implementation-defined pragmas that give guidance to the compiler about how variables will be accessed

among tasks. Defining such pragmas in the language would promote portability. More on these problems can be found in [Dewar90].

!rationale

[RI-1040.2]
A "toned-down" version of pragma SHARED, with all of its properties except atomic access/update, seems easy to add to the language (possibly with an additional argument to the existing pragma SHARED). This would allow what appear to be perfectly reasonable uses of shared variables that technically cannot be programmed correctly in Ada83.

[RI-1040.3]
The problem described above concerning shared composite objects appears to be genuine. It seems arbitrary that the effect of pragma SHARED cannot be obtained for the components of shared composite object. It may make sense to restrict this to composite objects created by declarations and not by allocators. The above requirement is intended to be vague in this regard.

[RI-1040.4]
In a sense this requirement is too specific in that if the language were changed to require that shared variables be marked, it would be much more reasonable to expect implementations to insert synchronization code when accessing/updating components of shared packed composite objects. However, it must be noted that (1) such a language change is very undesirable for upward-compatibility reasons and (2) synchronization in Ada is only intended for the rendezvous. Hence the requirement is worded in terms of not requiring an implementation to use synchronization to implement shared variables.

[RI-1040.5]
For distributed systems with, e.g., local and global memory sections or non-coherent memory caches on individual processors, it is important for an Ada implementation to be able to distinguish variables shared among tasks from variables not so shared. This distinction can be critical in the way variables are allocated in memories, the need for cache flushing for particular variables, and the placement of variables with respect to cache lines. What is needed to solve this problem in a portable way is a mechanism in the language that can be used by the programmer to mark shared (or non-shared) variables. The requirement above is worded in such a way that does not imply that all shared variables must be marked. This would be a non-upward compatible language change but is clearly the right solution to this problem if the language were being defined from scratch. Providing a mechanism in the language that the programmer can optionally use to mark shared variables is a less ideal but upward-compatible solution. The requirement above is also vague about whether the mechanism provided in the language works for objects created by allocators or just to those created by declarations.

!appendix

Distinguished Reviewer Norm Cohen recently made some good comments on the Dewar paper in DR e-mail. Some of those are relevant to this RI and in particular to Item [2]. They include:

1. Is a read or write to a pragma-shared variable a synchronization point for that TASK or for that VARIABLE? If its for a task, [2] may be unnecessary.
2. More generally, is there a need for programmer-defined synchrnonization points?

There has been no time for the RT to re-visit this RI since these remarks on the Dewar paper were made.

An outstanding issue here has to do with memory mapped I/O and was initially addressed as follows:

> %requirement 1 2 3 1
> Ada9X shall provide a mechanism for declaring that references to a particular variable must touch only the bits that have been set aside to represent the variable (i.e., must not touch "surrounding" bits for the sake of convenience or efficiency).

This needs more thought. Another outstanding issue is associating something like pragma SHARED with a record *type* so that the effect of pragma SHARED could be obtained for the components of objects of the type created by an allocator. Again, more thought needed.

%reference RR-0119 Need synchronized reference to elements of shared composite objects

Would like to be able to tell the compiler about a shared composite object. The effect would be that for references to the scalar and access subcomponents of the object, the no-local-copy, must-load-store, must-be-atomic rules would be used. Do not want synchronization; just want to inhibit optimization and cacheing.

%reference RR-0434 Need atomic read/write operations on shared volatile memory
??

%reference RR-0515 Need indivisible update for shared data

Desires guarantee of atomic updates to certain objects. May be willing to live with this capability for scalar and access types only. Does not want to have to program synchronization to accomplish the atomic updates. May want the synchronization to be inserted automatically by the implementation. Given that, it is unclear why he distinguishes scalar and access types from others.

%reference RR-0521 Need synchronized reference to shared composite objects

Many systems are best designed by using shared memory for communication. Ada makes this difficult by requiring explicit task synchronization for all use of shared memory (other than that allowed by pragma SHARED). Furthermore, if the application uses a "lock"/"unlock" discipline using a central task to manage the lock but not to actually perform the operation, then it is difficult to ensure that the object will be unlocked, and to efficiently compose subprograms each of which needs exclusive access.

%reference RR-0544 NEED SHARED on record components, indivisible increment/decrement

Would like efficient increment/decrement for components of objects of record types. In particular, sharing support for fields of objects created by allocators.

%reference RR-0678 Pragma SHARED is not sufficient; need pragma VOLATILE

For sharing among tasks, need to able to say (1) every read or write of this variable or this record component must reference memory immediately and (2) the variable must be correctly represented in memory at this point in execution.

%reference WI-0312 Need a volatile access concept for objects; atomic updates

Need a way to indicate that certain objects are arbitrarily volatile with respect to the Ada program, so each read and write in the source must be mapped to a single read and write of the memory that represents the object. Where feasible, an implementation shall provide atomic read and write operations, and shall provide some mechanism to warn of situations that will be non-atomic.

%reference AI-00004 Packed composite objects and shared variables

There is an implementation problem with 9.11(4) in that it always allows the individual fields of a composite object to be accessed independently by two tasks, regardless of the representation of the object. This presents a problem with packed composite objects.

%reference AI-00142 Proposed solution to packed composite object and shared variable problem

By including all types (not just scalars and access types) in LRM 9.11(4,5), and by allowing pragma SHARED on composite objects, the AI-00004 problem can be solved.

!issue Asynchronous Transfer of Control
!number RI-0003
!version 1.6
!tracking 5.8.14.1.1

!Issue revision (1) compelling,moderate impl,upward compat,inconsistent

1. Ada9x shall provide a method to terminate the execution of a sequence of statements in a controlled way upon the receipt of a signal. After this sequence of statements has terminated, it shall be possible to begin executing an alternate sequence of statements in the same thread of control without re-elaborating the enclosing task.

!Issue revision (2) compelling,moderate impl,upward compat,mostly consistent

2. Ada9x shall provide a mechanisms to insure that data structures, locks, and resources are released when a sequence of statements is terminated as a result of [1].

[Since the sequence of statements can be terminated at any point during its execution by the receipt of a signal, it is important to finalize this sequence when it is terminated, see RI-0002]

!reference RR-0083
!reference RR-0196
!reference RR-0106
!reference RR-0384
!reference RR-0768
!reference RR-0742
!reference RR-0710
!reference AI-00450
!reference RI-0001
!reference RI-0002 (Task Finalization)
!reference
[Quiggle 89] Thomas J. Quiggle, "Ramifications of Re-introducing Asynchronous Exceptions to the Ada Language," TeleSoft, 3rd International Workshop on Real-Time Ada Issues, 1989.
!reference
[Adaletters 88] "International Workshop on Real-Time Ada Issues," Moretonhampstead, Devon, UK, 1-3 June 1988, Ada LETTERS, Vol VIII, Number 7, Fall 1988.

!problem

To accommodate mode changes and interrupts in embedded systems, it is often necessary to terminate one computation in favor of another. The need for this operation also occurs as a result of external interrupt, timer interrupts, and others.

Basically, the desire is for a method to handle asynchronous transfer of control without having to pay the costs of task creation and termination for each transfer of control or make task creation and termination inexpensive enough that the cost becomes acceptable.

!rationale

The current workaround requires the use of tasks and the ABORT statement. This workaround is considered too costly by the real-time community, and any dependence on the creation and destruction of tasks is also considered too costly. For example, nested tasks withing the task to be aborted have a significant cost on termination and re-elaboration.

Three proposals exist in the references that would solve this problem [RR-0083, RR-0768, and RR-0196]. Although these proposals are widely different, they distill one common need: there is a need to specify that a sequence of statements is to be executed under the condition that it may be terminated before it completes and started again (from the beginning) at a later instant in time.

Given that this is the true need, then the revision request concerned with the "safe shutdown" of task when aborted will also apply to the sequence of statement that may be terminated before it completes. Thus, the second requirements calls for similar mechanisms to facilitate the safe shutdown of a sequence of statements used in this fashion.

Furthermore, RI-0001 is applicable to the termination of the computation. Namely, the termination should occur as soon as possible, and for embedded applications, the termination should occur in a bounded and finite amount of time.

!appendix

The current workaround requires that the thread of control which may be terminated exists within a task and there must be another task, the controller, which waits for the appropriate signal or delay. When a signal arrives which requires a transfer of control, the controlling task must abort the task currently holding the thread of control and startup an alternate thread of control (through another task or using its own thread). This is assuming that the abort statement will result in immediate termination of the aborted task. An additional cost will no doubt be paid by restarting the aborted task so that it will be ready to begin a calculation on the next iteration of the controller.

```
task A;
  ...
  controller.startup(...);
```

```
 ...
 -- thread of control that may be terminated
 ...
 controller.fini_signal;
 ...
end A;

task controller;
...
accept startup(...); --for task A;
...
select
 accept fini_signal;
  -- task A finished on time
  ...
or
 delay 10.0*seconds;  --external time signal
  abort A;
  take_alternate_action;
  --might have to restart A for the next iteration
end select;
...
end controller;
```

It should be noted that if task A is inexpensive to elaborated, then restarting the task should not be difficult or time consuming provided A has no dependent subtasks. Furthermore, A would have to abort all dependent subtasks which, in Ada83, it cannot.

The proposal made in RR-0083 and RR-0196 would add another form of selective wait to the language. In analyzing that proposal several issues are raised: 1) it complicates the rules for selective waits, 2) it is defined similar to abort, i.e., the transfer of control cannot occur later than the next synchronization point which may be the end of the select statement. The advantage of this proposal is that it avoids the cost of aborting and recreating a task to carry a thread of control (task A in the workaround example), and it is easy to understand the programmers intent. Several questions must be answered: what happens to resources used by the code when it is terminated and in a similar line, what happens to tasks that may have been allocated (dependent tasks)?

Another possibility for implementing this requirement would be to provide an "abort exception." The abort exception would be raised in the task that is to be aborted. This task could then handle the exception by terminating or by branching back to its initial state where it would wait for another rendezvous to being processing. The upward compatibility of this new exception is questionable in cases where the "others" exception handler is used. Further problems arise when more than one exception is raised: 1) one or more exceptions could be lost, 2) reduced ability to perform optimization, 3) the aborted task could be blocked on an entry, select or accept, 4) will the exception be propagated to other tasks via the rendezvous, 5) and many others (see RR-0196, RR-768, and [Quiggle89]). Another variation of the asynchronous exception is a new type of task entry proposed in RR-768

Still another possibility is to check point and restart; this requires strict coding standards and can be difficult to maintain.

Basically, the desire is for a method to handle asynchronous transfer of control without having to pay the costs of task creation and termination for each transfer of control or make task creation and termination inexpensive enough that the cost becomes acceptable.

The language currently provides this facility with some limitations; however, the cost of implementing solutions with the language is seen as prohibitive by the user community. The current language requires that the thread of control which may be terminated exists within a task and there must be another task, the controller, which waits for the appropriate signal or delay. When the appropriate signal arrives the controlling task must abort the task currently holding the thread of control and startup an alternate thread of control (through another task or using its own thread). This is assuming that the abort statement will result in immediate termination of the aborted task [RI-0001]. An additional cost will no doubt be paid by restarting the aborted task so that it will be ready to begin a calculation on the next iteration of the controller. There is no problem in efficiently implementing this scheme in Ada provided there are no dependent tasks that must also be aborted. Task finalization could resolve the dependent task problems [RI-0002].

%reference RR-0083 Provide asynch transfer via entry call/selective wait construct.

There are many situations where it is useful to terminate on computation and start a new computation. Asynchronous exceptions are not recommended because of a number of problems; rather a new form of selective wait construct is proposed.

%reference RR-0196 Need asynchronous transfer of control (like RR-0083)

There is a need for asynchronous transfer of control. Asynchronous exceptions are not recommended. Instead use the solution in RR-0083.

%reference RR-0106 Provide asynch transfer

The solution to asynchronous transfer of control should:

1. stop a task immediately (ideally, preemptively);
2. force a task to resume on a different control path;
3. resume right away, or later, under programmer control;
4. be very fast, so that it does not add to processor load;
5. be repeatable without bound.

%reference RR-0384 Can't write subprogram which cause an exception after a specified delay

The user needs to implement a special timer that raises an exception if a given computation has not occurred with in a given amount of time.

%reference RR-0768 Need asynchronous interrupt followed by rendezvous

Some users needs a method to interrupt one task from another task and exchange data during the interrupt.

%reference RR-0742 Need software interrupts, "events", etc. for asynchronous control

Asynchronous communication and control is desirable in many parallel applications. The ability to Terminate a sequence of statements and set up for the next iteration of an algorithms is often needed.

%reference RR-0710 Need to tie task entries to asynchronous external events

There is a need to associate task entries with external events the may occur asynchronously to the execution of an Ada program, e.g., I/O request, timers, communications, ...

%reference AI-00450 Should allow raising of an exception in another task

%reference RI-0001 Termination of tasks before Synchronization Points

%reference RI-0002 Task Finalization

!rebuttal

!topic Asynchronous Communication
!number RI-0005
!version 1.7
!tracking 5.8.14.1.2

!Issue revision (1) compelling,severe impl,upward compat,inconsistent

[1] Ada9x shall provide mechanisms for the asynchronous communication of data between tasks.

!reference RR-0183
!reference RR-0587
!reference RR-0655
!reference WI-0409 Parallel and Distributed Systems WG
!reference WI-0410 Parallel and Distributed Systems WG
!reference
[Haberman and Nassi] "Efficient Implementation of Ada Tasks," Technical Report, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania, 1980.
!reference
[Hilfinger] "Implementation Strategies for Ada Tasking Idioms," Proceedings of the AdaTEC Conference on Ada, October, 1982, Arlington, Virginia, 26-30.

!problem

Users would like a portable and efficient way to specify asynchronous communications between tasks.

!rationale

Although the language does not directly supply an asynchronous message passing mechanism, it does provide the basic tools to specify and implement this mechanism. The problem is that the current workarounds require the use of a surrogate task to accept the message from the source and pass it on to the destination. This surrogate can be in the form of a task protected queue or multiple tasks that queue on an entry in the destination. Thus, one asynchronous communication requires at least two rendezvous. The main cost penalty of the tasking method is the two context switches it MUST perform to place a message into the queue while the semaphore method will only perform the context switches if blocked at the semaphores. There are current compiler optimizations that will recognize and modify the surrogate task so that it is equivalent to the low-level semaphore implementation [Haberman and Nassi; Hilfinger]. Since many users are unwilling to pay the cost penalty and do not have such optimizing

compilers available, asynchronous communication is being performed in numerous nonportable ways. Ada9X users should not rely on optimization techniques to implement these functions efficiently. Furthermore, the current method is difficult to maintain and read because the tasks communicate via a surrogate.

!appendix

Although this is an important need, there is no OBVIOUS method for adding this feature directly to the language without having an adverse impact. Thus, the implementation and model ratings are low; however, the user need is very high.

The question is, should a user be forced to represent his program so that efficiency and portability can only be achieved by using compilers that implement complex inter-procedural optimizations? We believe the answer is no; however, this is one of the tradeoff of high-level languages. There is a method to avoid the dependence on the compiler's ability to optimize without modifying the language, that is to rely on the compiler vendor's ability to write efficient code. Provided that a method for adding this feature directly to the language is not discovered and supplemental standards are allowed and supported, then synchronization is one area to be considered for supplemental standards.

%reference RR-0183 Asynchronous inter-task communication is not available
%reference RR-0587 Provide loosely coupled intertask communications, or asynchronous communications
%reference RR-0655 Add asynchronous message queues
%reference WI-0409 Provide asynch send w/o ack, including failure semantics
%reference WI-0410 Provide asynch rec w/o ack, including failure semantics

!rah ‘?q!

[NOTE: This has been changed to !Issue revision as recommended by the DRs.]

[for !Issue revision instead of !Issue secondary standard]

It is my opinion that the need for intertask communication without explicit synchronous communication is pervasive. Fine-grain parallelism, loosely coupled multiprocessors, and fault-tolerant systems all need the ability to decouple the synchronization of tasks from the communication aspects.

The current Ada model has a fairly complete paradigm for synchronous multitask interactions using the rendezvous. Select mechanisms with delay and code execution

alternatives, parameter passing, entry guards, and critical regions in accept code blocks all provide abstraction levels and expressive power which would be unobtainable had these mechanisms been pasted on using a collection of library services.

It is inconceivable that the introduction of an asynchronous mechanism would not obtain the same level of support given to the existing model. A task must be able to effectively use asynchronous communication in harmony with the existing capabilities, such as selecting to accept a rendezvous, a message, or delay. Without compatible mechanisms, a designer's use of the constructs and language will be severely constrained, and her expressive power will be limited.

!topic Terminating Tasks before Synchronization Points.
!number RI-0001
!version 1.8
!tracking 5.8.14.1.3

!Issue implementation (1) compelling

1. Implementation dependent timing and inter-process communication properties of ABORT shall be defined in the Ada9x standard.

!Issue presentation (2) important

2. Ada9x shall make clear that the ABORT statement shall abort the designated tasks as soon as possible.

!Issue out-of-scope (un-needed) (3) compelling,severe impl,bad compat,inconsistent .sp
3. Ada9x shall provide a mechanism that terminates a task within a predetermined amount of time.

[Rationale] It is not feasible to place finite bounds on the amount of time needed to terminate a task. For example, some operating systems do not have the facilities to implement this semantic, and in the most general case of distributed processing with priorities, it is not possible to guarantee the amount of time required to terminate a remote task because of the communications involved.

!reference RR-0335
!reference RR-0063
!reference RR-0083 related
!reference RR-0196 related
!reference RR-0384 related
!reference RR-0106 related
!reference Ada83 9.10

Note: Some of these RRs ask for termination of a segment of code, not just a task (see Stopping a Computation RI-0003).

!problem

Abort statement is loosely defined in Ada83. The abort does not have to occur until the aborted task reaches a synchronization point [Ada83, 9.10]. In the worst case, the aborted task can consume unbounded amounts of processor time and resources between the time the abort is issued and the time the task actually terminates.

!rationale

There is a need, in time critical environments, for a language construct to abandon a computation immediately (within a predetermined amount of time), and there are only vendor specific workarounds. In the most general case of distributed processing with priorities, it is not possible to guarantee the maximum amount of time required to terminate a remote task because of the communications involved.

As a result, implementors should be required to report the method used to abort tasks including the maximum time required to abort a task after the execution site receives notification that a task executing on its resources should be aborted. Further, the communication between distinct processors should be described in enough detail so that the user can determine the effect of an abort, e.g., what is the priority given to an abort message when it must compete for the bus in a real-time system? In this way, a system developer can purchase a compiler with the right properties for his application.

!appendix

For a single processor implementation, this should include the maximum time required to abort a task. For a multiprocessor or distributed implementation, this should include a description of the communications involved with the abort and the maximum time required to abort a task after the processor it executes on receives notification of the abort. The maximum time does not have to be stated if that time is not bounded, e.g., systems that wait for the next synchronization point cannot bound the time to abort.

%ref RR-0335 Abort stmt too vaguely defined, too many special cases. A more stringent definition of ABORT is needed. abort immediately is preferred over abort at the next synchronization point.

%ref RR-0063 Make use of abort statement user-switchable

The inclusion of an abort statement makes it difficult to prove anything about program states, deadlock states, etc. Because the allowable time lapse before termination is not defined by the language, the abort statement may be inappropriate for critical function. All of the space is not reclaimed after a task has aborted. Abort finalization is also needed.

%ref RR-0083 (related) Provide asynch transfer

There are many situations where it is useful to terminate on computation and start a new computation. Asynchronous exceptions are not recommended because of a number of problems; rather a new form of selective wait construct is proposed.

%ref RR-0196 (related) Provide asynch transfer

There is a need for asynchronous transfer of control. Asynchronous exceptions are not recommended. Instead use the solution in RR-0083.

%ref RR-0384 (related) Provide asynch transfer

%ref RR-0106 (related) Provide asynch transfer

The solution to asynchronous transfer of control should:

1. stop a task immediately (ideally, preemptively);
2. force a task to resume on a different control path;
3. resume right away, or later, under programmer control;
4. be very fast, so that it does not add to processor load;
5. be repeatable without bound.

!rebuttal

!topic Safe Shutdown of Tasks
!number RI-0002
!version 1.9
!tracking 5.8.14.1.3

!terminology

In this RI we will use the term "running" to describe a task the has successfully completed its "activation" and has begun executing the sequence of statements associated with its body [LRM 9.3].

There are several conditions that will cause a "running" task to finish its execution [LRM 9.3]: 1) when it finishes executing the sequence of statements associated with the task body, 2) when it receives an exception without a corresponding handler, 3) when it receives an exception and completes the execution of the corresponding handler, 4) when it is aborted, 5) or when it is terminated by an open TERMINATE alternative.

After "running" task is finishes its execution, it is marked as either "terminated" if the cause is an open TERMINATE alternative, or "completed" if the cause is not an open TERMINATE alternative [LRM 9.3].

!Issue revision (1) compelling,moderate impl,upward compat,mostly consistent

1. (Finalization) Ada9x shall provide a method for specifying a finalization sequence for all tasks. These finalization sequences are to be executed immediately after a "running" task finishes its executions but before the task is marked as either "completed" or "terminated."

!Issue revision (2) desirable,moderate impl,upward compat,inconsistent

2. (protected regions) Ada9x shall provide a mechanism to designate a segment of code and a duration so that a task may not be aborted while executing the segment as long as it exits the segment within the duration specified.

!reference RR-0335
!reference RR-0063
!reference RR-0770
!reference RR-0651
!reference RR-0203
!reference AI-00581
!reference AI-00360
!reference RR-0083 (related)
!reference RR-0196 (related)
!reference RR-0384 (related)
!reference RR-0106 (related)
!reference RR-0768 (related)
!reference RR-0742 (related)

!reference RR-0710 (related)
!reference RR-0370 (related, library tasks)
!reference AI-00399 (library tasks)
!reference RI-0003 (Stopping a Computation)
!reference RI-0001 (Abort)
!reference RI-2022 (Finalization)
!reference
Quiggle, Thomas J., "Ramifications of Re-introducing Asynchronous Exceptions to the Ada Language," 1989.
!reference
International Workshop on Real-Time Ada Issues, Moretonhampstead, Devon, UK, Ada Letters, Volume VIII, Number 7, Fall 1988

!problem

The current methods to terminate a task are the ABORT statement, the TERMINATE alternative, or completion of the body; however, the ABORT does not guarantee "safe shutdown," the TERMINATE does not allow the task to transition to a safe state, and the completion of the body does not guarantee a safe state when an unhandled exception is the cause. In most cases, finalization is needed to properly release data structures, locks, and resources held by a task.

In practice, the ABORT may occur at any arbitrary point before or during the elaboration and execution of a task. The only way to guarantee the safe state of the system is to either prohibit the ABORT from interrupting a task in particular critical regions of the code or provide safe startup and finalization for the tasks. Examples of actual and potential problems can be found in [RR-0335, RR-0063, RR-0203, AI-00581, AI-00360]. Without safe shutdown, it is difficult to prove anything about the program or guarantee the state of a program when the ABORT statement is used.

In the case of the open TERMINATE alternative, the user may have to force the system from a known state into a safe state, and there is no mechanism provided.

!rationale

A portable method that allows the safe termination of a task when the task completes would increase the reliability and maintainability of programs. This would decrease the risk of errors, increase the safety of programs, and give the programmer more control over resource utilization. Task finalization provides a method to properly release data structures, locks and resources held by a task.

[1] Finalization is needed to bring the system from a potentially unsafe state into a safe state after an ABORT and after an unhandled exception. Finalization is also needed to enter a safe state from an open terminate alternative. [If finalization is provided for other Ada entities (e.g., packages, objects, or types) then the finalization for these entities must also be executed.]

A task should NOT execute its explicit finalization sequences if it does not activate

correctly because of an exception or abort during elaboration. If an exception or abort occurs during elaboration, the parent receives a TASKING_ERROR and the task is marked "completed" [LRM 9.3]. If finalization is provided for other Ada entities (such as packages, objects, or types), then any declarations that were successfully elaborated during the task activation process should be finalized.

[2] Protected regions are beneficial in when a task is managing critical resources and establishing check-points. Protected regions should take a "structured" form. Namely, this mechanism should be in the form of protected procedures, functions, selects, or block statements, and NOT if the form of and on/off switch [RR-0063] controlled by a pragma or procedure because of the potential for errors.

!appendix

Asynchronous exceptions of the form FINALIZE [Ada80] are not recommended because of the potential for multiple exceptions and race conditions.

%reference RR-0063 Make use of abort statement user-switchable

It is impossible to guarantee a safe state or predict the behaviour of a program that uses abort. Providing a pragma to turn off/on the effect of an abort statement would solve these problems.

%reference RR-0335 About statement too vaguely defined, too many special cases, is useless. Restrict the definition of abort.

%reference RR-0651 Allow one task to raise an exception in another task.

The only method to stop a task is through the abort statement. This guarantees nothing since the task may not reach a synchronization point, and it does not allow the aborted task to perform any cleanup actions. Asynchronous exceptions would solve this problem.

%reference RR-0770 Make aborting yourself cause instant completeness

Require immediate abort when aborting yourself. (Related more to RI-0001 than RI-0003)

%reference RR-0203 Allow termination code for packages and tasks.

Packages and tasks need the ability to finalize.

%reference AI-00581 Abort and undefined variables

Abort during an update causes the updated object to become undefined. If the update is on a component then any access to other fields of the object causes the execution to become erroneous.

%reference AI-00360 Abort of several tasks

When an abort statement is executed, some of the aborted tasks and their dependents may become abnormal before others. Any such abnormal tasks can have an effect on other tasks before all the tasks have become abnormal.

%reference RI-0003 Asynchronous Transfer of Control

%reference RI-0001 Abort timing should be known to the user

!rebuttal

Some users feel that certain conditions require a task to terminate without finalization. These users believe that the task should be terminated immediately, and all finalization sequence, both those explicitly associated with the task as well as those that may be inherited from packages and objects, should be ignored.

As an options, these users would like finalization on ABORT or unhandled exceptions to be user selectable.

!topic Notification of abnormal tasks
!number RI-0009
!version 1.6
!tracking 5.8.14.1.3

!Issue presentation

[1] Ada9x shall specify the semantics for an attempt to activate a terminated or abnormal task.

!Issue out-of-scope (un-needed) (2) desirable,severe impl,bad compat,mostly consistent

2. (Provide an exception) Ada9X shall provide an exception to be raised when a program attempts to activate an abnormal task.

The user need is not substantial, and the additional checking required to raise the exception are not justified given the small demand for this feature.

!reference AI-00268

!problem

AI-00268 states, "No rule says a check must be made before activating a task to see if the task is terminated. In the absence such a rule, no such check can be made, and so no exception can be raised if an attempt is made to activate a terminated task."

Furthermore, no semantics are given for an attempt to activate a terminate task.

!rationale

Although no semantics are given explicitly for an attempt to activate an abnormal task, AI-00268 concludes that only unactivated task can be activated. Ada9x should specify the semantics even if it is identical the that concluded by this AI.

!appendix

!QUESTIONS REMAINING:
Is the conclusion of AI00268 appropriate for Ada9x?

%reference AI-00268 Activation of already abnormal tasks

!rebuttal

!topic Fine-grain parallelism
!number RI-0201

!reference RR-0514
!reference RR-0747

!problem

The user has no method within the language to specify fine-grain parallelism, e.g., parallel loops. The task concept is too burdensome to use for this type of parallelism, and the synchronized communication is too expensive for the fine-grain nature of many problems. Conventional optimizers are prevented from performing may types of parallelization because of the restrictions of LRM 11.6.

!appendix

%reference RR-0514 Provide support for parallel loop and threads

The user has no method within the language to specify fine-grain parallelism, e.g., parallel loops. The task concept is too burdensome to use for this type of parallelism, and the synchronized communication is too expensive for the fine-grain nature of many problems.

%reference RR-0747 Provide support for parallel programs via Linda

Linda is a simple shared memory model for parallel and distributed programs. The Linda provides support for many different programming paradigms. Some of these paradigms may be useful in Ada.

!topic accept smts in subprograms
!number RI-2107
!version 1.1
!tracking 5.8.15

!Issue out-of-scope (un-needed) (1) not defensible,severe impl,upward compat,consistent

1. Ada9x shall allow accept statements to appear in the bodies of subprograms that are nested in the body of the task that contains the corresponding entry declaration.

[Rationale] if this facility were to be allowed, we would run into an impossible situation whereby an accept statement could be executed by a task that does not own the entry. Consider the following example:

```
task T is
   entry E;
end T;

task body T is
   procedure P;
   task INNER;
   procedure P is
   begin
accept E do . . . end E;
   end P;
   task body INNER is
   begin
P;   -- this call is legal and would cause INNER to
     -- execute an accept for E!
   end INNER;
begin
   ...
end T;
```

In order to resolve this problem, one would have either to make a special rule (which Ada9x should try to minimize), or to reconsider it in the context of a much more drastic change: get rid of nesting altogether.

!reference RR-0543 Allow accept stmts in subprograms nested inside tasks
!reference RR-0580 Allow accepts within subpgms/pkgs nested inside tasks
!reference AI-00214 Allow accept statements in program units nested in tasks

!problem

Due to the problem described above, Ada83 restricts accept statements to appear only in the sequence_of_statements of a task. This is considered sometimes as irritating in that it constrains the programming style of the user, and makes it difficult to program certain problems, e.g. the recursive traversal of a data structure by a task.

If this request for change is considered in isolation, the need does not seem to be sufficiently strong to balance the language and implementation costs. This issue may be revisited if deeper changes are envisioned to the tasking model, such as, e.g., the introduction of entry types, or any change that would result in disconnecting the notion of an entry declaration to that of a task declaration.

!rebuttal

RR-0580 suggests that in the case of an attempt to execute an accept statement on behalf of the wrong task, the exception TASKING_ERROR should be raised.

!topic Miscellaneous Issues with CSP a la Ada
!number RI-2108
!version 1.1
!tracking 5.8.17

DRAFT DRAFT DRAFT DRAFT DRAFT DRAFT

!Issue out-of-scope (un-needed) (1) not defensible,small impl,very bad comp,inconsistent

1. (Remove Entry Families) The Ada83 concept of entry families shall not be carried into Ada9X.

[RATIONALE] Even if more appropriate mechanisms (than entry families) were added to better realize server-based scheduling implementations, the removal of entry families would not be upward compatible. Further, it has not been shown that entry families are detrimental to the language.

!Issue revision (2) important,moderate impl,upward compat,mostly consistent

2. (Intermixing Calls and Accepts in a Select) Ada9X shall provide a form of selective waiting construct that allows waiting-for-accept, waiting-to-call, and waiting-to-terminate in the same construct.

!Issue revision (3) important,moderate impl,upward compat,mostly consistent

3. (Changing Service Port Dynamically) Ada9X shall provide a mechanism by which a task may be parameterized by an entry with a given parameter profile so that references to a given entry parameter may be changed dynamically in response to a mode shift.

!Issue out-of-scope (un-needed) (4) desirable,moderate impl,upward compat,inconsistent

4. (Virtual Entries) Ada9X shall provide a mechanism by which an implementor of an outer task can specify that references to an entry of the outer task are to be realized as references to an entry on an inner task declared internally to the outer task.

[RATIONALE] There are two reasons why this is out-of-scope. First, this is a significant departure from the Ada model since the current model guarantees that two rendezvous on a task are not happening in parallel. Under the requirement, parallel rendezvous would be possible since the outer and inner tasks can run in parallel. Second, the need for this does not seem very strong.

!reference RR-0056
!reference RR-0134
!reference RR-0158
!reference RR-0173
!reference RR-0279
!reference RR-0380
!reference RR-0658

!reference RR-0697
!reference REFER ALSO RR-0436

!problem

There are a number of problems is Ada's realization of Hoare's CSP programming model that make things difficult for programmers. One of these is that all of the communications is a select statement must be in the same direction, i.e. all must be accepts or all must be calls. This creates quite a difficulty at the systems design level that results in the inclusion of a number of "adapter tasks" whose sole purpose if to change the direction of the communications. In addition, the particular entries to which a task refers are essentially hardcoded into a program. If such an entry is used to represent a service, then it is difficult to change the location of the service in response to a mode change.

!rationale

[2108.2] The requirement allows the task system of an Ada program to more accurately model the task system of a higher level design by supporting communications in either direction within a selective wait. The current workaround of adding "adapter tasks" is generally felt to be too inefficient.

[2108.3] Often an entry represents a service that a server task is providing to its clients. In some systems, a change in the environment will dictate a mode shift whereby that the location of the service must be moved. In such a case, Ada9X should provide a straightforward and efficient way to accomplish this.

!appendix

%reference RR-0056 Do not remove task entry families

RR-0056 wants entry families to be retained.

%reference RR-0134 Require re-evaluation of entry'count on abandoned entries

RR-134 notes that the use of the 'COUNT attribute is not safe in the case where timed calls care made on the entry. The suggestion is that such guards be reevaluated.

%reference RR-0158 Add terminate alternative to entry calls

RR-0158 wants to add a terminate alternative to entry calls.

%reference RR-0173 Allow rendezvous with higher-level entity, i.e., set of tasks

RR-0173 wants to have a sort of task aggregation idea where such an a-task is actually implemented by multiple lower level (a-)tasks and the entries of the higher level a-task correspond to the entries of the lower level (a-)tasks. The idea is a process decomposition abstract a la the ACTORS model of Hewitt.

%reference RR-0279 Language should not view main program as a task

RR-0279 wants the main program not to be viewed as running under an "environment task" under the (perhaps mistaken) impression that this adds significant overhead whether or not any other tasks are used.

%reference RR-0380 Need standard task_id type and operations

RR-380 wants to have a standard task-id and operations; the main rub seems to be that tasks cannot be appropriately parameterized at start_up. This aspect is covered in RI-2012 on Initialization.

%reference RR-0658 Allow accept statement possibility in a conditional entry call

%reference RR-0697 Allow entry call alternative in selective wait

RR-0658 and RR-0697 want to be able to mix accept statements and entry calls in a single accept.

%reference REFER ALSO RR-0436 (3.1)

RR-0436 wants a more precise definition of the synchronization points in a task.

!rebuttal

!issue Controlling the service order of client tasks
!number RI-2001
!version 1.11
!tracking 5.8.18

DRAFT DRAFT DRAFT DRAFT DRAFT DRAFT

!Issue revision (1) compelling,moderate impl,upward compat,mostly consistent

1. (Service Order) Ada9X shall provide a mechanism enabling a server task to control service order based on parameters of the client requests. The mechanism shall not require more than one synchronization between the server and client for a specific service request; neither shall polling of entry queues be required.

!reference SDIO Ada9X Requirements Meeting, March 29, 1990.
!reference RI-2021

!reference

[Gehani and Roome 1988] Gehani, Narain H., and William D. Roome, 1988, "Rendezvous Facilities: Concurrent C and the Ada Language" TSE 14 11 (November): 1546-1553.

!reference

[Burns 1985] Alan Burns Concurrent Programming in Ada, Cambridge University Press, 1985.

!problem

A problem that frequently arises in concurrent programming based on a client/server message-passing model is that the server needs to be able to control the service order of the client requests based on information that is a dynamic property of the request (not of the client) in order to obtain appropriate throughput. Examples of this paradigm include a disk server that rearranges the order of requests based on locality of disk references and the scheduler of a discrete event simulator that rearranges the order of execution based on its notion of virtual time. In Ada83, the needed facilities may be implemented only by rather error-prone techniques using combinations of additional entries, tasks, synchronization points, and scanning/polling over entry families.

!rationale

The use of Ada83 as a concurrent programming language is well-studied. The deficiencies of Ada83 with respect to determining service order are discussed in [Burns, 1985] and [Gehani, 1988]. The difficulty is not whether such solutions to these problems can be programmed in Ada; rather, it is that the solution techniques are clumsy and error-prone. Further, the introduction of multiple rendezvous to model an interprocess synchronization that is in (in some sense) conceptually simpler than a single rendezvous decreases the desire of many users to use the tasking model at all. The concept has been called "abstraction inversion" by some in the Ada community.

!appendix

A real question is implementability, i.e. how easily can solutions to this problem be incorporated into Ada implementations. Fortunately, upward-compatible, easily implemented solutions are also well-known. [Gehani88] describes the solution adopted for the Concurrent C programming language--the dequeuing order is specified by optimization function over the input parameters of the entry. This solution is notationally very convenient. A more powerful but less convenient solution would be where FIFO queue ordering is maintained but the ability to suspend a rendezvous and requeue a client on another entry is provided. The client task is not released until a rendezvous exits without requeuing. (Some of the power of this solution is lost unless the language supports "entry types"; see RI-2021.)

!rebuttal

5.9 Exceptions

RI-1070

!topic exceptions
!number RI-1070
!version 1.5
!tracking 5.9.1, 5.9.2, 5.9.3, 5.9.4
DRAFT DRAFT DRAFT DRAFT DRAFT DRAFT

!Issue revision (1) important,moderate impl,upward compat,mostly consistent

1. (More Information Available in Handler) Ada9X shall further promote the construction of self-diagnostic software by allowing more information on a raised exception to be available in exception handling code. At a minimum, this information shall include the simple name of the raised exception. In addition, consideration shall be given to the language providing:

1. an unambiguous identification of the raised exception;
2. an indication of where the exception was raised;
3. an identification of the thread of control handling the exception;
4. data values explicitly passed at the point of the raised exception, and;
5. some identification of the cause of a predefined exception for those defined broadly in Ada83 such as CONSTRAINT_ERROR and STORAGE_ERROR.

!Issue revision (2) important,severe impl,upward compat,mostly consistent

2. (Make Exceptions more Flexible) An attempt shall be made to improve the usefulness of exceptions in Ada9X. Possibilities for improved usefulness include:

1. exceptions as parameters to subprograms and entries;
2. exceptions as generic formals;
3. allowing exceptions to be "parameterized" (i.e., with parameter values passed in when an exception is raised and read when the exception is handled), and;
4. grouping of exceptions to make exception handling easier and for the purposes of "constraints" on exception objects.

!Issue revision (3) important,small impl,moderate compat,mostly consistent

3. (Fix NUMERIC_ERROR/CONSTRAINT_ERROR Problem) Ada9X shall clarify the distinction between NUMERIC_ERROR and CONSTRAINT_ERROR or shall subsume the raising of NUMERIC_ERROR by the raising of CONSTRAINT_ERROR.

!Issue presentation (4) important

4. (Clarify Applicability of Pragma SUPPRESS) Ada9X shall clarify the semantics of pragma SUPPRESS when the name given is that of a subprogram, type, or subtype.

!Issue revision (5) desirable,small impl,upward compat,mostly consistent

5. (Force Compliance with Pragma SUPPRESS) Ada9X implementations shall be required to "support" pragma SUPPRESS. By "support" here, it is meant that:

1. any additional machine instructions that the compiler normally uses to implement the checking shall not be emitted, and;
2. where practical, hardware checking for the Ada exception conditions shall be disabled.

!reference RR-0005
!reference RR-0033
!reference RR-0036
!reference RR-0085
!reference RR-0101
!reference RR-0145
!reference RR-0219
!reference RR-0263
!reference RR-0399
!reference RR-0403
!reference RR-0407a
!reference RR-0407b
!reference RR-0416
!reference RR-0444
!reference RR-0477
!reference RR-0526
!reference RR-0582
!reference RR-0583
!reference RR-0621
!reference RR-0646
!reference RR-0752
!reference RR-0765
!reference RR-0772
!reference WI-0301
!reference WI-0302
!reference WI-0418
!reference AI-00299
!reference AI-00387
!reference AI-00542
!reference AI-00595

!problem

A variety of problems concerning exceptions in Ada83 are addressed herein.

[RI-1070.1]
For the purposes of debugging and building informative diagnostics into a program, it would be useful to be able to determine information about a raised exception in a handler for that exception. Examples of such information items include the "name" of the exception raised, "where" the exception was raised, an identification of the task in which the exception was raised, values of data items that are specific to the raising of the exception (see RR-0646), and information about the cause of a pre-defined exception. Presently, the language does not provide facilities for obtaining such information (particularly in multi-tasking applications where global variables cannot be used for this purpose).

[RI-1070.2]
It has been observed that exceptions would be more useful if they were treated more like other entities in the language. For example, if exceptions could be passed as parameters to subprograms, a caller could control what exceptions were to be raised under various circumstances. As another example, if exceptions could be grouped in some way (perhaps analogous to the grouping of a range of scalar-type values by a subtype), it would be considerably easier to write exception-handling code in programs that make use of large numbers of exceptions.

[RI-1070.3]
The distinction between CONSTRAINT_ERROR and NUMERIC_ERROR is presently cloudy in Ada83.

[RI-1070.4]
The LRM is unclear about various aspects of the applicability of pragma SUPPRESS.

[RI-1070.5]
Some very time-critical applications, such as signal processing, not only cannot afford the cost of runtime checks, but also cannot permit the cost of handling certain exceptions that might be detected by hardware. For example, transient hardware (e.g. sensor) faults may result in out-of-range data, which in turn may lead to numeric errors, such as overflow or division by zero. Such systems can be designed to tolerate occasional incorrect calculations or to check for output validity in a later less time-critical phase, but they cannot tolerate long delays such as would be imposed by pre-checking that the data cannot cause a check to fail, or raising an exception, handling it, and restarting the computation. The desired response to numeric and constraint errors may therefore be to continue with the computation. Pragma SUPPRESS in Ada83 does not work this need since it is allowed to have no effect. This is seen as being unacceptable.

!rationale

[RI-1070.1]

The most pressing requirement for information in a hander seems to be the name of the currently raised exception. A specific example of a need for this capability is the POSIX-Ada binding (see RR-0145). The "workaround" of supplying an exception handler for each possible exception is very inconvenient and furthermore requires visibility over each of these exceptions at the point of the exception-handling code. Based on the need of a user to obtain the name of the raised exception in a handler for that exception, several vendors (e.g., D.G./ROLM) supply a library package that provides this functionality; what appears to be needed is a language change that makes a solution to this problem portable.

What is much less clear is "how much" of a "name" for the current exception should be made available in a handler. Options here range from the simple name of the exception to an unambiguous "fully-qualified" exception name. It is plain that just the simple name of the exception is far from ideal; however, it is unclear exactly how more than this could reasonably be accomplished within the definition of the language. Providing more than the simple name of the exception would require specifying what these names look like in all cases (e.g., in the absence of block labels) within the language definition. This additional complexity in the language definition and in implementations need to be balanced against the gain realized by providing more than the simple name of the exception.

Beyond the name of the current exception, there appears to be a need to obtain additional information within a handler such as an indication of where (and under the control of which task) the exception was raised as well as "exception parameter" values that might be associated in some way with the raising of the exception. As with the unambiguous name of the exception, it is not clear how easy it is to work information about where an exception was raised into the language definition. The exception parameters idea appears to be a useful one and should be given consideration.

[RI-1070.2]

This is a high-level requirement with no specific compliance criteria because it is not clear exactly what is needed along these lines. Furthermore, the additional flexibility provided in the language should hinge to a large extent on the way [RI-1070.1] is satisfied. A way of tailoring the exceptions raised and handled in a subprogram or generic unit seems like it would add a lot to the flexibility of the language. The need for general exception objects is doubtful on its own (other than in combination with 'IMAGE to solve the name-of-exception- in-handler problem). Any introduction of exception objects requires making a decision about whether the choices in an exception handling block must be mutually exclusive (run-time enforced in general) or whether the ordering of the exception choices is significant and the first matching choice is applied. The case for grouping exceptions to ease the task of writing exception handling code seems valid; however, the language-change proposals that have been seen along these lines all seem awkward and un-Ada-like.

[RI-1070.3]

This requirement amounts to a statement that AI-00387 must be dealt with by Ada9X.

[RI-1070.5]

The problem described above was discussed extensively by the Real-Time Embedded Systems Working Group at the Destin Workshop. Requirement [RI-1070.5] is intended to remedy this problem and is in the spirit of recently approved AIs (such as AI-00555) that require implementations to comply with certain pragmas where the underlying hardware and system make it reasonable and practical to do so.

!appendix

%ref RR-0033 Make exceptions regular Ada types

Need to be able to get the name of an exception in a handler. Need to be able to pass an exception to be raised to a subprogram.

%ref RR-0036 Make exceptions regular Ada types

The idea here is to allow subtypes of exceptions to group exceptions in a hierarchical manner. The grouping is important to ease the writing of exception handling code.

%ref RR-0101 Make exceptions a regular Ada type

Wanted grouping of exceptions using subtype mechanism. Also apparently want exceptions as parameters. Perhaps "task types" under 2.) and 3.) of "Some other details ..." should be "exception types".

%ref RR-0526 Make exceptions regular Ada types /objects

Want to group them using something like the subtype mechanism, want to pass them as parameters, want to get 'IMAGE, etc.

%ref RR-0621 Generalize facilities available for exceptions for power/flexibility

Getting image of handler, get the current exception, pass them as parameters, limited assignment of exceptions, etc.

%ref RR-0752 Make various improvements to exception handling capabilities

Same basically as RR-621.

%ref RR-0085 Allow exception name or image to be exported
%ref RR-0145 Provide a way to get exception name from "when others" handlers
%ref RR-0219 Provide way to get name of raised exception, inc. out-of-scope
%ref RR-0403 Need to be able to get the name of the current exception
%ref RR-0772 Need to be able to get exception name in a handler
%ref AI-00595 Name of the "current exception"

There is a need to get the name of the currently raised exception in exception handling code.

%ref RR-0407a In handler need exception name, line # and unit name where raised

For logging errors and reporting errors to the user, it is desirable to be able to get name and unit name and line number where raised.

%ref RR-0477 Allow way to get name and location where raised in handler

Want to be able to print out the name of the raised exception and information about where it was raised.

%ref RR-0582 Need to get details on exception in WHEN OTHERS handler

For logging errors, would like to be able to obtain name of the exception and information about where raised plus trace-back data.

%ref WI-0418 Need to get name and context (where raised) of exception

Language shall (1) allow a raised exception to be identified, regardless of whether the raised exception is in scope and (2) allow the immediate context of where a raised exception was raised to be identified.

%ref RR-0263 Narrow number of cases that raise constraint error

Restrict cases giving rise to CONSTRAINT_ERROR. Separate meanings of pre-defined exceptions so that for any error situation, one pre-defined exception is defined to be raised. It would be nice to find out where an exception was raised in a handler.

%ref RR-0399 Break up overly broad pre-def exceptions, e.g., CONSTRAINT_ERROR

CONSTRAINT_ERROR is way too broad; need PARAMETER_ERROR, etc.

%ref RR-0416 Granularity of predefined exceptions is too coarse

Can't handle exceptions well enough for embedded system. Need better indication of why the exception was raised.

%ref WI-0301 Provide ways to distinguish between various forms of STORAGE_ERROR

Need LOCAL_STORAGE_ERROR, ALLOCATOR_ERROR, etc.

%ref RR-0005 Exceptions are inconsistent, generic code-sharing accordingly hard

Current definition of exceptions is a pain-in-the-neck for generic code sharing

---

%ref RR-0407b Do not allow tasks to die silently when unhandled exception

Exception handler is really no good. Want asynchronous exception raised in the parent.

%ref RR-0444 Allow specification of where a declared exception can be raised

Need private exceptions raiseable only in this package but handleable outside.

%ref RR-0583 Delete NUMERIC_ERROR if now subsumed under CONSTRAINT_ERROR

Remove definition of NUMERIC_ERROR (or provide it as a RENAME) if you adopt the AI recommendation.

%ref RR-0646 Allow exceptions to be parameterized with params read in handler

The parameterized exception idea.

%ref RR-0765 Allow "when package_name.others =>" as exception handler

This is like grouping exceptions. Want a single when choice to cover all the exceptions declared in a package.

%ref WI-0302 Improve mechanism to suppress raising exception for critical code

Time-critical applications do not want time spent doing the checking and also do not want time spent handling exceptions. There algorithms tolerate an occasional NUMERIC_ERROR, etc. A second need for this is in unpacking/packing data using numeric operations, do not want overflow there. A third need is modular arithmetic, do not want overflow there. A fourth need is suppressing ABE errors, they do not want to or cannot figure out the proper elaboration order of their library units.

%OTHERS

%ref AI-00299 Pragma Suppress and Subprogram Names

A received ramification on the meaning of pragma SUPPRESS wrt subprogram names.

%ref AI-00387 Raising CONSTRAINT_ERROR instead of NUMERIC_ERROR

An approved non-binding interpretation that says wherever the Standard requires that NUMERIC_ERROR be raised (other than by a raise statement), CONSTRAINT_ERROR should be raised instead.

%ref AI-00542 [BI,RE] Meaning of "base type" wrt pragma SUPPRESS

A received binding interpretation on the meaning of pragma SUPPRESS wrt base types and subtypes.

!rebuttal

5.10 Representation

RI-0100
RI-0101
RI-0102
RI-0103
RI-0104
RI-0111
RI-0105
RI-0106
RI-0109

!topic Internal Representation of Enumerations
!number RI-0100
!version 1.7
!tracking 5.10.1

!Issue revision (1) important,small impl,upward compat,consistent

1. Ada9x shall provide a method for casting between the integer form of the internal representation of an enumeration value and the enumeration value. A constraint check shall be performed during this casting.

!reference RR-0007
!reference RR-0040
!reference RR-0059
!reference RR-0187
!reference RR-0220
!reference RR-0465

!problem

Currently, the only way to acquire the internal representation of an enumeration literal is to use UNCHECKED_CONVERSION or write a function or lookup table to translate an enumeration literal into its internal form. The users of Ada do not wish to use UNCHECKED_CONVERSION and they believe that the function/lookup table method is error prone and needless considering that the language has already provided a mechanism to specify the internal form of an enumeration. Similar problems arise when translating from the internal form to the enumeration literal.

!rationale

Ada83 provide a mechanism to specify the internal representation of an enumeration; however, the language does not provide a method to discover the internal representation of an enumeration literal or to examine the enumeration literal that corresponds to the internal representation of the literal. The only language features provided to convert to and from enumerations and integers are based on the relative position of the enumeration literal in relation to the enumeration type ('POS 'VAL).

!appendix

Solutions to this problem could take the form of explicit conversion between integers and enumerations or additional attributes.

RI-2034 will solve these problems if it is accepted because the problems only occur when interfacing to the external environment.

%reference RR-0007 Ada should specify the default representation for enumerations types. Programmers must use enumeration representation clause when interfacing to the external environment even if the representation specified is trivial, e.g., starting at 0 and going up sequentially. The same applies to integer types; however, in the case of integer types, the user has no control for the representation. The LRM should state the the value of an integer is equal to its 'POS. (LRM 3.5.5(8) states that the position number of an integer value is the corresponding value of the type universal_integer).

%reference RR-0040 There is no language feature for determining what internal codes are being used to store enumeration literals. The language provides a method for specifying the internal codes but does not provide a method to discover the codes being used.

%reference RR-0059 There is not way to get the internal code of an enumeration literal. There is no way to get an enumeration literal that corresponds to an internal code.

%reference RR-0187 Sign extension with the length clause The programmer cannot use enumerations representations with the length clause to guarantee that the representation will match the hardware demands. %reference RR-0220 There is no portable way to retrieve the internal representation code for enumeration literals.

%reference RR-0465 Add representation attributes to enumeration types. There is no implementation-independent mechanism to find the representation of an enumeration literal or to find the enumeration literal which corresponds to a representation value.

!rebuttal

!topic Length Clause (SIZE)
!number RI-0101
!version 1.9
!tracking 5.10.1 5.10.2.1.5 5.10.2.1.8

!Issue revision (1) desirable,moderate impl,upward compat,consistent

1. Ada9x shall provide a mechanism to specify the exact number of bits used to represent the objects of a given scalar type. It shall be possible to override this specification when a scalar type is used as a component of another type.

!reference RR-0187
!reference RR-0463
!reference RR-0062

!problem

The SIZE specification given in a length clause specifies an UPPER bound on the number of bits used to hold an object but does not have to alter the number of bits used to represent an object. The programmer need a method to specify the number of bits used to represent the object. This is especially important for specifying external interfaces.

A simple example of the problem is given below:

```
type interface_to_host_condition_codes (neg,zero,pos);
for interface_to_host_condition_codes use
   (neg=> -1, zero => 0, pos => 1);
for interface_to_host_condition_codes'size use 32;
```

This solution does not work because objects of this type may be represented in fewer than 32 bits.

A workaround could be:

```
...
type interface_to_host_condition_codes_plus (neg,zero,pos,big);
for interface_to_host_condition_codes_plus use
   (neg=> -1, zero => 0, pos => 1, big => 16#3FFFFFFF);
for interface_to_host_condition_codes_plus'size use 32;
...
...
subtype interface_to_host_condition_codes is
        interface_to_host_condition_codes_plus range neg..pos;
...
```

but, this corrupts the basic enumeration type, adds additional types (subtypes) to achieve the desired result, and does not always work.

!rationale

Ada83 provides no method for specifying the exact number of bits to be used to represent an object. The length clause, SIZE, only specifies an upper bound on the number of bits, and the record representation clause only specifies the layout to be used to hold objects in memory. Programmers need the ability to control the space used to store objects (currently provided in the language) as well as the space used to represent objects. This requirement should not be taken to either preclude or require biased representations of data. In fact, the users requesting this facility are not requesting biased representation of data.

!appendix

Overriding the size specification for a scalar type is not as important as the ability to specify the exact size used to represent an object of a scalar type.

RI-2034 would solve these problems because they only occur when interfacing to the external environment.

%reference RR-0187 Sign extension with the length clause

The programmer cannot use enumerations representations with the length clause to guarantee that the representation will match the hardware demands.

%reference RR-0463 There should be a distinction between the logical size of a type and the space allocated by the compiler for objects in various contexts. The author suggests a new attribute, SPACING, for determining the number of bits allocated to consecutive objects of a type, and determine the amount of storage used (including padding) for individual objects.

%reference RR-0062 Provide explicit control of the size used in memory access. In interfacing to hardware devices, the size of an object read or written is important. The language does not guarantee the size of an object or the size (byte, half-word, words, ...) used to access an object.

!rebuttal

!topic Storage Conservation
!number RI-0102
!version 1.9
!tracking 10.1

!Issue out-of-scope (un-needed) (1) desirable,severe impl,upward compat,consistent

1. Ada9x should provide a machine independent mechanisms to specify that dense packing of memory for all types and objects is desired.

[RATIONALE] The current "good" compiler techniques appear to satisfy the largest number of programmers. Compilers will pack data on the program stack as tightly as the hardware will allow them to efficiently access that data. Programmers using systems with small address spaces should be aware of their hardware limitations when designing their data structures. If the program requires large numbers of scalar data types then the programmer may consider packing these into records.

!reference RR-0699

!problem

Some users are more interested in conserving storage space than in efficient execution time of their programs. These programmers are wiiling to pay the alignment penalties to have their data packed as tight as possible. In Ada83 the user can only achieve this in a machine independent fashion is through pragma PACK; however, pragma PACK only applies to records and arrays. The users would like this to be extended to apply to all storage. In other words, two consecutive objects allocated on the program stack should be packed tightly together without gaps.

!rationale

!appendix

No language change is needed to have compilers provide this function since it could be realized through pragma pack. Changes to the semantics of pragma packed may be needed, however.

The language does provide pragma OPTIMIZE(SPACE); however, this must appear in the "declarative part" of a unit, and packages do not have a "declarative part." Furthermore, it would be legitimate for a compiler to use pragma OPTIMIZE to optimize code and not data structures and types. For example, pragma OPTIMIZE may control the unrolling of loops rather than the packing of data types.

%reference RR-0699 Machine Independent Storage Size Specification for Objects

An unaccepted length clause for a type is treated as an error; therefore, there is no machine independent method for specifying that the minimum size for objects is to be used. The programmer wants memory conservation without machine dependent code.

!rebuttal

!topic Multi-dimensional Array Storage
!number RI-0103
!version 1.8
!tracking 5.10.1, 5.10.2.6, 5.10.2.1.6

!Issue revision (1) desirable,small impl,upward compat,consistent

1. (Determining the Storage Order) Ada9x shall provide a method for determining the storage order of multi-dimensional array storage. At a minimum, the program shall be able to determine if multi-dimensional arrays are stored in row major, column major, or neither row/column major order.

!Issue revision (2) desirable,small impl,upward compat,consistent

2. (Choosing the Storage Order) Ada9x should provide a method for choosing a storage order for all multi-dimensional arrays on a program wide basis. Ada9x shall provide a method to insure that all multi-dimensional arrays in a program are stored in the same order.

!Issue out-of-scope (un-needed) (3) desirable,small impl,upward compat,consistent

3. (Storage Order Rep Spec) Ada9x should provide a method for choosing a storage order for each multi-dimensional array type.

[Rationale] Linear algebra algorithms are written with row-, column-, or both row- and column-oriented variants. Therefore, it is only important to know which variant to use, or to specify that one order or another is to be used for the entire program. Selecting different orders for each matrix in the same program will create maintenance problems.

!Issue revision (4) desirable,small impl,upward compat,consistent

4. (Representation Clauses for Arrays) Ada9x shall provide a method for specifying the storage unit alignment of an array and the corresponding alignment for all components of an array. The method need only be general enough to provide the same alignment for each component of an array.

!reference RR-0507
!reference RR-0418
!reference AI-00550 Position record components across storage boundaries
!reference AI-00553 Size specification for a record type
!reference AI-00554 Pragma PACK for record type
!reference AI-00555 Pragma PACK for array type
!reference AI-00556 Giving a size specification for an array type

!problem

[1, 2, and 3] In numerically intensive array computations, it is important that the storage order of the arrays is known. There is a significant efficiency problem if arrays are indexed in an order different than the storage order used by the compiler.

[4] Ada83 currently provides a method for specifying the layout of records and their components; however, a similar specification is not allowed for arrays. The current workaround involves wrapping the array (and possibly its components) within a record and using the facilities provided by the language for records to align the array and its components. This workaround reduces the understandability of the source code because of the additional names introduced in accessing components of an array. Additionally, the user may use pragma PACK for the array and specify the the representation of the component types; however, this does not allow the user to control the alignment of the array type [AI-00555].

!rationale

[1] In numerically intensive array computations, it is important that the storage order of the arrays is known. There is a significant efficiency problem if arrays are indexed in an order different than the storage order used by the compiler. Many algorithms have both row- and column-oriented variants; however, the algorithms cannot determine information on the storage mode used.

[2] For algorithms that do not have both row- and column-ordered code, a method for specifying the preferred indexing techniques is desirable. The indexing technique should be specified before compiling any units of the program, and be apply to all units.

[4] Ada83 does not provide a method for specifying the exact layout of arrays. Workarounds do exist; however, they reduce the understandability of the source code. This requirement is designed to allow the programmer to specify the alignment of an array, and the alignment of the array component. For example, the array will start on a quad-word boundary, and the components will be aligned on half word boundaries. This is not designed to specify the different alignments for each component of an array, e.g., the five, seven bit characters plus 1 bit alignment on a PDP-10.

!appendix

RR-0507 suggest adding an attribute and enumeration type to package SYSTEM:

```
type MATRIX_STORAGE_MODE is (ROW_MAJOR, COLUMN_MAJOR,
OTHER);
```

%reference RR-0507 Multi-dimensional Array Storage

A method to determine and/or set the storage order of arrays is needed. Either column-, row-, or some other ordering should be available.

%reference RR-0418 Representation Clauses for array types

Representation clauses should be allowed for arrays. The need is to express the alignment of an array and the alignment of the components of an array.

%reference AI-00550 Position record components across storage boundaries

%reference AI-00553 Size specification for a record type

%reference AI-00554 Pragma PACK for record type

%reference AI-00555 Pragma PACK for array type

%reference AI-00556 Giving a size specification for an array type This AI also includes a discussion of pragma PACK for array types.

!rebuttal

!topic Definition of Address Clause
!number RI-0104

!reference RR-0291
!reference WI-0303

!problem

The LRM states "An address clause specifies a required address in storage for an entity." Even though the intent of this statement is clear, the responsibility on the implementation to locate objects a these addresses is not spelled out.

!appendix

%reference RR-0291 Ambiguity in the definition of Address Clause

The sentences in 13.5 should be changed to: "The use of an address clause specified the address at which the Ada implementation is to allocate the entity." Change the definitions in 13.5:4-6 similarly.

%reference WI-0303 The effect of an address clause shall be more precisely define.

!rebuttal

!topic Placement of Objects
!number RI-0111

!reference RR-0110
!reference WI-0303
!reference WI-0304
!reference WI-0305
!reference WI-0305M
!reference WI-0306
!reference WI-0307
!reference WI-0416
!reference AI-00573
!reference RI-0104 (meaning of an address clause)
!reference RI-2034

!problem

The current method for interfacing through memory to the external environment is deficient. Ada allows address clauses for locating objects a specific addresses; however, Ada provides default initialization and allows implementations to optimize and modify memory accesses. This behaviour is contrary to the protocol expected by many external environments. Users trying to interface to the external environment are faced with many similar problems. The users would like more control over the layout, access, initialization (or lack thereof), and location of objects and collections of objects.

!appendix

%reference RR-0110 Provide explicit control of memory usage

The language makes no distinction between physical addresses and logical addresses, nor does it provide for systems with mixed memories (e.g., RAM and ROM) or fragmented non-contiguous memories.

%reference WI-0303 The effect of an address clause shall be more precisely define.

%reference WI-0304 A way shall be provided to specify the address that the compiler shall use to refer to an object when reading and writing it, and as the subprogram entry-point when calling a subprogram. The compiler shall not generate code to initialize such an object if it is declared as a constant. It shall be possible to specify an address determined at run time or via a generic formal parameter. Specifying the same address for two objects shall cause them to share memory locations (implying that if they have the same type they will have the same value).

%reference WI-0305 A way shall be provided to specify the address that the compiler shall use to call a subprogram. The compiler shall provide code for the subprogram if and only if the pragma INTERFACE is not specified for it. For any Ada subprogram declared as a library unit or immediately within the specification of a library package, a way shall be provided to obtain an address such that if this address is specified for a second subprogram calls to the second shall have the same effect as calls to the first subprogram.

%reference WI-0305M Minority opinion on WI-0305; The current facilities work.

%reference WI-0306 A way shall be provided to specify where in memory an object or code unit generated by the compiler should be located. It shall be possible to optionally either specify the exact address or to specify a logical category of storage and let the compiler choose the exact address. It shall be possible to modify such an address specification for an entity without modifying the Ada source code of the compilation unit containing the entity. This may be via a post-compilation specification, within or without the Ada language, whose form may be dependent on the compiler or target machine architecture. If such an object has a specified value, the compiler is required to provide that the storage at the specified address is initialized to this value. Similarly, for a program units the compiler is required to provide that the code for the unit is loaded at the specified address.

%reference WI-0307 An Ada language implementation shall be permitted to define a private type in package SYSTEM which may be used as an alternative to SYSTEM.ADDRESS in specifying bindings between interrupts and task entries.

%reference WI-0416 The language shall provide explicit control for the location of objects and storage allocation for objects by:

- — Providing the ability to name, locate, and size specific regions of storage,
- — Constraining the location of objects within a named region,
- — Providing the ability to locate a storage allocation for objects to a predetermined location, and
- — Constraining the location of a storage allocation for objects within a named region.

%reference AI-00573

An implementation should be allowed to omit the initialization of a variable that has an address clause applied.

!rebuttal

RI-0105

!topic Storage Order in Representation Clauses (Big endian / Little endian)
!number RI-0105
!version 1.5
!tracking 5.10.2.2

!Issue revision (1) important,small impl,upward compat,consistent

1. Ada9x shall provide facilities for specifying and controlling the representation of data at the bit level.

!Issue revision (2) desirable,moderate impl,upward compat,consistent

2. Ada9x shall provide a method for specifying the ordering of bits and storage units in a representation clause for a type not containing implicit or explicit addressing.

[This does not preclude a program wide solution. A program wide approach may be the most desirable; however, there are cases when a single bit ordering for a program may not be appropriate, e.g., a network gateway program handling data from many different machines.]

!reference RR-0411
!reference RR-0137
!reference RR-0290
!reference RR-0460
!reference RR-0459 (related)
!reference RI-2034

!problem

Ada does not specify the order of bits or storage units in representation clauses. As a result, unnecessary machine dependencies are introduced into the source code. One would think that representation clauses should not be machine independent; however, many projects require a "software first" approach. In this approach, code is initially developed on a host processor and then ported to the target systems. When the host and targets are different, porting the software requires undue effort. Furthermore, the manifestation of differences between right-to-left and left-to-right ordering of storage in the representation clause is not intuitive to the human programmer. Basically, the users would like a consistent view of the storage (the order bits and bytes is the same) for different target architectures.

!rationale

[1] is a high-level requirement expressing the overall need programmers have with respect to data representation.

[2] provides a method for users to specify the bit and storage unit ordering for objects. This requirement specifically omits the case when access types or other forms of references are used in the representation of a type. This is done to prohibit the users from introducing unnecessary operations when dereferencing addresses; also, there appears to be no need for this type of ordering on access types and references. This restriction could be removed in the interest of "regularity"; however, it may cause serious programming errors when programmers are interfacing to the external environment.

Programmers needing this type of control will be granted a more portable method for representing data. Compilers would have to perform the transposition of fields within records when the underlying hardware differs from the conventions specified by the program.

!appendix

Note: RI-2034 would solve these problems.

%reference RR-0411 Record presentation clause is excessively machine-dependent and nonintuitive. This RR includes a long example with floating point numbers.

%reference RR-0137 Bit/Storage Unit Addressing Convention.

This RR is submitted by a user that must develop code on the host then port the code to the targets. The RR notes the lack of portability in representation clauses. This includes byte swapping and sign-bit locations. The current workaround for portable code requires that the program calculate all possible single bit address as well as start and stop addresses during initialization. Then knowing, from package SYSTEM, that the processor is either big endian or little endian bit addressed, it calculates n**2 start and end locations for all possible multiple and single bit fields. The problem with this workaround is that it requires 1024 constants for 32bit processors.

%reference RR-0137 Unsigned Arithmetic (bit addressing conventions)

In a bit array, what is bit i? Is it 2**i or 2**(nn-1-i)?

%reference RR-0290 Use of range notation in representation clauses is unclear

The range notation in record representation clauses confuses many users. It is not directly intuitive to the user. A forced left-to-right ordering of bits and a simpler notation would be helpful.

%reference RR-0459 Interoperability of programs.

A simple example of the problem on a VAX is the single-precision floating point numbers:

```
 31       16     14   7       0
+-------------------------------+
 LSB_fraction  Sign Exponent MSB_fraction
+-------------------------------+
```

```
type f_float_format is
   record
     MSB_fraction: bits7;
             exponent   : byte;
             sign       : bit;
             LSB_fraction: word;
 end record;

for f_float_format use
   record at mod 4;
     MSB_fraction  at 0 range 0..6;
             exponent     at 0 range 7..14;
             sign         at 0 range 15..15;
             LSB_fraction  at 2 range 0..15;
 end record;
```

This ordering is opposite of the users intuitive understanding of the floating point number.

!rebuttal

Ada83 has not done a good job to dispel a misguided belief that many programmers posses. Namely, many users believe that the material in LRM Chapter 13 should be portable between implementations and target architectures. The following should be stressed in Ada9x:

> "Ada9x shall specify that it is ridiculous to expect a program that contains rep specs to port with out modification."

!topic Unsigned Integers and Bit-vectors
!number RI-0106
!version 1.7
!tracking 5.10.3, 1.3, 5.2.3.3

!Issue presentation (1) compelling

1. Ada9x shall require that packed array of BOOLEAN is packed without any gaps, that each component of the array occupies exactly one bit of storage, and that adjacent components occupy adjacent bits.

!terminology

A WORD is a data type represented by a tuple of bits.

!Issue revision (2) compelling,moderate impl,upward compat,mostly consistent

2. (WORDS) Ada9x shall provide the WORD data type with the following operations: shift and rotate (left and right), bit-wise logical operations (and, or, not, xor), comparison operations (=, /=, <=, >, <, >=) bit selection (reading and writing a particular bit in a word), and slicing (treat a consecutive group of bits in a WORD as another WORD).

Ada9x shall provide safe and efficient modulo arithmetic operations on the cardinal representations of WORDS, and Ada9x shall define relation between the bit ordering of WORDS and their unsigned integer (cardinal) representation, e.g., the i(th) bit of a WORD corresponds to the value 2**i.

The size of WORDS may be restricted by Ada9x and additionally restricted by implementations. Nevertheless, Ada9x shall specify a minimum size for WORD that must be supported by all implementations.

!reference RR-0044 There is no need to add unsigned integers to Ada
!reference RR-0136 (related) Support bit-level shift operations
!reference RR-0138 Need full-sized unsigned integers
!reference RR-0139 Provide shift a rotate for boolean arrays
!reference RR-0188 There is a need for unsigned integers in the language
!reference RR-0332 Provide unsigned integer capability
!reference RR-0389 (related) Cyclic types
!reference RR-0433 There is a need for pre-defined unsigned integer types
!reference RR-0460 Ada needs to provide support for unsigned integer types
!reference RR-0633 Provided logical operations for integers
!reference RR-0634 Provide basic support for extended precision arithmetic
!reference RR-0640 Need to access chucks of a bit vector
!reference RR-0704 (related) Make every bit available
!reference RR-0721 Try to add unsigned integers to the language
!reference RR-0766 Provide shift and logical operators for integers

!reference WI-0311 Provide shift, rotate, and other set operations on bits
!reference WI-0311M Need bit arrays and operations
!reference WI-0314 Provide unsigned integer capability with modulo arithmetic
!reference WI-0314M Provide unsigned integer capability with overflow checking
!reference RR-0635 (slightly related)
!reference AI-00402 Unsigned integer types are not predefined types
!reference AI-00597 Unsigned integer types can be provided
!reference AI-00600 Why We Need Unsigned Integers in Ada
!reference AI-00555 Pragma PACK for an array type
!reference RI-0105 Storage Order in Representation Clauses

!problem

Integer values in the range 0..(2**N)-1 cannot be represented in N bits in Ada83 because integer base types are required to include negative numbers. Examples where full-word unsigned integers are needed include calculating hardware addresses and manipulating the contents of hardware device registers.

Furthermore, many algorithms require bitwise operations on "integer" data (e.g., communications protocols, error detecting and correcting codes, and priority interrupt buffers just to name a few). Ada83 provides bit-vectors in the form of packed arrays of boolean; however, the operations are not sufficient. First, many bit-style operations are not provided: shift (right and left), rotate (right and left), and find the first bit that is set. Second, modulo operations such as addition and subtractions are not provided and require the use of UNCHECKED and unsafe programming to achieve efficiently.

!rationale

Ada must support common interpretations of values held in machine registers and memory, and should allow implementation of efficient, portable, and safe operations on such values.

[1] The need for bit-vectors and associated operations is broad. Most languages, even Ada83, provide the concept of bit vectors. The problem not addressed by Ada83 is the set of operations needed for bit-vectors. Ada83 provides the logical operations, bit selection, and slicing; however, the language does not provided shift and rotate operations or modulo arithmetic operations.

[2] The particular method for implementing modulo operations on bit-vectors is not specified in these requirements. Instead, these requirements specify that the mapping between the bit-vector representation and its corresponding unsigned integer representation be stated as part of the language. A standard mapping will increase the portability and safety of these operations. In general, the need for bit-vectors as cardinals is restricted to small bit vectors (some multiple of an addressable memory unit like a register or pair of registers). For this reason, [2] provides an escape for implementing cardinal operation on bit-vectors. It is expected that the Ada9x mapping process will determine if such restrictions are in order. At a minimum, Ada9x should provide arithmetic operations for bit-vectors that or the same size as the predefined integers.

!appendix

Several revision requests include shift, rotate, and mask as desirable and efficiently implemented operations on unsigned integer values. While shifting can be equated to multiplying or dividing by powers of 2, I don't know how to argue for the others as reasonable numeric operations.

An unresolved question is whether predefined operations on unsigned integers should raise exceptions on overflow. The modular arithmetic camp wants to do without, while range checks sound like a good idea for address calculations. I can imagine wanting to mix modular and range- checked operations in a single expression, which seems to rule out pragma SUPPRESS as a general solution.

Uninterpreted base types seem to be preferred over packed arrays of BOOLEAN by those who twiddle bits. Ada's record representation specs provide a way for users to avoid bare bit-twiddling operations and capture some of the semantics of packed data structures such as contents of hardware registers.

The current method for realizing bit-vectors is through a packed array of boolean. The array approach complicates the addition of shift operations to the language. If shift operations are provided for arrays of boolean, should they also be provided for all one-dimensional arrays or is this another special case for LRM 3.6.2?

Ada9x must specify the relation between the bit ordering of a bit vector and its unsigned integer representation. If the order is not specified then severe implementation dependencies will be introduced into the language. This includes byte-swapped implementations, right-to-left, and left-to-right orderings.

Furthermore, there is some dispute as to the nature of the modulo operations on these bit-vectors. In the worst case, for example, error detecting and correcting codes, there is a need for logical operations, shifting, selecting bits, and addition and subtraction of bit-vectors. The full set of integer operation do not seem useful in this context; however, there may be applications where multiplication is needed (e.g., address calculations).

One question is purposely left open by these requirements: should modulo operations be supported on bit-vectors or should bit-vectors be explicitly converted to some other data type to perform modulo operations? A strict mapping of bit-vectors to unsigned integers may not be appropriate since unsigned integers will have an implementation dependent range while application needing bit-vectors may require implementation independent code (e.g., x-modem's CRC encode and decode).

%reference RR-0044

The need for unsigned integers is only critical for some 16-bit machines. The need for modular arithmetic is independent of unsigned numbers.

%reference RR-0136 Need bit-level operations on integer data types.

Ada should provide bit-operations for integer data types. Included are the logical operations and shift operations; multiplication and division by powers of 2 are inadequate due to overflow problems and implementation dependencies.

%reference RR-0138

Unsigned integers are needed to interface with hardware devices and non-Ada software.

%reference RR-0139 Need shift and rotate operations for boolean arrays.

Shift and rotate are commonly used in signal processing, communications, and cryptography. These operations are lacking in Ada and should be provided.

%reference RR-0188

Ada's lack of unsigned integers and associated bit-wise logical operations limits Ada's applicability to embedded software.

%reference RR-0332

Provide unsigned integers. Includes discussion of possible solutions.

%reference RR-0389

Provide cyclic types with modular arithmetic operations.

%reference RR-0433

Need predefined unsigned integer types for hardware address calculations and handling full-word, unsigned application data.

%reference RR-0460

Provide standard support for unsigned integer types. Several vendors support limited forms of unsigned integers using different approaches.

%reference RR-0633 Logical operations on integer data types.

Ada should provided logical and, or, not, and xor operations on integer data types.

%reference RR-0634 Need shift operations on integer data types.

Shift operations are provided in most languages, but not in Ada. Ada should provided shift operations on integer data types. The author notes that bit-vectors could be used.

%reference RR-0640 Bit-vectors and set operations.

Ada provides partial support for what is needed. Operations such as find-first-bit, shifting, and conversion of bit-vectors to integers would be useful. Conversion of bit vectors to integers should be set so that bit i becomes the coefficient 2**i.

%reference RR-0704

Find a unified solution to full-word address values, bit strings that cross word boundaries, unsigned integers, full accuracy of floating point numbers, and 8-bit characters.

%reference RR-0721

Seriously consider providing unsigned integers.

%reference RR-0766 Logical and shift operations are needed.

Communications and error detecting and correcting codes commonly need bit operations for shifting and computing checksums. Ada needs to provided and, or, xor, and not operations along with shift-left and shift-right. Modulo arithmetic would also be useful.

%reference WI-0311 Ada should support bit operations.

Operations suggested are: logical operations, sift operations including rotate, test and set bits, and find first bit for word length bit strings.

%reference WI-0311M Same as WI-0311 except that WI-0311 suggest a secondary-standard as a solution and the minority opinion is that these operations should be provided as predefined operations in the language.

%reference WI-0314

There must be a type which does not cause NUMERIC_ERROR (or CONSTRAINT_ERROR) but wraps around. Such a type with range 0..2*INTEGER'LAST + 1 is necessary, and additional ranges should be provided.

%reference WI-0314M

Minority position: Provide unsigned integers with overflow checking.

%reference AI-00402

The Ada83 LRM specifically disallows unsigned integer types as predefined types [LRM 3.5.4(7)].

%reference AI-00597

Although unsigned integers cannot be predefined types, implementations may provide unsigned integer types in packages other than STANDARD.

%reference AI-00600 Why We Need Unsigned Integers in Ada

%reference RR-0635 Need multiple precision integer operations.

Not directly related to bit-buckets but should be considered.

!rebuttal

!topic Representation Clauses
!number RI-0109

!reference RR-0287
!reference RR-0288
!reference RR-0290
!reference RI-3000

!problem

Two problems are represented in this RI. First, when interfacing to other languages and external devices, it is important to know that access types point directly to the objects and not to an intervening structure. In particular, languages the pass objects by reference need access types that guarantee this convention. Second, the current method for specifying representation clauses places the rep spec arbitrarily far from the type declaration. Further, the notation used for record representation clauses is confusing.

!appendix

%reference RR-0287 Access types point directly to objects

Implementation should be required to make access types point directly to the object and not to some intervening structure. This will make the interface to other languages less implementation dependent and more portable than the current methods.

%reference RR-0288 Integrate representation clauses with declarations

The current separation of representation clauses from the type declarations increases the cost of compilation, maintenance, and generally reduces programmer's productivity.

%reference RR-0290 Use of range notation in representation clauses is unclear

The range notation in record representation clauses confuses many users. It is not directly intuitive to the user. A forced left-to-right ordering of bits and a simpler notation would be helpful.

5.11 Storage Management

RI-0115
RI-0107
RI-0116

!topic Storage error/recovery and Exceptions
!number RI-0115sp

!reference RR-0118
!reference RR-0120
!reference WI-0301
!reference AI-00133
!reference RI-1070 (exceptions)

!problem

When STORAGE_ERROR is raised because the available stack space is exhausted, it is likely that there is insufficient storage for an exception handler to executed in order to recover form the exception. As a consequence, a second STORAGE_ERROR may be raised in attempting to handle the first STORAGE_ERROR. It should be possible for an application to guarantee that the recovery action can be performed.

In some applications, memory may be temporarily exhausted. Yet, if the allocation were attempted a short time later, it would succeed. Handling STORAGE_ERROR and looping back after a short delay is an expensive method for solving the problem.

When STORAGE_ERROR is raised, Ada83 does not provide a mechanism for the exception handler to discern the type of storage that is exhausted, e.g., stack or heap. A more general concern is that Ada83 only defines general categories of builtin exceptions, e.g., NAME_ERROR, CONSTRAINT_ERROR, etc. The users need a means for determining, with finer detail, the cause of exceptions. The conditions under which an exception is raised should also be clarified.

!appendix

%reference RR-0118 Provided user-specified storage reserve for recovery from STORAGE_ERROR.

When STORAGE_ERROR is raised because the available storage for items commonly allocated form the "stack" is exhausted, it is likely that there is insufficient storage for an exceptions handler to execute.

%reference RR-0120 Handling unsuccessful attempts to allocate memory.

Ada should provide more flexibility by permitting an application to specify an alternative to raising STORAGE_ERROR when there is insufficient storage to create an object via an allocator.

%reference WI-00133

At minimum, the standard shall specify a way for programs to distinguish between exhaustion of local storage due to an attempt to create locally declared objects, temporary variables, or subprogram/entry calls and exhaustion of storage used for objects designed by values of access types.

For the general requirement, the language shall provide a finer-grained exception facility than is currently provided. At a minimum, the following exceptional conditions shall be distinguishable:

A) Exhaustion of local storage due to an attempt to create locally declared objects, temporary variables, or subprogram/ entry calling frames;

B) Exhaustion of storage used for objects designated by values of an access type (regardless of whether a representation clause has specified a collection size for the access type); and

C) Each condition represented by one of the checks allowed as parameters to program SUPPRESS.

%reference AI-00133 raising STORAGE_ERROR

The conditions under which STORAGE_ERROR may be raised should be clarified. If it may be raised whenever the available storage is exceeded the list of circumstances under which it may be raised is unnecessary. If on the other hand, it cannot be raised except under those listed circumstances, there are some important cases which are missing. For example, evaluating an aggregate may require storage allocation. Also any assignments involving slices assigned to slices could be implemented in a manner requiring that new storage be allocated.

!topic Memory Reclamation
!number RI-0107

!reference RR-0063 Abort statement
!reference RR-0112 Garbage collection
!reference RR-0113 Guarantee memory reclamation
!reference RR-0439 Automatic garbage collection
!reference RR-0493 Knowing if garbage collection is performed
!reference RR-0643 Garbage collection
!reference RR-0702 UNCHECKED_DEALLOCATION and garbage collection
!reference AI-00570 Heap storage associated with tasks
!reference AI-00589 Clarify the use of UNCHECKED_DEALLOCATION
!reference RR-0351 Scrubbing memory (Finalization) (reference moved to RI-2022 outline section 4.4.2)

!problem

Many application require that unused memory is always reclaimed. These applications cannot tolerate memory leaks in any form. Ada83 does not provide a means to meet these requirements.

Furthermore, Ada83 does not require garbage collection. This encourages the use of unchecked programming via UNCHECKED_DEALLOCATION and discourages the use of access types. Systems that provide garbage collection can only be controlled in a limited way from within the language. Many programs would like the safety of garbage collection, and control over the execution time required for garbage collection. In other words, these users would like explicit control of the garbage collection process: control over the start of garbage collection (never, automatic, user-control), control over the triggering event (start time, trigger, priority, duration), dynamic control of collection methods (depending upon the application's requirements during execution).

Some of the issues raised by these RRs are:

1. Unchecked_deallocation is only required to change its parameter to null.
2. Garbage collection is not required, and there is no way to discover if it is being performed on any particular collection, when it is performed, how long it will run, or to control if/when/how it is performed.
3. Many implementations do not recover memory allocated to tasks.

!appendix

%reference RR-0063 Make use of abort statement user-switchable

The inclusion of an abort statement makes it difficult to prove anything about program states, deadlock states, etc. Because the allowable time lapse before termination is not defined by the language, the abort statement may be inappropriate for critical function. All of the space is not reclaimed after a task has aborted. Abort finalization is also needed.

%reference RR-0112 Garbage collection

Ada should support schemes for controlled reclamation of deallocated storage (incremental garbage collection). Ada should provided explicit user control of garbage collection process, so that a user may:

a. control how garbage collection starts: never, under user control, or automatically.

b. tailor the automatic execution of garbage collection (e.g., the starting time or triggering event, priority, cycle, maximum duration).

c. exert dynamic control of garbage collection (e.g., vary the mode of garbage collection, depending upon application requirements, during application execution).

Ada should constrain the definition of garbage collection so that where present, garbage collection is defined to return all deallocated storage to the free pool. Ada must also define explicitly operation fo the user to determine the status of the storage pool, so the user may initiate garbage collection when required.

%reference RR-0113 Guarantee memory reclamation.

Real-time, long running, embedded, and other systems require that unused memory is always reclaimed. Conforming compiler/RTs shall never lose storage. This includes storage allocated by allocators, storage allocated to tasks, and storage allocated by the compiler/RT for its own use. Upon completion of an Ada program, all storage consumed by the program shall be returned to the underlying system. Require UNCHECKED_DEALLOCATION to do something. The language shall require that in the absence of an explicit application request to the contrary, storage occupied by an object which was created by an allocator shall be reclaimed, e.g. made eligible for garbage collection. In particular, this is important in certain error conditions when memory is allocated by cannot be referenced (example given in RR). %reference RR-0439 Automatic garbage collection The lack of automatic garbage collection make it difficult to build software and eliminate many bugs (storage leaks).

%reference RR-0493 Knowing if garbage collection is performed

The language should provide an attribute that allows an application to know if it performs garbage collection on a given storage collection, or if UNCHECKED_DEALLOCATION must be used. When garbage collection is not performed, UNCHECKED_DEALLOCATION should be required to do something more than setting its parameter to null. Also, other Ada constructs are not allowed to generate memory leaks.

%reference RR-0643 Garbage collection Garbage collection techniques have improved over the past decade. As a result, garbage collection is now a viable options for mission-critical and real-time systems. This RR gives a detailed discussion of several modern garbage collection techniques.

%reference RR-0702 UNCHECKED_DEALLOCATION and garbage collection

Allow user specified garbage collection procedures.

%reference AI-00570 Heap storage associated with tasks

Some implementation do not provide a method to recover the heap storage associated with tasks (LRM 13.10.1(8)).

%reference AI-00589 Clarify the use of UNCHECKED_DEALLOCATION

Insert notes making it explicitly clear that:

A. An implementation is required to compile and execute instantiations and calls to UNCHECKED_DEALLOCATION without raising an exception.

B. An implementation is not required to deallocate memory when a call to UNCHECKED_DEALLOCATION is made.

C. An implementation is not required to perform automatic garbage collection.

!rebuttal

!topic Memory Management
!number RI-0116

!reference RR-0374
!reference RR-0375

!problem

Ada has not concept of memory partitions and their management. For example, Ada does not have the concept of pages, segments, paged segments, or the algorithms to manage these structures. Vendors supply different degrees of support for these concept; however they are outside of the language and not standardized.

!appendix

%reference RR-0374 Ada should address memory management and partitions

Ada has not concept of memory partitions and their management. For example, Ada does not have the concept of pages, segments, paged segments, or the algorithms to manage these structures. Vendors supply different degrees of support for these concept; however they are outside of the language and not standardized.

%reference RR-0375

There is no method to define segments or pages of memory dynamically throughout the users memory space and allow for these areas to be locked or unlocked by qualified processes.

5.12 Compilation Library Units

RI-4017
RI-2109
RI-2110
RI-3572
RI-3154
RI-3279
RI-2111
RI-2112
RI-2113

!topic Control of elaboration order is unsatisfactory.
!number RI-4017
!version 1.7
!tracking 5.12.1.1 5.12.1.2

!Issue revision (1) important,moderate impl,moderate compat,consistent

1. Ada9x should provide mechanisms allowing programmers explicit control over the order in which library units and unit bodies are elaborated. It should define rules for the elaboration of library units and unit bodies that minimize the instances in which these explicit mechanisms must be used. As much as possible, these rules should avoid requiring clients of a library unit to know details of the implementation and the dependences of the library unit or its body.

!Issue revision (2) desirable,unknown impl,upward compat,mostly consistent

2. Ada9x should modify elaboration rules to facilitate "call-in" of Ada subprograms from non-Ada code (as when Ada is used to implement subprogram libraries), and to address implementation issues raised by the access-before-elaboration check.

!Issue revision (3) desirable,unknown impl,upward compat,mostly consistent

3. Ada9x should address the problems posed by modules that (logically) require non-standard control over operations normally performed upon unit elaboration, upon exit from subprograms, blocks, and tasks, or upon completion of the program. Such operations include elaboration, storage de-allocation, and task termination, among others.

!Issue revision (4) desirable,unknown impl,upward compat,mostly consistent

4. Ada9x should facilitate the implementation of modules that provide the effect of persistent data—modules that are shared among executions without being re-elaborated.

!Issue revision (5) desirable,unknown impl,upward compat,inconsistent

5. Ada9x should address the problem of long-running systems that require the effect of replacing the bodies of some of their modules with new implementations.

!reference RI-2002
!reference RR-0004
!reference RR-0218
!reference RR-0233
!reference RR-0370
!reference RR-0396
!reference RR-0546
!reference RR-0767
!reference WI-0206

!reference AI-0354
!reference AI-0355
!reference AI-0421
!reference RI-4019

!problem

(1) In the absence of explicit directives, Ada83 does not specify the order in which library units are elaborated, aside from the requirements that a library unit must be elaborated before its body, and before any library unit or unit body that "with"'s it. This rule often permits elaboration orders that cause entities to be accessed before the elaboration of their declarations. It also increases cases in which objects are accessed before being fully initialized.

A general, automatic solution of the elaboration order problem in Ada is difficult first because of the complexity of the conditions under which resources promised in a unit may be needed during the initializations performed in other units and bodies. Thus, the body of package X must be elaborated before unit or body Y if

1. an initialization in Y calls a subprogram in X (but not if Y simply contains a call of X); or
2. an initialization in Y calls a subprogram that calls a subprogram in X; or
3. Y declares an object of a type with a default value whose elaboration calls a subprogram in X (but not if the type has no default value or the default is not used), or
4. Y declares an object containing a component of a task type declared in X;

and so on. At the time Ada83 was codified, the rules looked sufficiently complex that the issue was, in effect, deferred.

Furthermore, even solving the access-before-elaboration problem would still leave the more general form of the problem: access before initialization. This occurs when a certain data structure declared in one library unit is initialized in another and used in a third.

Ada83 provides an explicit mechanism—the ELABORATE pragma—to allow programmers to specify additional constraints on the elaboration order of library units and unit bodies. This pragma allows one to specify that elaboration of a particular unit's body must precede elaboration of the compilation unit containing the pragma. Unfortunately, this mechanism has many drawbacks. Consider the following set of compilation units, and assume they are separately compiled.

```
package MIDDLE is ... end;

with MIDDLE;
pragma ELABORATE(MIDDLE);
package CLIENT is ... end;
```

This might work, but not if the body of MIDDLE is as follows.

```
with UTILITY_UNKNOWN_TO_CLIENT;
package body MIDDLE is ... end;
```

That is, MIDDLE makes use of UTILITY_UNKNOWN_TO_CLIENT, but does not require it for package initialization, and hence does not apply a pragma ELABORATE to it. To avoid errors, however, the body of UTILITY_UNKNOWN_TO_CLIENT must be elaborated before CLIENT, which is not guaranteed by the declarations above. The pragma on CLIENT must actually be

```
pragma ELABORATE(MIDDLE, UTILITY_UNKNOWN_TO_CLIENT);
```

requiring a violation of information hiding (the writer of CLIENT must know that MIDDLE's implementation uses UTILITY_UNKNOWN...). The impact of this problem on maintainability and portability of software is obvious.

(2) Ada's restriction on elaboration before use for subprograms requires checking that can be embarrassing, while providing nothing in return but safety from a rather rare family of errors. The rules complicate the procedure for calling Ada subprograms from foreign code, especially if the logical "main procedure" is not written in Ada, but Ada is being used to provide a subroutine library, for example. Looking ahead to distributed implementations of Ada programs, the same rules seem to require that distributed tasks must consult a central flag upon first calling any subprogram to insure that subprogram has been elaborated. The cost/benefit of this check is questionable and should be re-examined.

(3) Data (including tasks) that are naturally declared as variables in a library unit currently come into being before the main program executes and last until the entire program ends. Sometimes, however, different behavior is desirable. Examples include systems that are designed to restart in cases of recoverable catastrophes, and systems that go through phases in which different groups of packages participate.

(4) In particular, it is not always (logically) desirable that a package's data disappear at the end of a program and be freshly allocated at the beginning. One can imagine a package that represents a database, for example.

(5) There are examples of systems that must stay up continuously for very long periods of time (telephone switching systems are one example that has been cited). In such cases, it is desirable to be able to change components of the system without restarting.

!rationale

Requirement 4017.1 uses the word "minimize" with the understanding (shared by all RI's) that this means "consistent with keeping language rules simple and compiler implementation costs reasonable." As suggested in the problem statement, a maximally precise statement of rules may be unwieldy and, for practical purposes, unnecessary. The word "module" throughout the requirements is intended to be a neutral word that decouples the problems stated from any necessary relationship to library unit elaboration rules.

Some commentators (RR-0004, RR-0233, RR-0396) have suggested that an acceptable solution for requirement (1) would be to keep the current elaboration rules for the case where no ELABORATE pragma occurs, and then to redefine ELABORATE (or introduce a new pragma) to create elaboration-order dependences transitively for all bodies of units "withed" by units (and corresponding bodies) that are mentioned in a pragma ELABORATE. The wording of the requirement admits of such a solution.

The rules mandated by the requirements are likely to be conservative in the sense of introducing a stricter partial order than is sometimes required, leading to a possible source of incompatibility (with the Ada9x ordering becoming circular in cases where Ada83 did not). For the sorts of rules discussed by the Ada83 design team, such cases might arise when relations of the form "unit X contains an access to an entity in unit Y" are stronger than "unit X accesses (at execution time) an entity in unit Y." Such cases, however, are probably rare.

Besides the problems of avoiding access-before-elaboration (ABE), there are also potential problems stemming from situations in which, for example, bodies of packages A and B have no ordering required to avoid ABE errors, but package A's statements perform an initialization on package C, which both A and B use. These situations cannot be handled automatically. However, a mechanism that allows the programmer to impose any order on elaboration of A and B ought to cover the case (leaving aside the question of whether the situation presented represents good practice).

While the problems so far described stem from insufficient specificity about elaboration order, Ada83's rules also can be unnecessarily strict. Elaboration of a certain package may not be necessary until some subprogram or object in the package is actually accessed, which need not necessarily occur until after execution of the main program begins. Implementations, however, are currently forbidden to delay (or avoid) elaboration of such packages.

Ada83 already does allow a certain amount of delay. Nothing forbids an implementation to defer linking to the subprograms of a package body until they are actually called, for example, providing that the compiler (and linker) have previously determined that this linking will succeed. Likewise, nothing prevents the deallocation of resources used in a library package if an implementation can determine that no further use will be made of them (or, perhaps, a pragma indicates that further use of the package will render the program erroneous). As always, in other words, semantically transparent implementation choices are always allowed. No change in Ada83 is needed to permit such implementations. Mandating such functionality for all implementations could have

considerable impact, however, and we have not received sufficient call for it to justify a requirement. (The "small" implementation impact given for the out-of-scope items above refers only to the impact of allowing but not requiring implementations to provide the indicated functionalities)

Similarly, an Ada83 implementation could, in principle, defer loading and elaborating a package body that contained object declarations, if it could assure that elaborating those declarations after the start of a main program would have no effect on semantics. As a practical matter, however, assuring such a condition is likely to be so complex as to discourage the attempt. Here again, one could envision a pragma that declared the program erroneous if deferring elaboration had any visible effect, and such a solution is probably possible even with Ada83.

Only with a certain amount of dubious exegesis on the Ada83 standard could one justify deferred elaboration where the precise body chosen is computed at execution (e.g., after acquiring environment information about today's configuration, the system decides to use version 3 of the body of the TELECOMMUNICATIONS package). No amount of exegesis would allow changing (and re-elaborating) the body of a package during execution, using different bodies for distinct instantiations of the same generic package, or re-elaborating a package.

There seems to be some expressed need for such a capability. It is not clear that this has to addressed under the aegis of library unit elaboration, and so the requirement uses the terms "provide the effect". In particular, with certain other extensions currently being discussed (procedure values and possibly object-oriented features), one can arrange a system so that changing a subprogram's implementation amounts to using an implementation-dependent dynamic loading function and then changing a pointer. Changes that facilitate such an approach meet the intent of this requirement. The issue remains of the extent to which any such dynamic (re)-loading facility must be made standard.

It is not clear to what extent the problems posed in (3) are properly addressed in the guise of library unit elaboration control—hence the neutral term "module". RR-0370 suggests that these are problems of unit elaboration, but many of its examples involve tasks, and the termination of tasks waiting on terminate alternatives, for example, and the standard Ada83 "operation" of de-allocating storage does not currently force the shutdown of tasks.

In a somewhat different direction, there have been calls for some form of "persistent package", whose state persists from execution to execution and may even be shared among several concurrent main programs (tasks representing I/O devices are an example). It is almost certainly unrealistic to expect a standard such facility to be designed in all its detail for Ada9X. Much of the desired effect may be specifiable as a standard package (elaborated according to Ada83 rules), whose body would have to be provided by the implementation. If this is the case, and if the capability is not considered important enough to require of all Ada systems, then no language change may be necessary.

!appendix

Several commentators use TEXT_IO as an example of the problems with the ELABORATE pragma. If an implementation of TEXT_IO happens to require a system-specific package, then any programmer using TEXT_IO during library package initialization, it might seem, would be forced to know about that system-specific package and mention it in an ELABORATE pragma. Needless to say, no such program would be portable. The discussion above does not use TEXT_IO as an example, however, because the sentiment of the ARG and the FRT is clearly that such misbehavior is already a breach of AI-325; for the Ada-defined standard packages, no such clauses should ever be required, even in Ada83 (see AI-355).

%reference RR-0004 Pragma elaborate should be transitive

RR-0004 points out that a single statement invoking pragma elaborate is not sufficient to ensure the correctness of an access to the unit designated in the pragma. Potentially many pragma elaborate statements are necessary; hence, the RR suggests a pragma for transitive elaboration. Using a new pragma solves the upward compatibility problem. The RR also suggests allowing pragma ELABORATE to override the transitive elaborate in cases where the latter gives rise to problems.

%reference RR-0218 Make implementation find good library-unit elaboration order

Without transitive elaboration, the pragma must be used continually in the construction of libraries so that clients of the library will not run into difficulties later. RR-0218 points out that "pragma elaborate is used routinely instead of in special situations" which seems inconsistent with the general purpose of pragmas in Ada.

It might also be noted that pragmas are generally used to indicate optional information to a compiler that does not affect correct behavior. Pragma elaborate, on the other hand, must be followed by the compiler to avoid a program error.

%reference RR-0233 Pragma elaborate should be made transitive

The concern for a transitive elaboration is repeated in RR-0233 where it is also mentioned that some environments may not permit the free access to source code that is needed to analyze an elaboration order problem or to insert the source code changes needed to correct it. For this reason, it seems imperative to find a way to ensure the correct elaboration of library packages.

%reference RR-0370 Allow localized use of lib units; repeated elab/deelab of lib units

RR-0370 lists numerous problems that are caused by interactions of the current elaboration and task termination rules with commonly-desired program structures. For example, one cannot restart library tasks by exiting a scope and re-entering, and workarounds involving dynamically-allocated tasks often lead (in the absence of garbage collection or explicit deallocation) to storage leaks. As a solution, it calls for fine control by programmers of elaboration, re-elaboration, and and "de-elaboration" of library packages.

%reference RR-0396 Add lib unit elab ordering rules to reduce need for pragma elaborate

Dissatisfaction over the need to specify an order through pragmas when the correct order is evident in the majority of cases leads to the suggestion in RR-0396 that the rules of elaboration should be changed. The rule change would make illegal certain cases which are now legal with the proper pragma elaborate statements. This is not an upward compatible change, but deserves some consideration because the elaboration problem makes some current programs nonportable.

%reference RR-054 Eliminate need for pragma elaborate; pragma NOT_ELABORATE might help

RR-0546 points out that earlier versions of Ada did not have this problem because complex program structures were illegal. The LRM now allows these cases at the cost of requiring the majority of legal programs to state extensive elaboration instructions to the compiler through pragma elaborate. This RR suggests it would have been preferable to elaborate most programs automatically and provide a different pragma that defers elaboration in those cases where the designer has introduced a subtle order effect. Thus, a change to the rules is suggested with the added provision for a new pragma that would allow modification of the order so that programs that become illegal under the new rules can be compiled with an intentionally specified elaboration order. This is essentially the approach recommended in the requirements stated above.

%reference RR-0767 Solve elaboration order problem properly

RR-0767 provides a broad view of the problem and suggests that a satisfactory solution to this deeply rooted problem can be reached after a through study. While the study here may be less than the RR envisions, the results indicate that no totally satisfactory solution exists. The problem arises through an overabundance of features in Ada whose interaction is not always benign during elaboration.

As an aside, one might note that part of the problem is the provision in Ada of too many initialization features. On the other hand, Ada has no finalization features. In rectifying this asymmetric feature set, one should certainly stop short of providing so many finalization features that problems arise during finalization that mirror the elaboration order difficulties during initialization!

%reference AI-00354 On the elaboration of library units

AI-0354 reiterates the position stated in the Ada83 LRM to the effect that the elaboration of a body is not required to occur at any point earlier than required by the rules stated in the LRM. Thus, elaboration orders that obey the rules but lead to a run-time program error are, in fact, legal.

%reference AI-00355 Pragma ELABORATE for predefined library packages

AI-0355 requires that implementors provide control of elaboration order for the implementation defined packages. In effect, it requires the full use of pragma elaborate so that no access before elaboration will occur in the implementation defined packages. This direction to the implementors corresponds to the programming conventions mentioned earlier that require the use of pragma elaborate throughout the collection of software to prevent improper elaboration order.

%reference AI-00421 Eliminate pragma ELABORATE

AI-0421 contains an extensive discussion that is too lengthy to summarize here but it should be consulted for a complete understanding of the elaboration problem.

!rebuttal

!topic Library unit and subunit naming
!number RI-2109
!version 1.4
!tracking 5.12.2

DRAFT DRAFT DRAFT DRAFT DRAFT DRAFT

!Issue revision (1) important,small impl,upward compat,consistent

1. (Allow subunits with same simple name) Ada 9X shall allow different subunits that have the same ancestor library unit to have the same identifier, at a minimum in the case where subunits with the same simple name occur in different parent units.

!Issue revision (2) desirable,moderate impl,upward compat,consistent

2 - Allow separate compilation of overloaded subprograms and operators] Ada 9X shall not restrict the names of subunits; in particular, it shall be possible to separately compile subunits with the same expanded name (i.e., homographs) or functions whose name is given by an operator symbol.

!Issue out-of-scope (un-needed) (3) desirable,severe impl,moderate compat,consistent

3. (Allow overloaded subprograms at library level) Ada 9X shall allow subprograms that are homographs to be declared as library units.

[Rationale] Although this requirement would remove what can be viewed as an inconsistency in the language definition, it is felt that the linguistic mechanism that would be necessary to differentiate the particular subprogram in a WITH clause would be too costly compared to the minor benefit that would be gained. This in itself could be argued, because the semantics of a WITH clause could be to import all the subprograms with that name that are visible in the library, and apply the normal overload resolution algorithm; this could probably be made to work except that in that case the implications on the complexity of the library management system would be non trivial: for instance, when one tries to compile a subprogram for which one with a compatible signature already exists, is it meant to be a recompilation of the old one, or an illegal (but unintended) overloading? Also, it would likely not be compatible with the library management system of several existing compilers.

!Issue out-of-scope (un-needed) (4) desirable,small impl,upward compat,mostly consistent

4. (Allow separate compilation of nested units) In Ada 9X, it shall be possible to separately compile any secondary unit, irrespective of where the corresponding declaration appears. In particular, it shall be possible to separately compile the body of any unit, even if its parent unit is not a compilation unit.

[Rationale] If the separate clause is not sufficient to fully disambiguate the unit that is referenced, then this requirement would introduce more problems than it solves. Treating

library units and subunits differently is not desirable; if overloaded subprograms are not allowed at the library level, then they should not be allowed either at the subunit level. This problem could be solved elegantly if the rule for renaming declarations was changed to allow a renaming clause to appear as a body.

!Issue out-of-scope (un-needed) (5) important,severe impl,moderate compat,inconsistent

5. (Allow library units with same name, provide subsystems) Ada 9X shall provide a mechanism to limit the visibility of certain library units to a set of specified compilation units. Library units whose visibility is restricted to disjoint sets of compilation units will be allowed to have the same name.

[Rationale] This requirement addresses the problem of programming large systems where name clashes introduce either organizational problems or integration problems; it also aims at improving the amenability of the language to reflect hierarchical designs, where CSCIs may be refined in various levels of CSCs before CSUs are introduced. However, such a requirement can be seen as reaching beyond the definition of the language itself, and into the realm of the environment. Such a requirement can also be viewed as an attempt at providing feeble solutions to the much wider problem of configuration management, with the risk of getting in the way of more comprehensive solutions that could be introduced outside the language.

!reference RR-0038
!reference RR-0041
!reference RR-0073
!reference RR-0178
!reference RR-0402
!reference RR-0557
!reference RR-0607
!reference WI-0212
!reference AI-00458
!reference AI-00572

!problem

The major problem addressed here is the number of limitations on what can be separately compiled: in 1815A, the general model that any unit body can be compiled separately with the exact same effect as leaving it where it is originally declared suffers from disconcerting anomalies: operators cannot be separately compiled; overloaded subprograms cannot be separately compiled; and subprograms that are nested inside other units cannot be separately compiled if their parent unit is not itself a compilation unit. The most irritating of these restrictions is the one expressed in LRM 10.2(5), whereby all subunits with the same ancestor library unit must have different simple names.

All these restrictions were introduced in the hope of simplifying the implementation of the library management system; however, most of today's compilation system offer much more sophistication than the simple model that the language describes.

It is felt that the requirements given here will not place any major burden on existing implementations.

!rationale

Requirement 1 only removes the unnecessary limitation to have all subunits with the same ancestor library unit have different simple names; this is seen as a minimum requirement because it does not call for allowing overloaded subprograms in the same subunit to be separately compilable.

This latter requirement is included in requirement 2. It is only desirable, because it could be satisfied reasonably simply by allowing renaming clauses to appear in lieu of subprogram bodies. In this case, it would be trivial to rename the body of a subprogram, without affecting its specification; this would also provide a satisfactory solution for operators.

!appendix

Allowing separate compilation of subunits with the same simple name when they belong to different parent units may potentially lead to the need to disambiguate the reference to the parent unit in the separate clause, leading, e.g., to a form like

```
separate (A.B.C)
```

which is admittedly more awkward. It is anticipated that this need for qualification will rarely be encountered in practice: in the vast majority of the cases, only one level of nesting is likely to be encountered; the real need occurs when one wants to separately compile two subprograms that have the same name and that are declared in two different packages (the packages having different names).

Another case where the expanded name would have to be used is in the context of requirement 4, i.E., when the subunit is nested arbitrarily deep in the parent compilation unit.

The case of requirement 2 does not necessarily call for an extension of the syntax rules: the subprogram header in the real body will be sufficient to identify which subprogram is concerned; it is only at the level of library tools that some additional information will have to be provided when listing the contents of a library, or designating some units. The only case that would cause a real problem is if one tries to separately compile a subunit that is itself nested in an overloaded, separately compiled subprogram. Can this case suffer the introduction of a restriction?

It should be pointed cut that this RI does NOT call for a rule requiring the parameter/result profile of a subprogram to appear in a separate clause for reasons of disambiguation.

!rebuttal

Issue 4, which is marked out-of-scope, calls for allowing an arbitrarily nested unit body to be separately compiled. The original restriction had intended to limit the amount of symbolic information that had to be kept around by the library management system. Since complete information is generally left around for the debugger, it is believed that this requirement will not cause significant implementation problem.

!topic Dependencies and Recompilations
!number RI-2110
!version 1.2
!tracking 5.8.15

DRAFT DRAFT DRAFT DRAFT DRAFT DRAFT

!introduction

This RI addresses various problems linked to the notion of dependencies between compilation units, and its impact on recompilation. This RI does NOT address the issue of the structure of the program library, or of its interfaces.

!Issue out-of-scope (un-needed) (1) important,severe impl,upward compat,consistent

1. (Recompilation containment) When a compilation unit is recompiled, recompilation of its dependent units will only be required for those units that are EFFECTIVELY affected by the changes made to the original unit. Ada9X will provide a minimal list of the allowed changes to a compilation unit that will NOT affect a dependent unit.

[Rationale] LRM 10.3(5) defines how a change in a compilation unit "potentially affects" a dependent unit, and further indicates that a dependent unit becomes obsolete if it is "potentially affected" by a change in a compilation unit. Although the LRM allows an implementation to reduce the amount of recompilation that would be necessitated, most implementations provide only a strict interpretation of the LRM, and force recompilation of all dependent units without any consideration for the changes actually made. This is sometimes considered as irritating, especially if the change only affects a comment!

HOWEVER, imposing a language change such as the one described in (1) would have a drastic impact on a large number of implementations: since it is in general impossible to control how the modification is made (for instance, the change may result from an automatic process, echoing a change made to a design diagram), all implementations would be forced to perform some kind of comparison between the old and the new version. This is a penalty that would have to be paid by all users, and it may also affect the performance of compilers in more subtle ways.

It seems better advised to let implementors have the option to treat this problem as a place for compiler optimization. If this becomes a recurring demand, implementors will be driven to provide it, possibly in more sophisticated ways.

Giving a list of features that should not affect recompilation is dangerous, because it will make assumptions on implementation techniques which may be either too complex or too naive. Even the simplest recommendation, e.g., a change in a comment, may turn out not to be a good candidate if this comment happens to contain an assertion in a language like ANNA.

!Issue out-of-scope (un-needed) (2) important,small impl,upward compat,consistent

2. (Deletion of obsolete bodies) When a change in a library package results in a specification that does not require a body, and if a body for that unit had been previously compiled, then the recompilation of the specification shall cause the corresponding obsolete body to be deleted from the program library.

[Rationale] The problem addressed here stems from the Note LRM 10.3(16). It may be the case that a change to a package specification causes the body to be no longer necessary, e.g., if the change corresponds to deleting all subprogram declarations. If the obsolete body is not deleted, then it may be inadvertently linked with the application, and may cause unwanted effect. Although it would be a trivial change to force the body to be deleted when the specification is recompiled, this is not necessarily desirable: the fact that a body is not necessary does not mean that there should not be one; in particular, it may be necessary to perform some initializations in a sequence_of_statements, or the body may contain a task declaration that does all the work. Leaving the obsolete body in the library gives a system the opportunity to detect that the body is obsolete, and emit a warning (in fact, any decent linker should notice that it is attempting to link a body that is out of date). Deleting the body will on the other hand result in losing track of the need for such a body, leading to its inadvertent omission in the case where it is needed. In fact, from a configuration management point of view, if there had been such a body, it is preferable to leave an empty one than to delete it altogether. Note that in the case where the body is deleted from the library but is still desired, it would be omitted by the automatic recompilation tools provided by many implementors. It seems therefore preferable to let compilers provide an option to force the deletion of dependent units, if this is indeed what is desired.

!Issue out-of-scope (un-needed) (3) desirable,severe impl,upward compat,consistent

3. (Recompilation of bodies in library units) Recompilation of a subprogram body that also acts as a library unit shall not render obsolete the (implicit) library unit declaration unless the subprogram specification has actually been modified.

[Rationale] LRM 10.1(6) allows a subprogram body to be compiled without any previous corresponding subprogram specification; this feature is

primarily intended for teaching the language, as it allows one to write, compile and execute simple programs without being taught anything about units and separate compilation. When such form of library subprogram is used, a change to the subprogram body will induce a recompilation of the (implicit) subprogram specification, and will therefore "potentially affect" all the units that refer to the subprogram in a with clause. It would be therefore highly desirable to get rid of this unwanted recompilation effect.

The primary reasons for rejecting this change are a. that it is trivial (and generally a good practice) to circumvent this effect by providing a proper subprogram specification; b. that the implications on implementations are far from trivial, as it would require implementations to detect whether the subprogram specification has been modified or not.

!reference RR-0065
!reference RR-0142
!reference RR-0688
!reference RR-0689
!reference AI-00400

!problem

The rationale given with each of the requirements describes the corresponding problem. Although these problems are valid, the general implication would be to go from a model of separate compilation to one of incremental compilation, in the sense that each implementation would be required to determine what has changed from the previous version when a unit is recompiled.

This determination can be relatively easy when a syntax-oriented front-end is being used, but it otherwise forces a compiler to compare the internal representations of two versions, and understanding the significant differences is not an easy task (actually, even a syntax-editor can be defeated, e.g. if one changes a complete declarative item in order to modify the initialization part).

!appendix

%reference RR-0065 Differentiate between compilation and post-compilation information

RR-0065 discusses the general problem brought forth here, but from the point of view of those changes that do not affect code generation at all, but only program construction. Those cases are only a subset of the more general problem discussed in (1).

%reference RR-0142 Reduce cases where recompilation of bodies and subunits is needed

RR-0142 addresses the general problem of recompilation as covered by (1). It calls for a list of features that can be changed without affecting dependent units.

%reference RR-0688 Need to re-consider dependency rules for subpgm library units

RR-0688 addresses the problem covered by (3).

%reference RR-0689 Disallow linking with obsolete optional bodies %reference AI-00400 Obsolete package bodies

Both RR-0689 and AI-00400 discuss the problem of obsolete package bodies, as presented in (2).

!rebuttal

!topic Access to a standard set of Ada compiler library operations.
!number RI-3572

!reference RR-0177
!reference RR-0226
!reference RR-0237
!reference RR-0368
!reference WI-0209
!reference WI-0209M
!reference WI-0210
!reference WI-0210M
!reference WI-0211

!problem

It is perceived that the difficulty in integrating an Ada compiler with other elements in the software development process such as a configuration management system or Integrated Project Support Environment (IPSE) is due to lack of standardization of an interface to the Ada library and inability to get at information only available on a proprietary basis. It is also claimed that this lack of a standard interface to the Ada library is the reason for the difficulty in using multi-vendor environments for various target environments.

!topic Program building
!number RI-3154

!reference

RR-0171 Separate program build-info from source
RR-0283 Need convenient way to set global compilation parameters
RR-0373 Need to declare constants whose value is supplied after linking
RR-0698 Need pragma to identify machine-dependent pieces of program

!problem

When an Ada program is built, often specific information is required to guide the tools. Some of this information, such as representation specifications, is contained in the source code next to the program units it modifies, while other information is linker or loader specific. This section addresses how complete this source-level information is, and if it the target-dependencies it introduces are acceptable.

!topic Controlling the compilation process.
!number RI-3279

!reference RR-0065
!reference RR-0171
!reference RR-0283

!problem

Ada83 provides a number of pragmas to control the compilation process. These pragmas are included inline in source programs, and must therefore be subject to the same requirements as source programs (configuration management, library updates, etc.) during the software development process. Changes that affect only such compiler directives may require significant unnecessary recompilation because of cascading effects. Many vendors have provided implementation dependent options (primarily through the use of command line arguments) to allow this information to also be specified on a per-compilation basis.

!topic Subunits
!number RI-2111
!version 1.3
!tracking 5.12.8

DRAFT DRAFT DRAFT DRAFT DRAFT DRAFT

!Issue revision (1) compelling,severe impl,upward compat,inconsistent

1. (Compilation Containment) Ada9X shall define a mechanism by which a new program unit may be constructed by effectively adding package and subprogram declarations to another existing unit. Ada9X shall define a means to designate such new units in a context clause. Changes in the newly created unit shall not force recompilation of the old unit or any clients of the old unit that do not name the new unit.

!Issue out-of-scope (un-needed) (2) desirable,moderate impl,upward compat,consistent

2. (Internal Subunits) Ada9X shall be defined so that subunits may appear at other than the outermost compilation level. Specifically, subunits should be allowed whenever the name of the parent declarative region is unambiguous.

[Rationale] While this facility may be desirable, it does not seem that the need is sufficient for a requirement.

!reference RR-0545
!reference RR-0154
!reference RR-0262
!reference RR-0448
!reference RR-0684

!problem

Ada was intended to support separate compilation to a large degree. However, users are still finding that wholesale structural changes are needed to effect compilation containment. For example, a procedure may be needed to emit some debugging information. If this procedure needs visibility of some private types, then it must be added to the specification and therefore all users of the package would need to be recompiled.

!rationale

RI-2111.1 would solve a frequent problem in system development where massive recompilation is required simply to add a procedure to a system that requires high visibility of a package's internals.

!appendix

%reference RR-0545 Subunits should not have to be at outermost compilation unit level

%reference RR-0154 Subunits should not have to be at outermost compilation unit level

RR-154 and RR-545 do not want to restrict subunits to the outermost nesting level.

%reference RR-0262 Do not require existence of subunit for body stubs

RR-262 does not want to require a stub for a subunit arguing that the linker can create the stub automatically.

%reference RR-0448 Allow separate compilation for subunit spec

RR-0448 proposes a mechanism for adding separately compiled units to a parent unit after compilation of the parent. Visibility of these added units is gained by other units by an extended WITH syntax.

%reference RR-0684Don't cram knowledge of a private type into a single package

RR-684 brings up a very commonly occurring problem in Ada: that different users need different views of a package based on the level of naiveness (if you will). The specific proposal is to modify the WITH clause so that an importer could specify that the private stuff is to be visible.

!rebuttal

!topic Context clause restrictions
!number RI-2112
!version 1.1
!tracking 5.12.9

DRAFT DRAFT DRAFT DRAFT DRAFT DRAFT

!Issue out-of-scope (un-needed) (1) desirable,small impl,upward compat,consistent

1. (Eliminate Name Restrictions in Context Clauses) Ada9X shall not require that the names mentioned in USE clauses come from the same context clause.

[Rationale] The need for this facility does not seem to support a requirement.

!Issue out-of-scope (un-needed) (2) desirable,small impl,upward compat,consistent

2. (Eliminate Need for Body) Ada9X shall allow an ELABORATE pragma to be specified for a (library) package with no body.

[Rationale] The need for this facility does not seem to support a requirement.

!problem

!rationale

!appendix

%reference RR-0095 Extend context clauses regularly to body/subunits

RR-95 wants the restriction that the names in USE and ELABORATE constructs must come from the same context clause; the argument is that names from any applicable context clause should be allowed.

%reference RR-0581 Remove unnecessary restrictions on pragma elaborate

RR-581 wants the restriction on names removed as in RR-95; in addition, the proposal is that ELABORATES should be allowed to be interspersed in context clauses and also that one should be able to elaborate units that have no body.

%reference AI-00226 ???

AI-226 simply explains that context clauses for a package/procedure do in fact apply to subunits of the BODY.

!rebuttal

5.13 Generics

RI-1012
RI-1013
RI-1014
RI-1015
RI-1016

!topic generalize parameterization allowed for generic units
!number RI-1012
!version 1.8
!tracking 5.13.1

DRAFT DRAFT DRAFT DRAFT DRAFT DRAFT

!Issue revision (1) important

1. (Generalize Generic Parameterization) The allowed parameterization of generics shall be generalized in Ada9X to the maximum extent possible within the constraints of (1) not damaging the fabric of the language and (2) not imposing undo hardships on implementations. In particular, for any kind of declaration in Ada9X, consideration shall be given to allowing an entity of this kind as a generic formal parameter.

!reference RR-0228
!reference RR-0383
!reference RR-0408
!reference RR-0424
!reference RR-0455
!reference RR-0468
!reference RR-0486
!reference RR-0488
!reference RR-0505
!reference RR-0621
!reference RR-0627
!reference RR-0659
!reference RR-0671
!reference RR-0706
!reference RR-0722
!reference RR-0752
!reference WI-0213
!reference AI-00451
!reference AI-00452

!reference
Bardin, B. and C. Thompson, "Composable Ada Software Components and the Re-Export Paradigm," Ada Letters, Vol. VIII, No. 1, January 1988, pp. 58-79.

!reference
Cohen, N., Summary of the OOP session at the Ada Reuse Workshop, Deer Isle, Maine, September, 1989.

!problem

The allowed parameterization of generic units in Ada83 is relatively restrictive. The language would allow more re-usable and modular software to be developed if there was some way generic units could be parameterized with respect to entities such as exceptions, record types, named numbers, task types, numeric types, packages, subtypes, derived types, task entries, and generic units. In attempting to develop truly reusable software, Ada83 programmers have found that the abstract functionality required in a reusable module cannot be programmed in a natural and convenient way due to restrictions on the way generic units may be parameterized.

!rationale

This is a high-level requirement with no specific compliance criteria because it is not clear exactly what will be needed along these lines in Ada9X. For example, potential changes in Ada9X in the nature of exceptions or towards the end of object-oriented programming seem to have a significant impact on the appropriate parameterization of generics in Ada9X. Hence the guidance here is to attempt to generalize the allowed parameterization of generics as seems appropriate within the context of other language changes both in an effort to improve the usefulness of generics and to make the language more uniform. It is noted that with respect to Ada83, potential candidates for increased parameterization of generics seem to be: exceptions, entries (that is, being able to treat a generic parameter as an entry within the generic unit), records (matching a required set of components), derived types, subtypes, and packages (either matching a required set of components or being any instantiation of a particular generic).

!appendix

%reference RR-0424 Allow pkg instantiation to control names of visible decls

Ada9X shall allow a module to be parameterizable wrt the names of the entities it exports.

%reference RR-0486 Allow generic formal task types as well as generic formal ltd types

Ada9X shall allow a program unit to be parameterizable wrt task types.

%reference RR-0505 Provide extendable record types, records as generic parameters

Ada9X shall allow code to be written that is parameterizable wrt record types having a common core set of components.

%reference RR-0627 Need a generic formal type for records

Ada9X shall allow code to be written that is parameterizable wrt sub-record types, including types of sub-record components.

%reference RR-0706 Allow exceptions and packages as generic parameters

Ada9X shall allow code to be written that is parameterizable wrt packages and exceptions.

%reference RR-0722 Need generic formal record types

Ada9X shall allow code to be written that is parameterizable wrt record types, including types of components.

%reference RR-0228 Allow generic parameterization with exceptions
%reference RR-0383 Need generic exceptions for truly re-usable generic units
%reference RR-0468 No generic way to handle exceptions raised by generic fml subpgms
%reference RR-0621 Generalize facilities avble for exceptions for power/flexibility
%reference RR-0671 Allow exceptions as generic parameters
%reference RR-0752 Make various improvements to exception handling capabilities

Ada9X shall allow code to be written that is parameterizable wrt exceptions.

%reference WI-0213 Allow all program units to be generic formal parameters

Ada9X shall allow code to be written that is parameterizable wrt any kind of Ada entity, including tasks, packages, entries, generics, records, and numerics.

%reference RR-0408 There is a need for generic formal entries
%reference RR-0488 Allow generic formal entries as well as generic formal subpgms
%reference RR-0659 Need to make entry call on a generic formal parameter
%reference AI-00451 Need entries as formal generic parameters

Ada9X shall allow code to be written that is parameterizable wrt task entries.

%reference AI-00452 Need generic record types

Ada9X shall allow code to be written that is parameterizable wrt record types, including number of and types of components.

%reference RR-0455 The import and export mechanisms of Ada are too limited

Ada9X shall allow code to be written that is parameterizable wrt named numbers, exceptions, subtypes, derived types, record types, packages, generic subprograms, generic packages.

The following is a more-concrete set of requirements that was once proposed by the RT. It is included here for information purposes only. Reviewers felt (1) the need for some of the specifics asked for was unconvincing and (2) it was a mistake to put forth concrete requirements for improved parameterization of generics outside the context of many other potential changes to the language.

%requirement 2 3 3 2

[1] Ada9X shall provide a facility for parameterizing a generic unit with respect to an exception. It shall be possible to both raise and handle such an exception within the generic unit.

%requirement 1 2 3 2

[2] Ada9X shall provide a facility for parameterizing a generic unit with respect to a task entry that conforms to a given parameter profile (analogous to the conformance requirements for a generic formal subprogram).

%requirement 1 3 3 3

[3] Ada9X shall provide a facility for parameterizing a generic unit with respect to a subtype of a particular base type.

%requirement 2 2 3 2

[4] Ada9X shall provide a facility for parameterizing a generic unit with respect to a general aggregation mechanism (e.g., a record type or package). Conformance rules for such a parameter shall be based on a set of required components assumed in the generic unit. The aggregation mechanism used for this purpose shall allow subprograms as components.

%out-of-scope 1 2 3 1

[5] Ada9X shall provide a facility for parameterizing a generic module with respect to the names of the entities that it exports.

[Rationale] This notion appears contrary to the design goals for Ada generics. Other

visibility-control changes in Ada9X could potentially solve this problem not only for a package obtained by instantiation but for uses of a non-generic package as well.

%out-of-scope 1 2 3 2

[6] Ada 9X shall provide a facility for parameterizing a generic unit with respect to a numeric (i.e., integer, fixed-point, or floating-point) type.

[Rationale] This does seem somewhat desirable but difficult to accomplish in a useful way with the present language due to the different ways arithmetic operators "*" and "/" work with respect to fixed-point and floating-point types. Using a private type and passing in the required operators and constants as additional generic parameters does not seem too unreasonable as a workaround.

%rationale

Requirements [1]-[4] above identify what appear to be the most important needs concerning increased parameterization of generic units. That is not say that the other forms of generic parameterization would be of no value; only that in the interest of limiting language complexity and minimizing impact on existing implementations, the line needs to be drawn at some point.

The need for generic parameterization with respect to exceptions is relatively strong; this need will be lessened slightly if exception parameters are added to the language (e.g., generic formal subprograms could be passed exceptions to be raised). Introducing generic formal exceptions into the language would require special-purpose rules concerning duplicate exceptions in generic exception handling code. However, this additional language complexity appears to be a worthwhile price to pay for this improved functionality of generic units.

The need for generic parameterization with respect to entries is also significant; the workaround of passing a subprogram as a generic formal that performs the conditional or timed entry call is distasteful. Generic formal entries seem to be relatively easy to add to the language. An alternate approach of allowing timed and conditional entry calls on generic formal subprograms also seems like a fairly easy language change to make.

Generic parameterization with respect to subtypes is an important part of re-export paradigm approach to reuse discussed in [Bardin88]. It is also convenient for developing subprograms in generic units that operate on data of the same generic type but with different constraints (e.g., as is sometimes appropriate in generic math packages).

Generic parameterization with respect to an aggregation mechanism such as a record type or package would provide important support for the object-oriented programming style as well as for the re-export paradigm of software reuse referenced above.

!rebuttal

!topic parameterization beyond generics
!number RI-1017
!version 1.7
!tracking 4.6 5.13.1

DRAFT DRAFT DRAFT DRAFT DRAFT DRAFT

!Issue out-of-scope (un-needed) (1) desirable,moderate impl,upward compat,inconsistent

1. (Generic Record-Object Processors) Ada9X shall allow code to be written that operates on the components of a record type where no knowledge of the names and types of the components or the number of components is built into the code.

[Rationale] This proposal makes little sense for a strongly typed language such as Ada. If this functionality is required, it clearly should be addressed outside the language.

!Issue revision (2) desirable,small impl,upward compat,mostly consistent

2. Ada9X should provide further support for the notion of conditional compilation beyond that provided in Ada83. In particular, it shall be possible to conditionally include (or omit) a set of declarative items in a declarative part (or package specification) at compile time. Similarly, it shall be possible to conditionally include (or omit) a set of context clauses or a set of component declarations in a component list.

!reference RR-0027
!reference RR-0174
!reference RR-0343
!reference RR-0529

!reference AI-00452

!reference RI-1011
!reference RI-1012

!problem

The facilities in Ada83 appear to be inadequate for the construction of truly re-usable, modular, and easily-maintained software. This problem is partially addressed by the requirements for generalizing the allowed parameterization of generic units discussed in RI-1012. The present subject, however, is support for re-usable software components beyond that which is felt to be providable by extending Ada generic units.

As an example, consider a package containing some data and a set of operations on that data. Further suppose that for certain multi-tasking applications, there is a need for mutually-exclusive access to the data during these operations, and, for other applications, there is desire not to force mutually-exclusive access to the data. For maintainability reasons, there is a need to keep a single copy of the package but there is also the need to allow these two kinds of uses of the package.

As a second example, suppose there is a need for a package that deals with objects of an arbitrary record type. Furthermore, in the package, suppose there is a need to sequentially process each of the individual components of these objects without building anything into the package concerning the number of and types of the components (see, e.g., AI-00452).

As a final example, consider the need to develop a new software unit that is a slight variant of an existing unit in the sense that it requires several additional (or different) items (e.g., WITH clauses, pragmas, declarations, and statements) beyond those provided in the original unit. The difficulty here is the development of the new unit in a way that does not introduce maintenance problems in the software that is common among the units.

!rationale

Generic units are inherently limited in Ada due to the safety that is built in to their design and due to the fact that generic units are meant to be compilable on their own. It is limitations such as these that make them inadequate for solving the problems described above. It does not seem to be desirable to alter these characteristics of Ada generics units to make them drastically more powerful. Generic units in Ada do provide good solutions to a number of software engineering problems and it seems unwise to alter the principles behind their design.

One potential way of addressing at least some of these problems is a language change that extends the support for conditional compilation. Support for conditional compilation in Ada83 is limited to regions of program text containing sequences of statements. There is also no requirement that this form of conditional compilation actually be supported by an implementation. Generalizing this support and requiring compliance would lessen the kinds of problems described above.

!appendix

%reference RR-0027 Need additional record type attributes

There is a need to write a generic report generator that operates on data on an arbitrary record type. The RR suggests the addition of a variety of attributes that would allow the report generator to iterate through the fields of the record, without building the number of fields (or the types of the fields) into the code.

%reference RR-0174 Allow packages to be generic wrt concurrency protection

There is a desire here for a single package that is parameterized with respect to whether concurrency protection (forced mutually-exclusive access) to the data in the package is desired. Without concurrency protection, the package is a normal package. With concurrency protection, the package is approximately a task with the visible subprograms of the package being entries. Hence with concurrency protection, the data in the package is only accessed by a single thread of control. It is important that only one "copy" of this package be maintained.

%reference RR-0343 Provide better facilities for conditional compilation

Wants conditional compilation facilities for easy maintenance of multiple "versions" of a module, e.g., a "production" version, a "debug" version, a space-optimized version, etc. Ada "conditional compilation", i.e., if statements with static boolean expressions, does not do the job for conditional declarations or conditional WITHs.

%reference RR-0529 Provide dynamic run-time types

There is a desire here to allow user-software to access run-time type descriptors so that very generic software modules can be written that can operate on data of a wide of a wide variety of types. The software would use the type descriptor to know how to properly interpret the data it is to process.

%reference AI-00452 Need generic record types

The desire here is for a generic formal record type capability with no built-into-the-code assumptions about number and types of any required components. Coupled with this idea are attributes that can be appled to an object of the generic formal record type within the generic unit to obtain the values (and types?) of the components.

!topic Contract model for generics
!number RI-1013
!version 1.8
!tracking 5.13.2

DRAFT DRAFT DRAFT DRAFT DRAFT DRAFT

!Issue revision (1) important,moderate impl,upward compat,consistent

1. The legality of an Ada9X generic instantiation shall not depend on the ways in which generic formal parameters are used within the body of the generic unit being instantiated.

!reference RR-0006
!reference RR-0342
!reference RR-0446
!reference RR-0472
!reference RR-0549
!reference RR-0584
!reference WI-0213

!problem

The purpose of a generic specification in Ada is to act as a "contract" between the instantiator and the generic unit. This contract is intended to capture the assumptions that underlie the correct usage of the generic. Generic specifications in Ada83 do not completely fulfill this role. In particular, a generic instantiation is illegal if an unconstrained subtype is passed for a formal private type and the formal private type is used in the generic body in a place that would require a constrained subtype. This fact contributes to the undesirable dependencies between instantiations and generic bodies, makes generic code sharing more difficult, is confusing to users, significantly hampers program maintenance, and necessitates the LRM 12.3.2(4) checks which are difficult and inefficient to implement.

!rationale

It seems this problem is genuine and important to solve; the only real requirements issue is whether a good solution can be found. The RR-0006 solution is upward compatible and in the right direction but is not sufficient for compliance with the requirement above. The RR-0472 solution is upward compatible, complies with the requirement, but has the disadvantage of putting off detection of Ada83 error described above until run time. A third possibility is a non-upward compatible change requiring distinction between constrained and unconstrained formal private types. Finally, a fourth possibility is removing from the language in some way the restrictions on the uses of unconstrained subtypes (e.g., allow variable declarations for objects of unconstrained subtypes).

The above requirement is justified on the basis of a strong need for fixing this problem together with the fact that some of the solutions mentioned above appear reasonable (e.g., the RR-0472 solution).

!appendix

The RT looked at three possible contract model problems for this RI. Besides the unconstrained/constrained problem described above, these were:

(1) legality of instantiation depending on the body of the generic unit due to the recursive instantiation error, and

(2) the alleged contract model problem in the Ada Rationale:

```
generic
 type T is private;
package OCTETS is
 type R is
  record
   A : T;
  end record;
 for R'SIZE use 8;
end;
```

Item (1) here seemed like not really a contract model problem and in any case hard to fix without simply putting off the detection of the error until run-time. Item (2) does not really appear to be a problem since the above length clause violates 13.2(6), regardless of how the generic unit is instantiated.

%reference RR-0006 Distinguish unconstrained/constrained generic formal types

Improve the contract model.

%reference RR-0342 Don't implement requests which will break generic code sharing

Code-sharing compilers are hard to build because the contract model is inadequate.

%reference RR-0446 Re-store contract model; distinguish const./unconst. generic types

Fix the contract model and don't break it again.

%reference RR-0472 Distinguish unconstrained/constrained generic formal types

Gives an upward compatible solution that allows making this distinction.

%reference RR-0549 Eliminate dependence on body for validity of generic actuals

Distinguish constrained and unconstrained private types.

!rebuttal

!topic support for generic code sharing
!number RI-1014
!version 1.6
!tracking 5.13.3

DRAFT DRAFT DRAFT DRAFT DRAFT DRAFT

!Issue revision (1) important

1. To the fullest extent possible given other constraints on the language revision, Ada9X should allow an efficient implementation of generics via code-sharing.

!Issue revision (2) desirable,small impl,upward compat,consistent

2. Ada9X shall include a mechanism for indicating whether shared or inline implementation of a generic unit is desired.

!reference RR-0005
!reference RR-0342
!reference RR-0445
!reference RR-0584
!reference RR-0585
!reference RR-0586
!reference RR-0693
!reference AI-00409

!reference RI-1011

!reference Rationale 12.4.1

!reference
Fowler, F. J.,"A Study of Implementation-Dependent Pragmas and Attributes in Ada", Software Engineering Institute Special Report SEI-89-SR-19, November 1989.

!reference
Gary Bray, "Implementation Implications of Ada Generics," Ada Letters 3, No. 2 (September-October 1983), pp. 62-71

!problem

In Section 12.4.1 of the Rationale, it is stated that the nature of generic units in Ada "offers distinct advantages in terms of efficiency, since compilers can easily identify the existing instantiations and, in some cases, achieve optimizations such as sharing of code among several instantiations of the same generic unit."

Experience with Ada suggests that there are several problems associated with this idea of sharing code among instantiations of a generic unit.

First, there are characteristics of Ada that make it difficult for a compiler to efficiently implement shared-code generics. These include the nature of exceptions (not being created by elaboration), the rather weak checking of constraints that is required for a generic instantiation, the fact that pass-by-copy semantics are required for generic formal private types when the corresponding actual type is scalar, and the instantiation-dependent rules on the order in which generic actual parameters are evaluated.

Second, there are characteristics of Ada that are intended to allow shared-code generics, but, unfortunately, also severely limit the usefulness of Ada generic units. These include the rules of Ada that disallow expressions derived from generic formal parameters from being static and the rules concerning generic formal types with respect to case statements and discriminants.

Third, just as there exists a pragma for subprograms that states a special desired implementation of the subprogram (pragma INLINE), it would be useful if there were a similar pragma that states a desired implementation of a generic instantiation (e.g., pragma SHARED or pragma INLINE).

!rationale

Concerning the first aspect of the problem discussed above, it is difficult to assess the merit of any of the specific proposals for making generic code-sharing easier or more efficient without considering other potential language changes that might be made during the 9X process. Hence we simply note the need to allow an efficient implementation of generics by code-sharing and suggest that the specific suggestions described in the referenced RRs be considered during the Ada9X mapping effort.

The second problem aspect discussed above is covered under RI-1011.

A pragma that can be used to specify a desired implementation strategy for a generic instantiation, while not critical, would be desirable in the language. Although there are implementations that support this need with an implementation-defined pragma, Ada software would be more portable if the nature of the pragma were standardized as part of the language (see [Fowler89]). [Bray83] discusses the various degrees in which generics can be shared.

!appendix

A Distinguished Review provided the following discussion on the problems mentioned in the third paragraph of the problem section, above:

- The first point addressed is "the nature of exceptions (not being created by elaboration)...." This must be a reference to the rule in 11.1(3) that "The particular exception denoted by an exception name is determined at compilation time and is the same regardless of how many times the exception declaration is elaborated." This inconsistent rule, intended to cope with the possibility of an exception declaration in the declarative part of a recursive subprogram, leads to some difficult dilemmas discussed in AI-00336. One of these dilemmas is whether an exception declaration in a generic template results in one exception per

instantiation or one exception shared by all instantiations. AI-00336 (alas, still just a work item, though it has been discussed at least once by the ARG) suggests that each instance should be viewed as having its own copy of the declaration. However, it is not clear what should happen if the template is declared in the declarative part of a recursive subprogram! In any event, the problem here is with exception declarations, not generics.

- The next point addressed is "the rather weak checking of constraints that is required for a generic instantiation." I had to turn to RR-584 to get any inkling of what the real issue is here. The writer of the RR objects that the constraints implied by the type mark of a generic formal object, or the type mark of a generic formal subprogram parameter or result, are ignored. (It is the analog of the objection raised in RI-5061 for renaming declarations. Since the writers of the RM strove meticulously to make generic formal parameter declarations behave like renaming declarations, it would be a good idea to consider these issues together.) At first glance, the fact that

```
generic
  type T is (<>);
  with procedure P(X: in out T);
package G is
  ...
end G:
```

really means

```
generic
  type T is (<>);
  with procedure P(X: in out T'Base);
package G is
  ...
end G:
```

appears to SAVE parameter-passing constraint checks for calls on P from within the body of G, since only the checks associated with the generic actual subprogram need be made. (No additional checks are necessary to determine that the parameter value satisfies the constraints associated with the actual subtype corresponding to T.) The writer of the RR points out, however, that if a check were made to ensure that the subtype of the parameter of the actual subprogram matching P were identical to the actual subtype matching T, constraint checks could be moved from the body of the actual subprogram to the sites of calls on that actual subprogram. The advantage of performing the checks in the calling context is that knowledge of the subtype of the actual parameters could allow the checks to be optimized away. In particular, it would be safe to call the corresponding generic formal procedure, P, from within G without any additional constraint checks.

There is a precedent for instantiation-time checks that the subtypes associated with a generic formal are identical to the corresponding subtypes associated with a generic actual. Such checks are performed for the index subtypes and component subtype of a generic formal array type and for the designated subtype of a generic formal access type.

- The next point addressed is "the fact that pass-by-copy semantics are required for generic formal private types when the corresponding actual type is scalar...." (What the writer of the RI means, of course, is that since the shared generic code must account for the scalar case, where pass by copy is required, pass by copy will apparently result even when the corresponding actual type is NOT scalar.) One possible response to this observation is that sharing is not appropriate for instances dealing with scalar types and instances dealing with large composite types; if we are interested in efficiency, we should settle for, say, one generic instantiation shared by all large composites and one shared by all types that fit in a word. Another response is to suggest that two implicit subprograms be passed with each generic formal type:

  — for setting up calls within the generic unit, a subroutine that places either a value of the generic formal type or the address of such a value on the stack, depending on the class of the actual subtype

  — for evaluating formal parameters of the generic formal type within subprogram bodies inside the generic unit, a "thunk" that expects to find either the value of the parameter or the address of the value at a given location

  (Come to think of it, these two responses are not at all incompatible.)

- The last issue raised is "the instantiation-dependent rules on the order in which generic actual parameters are evaluated." Again, I had no idea what the problem was until I turned to RR-586. That RR, written by an implementor, observes that 12.3(17) requires all explicit generic actual parameters to be evaluated before all default values for omitted generic actuals, so that values for generic actuals will be obtained in different orders on different calls. The implementor would prefer to evaluate the actuals in the order in which they will be stacked. (The example given in the RR is not compelling, because it involves generic formal objects of a fixed-size subtype, allowing the stack frames to be allocated in advance and filled in in whatever order the actual values are computed; however, this would not be possible for generic formal objects of type String.) I am not convinced that this is really a code-sharing issue: It is hard to imagine that the order of parameters on the stack would not be fixed even if generics were not to be shared. In any event, the resulting inefficiency (to set the stack up in a predictable order) is only encountered once per instantiation, so it is not a serious problem. The problem with allowing generic actual parameters and defaults for omitted parameters to be evaluated in an intermixed order is that generic parameter declarations are elaborated in the order in which they

appear, so the default value of one parameter may depend on the value obtained for an earlier parameter. (The rules for subprogram parameters are much more liberal. The only constraint on actual parameters--in 6.4.1(2)--and default expressions for omitted parameters--in 6.4.2(2)--is that they all be evaluated "before the call." But there is a crucial difference: As nice as it would be to write

```
function Substring
  (S          : Vstring;
   Starting_At : Positive;
   Ending_At   : Natural := S'Last)
  return Vstring;
```

the default expression for one subprogram parameter cannot refer to the value of another parameter of the same subprogram.)

%reference RR-0005 Exceptions are inconsistent, generic code-sharing accordingly hard

Shared code would be easier if exceptions could be created by elaboration.

%reference RR-0342 Don't implement requests which will break generic code sharing

The language must be such that compilers that implement generic code sharing can be reasonably built.

%reference RR-0445 Non-staticness of generic formals poses problems

The value of generic code-sharing is probably overrated.

%reference RR-0584 Need stricter checking of constraints on generic instantiations

This would make for more efficient code-sharing.

%reference RR-0585 Need pragma to specify code-gen. strategy for generic instantiation

This would be analogous to pragma INLINE.

%reference RR-0586 Tighten up rules for evaluation order of generic actuals

Evaluation order of generic actuals should be independent of the instantiation to make code-sharing easier.

%reference RR-0693 Param. pass distinction of scalars makes hard generic code sharing

Generic code-sharing would be easier to implement if one was allowed to pass a formal generic private type by reference regardless of how the unit was instantiated. Rules concerning the passing of scalar parameters make this impossible.

%reference AI-00409 Subtypes in an instance can be static

This is seen by RR-0342 as making generic code-sharing harder.

!rebuttal

!topic merged spec and body for generic subprograms
!number RI-1015
!version 1.5
!tracking 5.13.4

!Issue revision (1) desirable,small impl,upward compat,consistent

1. The Ada9X rules on the construction of generic subprograms with respect to allowing merged declaration and body shall be consistent with those for non-generic subprograms.

!reference RR-0547
!reference RR-0604
!reference RR-0760
!reference AI-00382

!problem

Ada83 allows a non-generic subprogram body to act as the declaration of the subprogram but the same kind of flexibility is not allowed for generic subprograms. This is inconvenient and inconsistent.

!rationale

There seems to be nothing gained by not allowing this flexibility for generic subprograms. Allowing the flexibility would eliminate a source of error, would make the language more consistent, and would be an upward compatible change to the language.

!appendix

It may make sense to aggregate this issue with other similar "irritants" in the language.

%reference RR-0547 Like non-gen. subpgms, allow merge of spec/body for generic subpgms

Do it.

%reference RR-0604 Like non-gen. subpgms, allow merge of spec/body for generic subpgms

Do it.

%reference RR-0760 Like non-gen. subpgms, allow merge of spec/body for generic ones

Do it.

%reference AI-00382 Like non-gen. subpgms, allow merge of spec/body for generic ones

Do it.

!rebuttal

!topic true separate compilation of spec and body for generics
!number RI-1016
!version 1.7
!tracking 5.13.5

DRAFT DRAFT DRAFT DRAFT DRAFT DRAFT

!Issue revision (1) important,severe impl,upward compat,consistent

1. Ada9X shall not allow an implementation to create a dependence on a generic unit body such that successful re-compilation of the body makes previously compiled units obsolete if they contain an instantiation of the generic unit.

!reference RR-0562
!reference WI-0217
!reference AI-00408

!problem

Ada83 allows an implementation to create a dependency on a separately compiled generic unit body from a unit containing an instantiation of the generic (AI-00408). This is inconvenient for users in that modifying a separately compiled generic unit body may necessitate recompiling all the units containing instantiations of the generic. Also, since implementations are allowed flexibility in this area, portability problems have been observed.

!rationale

While its true that some implementations take advantage of the freedom granted in AI-00408, reversing this AI would make generics much more friendly to the user and would make maintenance easier for programs that frequently employ generics.

It should be pointed out that there is an implementation approach where the body of the instantiation is made into an implicit subunit of the unit containing the instantiation. With this approach, reinstantiation can be performed after a generic body is recompiled, without recompiling the unit enclosing the instantiation. This seems to be a reasonable implementation technique for the above requirement, although it will admitted be difficult for certain implementations to transition to such generic instantiation strategy. On the whole, it appears now is the time to go ahead and fix the recompilation problem for generics.

!appendix

%reference RR-0562 Need true separate compilation of generic bodies & subunits

The recompilations that are required are painful and portability suffers.

%reference WI-0217 Need true separate compilation of generic bodies & subunits

Portability seems the major problem; files have to be re-organized etc. Also makes generics more difficult to use.

!rebuttal

The above "out-of-scope" should be a "requirement" (perhaps with the same ratings). This recompilation issue is one of the areas where generics are painful. It is not acceptable to have to recompile thousands of lines of code because I make a small change in one generic.

We cannot make current compilation technologies a reason for not fixing a language problem. It is very possible for compilers to produce nonoptimal code and fix it during optimization. It is also possible to generate all code at program build time.

Generics in Ada seem to "sit on the fence" and their net usability would be improved if they made a firm commitment one way or the other. That is, they either need to be like macros or they need to be truly separately compiled. The latter is to be preferred but the former is better than this fence-sitting.

5.14 Binding to Foreign Systems

RI-3514
RI-3657

!topic Interfaces to other languages.
!number RI-3514

!reference RR-0039
!reference RR-0345
!reference RR-0527

!problem

Ada83 makes it difficult to interface to software written in other languages and to use paradigms available in those other languages. For example, it is not possible to pass the name of a procedure to be iterated over in a numerical computation, thus limiting the ability of Ada solutions to numerous numerical problems requiring parameterization over a function. Furthermore, since the interface that is defined is implementation dependent, there is no guarantee that any solution for a particular implementation will allow interface to the equivalent set of routines on another implementation.

!topic Machine Code Insertions.
!number RI-3657

!reference RR-0043
!reference RR-0284
!reference RR-0371
!reference RR-0489
!reference RR-0490
!reference RR-0691
!reference RAT-15.7
!reference LRM-13.8
!reference

Fleck, Thomas J., "A Specification for Ada Machine Code Insertions", Ada Letters, VI(6), pp. 54-60, Nov.-Dec. 1986.

!problem

The Ada83 syntax for machine code insertions (LRM 13.8) is inconsistent with the approach taken by the remainder of the language. As a result, not only must a conceptual change of mind be made when using machine code insertions, but additional support code must be written in order to effect a return to the Ada style of programming.

The particular inconsistencies that require additional programming include:

1. Allowing machine code insertions only within procedures. This requires the creation of a dummy machine code insertion procedure when a machine code insertion is desired within a function.
2. Allowing only machine code insertions within such a procedure. This precludes the possibility of having an associated error handler, instead requiring an additional level of subprogram be created to simply handle any potential error situations.

Some additional limitations are presented in [Fleck]:

- Machine code insertion procedures are difficult to write, and to read, when compared with direct assembly language.
- The overhead involved with invoking a machine code procedure, in-line or otherwise, does not make it preferable to foreign assembly code.
- There is no mechanism to specify where a program variable is to reside prior to execution of a code statement.
- Parameter passing to machine code procedures is limited or absent.

The resulting situation is that it is often found more convenient to provide the functionality of machine code insertions through assembly language subprograms implemented through pragma interface.

5.15 Control Structures

RI-5250

!topic control structures
!number RI-5250
!version 1.2
!tracking 5.15

DRAFT DRAFT DRAFT DRAFT DRAFT DRAFT

!Issue revision (1) desirable,small impl,upward compat,consistent

1 – Generalize control structures] An attempt shall be made to remove unnecessary restrictions and inconsistencies in control structures in Ada9x.

!reference AI-00211 control of LOOP statements
!reference AI-00477 case choices should not have to be static
!reference RR-0312 generalize case statement to decision table
!reference RR-0317 better looping facilities
!reference RR-0320 generalize types of case guards, include REAL
!reference RR-0491 (related)
!reference RR-0538 loop structure without EXIT
!reference RR-0561 string processing
!reference RR-0615 (related)
!reference RR-0618 ban GOTO statement
!reference RR-0620 ban RETURN statement except inside functions
!reference RR-0642 label variables
!reference RR-0650 non-static case choices, non-discrete expressions
!reference RR-0717 step size in FOR loops
!reference RR-0743 step size in FOR loops
!reference RR-0744 non-discrete (fixed-point) FOR parameters

!problem

The revision requests represented here reflect a number of minor difficulties and/or inconsistencies in Ada's control structures. For example, case statement choices currently must be static, which is more restrictive than "constant and known at compilation time" (AI-00477). Another example is that loop statements, subprograms, and entries can all be completed from within nested statements – but blocks cannot (RR-0491).

!rationale

None of the identified difficulties or inconsistencies are impossible to work around or to live with. Most of the requested changes reflect fine tuning of a part of the language that is not really broken. However, if other language changes allow some or all of these difficulties to be removed, then the necessary changes should be included in Ada9x.

!appendix

%reference AI-00211

Add a "continue" statement to Ada's loop structure a la C.

%reference AI-00477

Case choices should not have to be static. At least remove the restriction on constants declared using explicit conversions.

%reference RR-0312

Complex logical conditions are often easily represented by decision tables. Ada should support them.

%reference RR-0317

Ada should provide facilities to loop over real numbers. Increment sizes other than 1 (and -1) should be supported.

%reference RR-0320

Ada should allow real numbers as case statement choices.

%reference RR-0491

To be consistent with loop statements, subprograms and entries, Ada should provide a mechanism to exit a block from within nested statements inside the block.

%reference RR-0538

Ada should provide a form of loop structure with an iteration scheme that does not allow the use of an EXIT statement.

%reference RR-0561

Ada should support string values as case statement choices.

%reference RR-0615

Since Ada supports FOR and WHILE loops, it should also support a LOOP/UNTIL construct with the test at the bottom of the loop.

%reference RR-0618

Ada should eliminate the GOTO construct.

%reference RR-0620

Ada should restrict use of the RETURN statement to returning values from functions. They should not be used to return from the middle of a procedure.

%reference RR-0642

Ada should support a restricted form of label variable to simplify the coding of state machines such as machine language emulators.

%reference RR-0650

It would be useful to allow expressions in case statement choices.

%reference RR-0717

Ada should support step sizes other than 1 (and -1) in FOR loops.

%reference RR-0743

Ada should support step sizes other than 1 (and -1) in FOR loops.

%reference RR-0744

Ada should allow real (or at least fixed-point) numbers in FOR loop iteration schemes.

!rebuttal

APPENDIX A: ADA 9X REVISION ISSUES - Indexed by RI Number

RI #	Page	Topic
1050	74	Complexity/surprises in overloading
1060	279	Reference to a task outside its master
1070	328	Exceptions
1081	106	Consistency of syntax
1082	107	Self-documenting syntax
1083	108	Clarity of syntax
1084	109	Convenience of syntax for programmer
2001	325	Controlling the service order of client tasks
2002	9	OOP (flexibility, maintainability)
2003	102	Character Set Issues
2011	168	Restrictive Pointers
2012	284	Initialization/Parameterization of Types
2021	21	Call-back style; Subprograms as Parameters and Objects
2022	64	Finalization
2023	218	Visibility of operators in a WITHed package
2024	221	Visibilty and Safety
2025	222	Fewer Restrictions on Overloading;
2027	226	Unexpected Consequences of Visibility Rules
2029	26	Support for control engineering programming
2032	124	Record discriminants
2033	130	Enhancing the Model of Record Types
2034	171	Typing at a Program Interface/Interoperability on Data
2035	177	Equality - redefinition and visibility
2101	268	Distributed systems
2102	16	User-defined assignment and equality
2105	192	Miscellaneous
2106	282	Library unit task termination
2107	320	Accept statements in subprograms
2108	322	Miscellaneous issues with CSP a la Ada
2109	376	Library unit and subunit naming
2110	380	Dependences and recompilations
2111	387	Subunits
2112	389	Context clause restrictions
2500	228	Visibility/Private/Hidden/Protected
3000	82	Information Hiding
3154	385	Program building
3279	386	Controlling the compilation process
3514	415	Interfaces to other languages
3572	384	Access to a standard set of Ada library operations
3600	211	The I/O Abstraction
3657	416	Machine code insertions
3700	205	I/O and External Environments
3749	189	Dimensional mathematics
3986	95	Portability
4017	368	Control of elaboration order is unsatisfactory
5010	158	Renaming declarations for types
5020	89	Subprogram body implementation

APPENDIX B: ADA 9X REVISION ISSUES - Indexed by Page Number

Page	RI #	Topic
9	2002	OOP (flexibility, maintainability)
16	2102	User-defined assignment and equality
21	2021	Call-back style; Subprograms as Parameters and Objects
26	2029	Support for control engineering programming
30	1030	Early messages from implementations for apparent problems
36	1031	Problems with erroneous execution & incorrect order dependence
39	1032	Strengthening subprogram specifications
43	1033	Fault tolerance
48	1034	Miscellaneous reliability issues
50	5040	Pre-elaboration
54	5080	Compile-time optimization
60	5100	Special case rules
64	2022	Finalization
67	5061	Constraints in renaming declarations
70	5062	Inconsistencies in renaming declarations
74	1050	Complexity/surprises in overloading
77	5110	Miscellaneous consistency and complexity issues
82	3000	Information Hiding
89	5020	Subprogram body implementation
95	3986	Portability
102	2003	Character Set Issues
106	1081	Consistency of syntax
107	1082	Self-documenting syntax
108	1083	Clarity of syntax
109	1084	Convenience of syntax for programmer
111	5120	Aggregates
113	5130	Array slices
115	5141	Variable-length strings
117	5142	Miscellaneous string issues
119	5150	Implicit subtype conversion
121	5160	Miscellaneous array issues
124	2032	Record discriminants
130	2033	Enhancing the Model of Record Types
134	5170	Uniformity of integers
136	0901	Fixed point range end points
139	0902	Implementation Freedom in Fixed Point
142	0903	Fixed-Point Support for Commercial Applications
145	0904	Restrictions in Fixed-Point
149	0905	Simplify Floating-Point Model
153	0906	Provide for Compatibility with IEEE Floating-Point
158	5010	Renaming declarations for types
161	5190	Multiple related derived types
163	5200	Type Declarations
166	0004	Syntax of relational expressions
167	0110	Non-contiguous subtypes of discrete types

APPENDIX C: REVISION ISSUES - January - March 1990

The Revision Issues (RIs) listed below are under consideration by the Distinguished Reviewers (DRs). An R indicates that the RI contained in this document is a revision which incorporates DR comments.

RI-0001
RI-0005-R
RI-0009
RI-1000
RI-1010-R
RI-1011-R
RI-1012-R
RI-1013-R
RI-1014-R
RI-1015-R
RI-1016-R
RI-1017-R
RI-1021-R
RI-1022-R
RI-1030
RI-1031
RI-1032
RI-1040 R
RI-2001-R
RI-2002
RI-2003-R
RI-2011-R
RI-2012-R
RI-2021-R
RI-2022-R
RI-2023-R
RI-2025 R
RI-2027
RI-2032 R
RI-2033 R
RI-2034-R
RI-2035 R
RI-2101
RI-3000-R
RI-3600
RI-3700
RI-3813
RI-3986
RI-4006
RI-4017-R
RI-5010
RI-5020
RI-5040-R

RI-5061
RI-5080 R
RI-5100 R
RI-5110 R
RI-5141
RI-5170 R
RI-5190
RI-5200
RI-5210 R
RI-7005
RI-7006 (withdrawn)
RI-7010
RI-7030
RI-7040

APPENDIX D: REVISION ISSUES - April-May 1990

The Revision Issues (RIs) listed below (37) are under consideration by the Requirements Team and the Distinguished Reviewers.

RI-0002
RI-0003
RI-0100
RI-0101
RI-0102
RI-0103
RI-0105
RI-0106
RI-0901
RI-0902
RI-0903
RI-0904
RI-0905
RI-0906
RI-1033
RI-1050
RI-1060
RI-1070
RI-2029
RI-2102
RI-2105
RI-2106
RI-2108
RI-2109
RI-2110
RI-2111
RI-2112
RI-3749
RI-5062
RI-5130
RI-5150
RI-5160
RI-5220
RI-5230
RI-5241
RI-5250
RI-7001

The 26 Revision Issues (RIs) listed below are incomplete in this document (problem statements only) because the full analysis has not been completed by the Requirements Team.

RI-0004
RI-0104
RI-0107
RI-0109
RI-0110
RI-0111
RI-0115
RI-0116
RI-0200
RI-0201
RI-1023
RI-1081
RI-1082
RI-1083
RI-1084
RI-2024
RI-3154
RI-3279
RI-3514
RI-3657
RI-3572
RI-5120
RI-5142
RI-7003
RI-7007
RI-7020