Ada 9X Project Repor

Ada 9X Revision Issues
Release 2

May 1990

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

PREFACE

This document contains the second release of the Ada 9X Revision Issues, based on the Ada 9X Requirements Team's analytical efforts from October 1989 to May 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 second release of RIs which incorporate Distinguished Reviewer (DR) comments and comments made by participants in the April Ada 9X Project Workshop, Soderfors, Sweden. 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:

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

The RI's presented in this document are under consideration by the Requirements Team and the Distinguished Reviewers as a possible basis for the formulation of Ada 9X requirements. However, any RI is subject to change or withdrawal as the requirements formulation process continues. This is a working draft document; its contents should not be construed as having an official status as Ada 9X requirements.

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 community at large. / KR)

There are three 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 a list of presentation AI's which are referenced in Section 3.1 of this report.

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 30 July.** Comments received after 30 July will be considered during the preparation of the Requirements Document.

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 21 May 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 a relatively low 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.

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

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 external standard: any standard that is developed or established outside the Ada 9X language revision effort. That is, no material specific to an external standard will be included in the Ada 9X standard.

[Note: The only responsibility that should be assumed by the Ada 9X Project Office is to ensure that Ada 9X will enable (even facilitate) the development and implementation of such standards.]

!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, an easy workaround exists, or a requirement more appropriate for Ada tools.

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

Implementation impact (on compilers)
- severe
- moderate
- small
- unknown

Compatibility impact (on existing code)
- very bad
- bad
- moderate
- upward compatible
- unknown

Consistent with the Ada model (design)
- inconsistent
- mostly consistent
- consistent
- unknown

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 wish 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. ISSUES CONCERNING THE STANDARD

This section presents the Revision Requests [RRs] that do not contain requirements for language revision. These RRs point to desired changes in the presentation of the LRM, suggest the need for some standards that are associated with Ada, and propose administrative management changes in some aspect of the Ada program. The three subsections contain a problem statement and the references that call for some action by the Mapping/Revision Team or by other organizations in the Ada community.

3.1 Presentation

Some members of the Ada community submitted Ada 9X revision requests [RRs] that call for editorial changes in the Ada Language Reference Manual [LRM] which do not affect the structure of the language. These RRs and the Ada Issue Commentaries, known as AI's, that concern editorial changes to improve readability and clarity of the LRM have been classified as "Presentation" issues.

!presentation

During the Ada 9X revision process, improvements in the readability and usability of the reference manual should be devised. The revision requests, revision issues, and AIs listed below are referenced only for completeness. No judgement is being passed on the suggestions provided in the referenced RRs.
!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 RI-0001
!reference RI-0009
!reference RI-0104
!reference RI-0106
!reference RI-0115
!reference RI-0901
!reference RI-1004
!reference RI-1031
!reference RI-1040
!reference RI-1050
!reference RI-1070
!reference RI-2003

!reference RI-2106
!reference RI-3657
!reference RI-7001

!reference
Presentation AIs - There are approximately 295 presentation AIs to be considered as part of this process. Appendix C provides the references for presentation AIs.

!problem

There are a number of areas where Ada users have found the Reference Manual to be deficient from their perspectiv, which is often incorrect or incomplete. The presentation issues cover suggestions for changes in presentation of the document and for specific textual changes for the purpose of clarifying the intent of the Reference Manual. The presentation AIs and the approved binding-interpretation AIs are now separate from the Reference Manual which necessitates the consultation of two reference reference documents.

!appendix

Note: any language revision requirements revealed by the referenced items are discussed and presented elsewhere.

Note: it is emphasized that the existence of the RR and AI references in this RI does not in any way imply that the specific suggestions in the referenced RRs and AIs are worthwhile or have any merit; no formal judgement is being passed on these suggestions during the requirements process.

%reference RR-0067

The technical terms in the LRM should be clarified and used more consistently.

%reference RR-0091

Section 10 should not describe the process of compilation, but rather the meaning of programs.

%reference RR-0204

The documentation of fixed-point predefined operators needs to be fixed up. In particular, Ada comments about the operators need to go in the text for package Standard.

%reference RR-0206

Paragraph numbers should be included in the RM cross reference.

%reference RR-0260

Ada RM is unclear, inconsistent, and incoherent in various ways.

%reference RR-0267

ARM does a bad job of distinguishing specifications and declarations.

%reference RR-0274

The language visibility rules are poorly presented in the LRM.

%reference RR-0277

Wording such as "all implementations" is inappropriate in the LRM.

%reference RR-0281

The delay statement is described poorly and not used in agreement with its semantics.

%reference RR-0292

Delete LRM 13.6 as it, strictly speaking, adds nothing to the language definition.

%reference RR-0298

The description of attribute prefixes in Appendix A is poor.

%reference RR-0300

Use an LR grammar to define the syntax of the language to ease the development of Ada compilers.

%reference RR-0301

There is bad wording concerning the definition of a "single compilation".

%reference RR-0305

The definition of the loop statement is technically not complete.

%reference RR-0309

The production of the new LRM should be automated to make help make it complete and consistent.

%reference RR-0326

Use a more expressive notation for defining the syntax of Ada, e.g., allow A ::= w <x|y> z.

%reference RR-0350

There is misleading LRM wording concerning types and default initial values.

%reference RR-0436

The concept of a synchronization point for a task (or for a variable) needs clarification.

%reference RR-0500

The LRM should be consistent in hyphenation of complex terms.

%reference RR-0501

The LRM should be consistent in delimiting section headings.

%reference RR-0502

The LRM should be consistent in use of upper and lower cases.

%reference RR-0622

LRM should use "metatype" in describing generic formal types.

%reference RR-0638

LRM should specify axioms for the built-in operations.

%reference RR-0683

LRM 11.6 is unclear about what replacements are allowed.

%reference RR-0732

Clarify semantics of instantiating Enumeration_IO with an integer type.

%reference RR-0757

Clean up definitions of program unit and compilation unit.

%reference RR-0758

There exists bad paragraph numbering in the LRM.

%reference RR-0769

Correct the wording in the definition of ancestor unit.

%reference WI-0106

Provide a formal specification of the context-sensitive language rules.

3.2 External Standards

Some RRs suggested that additional pre-defined library specifications or interfaces to environments external to Ada are needed. These user requests have been classified as "external standards". An "external" standard is any standard that is developed or established outside the Ada 9X language revision effort. That is, no material specific to an external standard will be included in the Ada 9X Standard. The proposed policy for the Ada 9X language revision process is that the only responsibility that should be assumed by the Ada 9X project is to ensure that Ada 9X will enable (even facilitate) the development and implementation of such standards.

!problem

Ada users do not have a set of standard Ada packages for mathematical processing but must develop their own libraries, at a higher cost than if standard libraries were available. There are also user needs to interface with software in an Ada environment which is not implemented in Ada. These interfaces, when they exist now, are not as reusable as if they were designed to a standard specification. The Ada community effort to obtain external standards is diffused into many ad hoc efforts.

!reference RR-0051
!reference RR-0068
!reference RR-0189
!reference RR-0296
!reference RR-0348
!reference RR-0719
!reference WI-0503
!reference RR-0074
!reference RR-0175
!reference RR-0181
!reference RR-0222
!reference RR-0355
!reference RR-0479
!reference RR-0480
!reference RR-0709
!reference WI-0505
!reference RR-0051
!reference RR-0068
!reference RR-0189
!reference RI-0906
!reference RI-2101
!reference RI-3572
!reference RI-3600
!reference RI-3657
!reference RI-3700
!reference RI-3813
!reference RI-5141
!reference RI-5220
!reference RI-5230
!reference RI-7007
!reference RI-7100

!appendix

%reference RR-0074 Define a standard run-time support environment interface

This RR would like to see interface from an Ada program to a standard run-time support environment. This would permit compilers to be used with different run time executives built for a target.

%reference RR-0175 Define interface between compiler- and target-specific RTS aspects

This RR would like to see interface from an Ada program to a standard run-time support environment. This would permit compilers to be used with different run time executives built for a target.

%reference RR-0181 Need standard means of communicating between Ada programs

This RR would like to have a secondary-standard specified to permit Ada programs to communicate with each other.

%reference RR-0222 Need additional predefined pkgs for process control/communication

This RR would like to have a standard method defined for communicating between multiple Ada programs, where each program forms a process on a multiprocess architecture.

%reference RR-0355 Standardize means of getting the OS command lines arguments

This RR would like a standard way of passing command-line arguments to an Ada program which has just been started by an operating system. It also requests a standard way of returning condition codes to the operating system upon termination of the program.

%reference RR-0479 Need standard subpgms to get user-interface info from OS

This RR would like a standard way of interfacing to an environment formed by an operating system. As well as passing command-line arguments to an Ada program which has just been started by an operating system, it would like standard ways of acquiring information about the user, terminal, execution mode, etc.

%reference RR-0480 Need std. means of communication (synchronization) between Ada pgms

This RR would like to have a secondary-standard specified to permit Ada programs to communicate with each other. Such programs need not be resident on the same processor or operating system.

%reference RR-0709 Need more portability in getting command line input

This RR would like a standard way of interfacing to an environment formed by an operating system, especially to pass command-line arguments to an Ada program which has just been started by an operating system.

%reference WI-0505 Interface Ada to existing stds (POSIX, X-WInd, etc.)

%reference RR-0051 Provide common math and other library package specs

Ada is suffering because no standard packages for numerics, string editing, etc are provided. Programmers are constantly reinventing such services.

%reference RR-0068 Need secondary standards for optional pre-defined lib units

Move TEXT_IO out from the Ada standard and make it a standard on its own

%reference RR-0189 Standard should include a floating-point math library interface

Numerical processing applications cannot be written in Ada, largely because it omitted a comprehensive set of mathematical functions

%reference RR-0296 Make pre-defined i/o packages optional if appropriate

Make Text_IO, Sequential_IO and Direct_IO packages optional for certain implementations. The LRM should explicitly state the optionality of these packages.

%reference RR-0348 Need predefined functions for real numbers, e.g., trig, log, etc.

Trig, log, etc. functions need to be provided by Ada.

%reference RR-0719 Need standardization with regards trig. functions, sqrt, etc.

The use of trigonometric functions, square root and others is so common that it should be standardized.

%reference WI-0503 Provide essential functions for mathematical processing

3.3 Administrative

There is another group of RRs that suggest initiatives which are not directly related to the Ada 9X language revision process but they reflect Ada community member views of desired change in administrative policy. These suggestions include, for example, making the LRM more usable by a blind person, rewarding compiler vendors for providing standardized libraries, and setting up a process for establishing and maintaining Secondary Standards [external standards].

!problem

Many members of the Ada community have ideas about changes to the administrative management of some aspect of the Ada program which they perceive as being part of the Ada 9X language revision effort.

!reference RR-0299
!reference RR-0314
!reference RR-0602
!reference RR-0681
!reference WI-0501
!reference WI-0502
!reference RR-0143
!reference RR-0176
!reference WI-0102
!reference RR-0630
!reference RR-0728
!reference RR-0318
!reference RR-0325
!reference RR-0435
!reference RR-0481
!reference RR-0528
!reference RR-0644
!reference RI-1030
!reference RI-3346
!reference RI-3700
!reference RI-3986

!appendix

RR-0299	Make everything in the ALRM "part of the standard"
RR-0314	Add to the standard definition of minimum-quality error diagnostics
RR-0602	Give merit badges to impls. for supporting standardized libraries
RR-0681	A definition of an Ada Line Of Code (LOC) should be standardized
WI-0501	Set up a process for establishing and maintaining secondary stds
WI-0502	Set up mechanism to facilitate interfacing Ada to other standards
RR-0143	Enforce implementation dependency documentation requirement
RR-0176	Require vendors to document run-time system behaviour
WI-0102	Require vendors to document all imp choices/ behaviour
RR-0630	Due to high implementation costs, define/allow Ada subsets
RR-0728	Need simple Ada subset for distributed memory MIMD architectures
RR-0318	Make available machine-readable LRM (with embedded mark-up)
RR-0325	Allow implementations to optionally take an Ada super(or sub)set
RR-0435	Need secondary standard for simple ada subset for safety appls
RR-0481	Make Ada documentation available in SGML format
RR-0528	Change Ada symbol names to recognized names for verbal communication
RR-0644	The LRM should implement guidelines for algorithms.

RI-1030

!administrative

[3 - Encourage Good Compiler Feedback] Ada 9X 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 by the implementation.

RI-1030

!administrative

[6 - Compilation Mode for No-Compile on Warning] Ada 9X 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.

RI-3346

!administrative

The language shall provide a mechanism to locally negate the effect of a compiler directive's applicability.

RI-3700

!administrative

[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.]

RI-3986

!administrative

[2 - Dependency Documentation] An implementation shall provide as much information on implementation dependent characteristics as necessary to support portability.

4. FUNCTIONALITY OF THE LANGUAGE

This section presents the Revision Issues (RIs) that deal with perceived user needs to improve the language in ways that increase its suitability for various programming styles, software engineering disciplines, and its general consistency and simplicity.

4.1 Generality (Suitability for Various Programming Styles)

The RIs in this section respond to concerns about the suitability of Ada for various programming styles. Of primary concern are object-oriented programming, the call-back style of programming and support for control engineering. The user input suggests language changes and extensions to enable the development of applications using these programming methodologies. Support for these programming styles can also involve binding to foreign systems which is the subject of the next section [4.2].

!topic Inheritance, polymorphism
!number RI-2002
!version 2.1

!introduction

This RI is intended to cover some of the evolutions that are considered desirable to improve the capacity of Ada 9X 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.

A language may be said to support object-oriented programming when its type systems supports encapsulation and inheritance. Key encapsulation capabilities include (1) the ability to ensure that an abstract object is initialized to a safe start state, (2) the ability to reclaim storage, and (3) the ability to hide various internal aspects of an implementation. These capabilities are not covered here–rather, they are covered in RI-2012, RI-2022, and RI-2500, respectively.

The basic idea of inheritance is that a type can be defined in terms of another type with the former type having the same fields and operations (see !terminology below) as the latter and some additional fields and operations as well. The code associated with the original type is "polymorphic" in the sense that it now works for many structures instead of just the original. A final important aspect of inheritance is dynamic binding–this aspect is covered under RI-2021 which calls for subprogram variables.

!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) compelling,moderate impl,upward compat,mostly consistent

1. (Inheritance/Polymorphism) Ada 9X 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. Ada 9X shall also provide an inheritance mechanism whereby new entity types can be defined based on existing ones, with additional properties, or with different implementations of existing properties. Ada 9X shall allow a form of reference where the possible referents are all objects of those entity types that may be obtained from a given entity type by zero or more applications of the inheritance mechanism.

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

2. (Implementation by Variance) Ada 9X 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 RI-2022 - Finalization
!reference RI-2500 - Improved Model of Privating
!reference RI-2021 - Call-back style; Subprograms as Parameters and Objects
!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 values 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.

!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 and polymorphism. 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 Ada 9X 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.

1. 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.
2. 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.
3. The requirements do not explicitly call for multiple inheritance, nor do they rule it out.
4. 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.

!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 Ada 9X 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 implementations 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 was some discussion at the Sweden workshop, that all of the notions of program composition–including Initialization (RI-2012), Finalization (RI-2022), Increased Visibility Control (RI-2023), Privating (RI-2500)–should be presented in a single RI. This suggestion was not followed for this version of the RI document.

!topic User-defined types vs predefined types
!number RI-2102
!version 2.1

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

1. (Overloading of Assignment) Ada 9X shall 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 Ada 9X by which a user-defined type may be bundled with an equality operation and appropriate assignment operations to yield an ADT so that clients would see it more as a nonlimited type. These ADTs should have as many characteristics as possible of predefined types, possibly including capability of having user-defined attributes and user control over copying and finalization. Other 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 RI-3333 User-defined attributes.
!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 (RI-2035).
3. lack of sufficiently powerful type initialization (RI-2012).
4. lack of type finalization (RI-2022).

User-defined assignment would reduce the need for user-defined subtype and object constraints. Also, predefined types are notationally more convenient since appropriate assignment, equality, and attribute operations may be defined.

!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 there is a great potential for interaction between this requirement and other parts of the language.

!sweden-workshop

Participants felt that the issue of good notation for ADTs was important but that overloading ":=" to achieve this was bound to be too complex.

!appendix

From 5.4.1.1

%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.

From 5.4.1.2

%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

%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.

From 5.7.2.2

%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

RI-2021

!topic Call-back style; subprograms as parameters and objects
!number RI-2021
!version 2.1

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

1. (Subprogram and Entry Types) Ada 9X shall provide types whose values are references to subprograms (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. Ada 9X shall restrict the available values of subprogram and entry types based on safety considerations.

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

2. (External Call-Back) Ada 9X 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 RI-3514
!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

!reference sweden-workshop

!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 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 problems 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. For this reason, the requirement is stated in terms of programming language facilities instead of problem domain. 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; this is particularly important for applications that are "prototyped" in a more dynamic language (say, Lisp or ML) and then translated into Ada. Also, the use of Ada with other programming languages will be significantly strengthened if external call-back were supported. A related issue, "Call-In", is discussed in RI-4017. RI-3514.1 is similar to RI-2021.2; the focus here is on the appropriate support of the call-back paradigm while the focus of RI-3514.1 is reuse of existing libraries that use call-back.

!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.

From section 4.1.2

%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

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

There shall be an Ada language mechanism for passing Ada subprograms and task entries to a non-Ada program.

From section 5.5.2

%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.

%reference sweden-workshop

An earlier version of this RI incorporated a requirement concerning Call-In in RI-2021.2. The recommendation was to eliminate this; this suggestion was followed.

Recommendation were also made (1) to separate entry types into a different RI and (2) to change from a more problem oriented requirement. The former was not followed because there seemed no point in doing so. The latter was not followed because it is felt that 30 years of programming language research have already indicated the appropriate solution to this problem.

!rebuttal

RI-2029

!topic Support for control engineering programming
!number RI-2029
!version 2.1

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

1. (Time-out on Operations) Ada 9X 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 is abandoned.

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

2. (Locking on Shared Variables) Ada 9X 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. Ada 9X shall provide a mechanism by which a task can ascertain if a variable is locked before attempting to acquire it.

!Issue out-of-scope (3) desirable,severe impl,upward compat,inconsistent

3. (Make Ada the Single Tool Needed for Real-Time Programming) Ada 9X 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-0002 Safe shutdown of tasks
!reference RI-2022 Finalization
!reference RI-7005 Priorities and Ada's Scheduling Paradigm
!reference RI-7007 Alternate Task Scheduling Paradigms
!reference RI-0003 Asynchronous Transfer of Control
!reference RI-5220 Provide construct for super-fast mutual exclusion

!problem

Many in the control programming communities preceive Ada83's facilities for real-time programming insufficient for all of 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.

!sweden-workshop

It was noted at the Sweden workshop that RI-0003 could be used as a solution to RI-2029.1. Another group recommended that both RI-2029.1 and RI-2029.2 be handled by external standards.

!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 Binding to Foreign Systems

Interfacing to foreign systems sometimes requires interfacing to a programming style which is not directly supported in Ada. These interfacing issues include the need to interface with software written in other programming languages and the portability and usability of machine code insertions.

!topic Interfaces to other languages.
!number RI-3514
!version 2.1

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

1. (Procedure Parameters) Ada 9X shall provide a mechanism to pass a procedure entry point to procedures written in a language other than Ada (specifically, C or Fortran) so that subsequent to the invocation of the foreign procedure, that procedure may invoke the Ada procedure whose entry was passed to it.

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

2. (Value-returning Procedures) Ada 9X shall provide a mechanism to interface with a value-returning procedure written in a language other than Ada (specifically, C or Fortran); such procedures not only return a value (as does an Ada function) but may also update their parameters (unlike an Ada function).

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

3. (Parameter Transmission for Foreign Subprograms) Ada 9X shall provide a standard method to specify the parameter transmission method for the parameters of subprograms written in other languages.

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

4. (Binding to an Externally Allocated Object) Ada 9X shall provide a standard method to specify that a declared objects corresponds with some object that is externally allocated.

!Issue revision (5) important,small impl,upward compat,consistent

5. (External Names for Objects) Ada 9X shall provide a standard method to specify an external link name for an Ada object, subprogram, or package. The external link name need not conform to the rules for Ada identifiers.

!Issue out-of-scope (6) desirable,moderate impl,moderate compat,inconsistent

6. (Pragma INTERFACE for Ada83) In order to ensure usefulness of Ada83 code with Ada 9X, Ada 9X shall provide a mechanism for calling subprograms written in Ada83 from an Ada 9X program.

[Rationale] This item assumes non-upward compatibility of the language revision. Although the intent may be to absolutely ensure that code written in Ada83 can be used in Ada 9X, the approach taken is entirely antithetical to the revision process.

!reference RR-0039

!reference RR-0345
!reference RR-0527
!reference RI-2021 Call-Back Style; Procedures as Parameters and Objects
!reference sweden-workshop

!reference

[Fowler 89] Fowler, Kenneth J., "A Study of Implementation-Dependent Pragmas and Attributes in Ada," Special Report, SEI-89-SR-19, November 1989.

!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.

In many organizations, the reuse of existing code is an important factor in the selection of software development plans. The introduction of Ada to these organizations is inhibited if there are large libraries of Fortran or C code. It should be possible to use these libraries with Ada without rewriting their contents. Furthermore, these organizations feel that their tested libraries of reusable code will be more reliable than newer Ada libraries, at least in the near future.

Ada83 offers pragma interface as a workaround, but the pragma is not standardized and needs additional support from the type system. For this reason, a secondary standard is needed.

Three features of Fortran and C libraries do not fit the Ada model. First, the data types are specified by name and their realization is implementation dependent. Second, procedure entries are commonly passed to another procedure to specialize its behavior. Third, functions are permitted to return a parameter and also modify their parameters (in-out mode function parameters). The required revisions to Ada are difficult because of these differences.

!rationale

[RI-3514.1] This is primarily a reuse issue--that is, there is a body of existing code that utilizes the notion that a procedure's behaviour be customized by a procedure parameter called at the appropriate time. This is not the issue of whether this "call-back" mechanism should be supported within an Ada program; within-Ada call-back is addressed in RI-2021.

[RI-3514.2] While Ada functions are enjoined from updating their parameters, value-returning procedures in other languages are not so enjoined. A standard way is needed to interface with existing libraries that contain such procedures. Fortran and C are specifically called out in the requirement because of the existing libraries of Fortran and C code. It will be important to consider other languages as well, particularly COBOL.

[RI-3514.3, RI-3514.4, and RI-3514.5] Many projects attempt to combine Ada components with components that are implemented in other languages. A common source of difficulty arises in (a) specifying parameter transmission modes for foreign functions, (b) establishing a correspondence between an Ada object and an externally allocated (and initialized) object, and (c) providing external (usually linker-compatible) names for Ada entities.

Current implementation address these issues by providing implementation-specific pragmas. This is a common cause of portability problems. [Fowler 89] specifically recommends that these problems be addressed by extending pragma INTERFACE to provide this information; this recommendation is felt to be too solution-specific for the requirements phase.

!appendix

%reference RR-0039 Make it easier to access FORTRAN libraries

This RR wants to have procedure arguments in order to be able to take advantage of paradigms used in FORTRAN for numerical computations.

%reference RR-0345 Need standardized interface to other ANSI languages

This RR wants to be able to have a consistent interface to other ANSI standized languages rather than have it done on a per-implementation basis in order to improve portability among ANSI conforming implementations.

%reference RR-0527 Extend defn. of pragma interface for portability, access to Ada83

Wants to treat Ada83 as a language that can be used in a pragma interface statement.

%reference sweden-workshop

There was a feeling in Sweden that one of the problems in interfacing with existing code written in other languages is that this code may contain functions that modify parameters. This is a problem because function parameters in Ada must be of mode IN.

!rebuttal

!topic Machine Code Insertions (MCIs)
!number RI-3657
!version 2.1

!Issue out-of-scope (1) desirable,severe impl,upward compat,inconsistent

1. (Consistent Syntax) Ada 9X shall support a syntax for machine code insertions that is more like assembly language than the current specification of record aggregates.

[Rationale] The original Rationale stated that the style for machine code insertions was designed in such a manner as to inhibit overuse of the feature.

!external-standard (2) desirable,unknown impl,upward compat,mostly consistent

2. (Portable Syntax) The standard shall specify a procedural interface to underlying machine code instructions in which access to Ada variables shall be allowed via parameter reference.

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

3. (integrating MCIs with Ada model) Ada 9X shall permit MCIs that are functions as well as procedures. Ada 9X shall standardize the notation used to return a value from a function or raise an exception from within a MCI.

!Issue presentation (4) desirable,small impl,upward compat,consistent

4. Ada 9X shall stress that machine code insertions are not synonymous with assembler code insertions. Assembly languages belong to LRM 13.9, "Interface to Other Languages."

!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 within such a procedure precludes the possibility of having an associated error handler. This requires an additional level of subprogram be created to simply handle any potential error situations.

Some additional limitations are presented in [Fleck]:

1. Machine code insertion procedures are difficult to write, and to read, when compared with direct assembly language.
2. The overhead involved with invoking a machine code procedure, in-line or otherwise, does not make it preferable to foreign assembly code.
3. There is no mechanism to specify where a program variable is to reside prior to execution of a code statement.
4. 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.

!rationale

[2] The specification of machine code inserts can be improved by specifying the instructions in the form of procedures in which access to Ada variables are allowed via parameter references. Some vendors already provide access to Ada variables via implementation dependent attributes.

[3] Many users feel the necessity for machine code inserts that are functions as well a procedures, and the necessity to raise exceptions from within machine code inserts. To promote portability, [3] requires a standard notation to be used to return a value from a function or raise an exception.

[4] Users want to write conventional assembly code inside of machine code inserts. If Ada 9X is not going to define the assembly code then the language standard should emphasize that machine code inserts are not intended to replace pragma INTERFACE and assembly code, and users that want the convenience of assembly language should use assembly language.

!appendix

The primary overhead to use the current approach is the cost associated with preparing a wrapper procedure around the machine code insertion procedure in order to pass parameters. The current approach has the additional benefit that machine code insertions promote portability, by forcing the abstraction of the machine code statements into a procedure, rather than allowing them to occur inline.

The remaining issue is how to gracefully cause an Ada exception to be raised within a machine code insertion. The RT is somewhat divided on this issue - some feel that attempting to raise an Ada error within a machine code insertion is a mixing of the metaphors (should only use Ada concepts within Ada and machine code concepts within machine code), while others contend that there must be a way for the machine code insertion to propagate a return that indicates an error has occurred. The workaround possibilities are to return a value that indicates successful (or not) completion of the machine code insertion, or effecting the exception condition through the same mechanism as the underlying runtime does.

%reference Fleck - A specification for Machine Code Insertions

%reference rationale 15.7. Interface with Other Languages

A limited and implementation-dependent facility for machine code insertions has been included in Ada. This facility has the advantage of clearly isolating the use of machine language. Furthermore, its use is heavier than direct use of an assembler: the facility offered should be sufficient for cases where there is an actual need, but its style will inhibit overuse.

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

Really doesn't want to have assembler language code treated any differently than other "foreign" code.

%reference RR-0284 Machine-code insertions are unreadable; replace with INLINE macros

Need is expressed as the ability to have an INLINE assembler macro expanded. I think the complaint is that the current syntax for specifying a machine code insertion is inconsistent with current practice and with the perceived Ada style.

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

Contains pointer to Ada Letters, Volume VI, Number 6, pp 54-60.

%reference RR-0489 Allow machine-code insertions in functions as well as procedures

Expand the code insertions to subprogram bodies, rather than just procedures.

%reference RR-0490 Need successful/convenient recovery from exceptions in MC

The inability to have an exception handler in a machine code insertion procedure prohibits the use of normal Ada style for exception handling and requires the introduction of additional support code.

%reference RR-0691 Allow machine-code insertions in functions as well as procedures

Same issue as raised by RR-0489.

!rebuttal

4.3 Reliability

The RIs in this section respond to RRs that take the position that Ada should be changed to better facilitate the construction of reliable programs. Primary concerns are the early-detection of programming errors, consistent behavior across implementations, formal analysis of Ada programs, and fault tolerance.

RI-1030

!topic Early messages from implementations for apparent problems
!number RI-1030
!version 2.1

!Issue revision (1) desirable,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 Ada 9X.

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

2. (Tightened Legality Rules) To any extent possible, without overly complicating the language, 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, Ada 9X shall consider 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 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) desirable

3. (Encourage Good Compiler Feedback) Ada 9X 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 the implementation.

!Issue out-of-scope (problematic) (4) desirable,small impl,moderate compat,inconsistent

4. (Implementation-Defined Legality) An Ada 9X 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 (5) desirable,moderate impl,upward compat,inconsistent

5. (Define Warning Cases in Language) Ada 9X 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) desirable

6. (Compilation Mode for No-Compile on Warning) Ada 9X 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.

!sweden-workshop

One working group in Sweden felt the LRM should mandate warnings for correct but unperformed pragmas. This idea is captured in the first rebuttal, below, since it is felt that this problem is adequately addressed by [3] and [6].

Item [2] has been weakened and a caution added concerning language complexity.

!appendix

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

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

The reference manual should require implementations to give warnings for unperformed pragmas!

It is simply counter-productive to allow implementations to silently ignore a pragma!

!rebuttal

I have a major problem with the classification, rationale, and wording of requirement #4. (Rejection of erroneous programs, amongst others).

While I do agree with the OOS classification for the given wording, because it is close to a carte-blanche for implementers to do as they please, I see an important subset of situations, where I would want implementation-defined legality. Yet, the OOS of this requirement and especially the rationale argue against that. (and create a contradiction with requirement #2).

Part of the rationale for introducing the term "erroneous" in Ada-83 was to convey that the program "really won't work", but that compilers could not be expected to check for the erroneous situation in all its generality. For example: use of uninitialized variables. Such situations cannot possibly be turned into a language-defined legality check. Yet, users would be more than happy to be informed in no uncertain terms that there is a bug in the program. A language semantics that forces compilers into compiling the program successfully regardless of such bugs is an utter disservice to the user community. In that sense, a subset of erroneous situations should be allowed to cause rejection of the program. There is no way that the language definition can delineate the boundary of detectable erroneousness in an acceptable fashion. Hence, there would remain a (very useful) area for implementation-defined illegality.

To pick up on #2 in this context:

```
    x mod 0 – to be considered illegal

    zero: constant := 0;
    ...x mod zero... – to be considered illegal ?

    zero: integer := 0;
    ... if ... case... x mod zero ... – to be considered illegal ????
```

Where is the borderline between illegality and erroneousness here?

RI-1031

!topic Erroneous execution and incorrect order dependence (IOD)
!number RI-1031
!version 2.1

!Issue presentation (1) important

1. ("Effect" of Execution) Ada 9X shall either (1) clarify what is meant by the "effect" of the execution of a program or (2) not use this term in defining the semantics of the language (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 or require a predefined exception to be raised (without imposing, however, undue compile-time or run-time performance penalities);
2. bound the potential meanings of the execution (rather than defining the effect to be "unpredictable" for erroneous execution);
3. do not make the condition an error and provide 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-0236
!reference RR-0329
!reference WI-0101
!reference AI-00585
!reference AI-00832
!reference RI-1022
!reference RI-5080

!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 either be defined in the LRM or alternate phrasing should be used.

[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.

!sweden-workshop

Concerning [1], the safety-critical/trusted system working group felt clarifying "effect" was "futile" and that removing its use was preferred. There seems no harm, however, in not over-constraining the solution to this problem.

Item [2] has been revised considerably based on the results of the workshop. Although the group felt detecting conditions such as these at compile-time or run-time was not (in general) practical, this idea still appears here for the sake of completeness. However, a note of caution on performance concerns based on remarks at the workshop was added. The idea of not allowing an implementation-dependent choice to vary dynamically at run-time was dropped as too constraining on implementations. A working group felt 3) above is really an RI-5080 issue but it has been included here also for completeness.

!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-0236 Need better support for formal analysis of ada programs

There are two main sources of problem: (i) where the informal (LRM) definition is long and difficult, but where the meaning of the program is sorted out by a compiler front end. Here we are thinking of problems of name visibility and overloading (LRM sect.8). (ii) where the semantic behavior is deliberately left undefined. Here we are thinking of the profusion of erroneous or order- dependent constructs in Ada, and the differing implementations allowed by LRM sect. 1.6.[3,6]

%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-00585 [BI,RE] discriminant change after prefix evaluation

Describes an unusual situation that probably should have been erroneous in Ada83.

%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

RI-1032

!topic Strengthening subprogram specifications
!number RI-1032
!version 2.1

!terminology

[Pure Subprograms] A subprogram in an Ada 9X 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 of only 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 out-of-scope (1) desirable,upward compat,mostly consistent

1. (Clearly Pure Subprograms) Ada 9X 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 impurity 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 an object whose implicit initialization requires the execution of a clearly pure subprogram.

[Rationale] See [Rationale] under [2].

!Issue out-of-scope (2) desirable,moderate impl,upward compat,mostly consistent

2. (Assertions for Clearly Pure Subprograms) Ada 9X 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.

[Rationale] It is easy to be sympathetic with the goals of this requirement, but on the whole it seems to be better to achieve these goals outside of Ada, using an assertion language, than to complicate the programming language further. It appears that the restrictions necessary to make assertions enforceable by the compiler would be either extremely complex or too confining for the programmer.

The two potential benefits of strengthened specifications are stronger assurances about safe behavior and alerting the compiler to opportunities for optimizations. An annotation language seems to be a better vehicle for providing assurances about safe behavior because it can be more expressive than a programming language and can depend on more powerful checking mechanisms (e.g. formal proof) than we would ever demand from a compiler. It is not clear that the benefits for optimization would arise frequently enough, or be helpful enough, to justify the added complexity required for the compiler to enforce the assertions. Indeed, given the finite resources of implementers, the effort applied to enforcing the assertions could well reduce the effort applied to performing optimizations.

The notion of "purity" expressed above is overly simplistic. It is not clear whether the "states" of the program are affected by the use of allocators or by I/O operations, for example. Neither is it clear how the potential for raising exceptions affects the "purity" of a subprogram.

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

3. (Assertions on Exceptions) Ada 9X 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.

[Rationale] This is an item with relatively low user need that would be a significant complication to the language. A separate assertion language is a more appropriate way to address this problem.

!Issue out-of-scope (4) desirable,moderate impl,upward compat,mostly consistent

4. (Additional Constraints on Parameters) Ada 9X 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).

[Rationale] This is an item with relatively low user need that would be a significant complication to the language. A separate assertion language is a more appropriate way to address this problem.

!reference RR-0030
!reference RR-0517
!reference RR-0518
!reference WI-0108
!reference WI-0108M
!reference RI-5080
!reference Trusted System and Safety Critical Ada 9X Workshop; IDA Beauregard Facility, Alexandria, 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.

!sweden-workshop

The items in this RI have been changed from requirements to out-of-scopes based on the discussion at the workshop.

One working group labeled [3] "important", but suggested an extra-lingual (or perhaps !Issue implementation) solution.

!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.

%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

RI-1033

!topic Fault tolerance
!number RI-1033
!version 2.1

!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 implementation (1) compelling,severe impl,upward compat,mostly consistent

1. (Failure Semantics) The Ada09X distributed-system implementation standards shall define the effect of failures of remote operations on the executions of Ada09X programs.

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

2. (Language Support for Recovery) Ada09X shall attempt to provide additional language mechanisms to facilitate recovery and reconfiguration in fault-tolerant systems.

!reference RR-0111
!reference WI-0406
!reference WI-0407
!reference WI-0407M
!reference WI-0408
!reference WI-0408M
!reference RI-2101
!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

Language improvements to facilitate the development of fault-tolerant systems 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 portable.

Second, the language does not fully 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. In essence, 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]

A necessary 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.

Currently, implementations have relatively little incentive to detect failures in remote operations, and, in particular, to minimize the situations where a thread of control in an Ada program can be "left hanging" indefinitely due to a failure of a remote operation. For the purpose of portability, it seems important for detected failures of remote operations to manifest themselves in the execution of a program in standard ways.

It should be understood that defining the semantics of failures in the underlying system is not unprecedented in Ada. 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. An acceptable solution to this problem might begin with a description of various remote operations in the distributed-system implementation standards. These standards could then describe how failures of these remote operations should manifest themselves in the execution of a program (e.g., perhaps the raising of a particular pre-defined exception). These implementation standards may also detail the circumstances under which implementations are required to check for the various failures. Finally, this standard might include specification of "time-out" information and other implementation requirements on the detection of these failures.

[RI-1033.2]

The provision of direct support for distributed environments was a very important issue at the Destin Requirements Workshop. In order to support the development of the distributed, fault-tolerant embedded systems of the 1990s and 2000s, Ada09X should provide mechanisms to let developers recover and reconfigure fault-tolerant systems from within the language.

Constraints on these mechanisms consist primarily of the need to mesh nicely with the language, be consistent with the existing and proven software technology in this area, and finally, not add major complexity to the language.

Mechanisms that are candidates for consideration include:

1. improved control over the elaboration (and finalization) of library units to re-initialize aspects of a system;
2. implementation-defined "task characteristics" for interaction with an underlying recovery system;
3. facilities for dynamic re-allocation of program segments;
4. accessible unique identification of the threads of control in the program to be used for re-configuration and communication with run-time support software, and;
5. accessible information on exceptions, including the exception name and identification of code where an exception was raised.
6. a facility for asynchronous notification of tasks that specific failures have occurred and a new mode of operation is required;

!sweden-workshop

Some attendees thought there might be areas besides remote operations for which failure semantics were missing, but no one was able to name any.

The safety-critical/trusted system group in Sweden seemed extremely concerned about language changes that would add complexity to the language. These concerns have been reflected in the rationale discussion above.

!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

RI-1034

!topic Miscellaneous reliability issues
!number RI-1034
!version 2.1

!reference RR-0478
!reference RR-0763

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

1. (Safe Suppression of Checks) An attempt shall be made to make it safer for the user to suppress checks in Ada 9X.

!Issue out-of-scope (2) desirable,severe impl,upward compat,mostly consistent

2. (Protection in Hostile Environments) Ada 9X shall provide the programmer with mechanisms to promote safety in hostile environments. In particular, Ada 9X shall provide features to help protect against viruses and trojan horses. [Refer to RR-0478 for more information.]

[Rationale] The problems addressed here are not applicable to the vast majority of programs. Furthermore, it is not clear that solutions for these problems are feasible with existing technology. Given the appropriate technology, there is nothing in Ada83 that precludes a solution based on annotations, ordinary packages, and linker directives.

!problem

There are two reliability problems covered herein. They are:

1. The fact that the Ada83 pragma SUPPRESS can be dangerous to use because this may suppress checks in far-removed inner code regions that rely on checks for correct functioning, and;
2. Lack of features in Ada83 to protect against viruses and trojan horses and otherwise promote safety in hostile environments. A main aspect of this problem is the need to achieve a form of multilevel security in trusted systems.

!rationale

[RI-1034.1] Other than a way of marking a code segment as relying on checks for its correct functioning (which would prevent its checks from ever being disabled), it is far from clear what can be done to correct this problem. Such marking of code segments seems somewhat contrary to the Ada philosophy which is that code normally relies on constraint checks for proper functioning. Nevertheless, this problem appears to be genuine and should be looked into further.

!sweden-workshop

The working groups at the workshop reviewed an earlier version of this RI that included a wider-variety of reliability problems. Most of these, as suggested, have been integrated into other RIs. The treatment of the remaining issues here is consistent with the feelings expressed at the workshop.

!appendix

%reference RR-0478 Add language facilities for more safety in hostile environments

The problem here seems to be that Ada provides no explicit features to obtain the degree of multilevel security that is required in certain environments. The author cites the need for protection against viruses and trojan horses and as a possible solution gives an authentication protocol that he feels would help address this problem.

%reference RR-0763

The problem here is that an inner segment of code can rely on constraint checks, etc., for correct functioning but these checks can be mistakenly removed by a far-removed pragma SUPPRESS or a compiler switch.

!rebuttal

4.4 Optimization

There are concerns in parts of the Ada community that the language is not meeting necessary safety and performance criteria. This section deals with RRs that address the needs for predictable execution and optimization.

RI-5080

!topic Compile-time optimization
!number RI-5080
!version 2.1

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

1. (Predictable execution with optimization) Optimization of Ada 9X code, unless otherwise explicitly indicated, shall be limited to semantically neutral program transformations. (That is, unless explicitly indicated, all optimizations shall yield executions that are valid interpretations of the original source code.)

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

2. (Statement order optimizations) Ada 9X shall provide a mechanism to indicate that statements may be reordered to improve performance. If no exception is raised, statement reorderings shall be semantically neutral. Statements affected by reorderings shall be bounded by an appropriate syntactic "fire wall." Statement order optimizations should allow utilization of available vector and parallel machine operations. [Note: the language will not be able to specify which of any affected variables will have been assigned new values when an exception is raised in statement-order-optimized code. Hence, any program that depends on the values of such variables when an exception is raised will have to be considered erroneous.]

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

3. (Deterministic order of evaluation) Ada 9X shall define deterministic (canonical) orders for expression and subprogram parameter evaluation, select-alternative guard evaluation, subprogram result copy-back, and statement execution. The phrase "in an order not defined by the language" shall not appear in the Ada 9X reference manual.

[Rationale] While defining a deterministic order for these constructs would make program execution more predictable, it would also encourage program dependence on this order. This would lead to more obscure, brittle, and side-effect dependent programs. Hence, the net effect would not be a benefit for program safety or reliability.

!Issue out-of-scope (4) desirable,unknown impl,upward compat,mostly consistent

4. (Literal translation mode) Ada 9X compilers shall provide a mechanism (or mechanisms) to explicitly turn off all optimizations.

[Rationale] While most compilers have a "debug" mode that produces unoptimized code, it is impossible to draw a line between where good code generation ends and "optimization" begins. The need for ultra-reliable translations within the trusted and safety-critical systems community may be better served by other mechanisms such as a verifiable intermediate program representation. This, of course, does not preclude or prohibit implementations from providing an unoptimized compilation mode.

!Issue revision (5) desirable,moderate impl,upward compat,consistent

5. (Lack of side effects) Ada 9X 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. [Note: this information is intended for use in optimization, not for formal proofs of program correctness.]

!Issue out-of-scope (problematic) (6) desirable,small impl,bad compat,inconsistent

6. (Short-cut evaluation) Change the semantics of Boolean expressions 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 RI-1031 problems with erroneousness and incorrect order dependence
!reference RI-1032 strengthening subprogram specifications
!reference RI-1034 miscellaneous reliability issues
!reference RR-0265 allow implementations to short-circuit in general
!reference RR-0386 need standard way of telling the compiler not to optimize
!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-0729 language should provide way to turn off optimization
!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
!reference Ada 9X Requirements Workshop, Sweden, April 1990

!problem

Code optimization is essential to many Ada applications. Ada83's rules for optimization [LRM 11.6] allow transformations that can change the meaning of programs. At the same time, they allow too little freedom for reordering instructions to improve performance. In addition, Ada83's optimization rules limit the use of vector and parallel machine operations where they are available.

!rationale

Ada 9X must allow for more extensive optimization than is provided for by Ada83 [LRM 10.6 and 11.6]. At the same time, Ada 9X must provide higher safety and more predictable program behavior than is provided by Ada83. Issue [1] is an attempt to allow flexibility for optimization, while also guaranteeing predictable program behavior. Issue [2] addresses further optimizations that are semantically neutral in the absence exceptions. It also covers the problem of generating efficient code for vector and parallel hardware with Ada's current rules. Issue [5] is an attempt to give compilers program "design knowledge" or "intent" that enables optimizations but is impossible to deduce automatically.

!appendix

An example of a problem that might be raised by issue [4] is: Can I keep values of variables in registers across statements or must I reload them? Is this an optimization or a reasonable code generation strategy? .

%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-0386

Optimizers are the buggiest parts of compilers. It is important that there be a way of explicitly instructing the compiler not to try to optimize the instruction sequences that it generates.

%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-0729

Optimizers are the buggiest parts of compilers. It is important that there be a way of explicitly instructing the compiler not to try to optimize the instruction sequences that it generates.

%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.*

%reference Ada 9X Requirements Workshop, Sweden, April 1990

There should be a way to prevent the semantically observable optimizations allowed in Ada83 (LRM 11.6). There should be a "superparanoid" mode in which even semantically neutral optimizations are suppressed. The syntactic "fire wall" for statement-reordering optimizations should be the block.

!rebuttal

4.5 Simplicity/Consistency

The RIs in this section address a theme expressed in various ways, which is, that Ada in some ways lacks consistency or uniformity; or, that it contains arbitrary or unexpected restrictions. There is also a user perception that the language is unbalanced or asymmetrical in that it deals with initialization and elaboration but not in finalization or de-elaboration. Several RRs also suggest the Ada RENAME facility is inconsistent and arbitrary and should be made more regular. Other RRs point out that attributes are not defined types in a consistent, natural, and predictable fashion.

RI-5100

!topic Special case rules
!number RI-5100
!version 2.1

!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 Ada 9X.

!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-00382 allow merge of spec and body for generic subpgms
!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
!rcference RR-0427 complexity
!reference RR-0547 allow merge of spec and body for generic subpgms
!reference RR-0604 allow merge of spec and body for generic subpgms
!reference RR-0612 Should allow both delay and terminate alternatives
!reference RR-0664 Need 'IMAGE and 'VALUE for floating-point
!reference RR-0760 allow merge of spec and body for generic subpgms

!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:

1. Special handling of renamed constructs in deciding whether a component selection is ambiguous. (AI-00119)
2. Disallowing mixed values of universal and named integer types for array bounds. (AI-00161)
3. 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)
4. Restricting the location of pragma OPTIMIZE to a declarative part. (It should be allowed to appear anywhere a basic declaration may appear.) (AI-00280)

5. Disallowing the combination of a terminate and a delay alternative in the same select statement. (RR-0612)
6. Not providing attributes 'image and 'value for fixed point and floating point types.
7. Not allowing generics and packages to be defined by just a body (rather than distinct specifications and bodies).
8. Making the execution of a program 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)
9. Not allowing merged declarations and bodies for generic subprograms.

!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 RR-0547 allow merge of spec and body for generic subpgms **%reference** RR-0604 allow merge of spec and body for generic subpgms **%reference** RR-0760 allow merge of spec and body for generic subpgms **%reference** AI-00382 allow merge of spec and body for generic subpgms

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.

%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

RI-2022

!topic Finalization
!number RI-2022
!version 2.1

!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 Ada 9X, 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 Ada 9X. [Note: the order could be defined to be implementation-dependent.]

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

2. (Reclaiming by Reference Counting) Ada 9X shall provide facilities so that a programmer implementing an abstract data type in terms of dynamic data structures can use reference counting to determine when memory used for a given abstract object may be reclaimed. At a minimum, the programmer must be able to gain control whenever a value is copied and whenever a scope is deactivated.

!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
!reference RI-0107
!reference Sweden Workshop

!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. [See RI-0107, Memory Reclamation] 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 exacerbated 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.

Indeed, the problem of reclaiming storage is so severe that it is broken out in a separate requirement. The requirement is stated in terms of abstract objects; but, as was noted at the Sweden workshop, there can also be a problem with reclaiming storage associated with abstract values if they are returned from functions.

!appendix

From 4.4.2.1

%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

From 4.4.2.2

%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

RI-5061

!topic Constraints in renaming declarations
!number RI-5061
!version 2.1

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

1. (Subtype constraints on renamed objects) Ada 9X 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) Ada 9X 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 (3) desirable,moderate impl,upward compat,mostly consistent

3. (Constraint checking for results of renamed subprograms) Ada 9X 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. Renaming 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)). For example,

```
procedure NO_PROBLEM is
  X: INTEGER;
  Y: POSITIVE renames X;
begin
  Y := -5;  -- no problem, no exception, Y is not POSITIVE
end;
```

2. A slight variation occurs when access values are involved (LRM 4.8(3)); e.g.,

```
procedure NO_PROBLEM_2 is
  type T is access INTEGER;
  X: T;
begin
```

```
    X := new POSITIVE;
    X.all := -5;  -- no problem, no exception, not POSITIVE
  end;
```

3. Renaming declarations for subprograms similarly ignore subtype constraints for parameters and function results (LRM 8.5(8)). For example,

```
procedure MAIN_PROGRAM is

  X: INTEGER;

  procedure ORIGINAL ( X: out INTEGER );

  procedure RENAMING ( X: out POSITIVE ) renames ORIGINAL ;

  procedure ORIGINAL ( X: out INTEGER ) is
  begin
    X := -5;
  end;

begin
  ORIGINAL(X);
  RENAMING(X);  -- no problem, no type mismatch, no exception
end;
```

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.

Requirements [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 Ada 9X should probably live with.

!topic Renaming declarations
!number RI-5062
!version 2.1

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

1. (Renamed construct as prefix) Ada 9X 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 and the rule in the LRM is not as clear as it might be. But, the restriction 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.

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

2. (Renaming enumeration literals) Ada 9X 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 Ada 9X 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:

1. Renaming enumeration literals as character literals (in RR-0096)
2. 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

RI-1050

!topic Complexity/surprises in overloading
!number RI-1050
!version 2.1

!Issue presentation

[1 - Clarify Definition] Ada 9X shall clarify the language rules regarding overload resolution and implicit type conversions.

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

2. (Attempt Simpler Rules) An attempt shall be to simplify and make more natural the Ada 9X rules concerning overload resolution and implicit type conversions.

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

3. (Fix Specific Inconsistency) Ada 9X shall allow discrete ranges such as "-1..10" in an index constraint, iteration scheme, or entry declaration.

!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.

[RI-1050.3]

From the point of view of the run-of-the-mill Ada programmer, it is baffling that the loop iteration scheme

```
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, it seems quite unusual to users that this problem exists. 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 implementers 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

There is really no point behind item RI-1050.2. The rules concerning overloading and implicit conversions are basically sound and not in need of repair. They were carefully thought out and intensely scrutinized during the Ada 83 design process, and, despite the unsupported claim above that "some users view the rules as unnatural and counterintuitive," in practice they almost always produce the expected result.

RI-5110

!topic Miscellaneous consistency and complexity issues
!number RI-5110
!version 2.1

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

1. (Implicit conversion in overloaded operations) Ada 9X shall 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,moderate compat,mostly consistent

2. (Consistent semantic rules) Ada 9X 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 revision (3) desirable,moderate impl,moderate compat,consistent

3. (Default constant values) Ada 9X shall provide for defining constants with default values obtained from their type definitions.

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

4. (I/O operations for characters) Ada 9X shall 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 to 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 (5) desirable,small impl,upward compat,consistent

5. ('RANGE for Scalar Types) Ada 9X 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-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

!reference Ada 9X Requirements Workshop, Soderfors, Sweden, April 1990

!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. Consistent semantic rules for similar syntactic structures that appear in different parts of the language
3. 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.

Issue [3] was marked as out-of-scope in the previous version of this RI with the following rationale. It was moved back in scope based on comments from the Sweden workshop.

[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.

%reference AI-00136

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

%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.

%reference Ada 9X Requirements Workshop, Soderfors, Sweden, April 1990

Overall, miscellaneous consistency and complexity problems are important to correct. The issue on defining constants with default values should be in scope.

!rebuttal

4.6 Support for Information Hiding

The RIs in this section address RRs that point to problems encountered in the use of Ada 83 mechanisms for the separation of the specification from implementation. Certain features of the Ada specification/body rules make it difficult for developers to provide clean specifications for others while keeping implementation details to themselves. Some of these RRs suggest that full declarations for private types and deferred constants should be allowed/required in the package body instead of in the package specification. Other RRs deal with perceived problems specific to the "private" concept in Ada.

RI-3000

!topic Information hiding
!number RI-3000
!version 2.1

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

1. (Defer Implementation Details in Body) Ada 9X 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) Ada 9X 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. Ada 9X shall allow the declaration of visible non-constant objects of a private type or of a record type containing a private type within the package declaring the 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-0082
!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-0577
!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.

!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

RI-5020

!topic Subprogram body implementation
!number RI-5020
!version 2.1

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

1. Ada 9X 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.

Ada83's subprogram renaming mechanism is not an adequate 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.

!rationale

Improving separation of specifications from implementation details should be part of the Ada 9X "clean-up" revisions.

!appendix

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

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.

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 RI-5060

RI-5060 covers a broader scope of inconsistencies and restrictions in the renaming facilities in Ada83.

%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

4.7 Portability/Interoperability

The single RI in this section addresses general user perceptions that there are missing or weak restrictions in the language definition which can cause problems when porting an Ada program from one Ada implementation to another.

RI-3986

!topic Portability
!number RI-3986
!version 2.1

!Issue administrative (1) desirable

1. (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-0252
!reference RR-0253
!reference RR-0254
!reference RR-0333
!reference RR-0346
!reference RR-0365
!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

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

%reference Rationale 5.4

[....there are occasions where the demands for a very efficient implementation outweigh those for complete 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-0252 Math model leads to inconvenience, inefficiency, non-portability

The entire math model in Ada is insufficient to provide 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

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-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 ("error 1");
  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

[... because it evidences implementation characteristics. The solution to this 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. ISSUES CONCERNING SPECIFIC ASPECTS OF THE LANGUAGE

The Revision Issues [RIs] in this section concern the specific aspects of the Ada language that are not well understood; or, are understood but are seen as too restrictive or not sufficiently well defined to effectively support the variety of applications for which Ada is being used. The majority of Revision Requests [RRs] and Workshop Issues [WI's] submitted for the Ada 9X revision process address language or implementation issues that are perceived by users as in need of "tuning." Most of the RIs in this section are concern with removing restrictions from what is already provided and with providing more uniform and effective implementations of the Ada language model.

5.1 Syntactic and Lexical Properties

The RIs in this section address user concerns for the lexical and syntactic properties of Ada. Issues include internationalization of the lexical character set as well as a variety of suggested changes to make the syntax of the language more regular and predictable.

!topic Character set issues
!number *RI-2003*
!version 2.1

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

1. (Base Character Set) The base character set of Ada 9X shall be based on the ISO 8859-1 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) Ada 9X 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) Ada 9X 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) Ada 9X 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.]

!Issue implementation (6) compelling,moderate impl,upward compat,consistent

6. (Alternate Character Set Support) Ada 9X shall provide a mechanism to obtain input/output facilities and IMAGE attributes for alternate character sets (specifically, for EBCDIC) comparable (in both functionality and performance) with what is provided for the built-in character set.

!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 sweden-workshop
!reference
[Brender 1989] Ronald F. Brender Character Set Issues for Ada 9X 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 Ada 9X 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 wh·ch 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; ISO 8859 is recommended as a adequate solution.

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

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

%reference RR-0438 Allow 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-0201 Support use of in: char sets, comments & char literals minimum

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

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

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

%reference ==ai-00420 Allow 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-0736 Need 8-bit ASCII in Ada

%reference sweden-workshop

The participants of the Sweden Workshop indicated a strong (Essential) need to have the ability to obtain I/O for alternate character sets, specifically EBCDIC. 2003.6 was added for this reason.

!rebuttal

RI-1090

!topic Syntax of the language
!number RI-1090
!version 2.1

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

1. (Language Syntax) The syntax of Ada 9X should be as consistent, convenient, easy-to-read, and self-documenting as is reasonably possible given other constraints on and goals for the revision of the language.

!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
!reference RR-0199
!reference RR-0205
!reference RR-0340
!reference RR-0596
!reference RR-0673
!reference RR-0126
!reference RR-0250
!reference RR-0251
!reference RR-0264
!reference RR-0290
!reference RR-0397
!reference RR-0491
!reference RR-0534
!reference RR-0556
!reference RR-0632
!reference RR-0695
!reference RR-0708
!reference RR-0755
!reference RR-0031
!reference RR-0046
!reference RR-0049
!reference RR-0152
!reference RR-0221
!reference RR-0391
!reference RR-0504
!reference RR-0548
!reference RR-0614
!reference RR-0615

!problem

There are potential changes to the syntax of the language that might

1. improve the internal consistency of the syntactic rules;
2. make programming more convenient for the user;
3. make Ada 9X programs easier to read, and;
4. make Ada 9X programs more self-documenting.

Refer to the above-referenced items for details.

!rationale

Clearly, improvements to the syntax of the language should be made if they do not adversely impact other constraints on and goals for the language revision.

!appendix

Note: it is emphasized that the existence of the RR and AI references in this RI does not in any way imply that the specific suggestions in the referenced RRs and AIs are worthwhile or have any merit; no formal judgement is being passed on these suggestions during the requirements process.

%reference RR-0028

Add a semicolon terminator to the SEPARATE clause to make it look like other things.

%reference RR-0132
%reference RR-0141
%reference RR-0200
%reference RR-0362

Allow "when condition" on raise statement for consistency with exit statement.

%reference RR-0499

Like other "blocks", allow exception handlers in accept statements.

%reference RR-0625

Change EXIT/WHEN to WHEN/EXIT to parallel Ada IF and also English.

%reference RR-0751

Add a when/raise construct to the language for help with assertion checking.

%reference RR-0753

Make the syntax of task type declarations more consistent with the rest of the language.

%reference RR-0199

Allow IF, CASE, and SELECT constructs to be named.

%reference RR-0205

Allow a program unit name with "private", "begin", and "exception".

%reference RR-0340

Allow optional simple name on case, if, and select statements.

%reference RR-0596
%reference RR-0673

Allow "END type_name" to substitute for "END RECORD".

%reference RR-0126

Allow underscore before "E" in numeric literals.

%reference RR-0250

Use a clearer notation for expressing null ranges.

%reference RR-0251
%reference RR-0534
%reference RR-0556
%reference RR-0755

Parentheses are used for too many purposes in the language. This leads to confusing names, expressions, and aggregates.

%reference RR-0264

Discriminants need to stand out more from the rest of the components.

%reference RR-0290

Use of range notation in record rep specs is easily confused with the range notation used in range constraints.

%reference RR-0397

Replace keyword "pragma" with something capturing meaning better.

%reference RR-0491
%reference RR-0632
%reference RR-0695

For sake of clarity and consistency, allow an EXIT from a block statement

%reference RR-0708

Allow infix function calls for all functions regardless of designator.

%reference RR-0031
%reference RR-0046

```
    if X in 3 | 4 | 6 | 9 then ...
```

%reference RR-0049

Allow a special notation for the case where the same name is on both sides of :=.

%reference RR-0152

Allow e.g., a < b < c which would mean a < b AND b < c.

%reference RR-0221

There is a need to be able to write common code for group of exception handlers.

%reference RR-0391

The syntax for based numbers is clumsy, especially in aggregates.

%reference RR-0504

Add an "exchange" operator to the language.

%reference RR-0548

Allow convenient syntax for instantiating a nested generic unit.

%reference RR-0614

Allow WHEN/RETURN in functions similar to EXIT/WHEN in loops.

%reference RR-0615

Define a LOOP/UNTIL control structure as in Pascal.

An RT member provided the following review of the suggestions covered by this RI:

RR-0625 change exit/when to when/exit.

Upward compatibility problem.

RR-0753 Syntax of task type declaration

Upward compatibility problem.

RR-0251 et al. Alternatives to parentheses

"[", "{", etc. are part of the replaceable character from ISO-646. Even if we go to ISO-8859-1, there will be a problem for the international community for years to come. Brackets also allow Ada programmers to replace vectors with functions, etc. Many think this is a powerful feature.

RR-0397 replace "pragma"

Upwards compatibility problem, and many like it (pragma, that is), including me.

RR-0031, 0046 if X in 3 | 4 | 6 ...

If we get discontiguous subsets, then

```
        if X in some_discontiguous_subset
```

works well, otherwise, a set package can return the next value of an iterator.

RR-0152 a < b < c for a < b and b < c

< is overloadable. a < b returns a boolean. It could cause REAL problems.

RR-0504 "exchange" operator.

How specialized do we want to get? Of course it is also overloadable.

RR-0391 Change syntax for based numbers.

It may be clumsy, but upward compatibility will keep in in. I certainly do not want 2 different syntaxes for based numbers.

!rebuttal

5.2 General Issues With Respect To Types/Operations/Expressions

The RIs in this section address a variety of perceived problems with the general typing mechanism in Ada. These issues include general complaints of "inconvenience and awkwardness" or suggestions for additions, deletions, or regularization of Ada typing mechanisms. The thirteen RIs in this section attempt to deal with the variety of issues raised by RR's.

RI-5010

!topic Type renaming and re-exporting
!number RI-5010
!version 2.1

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

1. (Type renaming) Ada 9X shall provide a means to declare synonyms for types. Subtype and derived type declarations in Ada83 do not achieve the desired type renaming capability.

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

2. (Re-exporting types) Ada 9x shall allow packages to import types and then re-export them under new names. Re-exported types, under their new names, shall include the operations and literals of their original types plus any newly defined operations. Subtype and derived type declarations in Ada83 do not achieve the desired type re-exporting 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

Ada83 does not provide the kind of visibility control for types that it does for packages and subprograms through renaming declarations.

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

Ada 9X 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. Potential solutions should take into account possible interactions with Object-Oriented Programming (OOP) changes to Ada 9X'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
%reference RR-0172

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-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 2.1

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

1. Ada 9X 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.

%reference RR-0052
%reference RR-0482

Suggested solutions include multiple derived type declarations (RR-0052), and derived subprogram declarations (RR-0482). 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.

!rebuttal

RI-5200

!topic Miscellaneous issues concerning types
!number RI-5200
!version 2.1

!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 (problematic) (2) desirable,severe impl,bad compat,inconsistent

2. (Mutable types) Ada 9X 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:

1. Not being able to use a derived type to complete the declaration of a private or incomplete type with discriminants
2. Not being able to use a subtype to complete the declaration of a private or incomplete type
3. Not having a way to hide selected operations for derived types
4. Not having a way to combine enumeration types to form supertypes
5. 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

RI-0110

!topic Non-contiguous subtypes of discrete types
!number RI-0110
!version 2.1

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

1. Ada 9X shall provide non-contiguous subtypes of discrete types.

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

2. Ada 9X shall provide user-defined constraints for subtypes.

!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. Furthermore, constraint checks are important to many users of non-contiguous subtypes.

!rationale

Item [1] is the solution proposed by both RRs. It solves the simple example of:

```
type day is (Sunday, Monday, Tuesday, Wednesday, Thursday, Friday,
        Saturday);
for day use (0, 1, 2, 3, 4, 5, 6);

subtype bad_day is day'(Monday, Wednesday);
```

Item [2] is a more general proposal for the problem and represents a solution to [1]. With user defined constraints, then, it may be possible to define the subtype with a contiguous range and a user defined function to test the constraints.

```
subtype bad_day is day;
function F (D: in day) returns boolean;
for bad_day'user_constraint use F;
```

!appendix

%reference RR-0058

When interfacing Ada code to external devices the use of enumeration types, with representation clauses is used. Because values are defined for the enumeration type, via a representation clause, an order for the elements is implied. In the Ada code, however, subtypes of the enumeration type may need to be declared. The problem is that these subtypes may neeed to have non-contiguous elements form the base type.

%reference RR-0603

Subtypes of discrete types currently must consist of contiguous ranges of elements. Unfortunately, there are many cases in which a subtype consisting of a non-contiguous set of elements from a parent type is what is really desired.

!rebuttal

!topic General references
!number RI-2011
!version 2.1

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

1. (General References) Ada 9X shall provide a 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. Ada 9X shall provide operations for obtaining a reference for an object in any appropriate reference type; these operations shall maintain strong typing. Ada 9X shall also provide operations for converting machine addresses to any appropriate reference type. Ada 9X shall require that programmers explicitly mark any instances of obtaining references that have the potential to create dangling pointers.

!reference RR-0018
!reference RR-0258
!reference RR-0293
!reference RR-0338
!reference RR-0524
!reference RR-0726
!reference RR-0773

!reference sweden-workshop

!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 addres. 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 s. fety. Certainly, there are no safety problems with obtaining a reference to any object that is statically allocated or allocated by an allocator.

There is, of course, a safety issue involved in pointing into the stack. The conclusion of the Trusted Systems Working Group was that complicated rules for eliminating dangling references are more greatly to be feared than the dangling references themselves--particularly if a programmer is required to mark those that may be dangerous.

!appendix

An important aspect of this issue is 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, Ada 9X 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.

%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 convertability 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

The Ada model does not include the concept of a dangling reference except by UNCHECKED_DEALLOCATION. No further introduction of this problem should be introduced into the language.

!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.

RI-2034

!topic Typing at a program interface/interoperability on data
!number RI-2034
!version 2.1

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

1. (Data Interoperability) Ada 9X shall provide the means to specify the location and the bit/byte ordering of memory resident data shared between an Ada program and an external system component and shall also provide operations for accessing and updating such data.

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

2. (Data Validation) Ada 9X shall provide a mechanism to validate data that enters a program via an untyped interface. For this reason, Ada 9X shall prohibit an implementation from removing uses of this mechanism during optimization.

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

3. (Knowledge of Contiguous Allocation) Ada 9X shall provide mechanisms by which an executing program may determine if a particular object is contiguously allocated without *hidden fields*.

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

4. (Canonical UNCHECKED_CONVERSION) Ada 9X 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.

[Rationale] This idea is not consonant with the "UNCHECKED" aspect of UNCHECKED_CONVERSION – which is that only when the details of a compiler's data representations are known can the returned value for UNCHECKED_CONVERSION be predicted. If this is desired as a solution to requirement RI-2034.1, then the more general statement is more appropriate.

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

5. (Weakening Typing at Subprogram Boundaries) Ada 9X 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.

[Rationale] The concern that is addressed here is that much underlying systems code is inherently typed by memory addresses and one-dimensional arrays of memory units; thus, at system-call boundaries, there may be a great deal of redundant type casting. However, those concerned with programming language security would maintain that the type casting is not redundant – i.e., that the explicit casting specifications, though repetitious, are valuable to those maintaining the program. Ada 9X will continue to try to achieve the paradigm that a programmer should be aware whenever his code is not type safe.

!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
!reference RI-2011

!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 for external interface:

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 address of any region of memory used for external interfacing should be a parameter of a program for portability reasons. This aspect of the problem of data interoperability is covered in RI-2011 - General References.

!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".

RI-2034.2 addresses the issue that data coming in through an external interface will not always obey that data constraint rules of Ada objects. Some consideration must be given in the language to how such data can be validated 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 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. 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.

!sweden-workshop

Participants at the Sweden Workshop felt that a previous statement of RI-2034.1 was too detailed; it used the concept of an "External Interface Block" to describe the operations needed. The current version does not rely on this concept. Also, restatement at a higher level allowed part of the old RI-2034.3 to be subsumed. The need for pointers to statically-allocated memory areas was thought to be a subset of the concerns in RI-2011; thus, the item discussing this aspect was removed.

!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:

1. 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,
2. 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,
3. 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,
4. the combination of pragma Pack and a length clause for its component type must completely determine the top-level layout of unconstrainted array types,
5. length clauses applied to constrained array (sub)types must not include descriptors in the size,

6. at least signed and unsigned representations of discrete types must be supported for scalar objects and for scalar components of arrays and records,
7. 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),
8. 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.

From 5.3.3

%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

!topic Equality - redefinition
!number RI-2035
!version 2.1

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

1. ("="(x,y:same_type) return STANDARD.Boolean) Ada 9X 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) Ada 9X shall allow "=" to be overloaded with any user-supplied definition returning STANDARD.Boolean.

!Issue out-of-scope (problematic) (3) desirable,small impl,upward compat,inconsistent

3. ("="(x:a_type;y:any_type) return whatever_type) Ada 9X 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).

!sweden-workshop

Some members of the Trusted Systems working group were offended by the anomalies in overloading "=": (1) only limited types, (2) same operand type, and (3) overloading "=" also overloads "/=". This last aspect is not covered here. Another group thought the requirements were poorly stated; after reviewing them, they seem to be adequately stated.

!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

RI-1010

!topic Definition of static expressions (not wrt generic formals)
!number RI-1010
!version 2.1

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

1. The definition of a static expression in Ada 9X 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

RI-1011

!topic Staticness inside generic units with respect to generic formals
!number RI-1011
!version 2.1

!Issue out-of-scope (problematic) (1) important,severe impl,upward compat,inconsistent

1. Ada83 restrictions based on the "non-staticness" of generic formal parameters shall not be present in Ada 9X. 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
!reference RI-5010

!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 compileo. This makes it difficult to develop generic software. It also gives rise to frustrations in converting non-generic software to generic software.

!sweden-workshop

The workshop participants labeled this issue "nice-to-have" and "desirable" but appeared to recognize the difficulties in providing this kind of capability.

The re-export aspects of RR-0455 are covered in RI-5010.

!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

RI-5210

!topic Null ranges
!number RI-5210
!version 2.1

!Issue revision (1) desirable,small impl,moderate compat,unknown compat

1. (Null ranges) Ada 9X 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
!reference Ada 9X Requirements Workshop, Soderfors, Sweden, April 1990

!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 if the "effective" 'LAST were defined 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.

%reference Ada 9X Requirements Workshop, Soderfors, Sweden, April 1990

The safety-critical/trusted systems working group objected to the upward compatibility problems of both the unknown changes in program behavior due to the recommended requirement and the exceptions that would be raised if the rebuttal were followed. All of the code samples presented as examples of where super-null ranges might arise in practical programs would compile and execute correctly with the recommended language change.

!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.

RI-0200

!topic Anonymous Type Definitions
!number RI-0200
!version 2.1

!Issue out-of-scope (problematic) (1) desirable,severe impl,upward compat,mostly consistent

1. (Anonymous Types Tr :ated the Same as Types) Ada 9X shall shall allow the use of an anonymous type in any location when a type may occur.

[Rationale] This introduces multiple dependencies into the type structure of the language, and makes discriminant constraint checking unusually difficult. See appendix on NOTE-055.

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

2. (Inconsistencies in Anonymous Types) Ada 9X shall attempt to remove inconsistencies with respect to anonymous types. For example,

1. allow unconstrained array definitions in the syntax of constant array definitions [AI-00538]. For example, C: constant array(INTEGER range <>) of INTEGER := (1,2,3);
2. allow record types in object declarations.
3. allow component of a record to be any type or subtype that is allowed in an object declaration.

!Issue out-of-scope (problematic) (3) desirable,moderate impl,bad compat,mostly consistent

3. (Remove Anonymous Types) Ada 9X shall not allow anonymous types.

[Rationale] Anonymous types are currently in the language and their removal would break many existing program. This dramatic step is not needed since anonymous types are already restricted to only a few uses in the language.

!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
!reference RI-2033 (enhancing the model of record types)

!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.

!rationale

[2] The use of anonymous types is rather restrictive in Ada. Some of the restrictions are considered minor and should be removed. For example, unconstrained array definitions are not allowed in the syntax of constant array definitions [AI-00538],

```
    C: constant array(INTEGER range <>) of INTEGER := (1,2,3);
```

Other example include allowing record types in object declarations and allowing componets of a record to be any type or subtype that is allowed in an object declarations. This would reduce the need for "extra" type definitions.

!appendix

NOTE-055

Subject: Re: Double Dependence on Discriminant
Author: AH
Date: 23 Jul. 81

Yes, by eliminating anonymous arrays, the Emilion proposal eliminated double dependencies such as:

```
   type R (S: Integer) is
     record
        V: array (1..10) of T(S);
     end record;
```

where the type T is some discriminated record type.

```
   type T (W: Natural) is ...
```

Speaking for myself, I find the double-dependency of limited utility. I have a hard time constructing realistic examples of its usefulness.

I believe you are right that the implementation is unnecessarily difficult. In one straight-forward implementation of discriminants, each record object contains space for the discriminant field. Thus, for an object of type R, each of the 10 components of type T would contain its own copy of the W discriminant field. The object R itself contains space for the S discriminant field. (Recall that Hilfinger does NOT prefer this implemen ation, but would rather try to share a common copy of the discriminant).

Let us consider what must happen for checking discriminant constraints during assignment. Suppose we have two objects of type R:

```
A: R(i);
B: R(j);
```

and that we write the assignment statements:

```
A := B;              -- (1)
A.V(1) := B.V(1);    -- (2)
A.V := B.V;          -- (3)
```

The code for checking the discriminant constraints must access the discriminant fields. In assignment (1), the field B.S is compared against A.S. In assignment (2), the field B.V(1).W is compared against A.V(1).W. In both of these assignments, the discriminant fields are taken from the objects we are about to assign. We can think of the addresses of the bases of the objects as a common subexpression in the two operations which make up the assignment:

```
Check_Discriminant(SourceObjectBaseAddress.DiscriminantOffset,
                   DestinationObjectBaseAddress.DiscriminantOffset);
Copy( SourceObjectBaseAddress, DestinationObjectBaseAddress );
```

In assignment (3), the discriminant need only be checked once (rather than 10 times) as it is necessarily the same for the entire array V. The array component V does NOT have its own copy of the discriminant field. We must either (i) use the discriminant field of the surrounding object of type R, or (ii) reach into some component of the array (say, the 'First one) and use its discriminant field.

Both approaches complicate the code in the compiler for assignment, as they make component selection and indexing "context sensitive" – that is, when the compiler generates code for selected components and index components, e.g.,

```
X.Y(k). ... .Z
```

it must take into account whether the ultimate name "Z" is an anonymous-array-with-a-double-dependent-component or not. If it is, then (in approach (i)) the discriminant field of the outermost object (or even some intermediate component object) must be accessed for the discriminant check. Consider the record declaration D:

```
type D (S: Integer) is
  record
     E: array(1..2) of array(1..3) of array(1..4) of array(1..5)
        of array(1..6) of array(1..7) of T(S);
  end record;
```

Approach (ii) is particularly messy with the type D as an assignment

```
A.E := B.E;
```

might require us to reach down through arbitrarily many anonymous arrays until we find a component which has a copy of the discriminant.

Yet a third approach is to implement the double-dependency as a parameterized array type. The record type R is thus treated something like:

```
type ANON (U: Natural) is array (1..10) of T (W => U);

type R' (S: Integer) is
   record
      V: ANON (U => S);
   end record;
```

An object of a parameterized array type includes space for its discriminants – these are probably allocated in the array descriptor along with the index constraints, array multipliers (if any), etc. The point here is to give the array object its own copy of the discriminant. Of course, in the compiler, a record type declaration such as R must be analyzed carefully to decide if it has an anonymous-array-with-a-double-dependency, and that it should thus be implemented as a parameterized array type.

My conclusion is that double-dependencies aren't worth the implementation effort.

–Andy Hisgen

P.S. If we adhere to Lee Maclaren's suggested ground rules, there is really no problem in the language with double-dependencies, and they should thus stay in. Also, at one point Ike Nassi was a strong supporter of parameterized array types; he thus might have strong feelings about double-dependencies, and you should make sure you get his input.

%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:

```
type T (n: integer) is record
   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.

!rebuttal

!topic Dimensional mathematics.
!number RI-3749
!version 1.4

!Issue revision important,moderate impl,upward compat,somewhat consistent

[1] Ada 9X 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 r^present 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

5.3 Issues With Respect To Specific Kinds Of Types

This section deals with perceived problems with the basic kinds of types and corresponding operations that Ada provides. By "basic kinds of types" we are speaking of the various scalar types, record types, and array types.

RI-5120

!topic Array aggregates
!number RI-5120
!version 2.1

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

1. ("OTHERS" and "sliding" in array aggregates) An attempt shall be made in Ada 9X to uniformly allow both named associations with an OTHERS choice and "sliding" of index values for array aggregates. At a minimum, Ada 9X shall allow named associations for default array aggregates to include an OTHERS choice where no subtype conversion is applied to the aggregate; i.e., where no "sliding" of index bounds occurs. [Note: This minimum requirement reflects the intent of LRM 4.3.2(6) as described in AI-00473.]

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

2. An attempt shall be made in Ada 9X to reduce the anomalies that exist with array aggregates in Ada83.

!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

!rationale

A more uniform treatment of array aggregates would simplify the language and its implementations. The "minimum" part of issue [1] is recommended in AI-00473 as reflecting the intent of the design of Ada83.

!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.

!rebuttal

!topic Array slices and cross-sections
!number RI-5130
!version 2.1

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

1. (General array slicing) Ada 9X shall support slicing of multi- dimensional arrays. That is, a sub-array of the same dimension as the complete array may be selected by specifying a range of index values for each dimension.

[Rationale] Ada's one-dimensional array slicing is elegant and efficient. Similar elegance could be extended to matrix operations with moderate implementation impact. The impact of extending this capability to the general case, however, would be severe. It would essentially require dope-vector implementations for "sliceable" arrays, which could carry significant performance penalties for all arrays. Limiting the extension to two dimensions would just as inconsistent as limiting it to one. Creating a new class of "sliceable" arrays to limit the extent of the performance penalties would add another wart to the language.

!note-to-DRs

It seems to me that the BLAS could be implemented by unchecked conversion of matrices to one-dimensional arrays if the element storage order were known. Do we need pragma ROW_MAJOR?

!Issue out-of-scope (2) desirable,severe impl,upward compat,inconsistent

2. (General array cross-sectioning) Ada 9X shall support cross- sectioning of multi-dimension arrays. That is, arrays of smaller dimension may be selected from multi-dimension arrays.

[Rationale] While a cross-sectioning 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 cross-section indexing would increase the size of compilers and reduce compilation speed with no guarantee of improved run-time performance. The easiest implementation would probably be to copy the cross-section and manipulate it normally, which is not what the RRs seem to have had in mind. There is also the difficulty of defining the type of a plane (or row or column) sectioned from a higher dimension array.

!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.

For example, for the array

    A: array (1..10, 1..10, 1..10) of FLOAT;

a 3 by 4 by 5 slice might be represented by the expression

    A(3..5,1..4,6..10)

A one-dimensional array consisting of the elements A(3,7,N) for N ranging from 1 to 10 might be represented by the expression

    A(3,7,1..10) – or A(3,7,*)

!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

RI-5141

!topic Variable-length strings
!number RI-5141
!version 2.1

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

1. (Enable definition of variable-length strings) Ada 9X shall provide language mechanisms necessary to enable a common set of declarations, operations, and semantics for variable-length strings to be defined safely and efficiently.

!external-standard (2) desirable

2. (Standard package for variable-length strings) An external standard should be created to define a common set of declarations, operations, and semantics for variable-length strings.

!reference RI-2022 Finalization
!reference RI-2102 User-defined assignment and equality
!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
!reference Ada 9X Workshop, Soderfors, Sweden, April 1990

!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 variable-length string package defined as an external standard would satisfy the need for uniform solutions to string manipulation problems without building the solution into the language. This would require data abstraction facilities that are not presently in the language. Without such facilities there would be cause to consider adding variable-length strings to Ada 9X as a predefined type. Specific requirements for the necessary data abstraction capabilities can be found in RI-2022 and RI-2102.

!appendix

Additional data abstraction facilities are needed in Ada 9X for solving much more than the variable-length string problem. Variable-length strings are a good 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. (See RI-2022.)

Overloading of assignment and equality operations would add considerably to the look and feel of abstract data objects as first class objects. (See RI-2102.) 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 harder 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.

%reference Ada 9X Workshop, Soderfors, Sweden, April 1990

Workshop participants rated variable-length strings as "important" and wanted the specification for variable-length strings to be part of the Ada 9X standard.

!rebuttal

RI-5142

!topic Miscellaneous string issues
!number RI-5142
!version 2.1

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

1. (String handling facilities) An attempt shall be made to improve the string handling facilities in Ada 9X.

[Rationale] While it would not hurt to have additional string handling facilities, no language changes are needed to support their definition.

!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)
!reference Ada 9X Workshop, Soderfors, Sweden, April 1990

!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. Pattern matching and searching
5. String editing – insert, delete, convert
6. Output formatting – a' la printf and scanf in C

!rationale

!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.

%reference Ada 9X Workshop, Soderfors, Sweden, April 1990

Participants in the information systems working group rated additional string handling capabilities as "important."

!rebuttal

RI-5150

!topic Implicit subtype conversion
!number RI-5150
!version 2.1

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

1. (Implicit Conversion of Array Subtypes) An attempt shall be made in Ada 9X 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:

1. assignment of record aggregate values that contain array components [LRM 5.2.1(5)]
2. subprogram parameter associations [LRM 6.4.1]
3. returning array values from functions (as for universal numeric values [LRM 5.8(6)])

Renaming declarations [LRM 8.5(4)] do not provide a workaround for subprogram parameter associations because the specified subtype information is ignored.

!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 record aggregates 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.

Strictly speaking, this change is not upward compatible, since currently if array aggregate bounds are not correct without subtype conversion, CONSTRAINT_ERROR will be raised. Extending the rules for subtype conversion would change the behavior of programs that relied on this mechanism for raising exceptions.

%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 **%reference** RR-0749

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.

!rebuttal

!topic Miscellaneous array issues
!number RI-5160
!version 2.1

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

1. (Vector and array operations) Ada 9X shall provide efficient built-in vector and array operations for inner product, outer product, element-wise relational operations, reductions, etc.

[Rationale] Ada 9X 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. This is an area that could be served by an external standard.

!Issue out-of-scope (problematic) (2) not defensible,small impl,bad compat,unknown compat

2. (Anonymous array types) Ada 9X 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) Ada 9X shall allow the declaration of array subtypes in which index subtypes are constrained to more restrictive ranges without fixing the constraints. [Note: the solution should be consistent with that for partially constrained record types. (See RI-2032.4)]

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

4. (Subsets of index constraints) Ada 9X shall allow the declaration of array subtypes in which some but not all of the index constraints are fixed. [Note: the solution should be consistent with that for subsetting discriminants of record types. (See RI-2032.5)]

!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. There are two flavors of this problem: one is where an index subtype is restricted to a narrower range of possible values but not fixed; e.g.,

```
type A is array (INTEGER range <>) of INTEGER;
subtype A1 is A (POSITIVE range <>);
```

The other flavor is where some index constraints are fixed but others are not; e.g.,

```
type B is array (INTEGER range <>, INTEGER range <>);
type B1 is B (1..10, INTEGER range <>);
```

!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

Ada 9X 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

Ada 9X 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

RI-2032

!topic Record discriminants
!number RI-2032
!version 2.1

!Issue out-of-scope (1) desirable,severe impl,upward compat,inconsistent

1. (Nonstatic Discriminants in Aggregates) Ada 9X 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, Ada 9X 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 (problematic) (3) desirable,small impl,upward compat,inconsistent

3. (Setting Discriminants w/o Complete Assignment) Ada 9X shall allow assignment to discriminants without requiring assignment to the whole record.

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

4. (Partial Constraining of Record Types) Ada 9X 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. (Supplying a Subset of the Discriminants) Ada 9X 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) Ada 9X 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 (7) desirable,moderate impl,upward compat,inconsistent

7. (Late Constraining of Components) Ada 9X 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 perceived user need (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. The type of partial constraining implied by RI-2032.4 could result in significant space savings as is shown in the **!problem** above.

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.

!sweden-workshop

The consensus of the Trusted Systems Group in Sweden was the RI-2032.4 and RI-2032.5 added gratuitous complexity to the language, i.e. they felt that neither the (potentially significant) space savings or more general model of subtyping was particularly important as measured against (an unknown) increase in complexity. Another group, also concerned with additional complexity, cautioned a go-slow approach. Notwithstanding this counsel, the PUNs have been left as important.

!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

```
Y   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   Allo·7 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

RI-2033

!topic Enhancing the model of record types
!number RI-2033
!version 2.1

!Issue out-of-scope (1) desirable,severe impl,upward compat,inconsistent

1. (Generic Record Attributes) Ada 9X 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 (2) desirable,severe impl,upward compat,inconsistent

2. (Anonymous Component Types) Ada 9X 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 (3) desirable,small impl,upward compat,mostly consistent

3. (Defining by Component-Wise Application) Ada 9X 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) Ada 9X 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) Ada 9X 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 diffrnt. 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 variants in record types RR-568 show that a structure frequently 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

RI-5170

!topic Uniformity of integers
!number RI-5170
!version 2.1

!Issue revision (1) desirable,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 Ada 9X.

!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:

1. Operations defined for type INTEGER, but not for other integer types (i.e., exponentiation and fixed point multiplication and division).
2. The leading space in the 'IMAGE attribute of non-negative integers.
3. 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 Ada 9X, 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 2.1

!Issue out-of-scope (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:

```
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 favor 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

Ada09X 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 'SMALL, is that the declaration

```
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 Ada09X 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.

RI-0902

!topic Implementation freedom in fixed point
!number RI-0902
!version 2.1

!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 Co-processors

This RR requests that compilers provide fixed-point support even if a floating-point co-processor 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 co-processor for support of floating-point, and the compiler is being used on a system with no co-cop processor. 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.

RI-0904

!topic Restrictions in fixed-point
!number RI-0904
!version 2.1

!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.

RI-0905

!topic Simplify floating-point model
!number RI-0905
!version 2.1

!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 accommodate 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 favorably 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 favor 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

RI-0906

!topic Provide for compatibility with IEEE floating-point
!number RI-0906
!version 2.1

!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 external 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:

1. Machine overflows is static which is not appropriate to the IEEE model of dynamic control over the handling of overflow.
2. 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.
3. 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.
4. 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 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-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 interfacing P754 to the current definition of Ada.

!rebuttal

5.4 Subprograms

The RIs in this section address RRs which point to user problems with parameters of Ada subprograms and difficulty or lack of selectivity with the inlining of subprograms.

RI-1000

!topic Reading non-input parameters
!number RI-1000
!version 2.1

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

1. Ada 9X 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 (problematic) (2) desirable,small impl,moderate compat,consistent

2. Ada 9X 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

%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. 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 "desirable"!

RI-1022

!topic User control over parameter passing mechanism
!number RI-1022
!version 2.1

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

1. Ada 9X 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.

In any event, it is noted that the effect of pass by copy can always be forced by an explicit assignment and the effect of pass by reference could forced by a mechanism allowing the creation of pointers to any variable. This latter idea is the subject of RI-2011.

!Issue implementation (2) important

2. Where Ada 9X 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.

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

3. The defined meaning of parameter passing in Ada 9X shall be bounded in all cases by the individual meanings associated with the allowed parameter-passing mechanisms. In particular, an execution of an Ada 9X program shall not be totally unpredictable because the program behaves differently for different choices of parameter-passing mechanisms. Finally, Ada 9X shall not consider dependence on a particular parameter-passing mechanism to be an error in the sense of allowing an implementation to raise PROGRAM_ERROR.

!reference WI-0103
!reference WI-0308
!reference AI-00349
!reference Trusted System and Safety Critical Ada 9X Workshop; IDA Beauregard Facility, Alexandria, VA; January 25-26, 1990.
!reference RI-2011
!reference RI-3514

!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

In certain safety-critical and real-time communities, it is considered essential to have documentation on how a compiler selects its parameter-passing mechanisms. This could be provided either statically in an appendix of the reference manual or dynamically in source listings. This is suggested here as an implementation requirement as it is not expected to be of concern to most Ada users.

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. Indeed, relying on the particular parameter-passing mechanism chosen by the implementation seems like no more of an error that relying on some minimum bounds for STANDARD.INTEGER.

!sweden-workshop

One working group thought [1] above might be important for "interfacing to external systems". This matter is covered in RI-3514. They also thought a further examination of the efficiency issues concerning [1] was warranted. This has not yet been performed.

Item [2] above was previously out-of-scope because (A) it encourages non-portable programming and (B) it is counterproductive to require documentation of those aspects of implementation behavior upon which the standard forbids the programmer to depend. There has been little sympathy for the portability concerns and the view at the workshop seemed to be that the standard should not forbid dependence on the parameter passing mechanism (hence the "no PROGRAM_ERROR" aspect of [3]).

As a result of the workshop, Item [3] above has been taken one step farther in that it bans PROGRAM_ERROR for dependence on the parameter passing mechanism.

A need of "essential" was suggested at the workshop for both [2] and [3]. Relatively speaking, this seemed somewhat high and therefore this advice has not been followed.

!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 implem[illegible]ation techniques."

%reference WI-0308 Allow user control of subprog parameter passing techniques

Need more control over parameter passing to, e.g.,

1. interface to foreign code,
2. control parameter results when subprogram is abandoned due to exception,
3. operations on memory-mapped objects,
4. prohibit aliasing,
5. 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.

!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.

RI-1021

!topic Pragma INLINE flexibility
!number RI-1021
!version 2.1

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

1. (Inline On Subprogram Basis) Ada 9X 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) Ada 9X 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" subprogr m 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

RI-1023

!topic Miscellaneous subprogram issues
!number RI-1023
!version 2.1

!reference RR-0269
!reference RR-0598
!reference RI-3514

!Issue out-of-scope (1) not defensible,small impl,bad compat,inconsistent

1. (No Recursion by Default) Subprograms in Ada 9X shall not allow for recursion unless this functionality is explicitly asked for by the programmer.

[Rationale] This is a non-upward-compatible change to the language. An implementation-dependent usage-restrictive pragma is a more appropriate way to address the problem described below.

!Issue out-of-scope (2) desirable,small impl,upward compat,mostly consistent

2. (Multiple Return Values from Functions) It shall be possible in Ada 9X to return from a subprogram an object whose constraints are determined by the subprogram as well as separate status information.

[Rationale] This seems fairly easy to do with a function that returns a two-component discriminated record, one component for the data (constrained by the discriminant) and the other for the status information. There is a related issue of interfacing to C and FORTRAN functions that return values as well as modify parameters, this problem is addressed in RI-3514.

!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.

!sweden-workshop

It seems that two working groups at the workshop felt [2] above should not be out-of-scope. However, it appears that their primary concern was the need for Ada to be able to call non-Ada (primarily C or FORTRAN) functions that return values as well as modify parameters. This matter is covered in RI-3514.

!appendix

%reference RR-0269

See (1) above.

%reference RR-0598

See (2) above.

!rebuttal

5.5 Visibility and Packages

The RIs in this section address RRs that present problems with the ways that the Ada "with" and"use" statements provide visibility to enumeration literals and operators, and the ways that overloading in Ada works in general. One RR perceives a problem with the Ada package structure. Several AIs that address specific aspects of visibility are also included in the references for the analysis presented in these RIs.

RI-2023

!topic Visibility of operators in a WITHed package
!number RI-2023
!version 2.1

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

1. (Visibility for Character Literals) Ada 9X shall provide a mechanism to establish direct visibility of 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 literal.

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

2. (Visibility for Infix Operations) Ada 9X shall provide a mechanism to establish direct visibility of the infix operations 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. The issue is more compelling for character literals than for infix operators since character literals cannot be renamed.

!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

RI-2024

!topic Visibility and safety
!number RI-2024
!version 2.1

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

1. (Read-Only Access to Objects) An attempt should be made in Ada 9X to provide a mechanism by which only read-only access is permitted via a given reference to an object. [Note: both names and access values should be considered.]

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

2. (Strong Coupling to Packages) Ada 9X shall provide a mechanism to allow declarations made directly visible from a visible package to hide otherwise directly visible but nonlocal declarations. The mechanism shall not require that each imported entity be explicitly mentioned.

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

3. (Restricting Naming Operations) Ada 9X shall provide a mechanism to specify that certain declarations and visibility operations are not allowed in inner scopes. The following sorts of specifications should be supported:

   1. No USE-clauses.
   2. No WITH-clauses.
   3. No hiding of a specific declaration.

[Rationale] While such restrictions may be important in a particular project, it is more appropriate to use extralingual mechanisms to solve these problems.

!reference RR-0270
!reference RR-0588
!reference RR-0589
!reference RR-0266

!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.

!rationale

[2024.1] Restricting write access via a given reference can be important. For example, the referrent might be in a read-only memory and it might make no sense to attempt to write it. The workaround for names of providing a function that returns the value seems relatively clumsy; note that compilers already support nearly the correct mechanism for by-reference IN parameters.

[2024.2] This requirement is aimed at the problem that occurs when the implementation of a subunit is intimately linked with an external package; changing the declarations in the parent unit should not make declarations from the coupled package invisible. This requirement would also work the problem that the declaration in package STANDARD are very difficult to override; for example, if a programmer wanted a subtype NATURAL that was 1..INTEGER'LAST, then every library unit would have to import this type explicitly, i.e. it couldn't be done with "just" a use-clause.

!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 2.1

!Issue out-of-scope (problematic) (1) desirable,severe impl,bad compat,mostly consistent

1. (Parameter Names for Overloading) Ada 9X 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 (problematic) (2) desirable,severe impl,upward compat,mostly consistent

2. (Generic Formal Names for Overloading) Ada 9X 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, Ada 9X 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 (problematic) (3) desirable,severe impl,upward compat,mostly consistent

3. (Trailing Attributes for Overloading) Ada 9X 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 (4) desirable,small impl,upward compat,consistent

4. (Additional Infix Operators) Ada 9X shall use introduce a number of additional infix operators that would be eligible for overloading.

!Issue out-of-scope (5) desirable,moderate impl,upward compat,mostly consistent

5. (Overloading of Indexing) Ada 9X 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

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

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

RI-2500

!topic Improved model of privating
!number RI-2500
!version 2.1

!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 Ada 9X 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 (2) desirable,small impl,upward compat,consistent

2. (Specification of Constant Components) Ada 9X shall provide a mechanism to specify that certain components of an object of a record type cannot be overwritten without completely rewriting the object. Ada 9X shall not arbitrarily restrict the type of these components nor require user-intervention in initializing them.

[Rationale] The need for this feature does not seem to warrant a language change.

!reference RI-2002
!reference RR-0229
!reference WI-0216

!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.

!sweden-workshop

Participants at the Sweden Workshop felt that this issue was important instead of compelling. The thought was also that a number of RIs including this one could be rolled up into an all-encompassing RI entitled "Program Composition"; this suggestion was not followed.

!appendix

An alternative to 2500.1 which is much more solution oriented would be like the following:

%requirement

[1] To some reasonable extent, Ada 9X 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.

%reference RR-0560

RR-560 also refers to the problem that some users of a package should be granted more access than others.

%reference RR-0679

RR-0679 is really all about the idea that selection should be definable. It's an interesting idea.

%reference AI-00327

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

RI-2105

!topic Own variables in packages
!number RI-2105
!version 2.1

!Issue out-of-scope (problematic) (1) not defensible,small impl,bad compat,inconsistent

1. (Require extra specification for own variables) Ada 9X shall require extra specification for the objects declared in library packages indicating that the objects are not deallocated during the execution of a program.

!reference RR-0271

!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

This idea is wholly upwards incompatible. It is likely to break essentially every existing Ada program that uses a library package.

!appendix

%reference RR-0271 Own variables in packages are inappropriate

RR-0271 wants what is stated in RI-2105.1.

!rebuttal

5.6 Parallel Processing and Concurrency

The RIs in this section deal with the perceived user problems in understanding and using Ada tasking. The RRs question not only interactions, timing and the priority scheme. but also the suitability of Ada for parallel processing. This section contains the largest number of RIs to deal with user issues presented by RRs, Workshops, and AIs.

RI-5241

!topic Complexity and efficiency of tasking model
!number RI-5241
!version 2.1

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

1. (Speedy rendezvous) Ada 9X real-time implementations shall be required to achieve standard levels of performance for rendezvous for (at least the non-distributed parts of) real-time applications.

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

2. (Low-level tasking facilities) Ada 9X shall provide efficient low-level tasking operations.

!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)
!reference Ada 9X Requirements Workshop, Soderfors, Sweden, April 1990

!problem

The general theme of these revision requests is the poor performance of implementations of Ada's rendezvous tasking mechanism. There are two problems here. The first 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. The second problem is that requiring multiple rendezvous to model interprocess synchronization schemes that are conceptually simpler than a single rendezvous cannot achieve respectable performance and makes programming more difficult than necessary. This problem of implementing low-level facilities using high-level constructs has been called "abstraction inversion."

!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 tasking facilities. Examples include low-level tasking library packages supplied by compiler vendors 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 problem in Ada 9X.

Item [1] addresses the rendezvous performance problem for at least the non-distributed parts of real-time applications via the implementation standards mechanism. (Performance of distributed operations, communications, etc. are separate issues.) It is not clear how the expected performance of a rendezvous can or should be specified. RI-5220 calls for a fast mutual exclusion mechanism, which is a more specialized solution for a particular kind of task synchronization operation.

Item [2] addresses the abstraction inversion problem. It seems quite impossible to solve this problem without introducing some appropriate set of lower-level tasking primitives. Having two ways to specify concurrent operations within the language has serious drawbacks. Unfortunately, it appears that this situation already exists. And what's worse, the low-level operations that real-time system developers feel compelled to use are not standard! RI-5230 calls for creation of an external standard for low-level tasking primitives, which is the solution preferred by several real-time Ada constituencies. It is not clear how closely the set of low-level primitives can or should be tied to underlying primitives used to implement rendezvous.

!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).

%reference Ada 9X Requirements Workshop, Soderfors, Sweden, April 1990

Workshop participants felt the issues presented in this RI were adequately covered in other RIs.

!rebuttal

RI-5220

!topic Mutual exclusion
!number RI-5220
!version 2.1

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

1. (Mutual exclusion) Ada 9X shall define a standard mechanism for exclusive access to data and operations.

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

2. (Fast mutual exclusion) Real-time implementations of Ada 9X shall provide efficient implementations of the standard mutual exclusion mechanism.

!reference RR-0185 (related)
!reference RR-0241 (related)
!reference RR-0461 provide standard package of semaphore operations
!reference RR-0590 standardize package-level mutual exclusion
!reference WI-0415 Need efficient mutual exclusion
!reference Ada 9X Requirements Workshop, Soderfors, Sweden, April 1990

!problem

Ada83 provides no highly-efficient, portable mechanism for mutual exclusion. Many real-time applications cannot afford the overhead of a general rendezvous to achieve simple mutual exclusion and are forced to adopt non-portable solutions. In particular, an efficient mechanism is needed for shared reading of data with locks for exclusive writing.

!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 direction from several of the RRs on this issue is to standardize the many variants of semaphore packages currently provided by vendors.

A specification of the mutual exclusion mechanism in terms of Ada tasks would imply synchronization points and overhead, which are the major causes of the problems in the first place. A procedural interface for semaphores, for example, would allow non-real-time implementations to provide a tasking solution while real-time implementations could use other techniques that do not incur the associated performance penalties.

%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 RR-0590

Writing explicit tasks for mutually exclusive access adds extra tasks to a design, which may carry significant task-switching overhead and may confuse readers. If hidden in a package body, these tasks could create timing or deadlock problems that would be difficult to debug.

%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).

%reference Ada 9X Requirements Workshop, Soderfors, Sweden, April 1990

The real-time working group recommended changing the external standard approach presented in an earlier draft of this RI to an "internal" standard, and clarified the need for shared-read-exclusive-write access to data. It also recommended increasing the need rating and decreasing the implementation impact rating.

!rebuttal

!topic Low-level tasking facilities
!number RI-5230
!version 2.1

!external-standard (1) important,unknown impl,upward compat,consistent

1. (Standard low-level tasking facilities) An effort outside the Ada 9X language revision effort should be established to standardize a low-level tasking interface suitable for building efficient alternative tasking or task scheduling mechanisms

!reference RI-7005 Priorities and Ada's scheduling paradigm
!reference RI-7007 Support for alternate run-time paradigms
!reference RI-7010 Flexibility in sclection for 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.
!reference Ada 9X Requirements Workshop, Soderfors, Sweden, April 1990

!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:

1. dynamic priorities
2. priority entry queuing
3. priority inheritance.

Examples of alternate task scheduling mechanisms that are needed but not currently supported include:

1. rate-monotonic
2. 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 non-portable or implementation-dependent solutions, Ada 9X should allow the specification of 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 Ada 9X 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 Ada 9X 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. Ada 9X should define a standard set of asynchronous real-time primitives that can be implemented directly in the run-time executive.

!reference Ada 9X Requirements Workshop, Soderfors, Sweden, April 1990

The real-time working group recommended limiting the scope of this RI to the capability to write customized schedulers. They also wanted it incorporated in the Ada 9X standard, rather than an external standard, and rated the need for the scheduling capability higher than the broader tasking capabilities presented above.

!rebuttal

RI-7005

!topic Priorities and Ada's scheduling paradigm
!number RI-7005
!version 2.1

!note_to_DRs

The Requirements team would like you to give serious consideration to both the proper body of this RI, and the **!rebuttal**, and to provide guidance to the RT as to your opinion of the best way for Ada 9X to handle this area. A fundamental question is whether Ada 9X should be a general-purpose language with a hefty "implementation" section for Real Time, or if it should be a Real Time language with "escapes" for those who need alternate behaviour.

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

1. (Consistent with Priority) Ada 9X 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:

1. Priority must be used as the main criteria for choosing the next candidate task when more than one choice is possible. 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.
2. Priority Inheritance, where an executing task inherits the priority of the highest priority task queued on any of its entry queues, is required.
3. Dynamic Priorities, where an executing task can modify its own, priority and to support mode-shift, is required.
4. New task interactions which may be defined as part of Ada 9X 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 (2) not defensible,moderate impl,bad compat,inconsistent

2. Ada 9X 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.

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

3 Alternative to Preemptive Scheduling] Ada 9X shall provide a mechanism for a program to obtain the effect of non-preemptive scheduling.

!reference RR-0013
!reference RR-0015
!reference RR-0020
!reference RR-0021
!reference RR-0072
!reference RR-0075
!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 AI-00594
!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).
!reference
Sweden Ada 9X Requirements Workshop

!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.

[3] An additional problem exists in that the priority model is sometimes overspecified. Although preemptive scheduling is mandated by the RM, and AI-00032, programs sometimes need alternate behaviour. The Ada solution of not using PRIORITY is seen as being too restrictive in AI-00594.

!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 expressed need for dynamic priorities is to support mode shift, or to enable a task to adjust its priority for special needs. It is felt that the general case of asynchronous setting of another task's priority is not needed.

[3] The current Ada approach to priorities and preemption is accepted by the realtime community as the appropriate model, with the fixes asked for in [1]. The policy for programs which need priorities with a different behaviour than the preemptive one is to not use Ada priorities, then to define another, implementation-specific priority and customize the RTE around that. This approach is unacceptable to many developers because of cost to develop an alternate RTE, cost of maintaining alternate code, or being forbidden by the customer from using "non-Ada" solutions. Ada 9X must recognize the needs of such classes of users, and provide relief, either as guidance on when it is

appropriate to use a non-preemptive model coupled with Ada's notion of "priority", or a language mechanism to specify the behaviour required.

!appendix

Discussion of restrictions on Dynamic Priority

Explicit 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-000594.

This RR opposes a position taken in (unapproved) AI-00594 that there could be times when it is sensible for a low priority task to have processing resources while a high priority task is waiting for the same resource.

%reference AI-00594 Sensible preemption by high priority tasks

%reference 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

%reference WI-0412 Don't prohibit scheduling by context, incl. dynamic

%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.

%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

%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

%reference Sweden Ada 9X Requirements Workshop

The Workshop Real Time group endorsed the statement of the !problem and requirements, except that it was felt that explicit mention of reasons for dynamic priority were required: mode shift and alternate scheduling paradigms. Since alternate scheduling paradigms is discussed in RI-7007, it is not included here.

!rebuttal

The problem discussed in this RI is certainly one that should be solved in the Ada 9X process. This rebuttal is mostly concerned with choosing the best mechanism for addressing the problem. The recommendation of the RI is essentially that a dynamic-priority-with-preemption model be adopted as the Ada standard with some escape mechanism to be available for those times when this model is not appropriate. This is counterproductive; a preferable resolution would be one that defined a wide range of allowed variation (i.e. as wide as the class of appropriate scheduling models) but that achieved portability via the mechanism of an internal, implementation standard. Therefore, the following method seems more appropriate:

[1 – Enable Alternate Scheduling Paradigms] Ada 9X shall not arbitrarily restrict the placement, binding time, or interpretation of scheduling parameters. [Note: Ada83 restricts both the placement and binding time of pragma PRIORITY; thus, the latest binding time for the priority of a task is when its specification is compiled.]

[2 – Preemptive Scheduling by Dynamic Priority] Ada 9X shall define an implementation standard that provides a model of task behaviour that is based upon

1. guaranteed preemptive scheduling--in a uniprocessor implementation, the highest priority unblocked task is guaranteed to begin/resume execution within some bound;
2. dynamic priority with priority inheritance--not only may the base priority of a task change dynamically during program execution, but the RTS will adjust the instantaneous priority of a given task based on the priority of the tasks enqueued on the entries owned by that task.
3. using priority to control essentially all aspects of scheduling--this would include not only selection of the next task to run but also (i) entry queue order and (ii) selection of an open alternative in a selective wait.

!rebuttal

RI-7007

!topic Support for alternate run-time paradigms
!number RI-7007
!version 2.1

!Issue revision (1) important

1. (Do not preclude alternate RT models) Ada 9X 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) Ada 9X shall remove restrictions which prohibit or have the effect of prohibiting alternate run-time paradigms. Ada 9X shall not specify new constructs or restrictions which have the effect of prohibiting the use of alternate run-time paradigms.

!Issue out-of-scope (problematic) (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 time-slice, run-to-completion, 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.

[Rationale] 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).

!Issue external standard (4) compelling

4. (Interface to Alternate Run-Time) The standard shall provide an interface specification to alternate run-time executives which support paradigms such as Time Slicing, 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 RI-5230
!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.
[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).
[3] J. Goodenough, L. Sha "Real Time Scheduling in Ada" Computer, April, 1990
!reference Ada 9X Sweden Requirements Workshop April 23-26 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:

1. Ada83's specification of static priorities
2. FIFO task entry queues
3. 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 Ada 9X), 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.

!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 (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 RR-0124

This RR points out that problems in Ada83 connected with task interactions and scheduling should be solved by expanding the run-time model, and by including other paradigms such as rate-monotonic.

%reference RI-7010

This RI addresses the need to control task entry queue behaviour and selection for rendezvous.

%reference Ada 9X Sweden Requirements Workshop April 23-26 1990

Ada 9X must be able to support the specification of a variety of run-time paradigms from within the language. For this reason it is believed that item 3 is a requirement, not Out-Of-Scope, and that items 1 and 2 should defer to RI-5230 (low-level tasking primitives). It is the strong feeling of the group that disclosure of alternate run-time behaviour should be made in the source code.

!rebuttal

This rebuttal is derived from discussions of the Real Time Group at the Sweden Requirements Workshop.

Ada 9X must be able to support the specification of a variety of run-time paradigms from within the language. For this reason it is believed that item 3 is a requirement, and that items 1 and 2 should defer to RI-5230 (low-level tasking primitives). It is the strong feeling of the group that disclosure of alternate run-time behaviour should be made in the source code.

RI-7010

!topic Flexibility in selection for rendezvous
!number RI-7010
!version 2.1

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

1. (Explicit user Control over Accepts) Ada 9X shall provide explicit user control over the mechanisms that determine which candidate is selected for a rendezvous, when more than one choice is possible. The choices shall include which candidate is selected 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. The choice mechanism must be notationally compact, and should be able to be implemented efficiently, without requiring polling of entry queues by the server at the time of the select.

!Issue out-of-scope (2) desirable,severe impl,upward compat,mostly consistent

2. Ada 9X shall provide alternate means of Inter-Process Communication, such as simultaneous entry call and wait-for-accept.

[Rationale] It is acknowledged that there is a class of problems which Ada83 is forced to solve by the addition of intermediate tasks or "helper" tasks. RI-0003 Asynchronous Communications and RI-2001 Service Order give good presentations of the problems and requirements for solution without having to dramatically alter the semantics of the selective accept mechanism.

!reference RR-0072
!reference RR-0121
!reference RR-0498
!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.
!reference Sweden Ada 9X Requirements Workshop April 23-26 1990

!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 modeled by a static priority model. It can be solved by providing more mechanisms at the task synchronization 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 multi-cpu 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 Ada 9X 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 multitasking 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 nonstandard 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.

It has been noted by some writers that the Ada Inter-Process Communication paradigm is incomplete in that the model lacks effective controls to provide a flexible ordering of service, when a (scarce) service is competed for by multiple requesters (tasks). When the service choice of the next requestor is to be made, the selection criteria may be based upon some global importance, or priority, or it may be based upon the state of the service provider, and known only to that provider.

In the situation where the selection criteria is determinable excluding the state of the service provider, then a predetermined queueing order or selection order is possible. In many real-time systems, the criteria needed is priority. In more general systems, or in system with multiple processors or complicated algorithms, other criteria, such as one or more parameters of the call, are required. In either case, the caller's importance can be evaluated at call time, and the caller queued in descending order of importance. Indexing methods are known such that the cost of the queue insert is O(logn) or less, where n is the number of requesters queued on a service. It is this this specific problem area that this RI is attempting to address.

In the situation where the selection of the next requestor depends upon the internal state of the acceptor at the time of the accept, the pre-indexing method described above is insufficient. The choice of which requester to choose depends upon evaluations of the service provider at the time. Two methods of choice are apparent, function evaluation of each requestor's parameters (including priority), or accepting/requeueing of inappropriate candidates. The function evaluation method will be of order O(n) for each selection in general. Accept/Requeue methods should be of order O(1) or O(logn), as long as the server can predict future states. (A disk server using the elevator algorithm is an example. Requesters cannot know whether to be placed in the up or down queue because they have no knowledge of the current location or direction of service. Since the server knows, it can requeue requests into the proper queue in the right order to maximize throughput.)

RI-2001 addresses the issues of out-of-order service where only the server can determine the next candidate for selection.

It is apparent that Ada 9X has to solve both requirements, the need to efficiently order requesters based upon preconditions which permit appropriately ordered queues or selection criteria to be built, and the need to control selection for rendezvous from the server, based upon state information determined at the time of the select. ??? Other requests, such as the ability to make an entry call while suspended in a select for accept, can be addressed by asynchronous communication

!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-0498 Make selective wait symmetrical wrt accept stmts. and entry calls

This RR asks that a task be able to make an entry call a select alternative in a Select for Accept structure.

%reference RR-0737 Allow preference control for entries in a select statement

This RR requests that Ada 9X 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 multi (&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

%reference Sweden Ada 9X Requirements Workshop April 23-26 1990

The feelings of the Real Time Group at the workshop are contained in the rebuttal section.

!rebuttal

From the Real Time Group at the Soderfors, Sweden Workshop

The subgroup recommends restricting the scope of this RI, in an effort to limit special "bells and whistles" per Conclusion 4. The requirement should be reworded as follows: "!Issue revision (1) important, moderate impl, mod compat, consistent 1. Ada 9X shall define which candidate is selected for a rendezvous, when there is a select statement with callers on more than one open accept." The objective of this restricted requirement is that a programmer should be able to predict which alternative will be given preference, and exert some control, without noticeable implementation overhead. For example, the lexical order of the accept alternative could be used for specifying the preference of selection. Bill Taylor provided an illustration of a minor syntax change to the select statement to support the lexical order preference:

select accept entry-1; or else accept entry-2; or else accept entry-3; end select;

RI-7001

!topic Time issues
!number RI-7001
!version 2.1

!Issue presentation

[1 - Specification of Synchronization Point] Ada 9X shall define the term "Synchronization Point", clarify what happens at a "Synchronization Point", and provide an itemized list of synchronization points.

!Issue presentation (2) compelling

2. (Resume execution after delay) Ada 9X shall require that the completion of a delay statement cause the enclosing task to become immediately eligible for execution according to its corresponding priority.

!Issue implementation (3) compelling

3. Implementation dependent timing and inter-process communication properties of the completion of a delay shall be defined in the Ada 9X standard.

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

4. Ada 9X shall provide a method of defining program-specific behaviour of all clocks and timers used by a run-time executive to support an Ada 9X program.

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

5. (Permit implementations with no clock) Ada 9X shall make provision for implementations with no hardware support for clocks or timers.

[Rationale] The need in some communities to target Ada programs to "interesting" hardware is significant. This does not mean that a general-purpose language such as Ada should be distorted for these architectures. AI-0325 gives relief in Ada83 in specifying that an implementation does not have to use or implement certain language features when the underlying implementation makes it unachievable. It is anticipated that the statement of this AI will be subsumed into Ada 9X, and that this provides the relief necessary.

!Issue revision (6) important,moderate impl,upward compat,consistent

6. The standard shall provide an interface for an application to set or manipulate the calendar clock. As a minimum, such an interface should provide discrete and incremental adjustment of the clock, advancing or retrograde, (to support Daylight Saving Time, Leap Year Leap Seconds, Time Zone adjustment, and clock synchronization), time before AD 1900, and controlled clock update facilities.

!Issue revision (7) compelling,severe impl,upward compat,mostly consistent

7. (Improve Clock Specifications) An attempt shall be made in Ada 9X to provide a more complete and general specification of time for delay, calendar, and for cpu timing.

!Issue revision (8) important

8. (Consistency in Conditional/Timed Entry Calls) Ada 9X shall require that execution behaviour of a conditional entry call be consistent with timed entry calls. This consistency shall remove semantic ambiguities associated with remote entry calls and nonpositive delay expressions.

!reference AI-00032
!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
!reference RI-7030
!reference
Third International Workshop on Real Time Ada Issues 26-29 June 1989, Farmington, PA.
!reference
Sweden Ada 9X Requirements Workshop, April 23-26, 1990

!problem

Ada83 has an inconsistent model and implementation of time. The following paragraphs highlight the problems encountered by users.

The Ada model of time is of a single time keeping kernel which provides a uniform view of time to all parts of the program, and manages tasks suspended on delay statements or delay alternatives. The specification of this time, however uses three different underlying types with potentially three different representations. The three are Calendar.Time'Base, Standard.duration, and System.tick. On top of this, there may be a different implementation tick rate for a calendar clock, and a different implementation tick rate for timers which provide time to the delay timer. Distributed programs may even have different timers for different parts of the program. Ada83 does not recognize these differences, and does not make specific provision for them.

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. This often requires complete access through the Run-Time Executive which can be prohibitively expensive to many projects.

Ada tasks cannot always get the behaviour they require when attempting to perform time-oriented functions such as delay, make timed entry calls, or make timed accept statements. The problems encountered here are as follows.

1. Implementations in Ada83 are free to awaken a task suspended on a "delay" at an arbitrary late time after the delay has expired. This unboundedness in the RM is unacceptable to real-time users.
2. The expiration of a delay statement is not necessarily a synchronization point. Some compilers reschedule tasks after a delay has expired, while some do not. This allowed non-standard behaviour encourages non-portability of code between implementations.
3. Tasks making a conditional entry call, or a null-length entry call have to actually test the condition at the acceptor's site, which may take considerable time. Tasks making "short" timed entry calls can abandon the entry before knowing the ability of the acceptor to rendezvous. This causes a discontinuity in which it may be faster to make a "short" entry call than a conditional one.
4. Tasks wishing to do a delay until a fixed calendar time cannot do so reliably. This problem is addressed in RI-7030.

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 term Synchronization Point is used in the Reference Manual without being defined. The manual does not clearly specify what a Synchronization Point is, what an Ada program does at such a point, and does not catalogue a minimum set of such points. Ada 9X must rectify this situation to improve uniformity of implementation and understanding of users.

There are more than one kind of Synchronization Point. Synchronization Points for data are usually treated differently that scheduling Synchronization Points. Ada 9X may identify other classes of Synchronization Point which apply to Ada Programs in general. Ada 9X must document allowed program behaviour by the class of Synchronization Point.

Implementation dependent Synchronization Points may occur because of remote procedure calls, IO, or unique hardware which affects the synchronization of program elements. Such implementation-dependent behaviour must also be required to be appropriately documented by an implementation.

[RI-7001.2]

AI-00032 dictates the preemptive scheduling of tasks which become ready for execution, say after the completion of a delay. The RM does not dictate that a delay should be a synchronization point where the task priorities are identical. It is the opinion of the RT that the completion of a delay statement or a delay alternative constitutes a synchronization point for the program, and that the task whose delay expired will be scheduled for execution in the shortest time possible within the constraints of Ada's priority model and scheduling rules. Ada 9X should document program behaviour in all cases of delay expiration.

[RI-7001.3]

There is a need in time critical environments, for a language construct which guarantees behaviour after a delay has expired. In distributed environments the inter-process communication protocols and timings may affect this behaviour. This puts a requirement on the language standard to tighten its allowed behaviour for real time systems, and for systems, especially distributed ones, to document their protocols for modeling and analysis.

[RI-7001.4]

The concept of time provided by the run-time executive may be different than the concept of time needed by the application program, or provided by the underlying hardware/OS. Examples of such differences are different values of tick rate than system.tick or duration'small, or programs which need simulation time instead of real time. Application programmers need the ability to unify the time concepts provided by the Ada executive, and that required by the application program.

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.

In order to assist compiler writers and implementers who need access to low-level time interfaces, Ada 9X must provide mechanisms to integrate application-level clock device handlers to the Ada executive. This may be accomplished by defining an interface package between the run-time executive and the lower level (OS or hardware). Such an interface could provide details about the number of time services available, their resolution, addresses and interrupt locations, etc. If such a package was specified and provided by all implementations, programmers could use the implementation-provided bodies, or could write their own bodies and integrate them into the program/executive without invalidating the RTE (as usually happens in current implementations).

[RI-7001.6]

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 Ada 9X 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-7001.7]

Ada 9X 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.

One of the needs stated by users is for "continuous" time, i.e. time as measured in cpu cycles, not by a delay timer. Certain operations are calculated in the amount of "cpu time" taken. Ada 9X should investigate the feasibility of including this notion in its time concepts, as stated in the Third International Workshop on Real Time Ada Issues.

[RI-7001.8]

The situation which exists today, where conditional entry calls on a (distributed) system may take indefinitely longer than timed entry calls, occurs because the delay time is measured locally, whereas the conditional call is determined at the remote site. Ada 9X should investigate the ramifications of changing the underlying model with a view to providing more consistent behaviour.

!appendix

%reference AI-00032 Preemptive Scheduling is Required

%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 dependent 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 to 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. Ada 9X must account for this and provide mechanisms to let the application developer manage such a system.

%reference

Sweden Ada 9X Requirements Workshop, April 23-26, 1990 Real Time group Better-defined behaviour upon delay completion seen as essential to Ada 9X. Application control of all clocks seen as essential, and a language issue, not a secondary-standard. Calendar Clock specifications seen as essential. Essential to rationalize and unify the *multiple notions of time in* Ada.

%rebuttal for Essential Rating The RT feels that sufficient alternate ways exist to describe and manipulate a calendar clock that an essential rating cannot be justified.

!rebuttal

!topic Interrupts
!number RI-7020
!version 2.1

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

1. (Correct Interrupt Setup) Ada 9X shall provide a mechanism for manipulating hardware and delivering hardware interrupts into a program in a manner which accommodates a wide diversity of computer architectures. These mechanisms shall be flexible enough to associate and disassociate a chosen interrupt to a selected entry of a specified task object at execution time. For efficiency reasons, Ada may place restrictions on the program constructs which are available in interrupt-handling code.

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

2. (Correct Interrupt Handling) Ada 9X shall provide mechanisms for handling interrupts by an Ada program which provide an abstraction of the interrupt handling compatible with the Ada execution/tasking model, and which can be efficiently implemented. At a minimum, these mechanisms shall:

   1. Permit the device handling code to mask its triggering interrupt while it is manipulating hardware which may cause the interrupt to occur;
   2. Permit the device handling code to mask its triggering interrupt while it is not ready to accept or handle the interrupt;
   3. Process interrupts without using the resources of the software scheduler, until interacting with Ada tasks.
   4. Permit the interrupt-handling code to effectively unmask the interrupt and continue processing, but ready to receive another interrupt in the maskable code section.

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

3. (Non-blocking Inter-Process Communication) Ada 9X shall permit a task to pass parameters to another visible task, or to unblock another visible task, without rendezvousing with that task. in any way.

!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
!reference Sweden Ada 9X Requirements Workshop April 23-26 1990

!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:

1. Interrupts are specifiable only in task declaration blocks, meaning that arrays of tasks, etc. to manage hardware is virtually unachievable.
2. 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.
3. 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.
4. 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".
5. 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.
6. 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.
7. Tasks cannot explicitly schedule other other tasks without doing a rendezvous with them.
8. 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 serious. In order to solve them, implementations are resorting to implementation-dependent 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)

Ada 9X 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. 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.

Possible program constructs which may be made unavailable in interrupt-level code are entry calls, entry accepts of non-interrupt entries, task creation or termination, or access to non-local or non-global objects,

[7020.2]

The mechanisms for handling interrupts, setting up hardware which will cause interrupts, and interacting with the rest of the program must be compatible with hardware systems in use, and with the overall Ada programming model.

a) and b) - A device handler or interrupt handler cannot tolerate interrupts occurring at uncontrolled times. Such handlers must be able to prevent interrupts from occurring, even while manipulating the hardware which may trigger such an interrupt. To do this, the ability to mask the interrupt is needed. Such masking cannot be restricted to a task ACCEPT block, as is done in Ada83.

c) The efficiency and latency requirements of interrupts dictate that no unnecessary interactions occur while interrupts are being processed.

d) Most software systems have a concept of a "forked processing" mode in handling interrupts. This may be thought of as the dispatch of another Ada task, executing with the interrupt unmasked while the interrupt handler waits for another interrupt, except that such dispatching should not involve the complete rescheduling of tasks in the program.

[7020.3]

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.

!appendix

It is the opinion of this team that the Ada tasking paradigm is appropriate for including interrupts in the language. The problem of excessive overhead resulting from Ada83's attempts to permit all tasking constructs in tasks designed to handle interrupts can be corrected, without a major paradigm shift. It is also felt that the accept statement fielding interrupts is a reasonable model once some of the Ada83 difficulties are fixed. This team would not to see the problem complicated by the addition of an additional structure with new interactions to write interrupt routines in Ada 9X.

%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 re-enable 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.

%reference Sweden Ada 9X Requirements Workshop April 23-26 1990

The real-time group felt that the the model for an interrupt handler should be different from a task or a procedure. The main concern with tasks as the vehicle for interrupt handling was that the software scheduler should have no input into the scheduling of these handlers, and that interrupts are less controllable, and more asynchronous than the current model permits.

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. Ada 9X must consider this case, and provide appropriate mechanisms that such interactions are properly handled.

!rebuttal

!topic Distributed systems
!number RI-2101
!version 2.1

!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".

!note-to-DRs

All of the "Distributed Systems" requirements are rolled into this one RI as was advised at the January DR Meeting. In fact, the first 13 items support the single-program paradigm where as the last two support the multiprogramming paradigm. The last two are reworked from the old RI-2005 and included here according to the January DR Meeting. There is sentiment among the RT that the two issues should be "re-separated".

!note-to-DRs

The RT has given a great deal of consideration to the rating of these requirements items. However, DR advice on the proper ratings is being sought as well as advice on what's in and what's out of scope.

!Issue revision (1) compelling,severe impl,moderate compat,inconsistent

1. An attempt shall be made in the design of Ada 9X 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:

   a. The exact semantics (behaviour and timing, including failure modes) of timed and conditional entry calls and of exception propagation are not well defined in Ada83 for a single program distributed across a distributed environment.

   b. The exact semantics and available recovery from hardware failure is not well-defined in Ada83 for a single program distributed across a distributed environment;

   c. 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.

   d. 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. Ada 9X 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. Ada 9X 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. Ada 9X 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.]

!external-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 targeting a distributed or shared-memory architecture. An Ada translation system not targeting 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. Ada 9X 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 Ada 9X program partitioned for execution on a distributed or shared-memory architecture, Ada 9X 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 Ada 9X program partitioned for execution on a distributed or shared-memory architecture, Ada 9X 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, Ada 9X shall support different scheduling paradigms in different parts of the program.

!Issue out-of-scope (10) desirable,severe impl,upward compat,mostly consistent

10. For the purposes of error reporting, load balancing, and load shedding, Ada 9X 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. Ada 9X 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. Ada 9X 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 (12) desirable,severe impl,moderate compat,inconsistent

12. Ada 9X shall incorporate a model of virtual memory and physical memory protection into its execution model.

!Issue out-of-scope (13) important,severe impl,moderate compat,inconsistent

13. Ada 9X shall incorporate a model of autonomous "producer-consumer-like" intertask communications between tasks executing in different execution sites.

!external-standard (14) compelling

14. (External Interface for Main Programs) An associated standard should be established to define the means for an Ada 9X 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 Ada 9X program to change its communications interconnections dynamically.

The standard shall define the means for an Ada 9X program (the "invoking program") to cause another Ada 9X 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 Ada 9X program to signal another Ada 9X program that the latter program is requested to terminate.

!Issue revision (15) important,small impl,upward compat,consistent

15. Ada 9X 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 Ada 9X Proceeding of Distributed Ada 1989 held December, 1989 at the University of South Hampton, 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 ditributed 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

An idea that came up at the March DR meeting and not covered here is the idea of parallel elaboration.

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

1. virtual memory.
2. memory protection and security.
3. propagation of exceptions across machine boundaries.
4. specification of partition and allocation
5. 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 mono-program 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-0402M 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-0404M Direct 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 preemption 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.

RI-1060

!topic Reference to a task outside its master
!number RI-1060
!version 2.1

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

1. (No Task Reference Outside Master) It shall be an error in Ada 9X 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 AI-00867/01
!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. Indeed, in light of this problem the ARG recently approved (pending editorial review) AI-00867 which calls for accessing a task outside its master to be erroneous (this AI has gone no farther in the approval process as of 18 May 1990).

If this problem is going to be solved in Ada 9X, situations such as those shown above must not be permitted in Ada 9X. If Ada 9X 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
> !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 AI-00867/01

This is an ARG-approved-only "pathology" AI that reverses AI-00167 and says it is erroneous to access a task from outside 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

RI-2106

!topic Library unit task termination
!number RI-2106
!version 2.1

!terminology

A "library task" is one that is declared directly in a package that is a library unit.

!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 (2) desirable,small impl,bad compat,inconsistent

2. (Forced Termination of Library Tasks) Ada 9X shall be defined so that a library task is terminated whenever positioned at a terminate alternative.

!Issue out-of-scope (3) desirable,small impl,bad compat,inconsistent

3. (Forced Termination of Library Tasks with Main) Ada 9X 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 (4) desirable,severe impl,bad compat,inconsistent

4. (Changing Masters and Termination) Ada 9X 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.

!reference RR-0023
!reference RR-0215
!reference RR-0496
!reference AI-00399
!reference REFER ALSO TO RR-0370

!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-00399 Status 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

RI-2012

!topic Initialization/parameterization of types
!number RI-2012
!version 2.1

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

1. (Default Expressions) Ada 9X 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) Ada 9X 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) Ada 9X 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. Ada 9X 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 Sweden Workshop
!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. One of the working groups at the Sweden Workshop felt that 2012.3 was the most important of the items.

!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 for task objects
!number RI-7003
!version 2.1

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

1. (Support for Ada Object attributes/Rep Specs) Ada 9X shall provide mechanisms for the specification of representation clauses and storage allocation on a task object basis as well as than a task type basis. Additional features of Ada 9X, such as (possibly) task attributes must also be applicable to both task objects and task types.

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

2. (User-defined defaults for tasking) Ada 9X 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
!reference RI-3279

!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.

1. 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.
2. 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.
3. 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

RI-7030

!topic Support for periodic tasks
!number RI-7030
!version 2.1

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

1. (Provide Periodic iteration) Ada 9X shall provide a mechanism to permit a code segment to be executed periodically without drift or jitter.

!Issue out-of-scope (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
!reference 3rd International Workshop on Ada Realtime Issues

!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 jitter. 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 simplest 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

This RR asks for better support for periodic tasking. 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-pror* .

!rebuttal

RI-7040

!topic Need for selective accept mechanism
!number RI-7040
!version 2.1

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

1. (Eliminate terminate alternative) The terminate alternative shall be removed from Ada 9X.

[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) Ada 9X 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 unaccessible 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

%reference WI-0206

The language should attempt to provide a mechanism allowing a user to (optionally) indicate when elaboration of a given unit must occur.

RI-2109

!topic Library unit and subunit naming
!number RI-2109
!version 2.1

!Issue revision (1) desirable,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 simple name, at a minimum in the case where subunits with the same simple name occur in different parent units.

!Issue out-of-scope (2) desirable,moderate impl,upward compat,mostly consistent

2. (Allow separate compilation of overloaded subprograms) Ada 9X shall not restrict the names of subunits; in particular, it shall be possible to have multiple subunits of the same parent unit with the same simple name.

[Rationale] There seems to be good reason for this restriction on Ada83. Suppose multiple subunits of a parent share the same simple name and they themselves contain subunits. The separate clauses that would introduce these latter subunits would be ambiguous unless parameter and result type information were somehow added to the separate clause. Working around this problem with other language legality rules seems to add undesirable complexity to the language. The use of renaming declarations appears to be an acceptable workaround, particularly if (in the case of subunits for subprograms in the visible part of a package declaration) the renaming could be hidden from clients of the package either by improved privating facilities (RI-2500) or by allowing subprogram body definitions to be given by a renaming declaration (RI-5020).

!Issue out-of-scope (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 revision (4) desirable,small impl,upward compat,consistent

4. (Separately-compiled operator-symbol subprograms) Ada 9X shall not require that separately-compiled subprograms have identifier designators.

!Issue revision (5) desirable,moderate impl,upward compat,mostly consistent

5. (Allow separate compilation of nested units) Ada 9X shall not require that all body stubs be within the specification of a library package or the declarative part of another compilation unit.

!Issue out-of-scope (6) important,severe impl,moderate compat,inconsistent

6. (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. [However, see **!rebuttal** below.]

!reference RR-0038
!reference RR-0041
!reference RR-0073
!reference RR-0154
!reference RR-0178
!reference RR-0262
!reference RR-0402
!reference RR-0545
!reference RR-0557
!reference RR-0607
!reference WI-0212
!reference AI-00458
!reference AI-00572
!reference Sweden-Workshop

!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

RI-2109.1 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.

RI-2109.4 is difficult in the sense that (1) this restriction in Ada83 seems quite arbitrary but at the same time (2) using a renaming declaration is an easy workaround for this problem. The language would be better on the whole if the restriction were removed and hence this item is posed as a requirement.

RI-2109.5 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 a significant implementation problem. A member of the large systems working group at the Sweden workshop insisted that the implementation problems for this are essentially the same as for nested generics.

!sweden-workshop

The large systems working group felt that this issue was an important one–in fact, the consensus was that all of the ıtems were important. The ratings assigned here continue to reflect the perception of the requirements team.

Also, there was some concern that RI-2109.6 was designated out-of-scope. The **!rebuttal** deals with this issue.

!appendix

%reference RR-0038

Asks for multiple subunits of the same ancestor unit with the same simple name. Apparently also wants for multiple library units with the same simple name.

%reference RR-0073

A single flat name space in the program library is inadequate for large systems. A hierarchical structure is required to avoid name clashes.

%reference RR-0154

%reference RR-0545

Argues that a subunit should not have to be positioned at the outermost level of a compilation unit.

%reference RR-0178

Name clashes in a program library are a problem. Lets either have a multi-level program library or somehow solve the re-compilation problem (so I wouldn't need so badly to have my program broken up into many small units).

%reference RR-0262

Does not want to require a stub for a subunit arguing that the linker can create the stub automatically.

%reference RR-0402

%reference AI-00458

%reference AI-00572

Argues that subunits of the same ancestor unit should be allowed to have duplicate simple names (provided they are in different parent units).

%reference RR-0557

The rename workaround for overloaded or operator-symbol subunits isn't really very good for subpgms in a package specification because the unique-name non-operator-symbol routines are visible to the user of the package. Perhaps some form of RENAME or REPLACE in the package body would work this problem.

%reference RR-0607

Allow overloading and operator symbols for names of compilation units.

%reference RR-0041

%reference WI-0212

Would like to be able to have multiple subunits of the same compilation unit with the same simple name.

!rebuttal

This rebuttal deals with the out-of-scope designation of RI-2109.6. The real need is for the language to define more configuration management capabilities so as to preclude name clashes that are possible when several contractors work on modules that are later to be combined into a single program. Further, better visibility control should be provided so that a concept of a library unit that is visible only to selected clients can be realized. Far from being an unwarranted intrusion into extralingual concerns, Ada 9X must address the problems of configuration management for large systems.

!topic Dependencies and recompilation
!number RI-2110
!version 2.1

!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 (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. Ada 9X 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 (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 revision (3) important,small impl,upward compat,consistent

3. (Package Bodies that are No Longer Needed) Ada 9x shall require implementations to warn the user when compilation units are obsolete and no longer needed.

!Issue out-of-scope (4) desirable,severe impl,upward compat,consistent

4. (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).

!rationale

LRM 10.3(5) says:

> If a compilation unit is successfully recompiled, the compilation
> units potentially affected by this change are obsolete and
> must be recompiled unless they are no longer needed.

[2110.3] addresses the problem that arises when a compilation unit is obsolete but no longer needed. The term "no longer needed" is not properly defined by Ada83. The user's definition of "no longer needed" may not be identical to the compiler's definition. As a result, situations arise where a compilation unit is not recompiled because the compiler has determined that the unit is "no longer needed" when in fact, the unit may be needed for the correct operation of the system. In this situation, the user shall be informed that the compiler has made the decision to not recompile the unit. This information should be given to the user prior to linking and executing the program.

!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 [4].

%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 and 3].

!rebuttal

RI-3572

!topic Library Operations
!number RI-3572
!version 2.1

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

1. (library management) Ada 9X shall not preclude any implementation of the library management system. [Note: this implies the deletion of the notion of "library file" as defined in LRM 10.4(1 and 2).]

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

2. (define the library) Ada 9X shall define all the aspects of the program library that may affect the order or necessity of compilations and recompilations, and only those aspects. Such aspects include (without limitation) the effect of importing a unit in a library or of removing a unit from a library.

!external-standard (3) desirable,moderate impl,upward compat

3. (standard interface) Ada09X shall define a standard interface to the Ada program library.

!reference RR-0177
!reference RR-0226
!reference RR-0237
!reference RR-0368
!reference RR-0698
!reference WI-0209
!reference WI-0209M
!reference WI-0210
!reference WI-0210M
!reference WI-0211

!problem

Users are experiencing difficulties in integrating Ada compilers with other elements in the software development process such as a configuration management system or Integrated Project Support Environment (IPSE). This 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. This lack of a standard interface to the Ada library causes difficulty in using multi-vendor environments for various target environments.

Users express that there is a lack of support for:

1. multiple variants and versions of compilation units;
2. access control between multiple users;

3. source code identification and derivation histories;
4. change control;
5. control of visibility;
6. importing and exporting of compilation units;
7. general reporting.

!rationale

[1] is intended to loosen the current definition of a library and preclude Ada09X from introducing unneeded restrictions on the program library.

[2] is intended to provide useful information concerning compilation and recompilation with respect to the program library to the configuration management system (or Project Support Environment).

[3] Standardization of compiler library environment interfaces should be coordinated with the Ada Programming Support Environment (APSE) effort. The functions needed are those that will support configuration managment for large projects. On the down side, requiring compiler vendors to either divulge or dispense with their proprietary efforts are not likely to be widely accepted. Furthermore, the specific interfaces for individual environments are likely to to be widely varying, thus requiring implementation dependencies, *which may be difficult to standardize.*

!appendix

There is doubt that compiler vendors would support any requirements for a standard library interface to a great level of detail. Doing so would be the first step in an attempt to provide completely "open" compilation systems, and there are certainly going to be proprietary concerns with doing that.

%reference RR-0177 Standardize interface between compiler and library for configuration management.

It is usually possible to use multiple Ada program libraries in order to hold multiple versions of a unit, under different levels of access protection, but it is difficult to exploit these fully within a more general project support environment because of the following sorts of problem:

1. Use of a large number of Ada program libraries can be expensive.
2. The available information about the units is not complete (e.g. the identity of the source file from which a unit was compiled), and is obtained via interactive commands rather than procedural interfaces.
3. There are often restrictions on copying units or libraries, and therefore it can be difficult or impossible to make a copy of a unit in a safe place.
4. What facilities there are differ from one compiler to another so much that an Ada workbench for an IPSE has to be written from scratch for each compiler, and the

provision of a useful, compiler-independent workbench (e.g. for a project using a variety of compilers for multiple targets and host testing) is almost impossible.

5. An Ada workbench is needlessly difficult to implement, and slow in use, because of the indirect way in which the operations on the program libraries are performed.

%reference RR-0226 Need better library support for configuration management.

The requirements placed on the implementation of the program library are limited to the minimum required for supporting the Configuration control of compilations in a project support environment mechanics of separate compilation with visibility of earlier compilation units. Failure either to define the program library sufficiently, or to express the requirements of separate compilation in a functional way, has resulted in compilation systems which are inadequate tools for the development of large systems, or which provide various essential facilities in a variety of ad hoc ways. Thus, recognizing that to provide no support at all for maintenance of the program library is unacceptable, and yet having no defined interface through which to work, compiler developers have been led to producing library management facilities which exhibit tight coupling between a compiler and a program library and which make integration of the program library with a proper configuration management system extremely difficult.

In consequence, the developer of a large application finds that there is a lack of standard support for:

1. multiple variants and versions of compilation units;
2. access control;
3. source code identification and derivation histories;
4. change control;
5. control of visibility;
6. importing and exporting of compilation units;
7. general reporting.

%reference RR-0237 Make separate compilation independent of a particular library model.

We cannot adequately configure large systems as the language now stands. There are no standard means of performing the kind of operations on library units generally considered desirable. These include:

1. creating a new variant or version of a compilation unit;
2. mixed language working, particularly the use of Ada units by other languages;
3. access control, visibility of units to other programmers;
4. change control and the general history of the system.

The inability to do these things arises out of a few loosely worded paragraphs in the LRM (in 10.1 and 10.4), which imply the existence of a single Ada program library, whose state is updated solely by the compiler. This can be an inconvenient foundation on which to build. The relationships between compilations in a project will be determined by the problem and the organization of work, and any automatic enforcement of a configuration control regime must come from a locally chosen PSE. Ada especially, as a language with large and diverse application, must have a separate compilation system which gives the greatest freedom possible in this area.

%reference RR-0368 Program library not sufficient for configuration management.

A specific issue of concern is related to the use of the program library to support the separate compilation rule (this includes compilation dependencies arising from a library unit's context clause.) A minor change to an existing unit can force a major recompilation, some of which may be unnecessary. The idea of separate compilation is important, and it has been introduced to "reduce compilation costs and to simplify development" (the Rationale, chapter 10) by ensuring that any change affected by this change" (LRM,10.3(5)) as obsolete. In the case of large and complex software systems; however, this is too restrictive and can have the opposite effect.

A second issue is the general deficiency of the vendor supplied means of access to the program library. Although the program library principally supports compilation rules, it should also (at least for the sake of assisting software engineering) allow the developed to use the program library to support his software development methodology. Therefore, it is recommended that access to the program library be open to any Ada program or to tool that the developer may develop as part of his APSE.

%reference RR-0698 Support multiple versions of non-portable units.

There are no constructs in the Standard that can be used to select code based on its implementation dependencies.

A portable program normally consists of a large collection of portable units and multiple versions of non-portable units.

%reference WI-0209 Don't indicate particular implementation of library management system.

%reference WI-0209M Need a minimal set of library mechanisms.

This WI asks for a set of useful library function to be implemented by all programming environments. This WI does not require a standard interface.

%reference WI-0210 Define only library aspects needed to determine compilation/recompilation order.

%reference WI-0210M Need a minimal set of STANDARD library mechanisms.

Same as WI-0209M, except that this WI request that the functions be standardized.

%reference WI-0211 Don't prohibit 3rd party tools.

!rebuttal

RI-3279

!topic Global Compilation Attributes
!number RI-3279
!version 2.1

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

1. Ada 9X shall promote the use of compile time directives that apply to all of the compilation units of a program. Whenever possible, these directives should be separated from the source code to reduce recompilation cascading effects.

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

2. (Provide program-level defaults) Ada 9X shall improve the availability of library-level or program-wide defaults. As a minimum, the following items must be specifiable on a library-wide or program-wide basis:

1. Task STORAGE_SIZE
2. Task default priority
3. size of INTEGER
4. size and representation of FLOAT
5. size and representation of CHARACTER
6. representation order of storage units and octets within computer words
7. The storage ordering of multidimensional arrays (row or column major)
8. Default allocation for unconstrained arrays
9. default settings for predefined or implementation-defined pragmas

Ada 9X shall specify which defaults may be overidden within compilation units.

!Issue out-of-scope (3) desirable

3. Ada 9x shall provide a method to declare constants that have no value in the source code, but are bound to values before execution.

[Rationale] This is better handled in RI-0111.

!reference RR-0065
!reference RR-0171
!reference RR-0283
!reference RR-0653
!reference RR-0698
!reference RI-0103 (multi-dimensional arrays)
!reference RI-2034 (interoperability on data)
!reference RI-3986 (portability)
!reference RI-0111 (placement of objects)

!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.

When an Ada program is built, there is certain information which is required by the compiler or follow-on tools which cannot be specified readily in Ada83. An example is task storage_size. The ability to place a program on a target may depend on having task storage_size smaller than the implementation provides as a default. This forces the developer to define every task as a task type with an explicit STORAGE_SIZE clause. He may also have to rearrange code because tasks so specified cannot be in a later declarative region. If the developer does not have access to the source of some modules, he may be stymied in his attempts to make the needed adjustments.

Similarly, a developer often develops common modules which are incorporated into multiple programs, but which have slight differences in those programs. Differences in executable code can be handled with conditional statements which depend on constants defined at build time. This cannot be done in the specification portion of a program in Ada83. An example is building an interface module where the hardware being exercised may have slight differences, but the basic algorithms are the same. The user wants to define a record which matches the hardware registers on the configuration being targeted. In Ada83 he has to edit the definition section to create the correct configuration, then likely edit every place where the registers are accessed. Current workarounds use textual preprocessors to selectively include the needed code fragments. This eliminates some of the manual editing, but almost never removes it entirely. It also means an extra pass over the code with a tool for the sole reason of providing "ifdef" capability.

The lack of support for program building causes developers to spend excessive time reworking programs to get the increased functionality or portability required. It is also playing a role in restricting the development of portable software and software tools.

!rationale

[1] Information controlling the compilation process should be specified external to the source program as much as is possible. This decoupling would provide a better split of the concerns of each aspect of the software development process. In order to codify existing practice, it might be beneficial to perform a survey of existing command line arguments that control the compilation process, in a manner similar to the way in which the survey of implementation dependent pragmas and attributes was performed.

[2] Part of the software engineering paradigm is eliminating redundancy and reducing complexity so that code is easier to understand and to maintain. The ability to specify default behaviour allows the specification of individual behaviour to be placed in the

affected module. The inability to specify default behaviour forces the developer to use the special notation throughout large segments of the program. This is unproductive, time-consuming and error-prone as some specifications may be made incorrectly, or missed, and there is no compiler assistance to detect these incorrect specifications. By widening the scope of Ada's default specifications, (possibly similar to pragma system_name), better control of a program's environment can be achieved without forcing developers to examine every module in their program.

!appendix

%reference RR-0065 Differentiate between compilation and post-compilation information.

This RR wants to be able to separate the material that is used to guide the compilation process (particularly pragmas and length clauses) from the source code, in order to allow it to be maintained separately.

%reference RR-0171 Separate program build-info from source

This RR points out that a lot of information presently placed in program source files is compiler or target-dependent. Source code which must be moved from compiler to compiler or to different targets increases the complexity of source control and configuration management. This RR would like to see such information placed in source files separate from the program source files.

%reference RR-0283 Need convenient way to set global compilation parameters

This RR is interested in establishing a context for a compilation that allows the global specification of pragmas and context clauses. Too much compilation material is included in the source code, which must be maintained under separate configuration control, for matters that are not really source code issues.

%reference RR-0653 Need to declare constants whose value is supplied after linking.

This RR states a need to be able to declare constants, but to defer the value to link or execution time.

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

This RR notes that portable programs consist of some portable units plus many versions of non-portable units. It asks for mechanisms to assist in the identification and selective build of portable systems, such as conditional compiles tied to SYSTEM.SYSTEM_NAME.

!rebuttal

RI-2111

!topic Compilation containment
!number RI-2111
!version 2.1

!Issue revision (1) compelling,severe impl,upward compat,inconsistent

1. (Compilation Containment) Ada 9X shall define a mechanism by which a new program unit may be constructed by effectively adding package and subprogram declarations to another existing unit. Ada 9X 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.

!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-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-0684 Don'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

RI-2112

!topic Context clause restrictions
!number RI-2112
!version 2.1

!Issue out-of-scope (1) desirable,small impl,upward compat,consistent

1. (Eliminate Name Restrictions in Context Clauses) Ada 9X 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 warrant a language change.

!Issue out-of-scope (2) desirable,small impl,upward compat,consistent

2. (Eliminate Need for Body) Ada 9X 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 warrant a language change.

!problem

There are two problems that have been pointed out with context clauses. The first is that one may have to repeat the with-clause from a parent unit in order to provide a use-clause or pragma ELABORATE for the named unit. Since the with-clause of the parent unit still applies, one may feel that it should not need to be repeated.

The second issue is that if a package is named in a pragma ELABORATE then it must have a body. However, it is possible that the body for a package is no longer needed because of some program change during maintenance. If so, the code for the clients of that package should not have to change.

!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

5.10 Generics

The RIs in this section address specific aspects of the Ada "generic" construct. The issues concern present limits on how generics may be parameterized.

RI-1012

!topic Generalize parameterization allowed for generic units
!number RI-1012
!version 2.1

!Issue revision (1) important

1. (Generalize Generic Parameterization) The allowed parameterization of generics shall be generalized in Ada 9X to the maximum extent possible within the constraints of (1) not damaging the fabric of the language or making it significantly more complex and (2) not imposing undo hardships on implementations. In particular, for any kind of declaration in Ada 9X, 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 RI-2002
!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 entity types (as defined in RI-2002), 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 Ada 9X. For example, potential changes in Ada 9X 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 Ada 9X. 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).

!sweden-workshop

The general feeling at the workshop was that caution was required by the mapping team in satisfying this requirement.

Participants felt this RI should be combined with other RIs that deal with program composition, including object-oriented programming, renaming, and import-for-export. This advice has not yet been followed.

!appendix

%reference RR-0424 Allow pkg instantiation to control names of visible decls

Ada 9X 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

Ada 9X shall allow a program unit to be parameterizable wrt task types.

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

Ada 9X 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

Ada 9X 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

Ada 9X shall allow code to be written that is parameterizable wrt packages and exceptions.

%reference RR-0722 Need generic formal record types

Ada 9X 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

Ada 9X shall allow code to be written that is parameterizable wrt exceptions.

%reference WI-0213 Allow all program units to be generic formal parameters

Ada 9X 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

Ada 9X shall allow code to be written that is parameterizable wrt task entries.

%reference AI-00452 Need generic record types

Ada 9X 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

Ada 9X shall allow code to be written that is parameterizable wrt named numbers, exceptions, subtypes, derived types, record types, packages, generic subprograms, generic packages.

!rebuttal

RI-1013

!topic Contract model for generics
!number RI-1013
!version 2.1

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

1. The legality of an Ada 9X 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).

!sweden-workshop

The implementation impact of the requirement was lowered to "small" based on discussion at the workshop.

!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

RI-1014

!topic Support for generic code sharing
!number RI-1014
!version 2.1

!Issue revision (1) important

1. To the fullest extent possible given other constraints on the language revision, Ada 9X should allow an efficient implementation of generics via code-sharing.

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

2. Ada 9X 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 Ada 9X 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:

1. 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. 2)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.

2. 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.)

3. 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

RI-1016

!topic True separate compilation of spec and body for generics
!number RI-1016
!version 2.1

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

1. Ada 9X 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, possibly leading to a cascade of necessary additional recompilations. 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.

!sweden-workshop

The problem statement has been revised based on comments made at the workshop. It was also suggested that this RI be merged into a more general RI on the recompilation problem. This has not been done as of yet, although it may well happen in the future.

!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

5.11 Exceptions

The single RI in this section deals with RRs that point to problems with the way that Ada exceptions fit into the rest of the language and the need for a way to get more details in an exception handler.

RI-1070

!topic Exceptions
!number RI-1070
!version 2.1

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

1. (More Information Available in Handler) Ada 9X 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 Ada 9X. 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) Ada 9X 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) Ada 9X shall clarify the semantics of pragma SUPPRESS when the name given is that of a subprogram, type, or subtype.

!Issue implementation (5) desirable,small impl,upward compat,mostly consistent

5. (Force Compliance with Pragma SUPPRESS) Ada 9X implementations shall be required to "support" pragma SUPPRESS. By "support" here, it is meant that:

a. any additional machine instructions that the compiler normally uses to implement the checking shall not be emitted, and;

b. 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 Ada 9X.

[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.

!sweden-workshop

A common idea expressed at the workshop was that if the programmer was not using any of the new exception functionality, he should not suffer any performance penalties.

Item [5] above was previously stated as a requirement. Several groups in Sweden felt it should be out-of-scope. Since this seems to be an implementation/performance issue and was felt to be important in Destin, it has been changed to an implementation item.

Some at the workshop felt strongly that exceptions as first-class values were a mistake. Specifically, proposals along these lines are generally complicated and introduce subtle interactions; the need does not strong enough to justify making language changes along these lines.

One group made the suggestion that [1] above should be generalized from obtaining information in a handler to obtaining information in a "failure situation". Presumably, this would include situations where an error status parameter was returned from a subprogram. However, it is not clear what Ada83 lacks along these lines.

One group thought generic formal exceptions were a bad idea because of the complexity they introduced, another felt them to be "very important" in their discussion of RI-1012.

!appendix

%reference 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.

%reference 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.

%reference 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".

%reference 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.

%reference RR-0621 Generalize facilities avble for exceptions for power/flexibility

Getting image of handler, get the current exception, pass them as parameters, limited assignment of exceptions, etc.

%reference RR-0752 Make various improvements to exception handling capabilities

Same basically as RR-621.

%reference RR-0085 Allow exception name or image to be exported

%reference RR-0145 Provide a way to get exception name from "when others" handlers

%reference RR-0219 Provide way to get name of raised exception, inc. out-of-scope

%reference RR-0403 Need to be able to get the name of the current exception

%reference RR-0772 Need to be able to get exception name in a handler

%reference AI-00595 Name of the "current exception"

There is a need to get the name of the currently raised exception in exception handling code.

%reference 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.

%reference 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.

%reference 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.

%reference 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.

%reference 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.

%reference RR-0399 Break up overly broad pre-def exceptions, e.g., CONSTRAINT_ERROR

CONSTRAINT_ERROR is way too broad; need PARAMETER_ERROR, etc.

%reference 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.

%reference WI-0301 Provide ways to distinguish between various forms of STORAGE_ERROR

Need LOCAL_STORAGE_ERROR, ALLOCATOR_ERROR, etc.

%reference RR-0005 Exceptions are inconsistent, generic code-sharing accordingly hard

Current definition of exceptions is a pain-in-the-neck for generic code sharing

%reference 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.

%reference RR-0444 Allow specification of where a declared exception can be raised

Need private exceptions raisable only in this package but handleable outside.

%reference 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.

%reference RR-0646 Allow exceptions to be parameterized with params read in handler

The parameterized exception idea.

%reference 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.

%reference 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.

%reference AI-00299 Pragma Suppress and Subprogram Names

A received ramification on the meaning of pragma SUPPRESS wrt subprogram names.

%reference 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.

%reference 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

5.12 Input/Output

The two RI's in this section address perceived user needs for additional input-output [I/O] capabilities or improvements in existing facilities.

RI-3600

!topic The I/O abstraction
!number RI-3600
!version 2.1

[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:

1. 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.]
2. Provide a mechanism for saving and restoring the standard input and output.
3. Remove the restriction on independence of standard input and output to allow better control over formatting.
4. Allowing per-instantiation DEFAULT values to be formal parameters (e.g., FORE, AFT, WIDTH).
5. Allow real number output in non-decimal bases.
6. Provide a function to determine line length.
7. Allow data of mode IN for SEND_CONTROL.

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

2. (I/O is Multi-tasking) Ada 9X shall disallow task starvation to occur in instances where one task executes an I/O instruction.

!Issue external 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 an external 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 available 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

RI-3700

!topic I/O and external environments
!number RI-3700
!version 2.1

[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 external 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 external 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:

   1. indexed sequential I/O.

   2. 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 (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 (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 (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 standard Ada

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

5.13 Control Structures

The single RI in this section addresses specific issues relating to Ada control structures, or statements.

RI-5250

!topic Control structures
!number RI-5250
!version 2.1

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

1. (Generalize control structures) An attempt shall be made to avoid unnecessary restrictions and inconsistencies in control structures in Ada 9X.

!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 Ada 9X.

!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

APPENDICES

A. INDEX OF RIS IN NUMERICAL ORDER

RI #	Page	Topic
RI-0001	371	Terminating tasks before synchronization points
RI-0002	375	Safe shutdown of tasks
RI-0003	361	Asynchronous transfer of control
RI-0005	367	Asynchronous communication
RI-0009	379	Notification of abnormal tasks
RI-0100	397	Internal representation of enumerations
RI-0101	399	Length clause (SIZE)
RI-0102	401	Storage conservation
RI-0103	403	Multi-dimensional array storage
RI-0104	407	Definition of address clause
RI-0105	415	Storage order in representation clauses (Big endian / Little endian)
RI-0106	419	Unsigned integers and bit-vectors
RI-0107	435	Memory reclamation
RI-0109	425	Miscellaneous representation clause issues
RI-0110	163	Non-contiguous subtypes of discrete types
RI-0111	409	Placement of objects
RI-0115	431	Storage error/recovery and exceptions
RI-0116	441	Miscellaneous memory management issues
RI-0200	187	Anonymous Type Definitions
RI-0201	381	Fine-grain parallelism
RI-0901	229	Fixed point range end points
RI-0902	233	Implementation freedom in fixed point
RI-0904	237	Restrictions in fixed-point
RI-0905	241	Simplify floating-point model
RI-0906	245	Provide for compatibility with IEEE floating-point
RI-1000	251	Reading non-input parameters
RI-1010	177	Definition of static expressions (not wrt generic formals)
RI-1011	181	Staticness inside generic units with respect to generic formals
RI-1012	483	Generalize parameterization allowed for generic units
RI-1013	487	Contract model for generics
RI-1014	491	Support for generic code sharing
RI-1016	497	True separate compilation of spec and body for generics
RI-1021	259	Pragma INLINE flexibility
RI-1022	255	User control over parameter passing mechanism
RI-1023	261	Miscellaneous subprogram issues
RI-1030	61	Early messages from implementations for apparent problems
RI-1031	69	Erroneous execution and incorrect order dependence (IOD)
RI-1032	73	Strengthening subprogram specifications
RI-1033	77	Fault tolerance
RI-1034	81	Miscellaneous reliability issues

RI #	Page	Topic
RI-1040	355	Shared variables
RI-1050	109	Complexity/surprises in overloading
RI-1060	337	Reference to a task outside its master
RI-1070	501	Exceptions
RI-1090	145	Syntax of the language
RI-2001	391	Controlling the service order of client tasks
RI-2002	27	Inheritance, polymorphism
RI-2003	141	Character set issues
RI-2011	165	General references
RI-2012	343	Initialization/parameterization of types
RI-2021	39	Call-back style; subprograms as parameters and objects
RI-2022	97	Finalization
RI-2023	265	Visibility of operators in a WITHed package
RI-2024	269	Visibility and safety
RI-2025	271	Less restrictions on overloading; enhanced resolution
RI-2029	45	Support for control engineering programming
RI-2032	217	Record discriminants
RI-2033	223	Enhancing the model of record types
RI-2034	169	Typing at a program interface/interoperability on data
RI-2035	175	Equality - redefinition
RI-2101	325	Distributed systems
RI-2102	33	User-defined types vs predefined types
RI-2105	279	Own variables in packages
RI-2106	341	Library unit task termination
RI-2107	383	Virtual entries; accept statements in subprograms
RI-2108	387	Miscellaneous issues with CSP a la Ada
RI-2109	457	Library unit and subunit naming
RI-2110	463	Dependencies and recompilation
RI-2111	477	Compilation containment
RI-2112	479	Context clause restrictions
RI-2500	275	Improved model of privating
RI-3000	119	Information hiding
RI-3279	473	Global Compilation Attributes
RI-3514	51	Interfaces to other languages.
RI-3572	467	Library Operations
RI-3600	511	The I/O abstraction
RI-3657	55	Machine Code Insertions (MCIs)
RI-3700	517	I/O and external environments
RI-3749	193	Dimensional mathcmatics.
RI-3986	133	Portability
RI-4017	445	Control over elaboration
RI-5010	153	Type renaming and re-exporting
RI-5020	127	Subprogram body implementation

RI #	Page	Topic
RI-5061	101	Constraints in renaming declarations
RI-5062	105	Renaming declarations
RI-5080	85	Compile-time optimization
RI-5100	93	Special case rules
RI-5110	113	Miscellaneous consistency and complexity issues
RI-5120	197	Array aggregates
RI-5130	201	Array slices and cross-sections
RI-5141	205	Variable-length strings
RI-5142	209	Miscellaneous string issues
RI-5150	211	Implicit subtype conversion
RI-5160	213	Miscellaneous array issues
RI-5170	227	Uniformity of integers
RI-5190	157	Multiple related derived types
RI-5200	159	Miscellaneous issues concerning types
RI-5210	185	Null ranges
RI-5220	287	Mutual exclusion
RI-5230	289	Low-level tasking facilities
RI-5241	283	Complexity and efficiency of tasking model
RI-5250	525	Control structures
RI-7001	311	Time issues
RI-7003	347	Representation clauses for task objects
RI-7005	293	Priorities and Ada's scheduling paradigm
RI-7007	301	Support for alternate run-time paradigms
RI-7010	305	Flexibility in selection for rendezvous
RI-7020	319	Interrupts
RI-7030	351	Support for periodic tasks
RI-7040	353	Need for selective accept mechanism

B. INDEX OF RIS BY PAGE NUMBER

Page	RI #	Topic
27	RI-2002	Inheritance, polymorphism
33	RI-2102	User-defined types vs predefined types
39	RI-2021	Call-back style; subprograms as parameters and objects
45	RI-2029	Support for control engineering programming
51	RI-3514	Interfaces to other languages.
55	RI-3657	Machine Code Insertions (MCIs)
61	RI-1030	Early messages from implementations for apparent problems
69	RI-1031	Erroneous execution and incorrect order dependence (IOD)
73	RI-1032	Strengthening subprogram specifications
77	RI-1033	Fault tolerance
81	RI-1034	Miscellaneous reliability issues
85	RI-5080	Compile-time optimization
93	RI-5100	Special case rules
97	RI-2022	Finalization
101	RI-5061	Constraints in renaming declarations
105	RI-5062	Renaming declarations
109	RI-1050	Complexity/surprises in overloading
113	RI-5110	Miscellaneous consistency and complexity issues
119	RI-3000	Information hiding
127	RI-5020	Subprogram body implementation
133	RI-3986	Portability
141	RI-2003	Character set issues
145	RI-1090	Syntax of the language
153	RI-5010	Type renaming and re-exporting
157	RI-5190	Multiple related derived types
159	RI-5200	Miscellaneous issues concerning types
163	RI-0110	Non-contiguous subtypes of discrete types
165	RI-2011	General references
169	RI-2034	Typing at a program interface/interoperability on data
175	RI-2035	Equality - redefinition
177	RI-1010	Definition of static expressions (not wrt generic formals)
181	RI-1011	Staticness inside generic units with respect to generic formals
185	RI-5210	Null ranges
187	RI-0200	Anonymous Type Definitions
193	RI-3749	Dimensional mathematics.
197	RI-5120	Array aggregates
201	RI-5130	Array slices and cross-sections
205	RI-5141	Variable-length strings
209	RI-5142	Miscellaneous string issues
211	RI-5150	Implicit subtype conversion

Page	RI #	Topic
213	RI-5160	Miscellaneous array issues
217	RI-2032	Record discriminants
223	RI-2033	Enhancing the model of record types
227	RI-5170	Uniformity of integers
229	RI-0901	Fixed point range end points
233	RI-0902	Implementation freedom in fixed point
237	RI-0904	Restrictions in fixed-point
241	RI-0905	Simplify floating-point model
245	RI-0906	Provide for compatibility with IEEE floating-point
251	RI-1000	Reading non-input parameters
255	RI-1022	User control over parameter passing mechanism
259	RI-1021	Pragma INLINE flexibility
261	RI-1023	Miscellaneous subprogram issues
265	RI-2023	Visibility of operators in a WITHed package
269	RI-2024	Visibility and safety
271	RI-2025	Less restrictions on overloading; enhanced resolution
275	RI-2500	Improved model of privating
279	RI-2105	Own variables in packages
283	RI-5241	Complexity and efficiency of tasking model
287	RI-5220	Mutual exclusion
289	RI-5230	Low-level tasking facilities
293	RI-7005	Priorities and Ada's scheduling paradigm
301	RI-7007	Support for alternate run-time paradigms
305	RI-7010	Flexibility in selection for rendezvous
311	RI-7001	Time issues
319	RI-7020	Interrupts
325	RI-2101	Distributed systems
337	RI-1060	Reference to a task outside its master
341	RI-2106	Library unit task termination
343	RI-2012	Initialization/parameterization of types
347	RI-7003	Representation clauses for task objects
351	RI-7030	Support for periodic tasks
353	RI-7040	Need for selective accept mechanism
355	RI-1040	Shared variables
361	RI-0003	Asynchronous transfer of control
367	RI-0005	Asynchronous communication
371	RI-0001	Terminating tasks before synchronization points
375	RI-0002	Safe shutdown of tasks
379	RI-0009	Notification of abnormal tasks
381	RI-0201	Fine-grain parallelism
383	RI-2107	Virtual entries; accept statements in subprograms
387	RI-2108	Miscellaneous issues with CSP a la Ada
391	RI-2001	Controlling the service order of client tasks

C. PRESENTATION AIS

AI #	Date	LRM Chapter#
AI-00052/01	83-10-10	02.04.01(02)
AI-00053/01	89-03-08	02.08(02)
AI-00054/00	83-10-10	03.05(10)
AI-00055/00	83-10-10	03.05.05(19)
AI-00056/00	83-10-10	03.05.06(06)
AI-00057/00	83-10-10	03.06(01)
AI-00058/00	83-10-10	03.06(11)
AI-00059/00	83-10-10	03.07(09)
AI-00060/00	83-10-10	03.07.03(06)
AI-00062/00	83-10-10	04.03(09)
AI-00063/00	83-10-10	04.03.02(13)
AI-00064/00	83-10-10	04.05.02(11)
AI-00065/00	83-10-10	04.09(12)
AI-00066/00	83-10-10	04.09(13)
AI-00067/00	83-10-10	05.02(09)
AI-00068/00	83-10-10	05.04(05)
AI-00069/00	83-10-10	05.04(09)
AI-00070/00	83-10-10	05.09(06)
AI-00071/00	83-10-10	06(02)
AI-00072/01	88-12-28	06.03(07)
AI-00073/00	83-10-10	06.04.01(04)
AI-00074/00	83-10-10	06.06(04)
AI-00075/00	83-10-10	07.04.04(09)
AI-00076/00	83-10-10	07.06(04)
AI-00077/00	83-10-10	08.05(02)
AI-00078/00	83-10-10	09.01(03)
AI-00079/00	83-10-10	09.01(04)
AI-00080/00	83-10-10	09.04(05)
AI-00081/00	83-10-10	09.04(06)
AI-00082/00	83-10-10	09.04(16)
AI-00083/00	83-10-10	09.06(09)
AI-00084/00	83-10-10	09.07.01(08)
AI-00085/00	83-10-10	09.11(01)
AI-00086/00	83-10-10	11.02(07)
AI-00087/00	83-10-10	11.05(04)
AI-00088/00	83-10-10	12.01.01(04)
AI-00089/00	83-10-10	13.03(03)
AI-00090/00	83-10-10	13.07(00)
AI-00091/00	83-10-10	14.02.04(17)
AI-00092/00	83-10-10	14.07(02)

AI #	Date	LRM Chapter#
AI-00093/00	83-10-10	Appendix C (21)
AI-00094/01	84-01-17	Index (with clause)
AI-00095/00	83-10-10	Postscript (00)
AI-00096/00	83-11-07	03.01(05)
AI-00097/00	83-11-07	03.01(06)
AI-00098/00	83-11-07	03.05(03)
AI-00100/00	83-11-07	03.08(06)
AI-00102/00	83-11-07	04.06(15)
AI-00104/01	88-12-20	05.01(03)
AI-00105/00	83-11-07	08.04(04)
AI-00107/01	88-12-20	10.05(01)
AI-00108/00	83-11-07	Index (rounding)
AI-00109/00	83-11-07	03.06.01(02)
AI-00110/00	83-11-07	08.03(22)
AI-00111/00	83-11-07	08.03(16)
AI-00112/00	83-11-07	10.06(02)
AI-00121/00	83-11-07	03.07.02(05)
AI-00122/00	83-11-07	08.05(08)
AI-00123/01	88-12-20	09.07.01(10)
AI-00124/00	83-11-07	11.05(06)
AI-00125/00	83-11-07	13.05(10)
AI-00127/00	83-11-07	Appendix D (evaluation)
AI-00130/00	83-11-07	08.01(04)
AI-00160/00	84-01-13	03.03.03(00)
AI-00164/00	84-01-13	04.06(12)
AI-00168/00	84-01-17	03.03.03(11)
AI-00182/00	84-01-25	08.04(10)
AI-00191/00	84-03-13	07.01(05)
AI-00204/00	84-03-13	Index(FREE)
AI-00206/00	84-03-13	03.05.08(00)
AI-00212/00	84-03-13	07.04.02(11)
AI-00213/00	84-03-13	07.06(04)
AI-00220/00	84-03-13	07.04.01(03)
AI-00227/01	88-12-20	10.04(01)
AI-00230/00	84-03-13	Index (entity)
AI-00259/00	84-05-26	Index (Static)
AI-00269/00	84-08-05	Index (type conversion)
AI-00272/00	84-08-27	03.07.04(04)
AI-00277/00	84-08-27	09.11(02)
AI-00281/00	84-08-27	Index (new)
AI-00342/00	85-06-18	03.05.09(07)
AI-00347/00	85-06-18	05.05(09)
AI-00351/00	85-06-18	07.04.01(02)

AI #	Date	LRM Chapter#
AI-00363/00	85-07-07	Index (00)
AI-00381/00	85-08-22	11.06(06)
AI-00395/00	85-10-06	13.05(08)
AI-00403/00	85-12-03	03.05.05(01)
AI-00413/01	89-03-08	01.04(01)
AI-00416/00	86-03-25	06.01(02)
AI-00439/00	86-07-10	08.05(16)
AI-00610/00	88-12-21	01(01)
AI-00611/00	88-12-21	01.05(04)
AI-00612/00	88-12-21	01.06(03)
AI-00614/00	88-12-21	01.06(07)
AI-00615/00	88-12-21	02.04.02(04)
AI-00616/00	88-12-21	Appendix F (01)
AI-00617/00	88-12-21	03.01(06)
AI-00618/01	89-06-22	03.02.01(06)
AI-00619/01	89-06-22	03.02.01(15)
AI-00620/00	88-12-21	03.02.01(16)
AI-00621/00	88-12-21	03.02.01(19)
AI-00622/00	88-12-21	03.02.02(03)
AI-00623/00	88-12-21	03.03.01(03)
AI-00625/00	88-12-21	03.04(11)
AI-00627/00	88-12-21	03.04(14)
AI-00628/00	88-12-21	03.04(20)
AI-00629/00	88-12-21	14.03.07(14)
AI-00630/00	88-12-21	03.05.02(01)
AI-00631/00	88-12-21	03.05.05(03)
AI-00632/00	88-12-21	03.05.05(13)
AI-00633/00	88-12-21	03.05.05(13)
AI-00634/00	88-12-21	03.05.05(18)
AI-00635/00	88-12-21	03.05.06(04)
AI-00636/00	88-12-21	03.05.06(07)
AI-00637/00	88-12-21	03.05.07(17)
AI-00638/00	88-12-21	03.05.08(19)
AI-00639/00	88-12-21	03.05.09(01)
AI-00641/00	88-12-21	03.05.09(14)
AI-00642/00	88-12-21	03.05.09(16)
AI-00643/00	88-12-21	03.05.10(03)
AI-00644/00	88-12-21	03.06(05)
AI-00645/00	88-12-21	03.06.01(02)
AI-00646/00	88-12-21	03.06.01(02)
AI-00647/00	88-12-21	03.06.01(02)
AI-00648/00	88-12-21	14.03.08(18)
AI-00649/00	88-12-21	03.06.01(04)

AI #	Date	LRM Chapter#
AI-00650/00	88-12-21	03.06.01(04)
AI-00651/00	88-12-21	03.06.01(06)
AI-00652/00	88-12-21	03.07(05)
AI-00653/00	88-12-21	03.07(07)
AI-00654/02	89-01-25	03.07.01(03)
AI-00655/00	88-12-21	03.07.01(04)
AI-00656/00	88-12-21	03.07.01(11)
AI-00657/00	88-12-21	03.07.02(05)
AI-00658/00	88-12-21	03.07.02(05)
AI-00659/00	88-12-21	14.03.09(11)
AI-00660/00	88-12-21	03.07.02(05)
AI-00661/00	88-12-21	03.07.02(05)
AI-00662/00	88-12-21	03.07.02(08)
AI-00663/00	88-12-21	03.07.02(08)
AI-00664/00	88-12-21	03.07.02(08)
AI-00665/00	88-12-21	03.07.02(08)
AI-00666/00	88-12-21	03.07.02(10)
AI-00667/00	88-12-21	03.07.02(10)
AI-00668/00	88-12-21	03.07.02(15)
AI-00669/00	88-12-21	03.07.04(03)
AI-00670/00	88-12-21	03.08(03)
AI-00671/00	88-12-21	03.08(04)
AI-00672/00	88-12-21	03.08(06)
AI-00673/00	88-12-21	03.08(06)
AI-00674/00	88-12-21	03.08.01(04)
AI-00675/00	88-12-21	14.03.06(13)
AI-00677/00	88-12-21	04.01.04(02)
AI-00678/00	88-12-21	04.02(04)
AI-00679/00	88-12-21	04.03(02)
AI-00680/00	88-12-21	04.03(07)
AI-00682/00	88-12-21	04.03.02(03)
AI-00683/00	88-12-21	04.03.02(04)
AI-00684/00	88-12-21	04.03.02(05)
AI-00685/00	88-12-21	04.03.02(06)
AI-00686/00	88-12-21	04.03.02(08)
AI-00687/00	88-12-21	04.03.02(11)
AI-00688/00	88-12-21	04.04(04)
AI-00689/00	88-12-21	04.05(06)
AI-00690/00	88-12-21	04.05.05(10)
AI-00691/00	88-12-21	04.05.07(08)
AI-00692/00	88-12-21	04.06(03)
AI-00693/00	88-12-21	04.06(13)
AI-00694/00	88-12-21	04.06(15)

AI #	Date	LRM Chapter#
AI-00695/00	88-12-21	04.06(21)
AI-00696/00	88-12-21	04.08(05)
AI-00697/00	88-12-21	04.08(05)
AI-00698/00	88-12-21	04.08(06)
AI-00699/00	88-12-21	04.08(14)
AI-00700/00	88-12-21	04.09(06)
AI-00701/00	88-12-21	04.09(11)
AI-00702/00	88-12-21	04.09(11)
AI-00703/00	88-12-21	04.09(12)
AI-00704/00	88-12-21	04.09(12)
AI-00705/00	88-12-21	13.10.01(06)
AI-00706/00	88-12-21	05.04(02)
AI-00707/00	88-12-21	05.04(03)
AI-00708/00	88-12-21	05.04(03)
AI-00709/00	88-12-21	05.05(06)
AI-00710/00	88-12-21	06.02(08)
AI-00711/00	88-12-21	06.04.01(09)
AI-00712/00	88-12-21	06.06(03)
AI-00713/00	88-12-21	07.04.01(02)
AI-00714/00	88-12-21	07.04.01(03)
AI-00715/00	88-12-21	07.04.01(04)
AI-00716/00	88-12-21	07.04.01(04)
AI-00717/00	88-12-21	07.04.01(05)
AI-00718/00	88-12-21	07.04.02(01)
AI-00719/00	88-12-21	07.04.02(06)
AI-00720/00	88-12-21	07.04.02(06)
AI-00721/00	88-12-21	07.04.02(11)
AI-00722/00	88-12-21	07.04.04(01)
AI-00723/00	88-12-21	07.04.04(08)
AI-00724/00	88-12-21	08(01)
AI-00725/00	88-12-21	08.03(01)
AI-00726/00	88-12-21	08.03(01)
AI-00727/00	88-12-21	08.03(01)
AI-00728/00	88-12-21	08.03(02)
AI-00729/00	88-12-21	08.03(05)
AI-00730/00	88-12-21	08.03(06)
AI-00731/00	88-12-21	08.03(15)
AI-00732/00	88-12-21	08.03(15)
AI-00733/00	88-12-21	08.03(17)
AI-00734/00	88-12-21	08.03(17)
AI-00735/00	88-12-21	Appendix B (03)
AI-00736/00	88-12-21	08.05(04)
AI-00737/00	88-12-21	08.05(05)

AI #	Date	LRM Chapter#
AI-00739/00	88-12-21	08.05(07)
AI-00740/00	88-12-21	08.05(07)
AI-00741/00	88-12-21	Appendix B (05)
AI-00742/00	88-12-21	08.05(10)
AI-00743/00	88-12-21	08.07(02)
AI-00744/00	88-12-21	08.07(03)
AI-00745/00	88-12-21	08.07(07)
AI-00746/00	88-12-21	08.07(13)
AI-00747/00	88-12-21	08.07(13)
AI-00748/00	88-12-21	08.07(19)
AI-00749/00	88-12-21	09.01(01)
AI-00750/00	88-12-21	09.04(01)
AI-00751/00	88-12-21	09.04(05)
AI-00752/00	88-12-21	09.04(13)
AI-00753/00	88-12-21	09.05(05)
AI-00754/02	89-03-08	09.06(05)
AI-00755/00	88-12-21	09.07.01(06)
AI-00756/00	88-12-21	09.07.01(10)
AI-00757/00	88-12-21	14.03.05(07)
AI-00758/00	88-12-21	09.08(05)
AI-00759/00	88-12-21	Appendix B (05)
AI-00760/00	88-12-21	10.01(03)
AI-00761/00	88-12-21	10.01(03)
AI-00762/00	88-12-21	10.01(06)
AI-00763/00	88-12-21	10.01.01(04)
AI-00764/00	88-12-21	10.02(03)
AI-00765/00	88-12-21	10.03(01)
AI-00766/00	88-12-21	10.03(03)
AI-00767/00	88-12-21	10.03(04)
AI-00768/00	88-12-21	10.03(16)
AI-00769/00	88-12-21	14.03.05(08)
AI-00770/00	88-12-21	10.05(01)
AI-00771/00	88-12-21	14.03.05(12)
AI-00772/02	89-03-08	10.05(02)
AI-00773/00	88-12-21	10.05(02)
AI-00774/00	88-12-21	10.05(04)
AI-00775/00	88-12-21	Appendix D (00)
AI-00776/00	88-12-21	11.01(06)
AI-00777/00	88-12-21	11.01(07)
AI-00778/00	88-12-21	11.07(10)
AI-00779/00	88-12-21	12.01.01(03)
AI-00780/00	88-12-21	12.01.01(04)
AI-00781/00	88-12-21	12.03(04)

AI #	Date	LRM Chapter#
AI-00782/00	88-12-21	12.03(14)
AI-00783/00	88-12-21	12.03(15)
AI-00784/00	88-12-21	12.03(17)
AI-00785/00	88-12-21	12.03.02(04)
AI-00786/00	88-12-21	12.03.06(03)
AI-00787/00	88-12-21	12.04(06)
AI-00788/00	88-12-21	13.01(03)
AI-00789/00	88-12-21	13.01(05)
AI-00790/00	88-12-21	13.01(06)
AI-00791/00	88-12-21	13.01(06)
AI-00792/00	88-12-21	13.01(07)
AI-00793/00	88-12-21	13.01(09)
AI-00794/00	88-12-21	13.01(10)
AI-00795/00	88-12-21	13.02(03)
AI-00796/00	88-12-21	13.02(12)
AI-00797/00	88-12-21	13.03(02)
AI-00798/02	89-03-08	13.03(04)
AI-00799/00	88-12-21	13.04(07)
AI-00800/00	88-12-21	13.05(10)
AI-00801/00	88-12-21	13.07.01(02)
AI-00802/00	88-12-21	13.07.01(03)
AI-00803/00	88-12-21	13.07.03(04)
AI-00804/00	88-12-21	13.07.03(10)
AI-00806/00	88-12-21	13.09(03)
AI-00807/00	88-12-21	14.06(03)
AI-00808/00	88-12-21	03.05.04(12)
AI-00810/00	89-03-07	03.03.01(04)
AI-00814/00	89-04-14	03.05.07(09)
AI-00815/00	89-04-14	03.05.09(18)
AI-00816/00	89-04-14	03.06(04)
AI-00817/00	89-04-14	03.09(06)
AI-00818/00	89-04-14	04.03.02(09)
AI-00819/00	89-04-14	04.03.02(11)
AI-00820/00	89-04-14	04.05.07(07)
AI-00823/00	89-04-14	11.01(04)
AI-00824/00	89-04-14	13.01(03)
AI-00826/00	89-06-14	14.01(05)
AI-00829/00	89-06-15	Postscript (00)
AI-00835/00	89-09-08	04.03.02(03)