Ada 9X Project Report

Ada 9X Requirements

December 1990

Office of the Under Secretary of Defense for Acquisition

Washington, D.C. 20301

Approved for public release; distribution is unlimited.

Copyright © 1990 by Carnegie Mellon University.
Reprinting permitted if accompanied by this statement.

Table of Contents

Foreword . . . . . **v**

1 Introduction **1**
1.1 Background 1
1.2 Scope of the Revision 1
1.3 Structure and Terminology 4
1.4 General Guidelines for the Mapping/Revision Team 5

2 General Requirements **7**
2.1 Presentation Requirements 7
Requirement R2.1-A(1) — Incorporate Approved Commentaries 7
Requirement R2.1-A(2) — Review Other Presentation Suggestions 7
Requirement R2.1-B(1) — Maintain Format of Existing Standard 8
Requirement R2.1-C(1) — Machine-Readable Version of the Standard . . 8
2.2 Efficiency, Simplicity, and Consistency 8
Requirement R2.2-A(1) — Reduce Deterrents to Efficiency 8
Requirement R2.2-B(1) — Understandability 8
Requirement R2.2-C(1) — Minimize Special-Case Restrictions 8
2.3 Error Detection 9
Study Topic S2.3-A(1) — Improve Early Detection of Errors 9
Requirement R2.3-A(2) — Limit Consequences of Erroneous Executions 10
2.4 Controlling Implementation-Dependent Choices 11
Requirement R2.4-A(1) — Minimize Implementation Dependences 11

3 Requirements for International Users **13**
3.1 International Character Sets 13
Requirement R3.1-A(1) — Base Character Set 13
Requirement R3.1-A(2) — Extended Graphic Literals 14
Requirement R3.1-A(3) — Extended Character Set Support 14
Requirement R3.1-A(4) — Extended Comment Syntax 14
Study Topic S3.1-A(5) — Extended Identifier Syntax 14

4 Support for Programming Paradigms **17**
4.1 Subprograms 17
Study Topic S4.1-A(1) — Subprograms as Objects 17
Requirement R4.1-B(1) — Passing Subprograms as Parameters 18
Requirement R4.1-B(2) — Pragma INTERFACE 18
4.2 Storage Management 19
Requirement R4.2-A(1) — Allocation and Reclamation of Storage 19
Study Topic S4.2-A(2) — Preservation of Abstraction 19
4.3 Composition of Program Units 20
Study Topic S4.3-A(1) — Reducing the Need for Recompilation 20
Study Topic S4.3-B(1) — Programming by Specialization/Extension . . . 21

Table of Contents

Study Topic S4.3-C(1) — Enhanced Library Support 22
4.4 Generics 22
Study Topic S4.4-A(1) — Generic Formal Parameters 22
Requirement R4.4-B(1) — Dependence of Instantiations on Bodies 23
Study Topic S4.4-B(2) — Tighten the ‘‘Contract Model’’ 23
Requirement R4.4-C(1) — Generic Code Sharing 23
4.5 Exceptions 24
Requirement R4.5-A(1) — Accessing an Exception Name 24
4.6 Input/Output 24
Requirement R4.6-A(1) — Interactive TEXT_IO 24
Requirement R4.6-B(1) — Additional Input/Output Functions 25

5 Real-Time Requirements 27
5.1 Time 27
Requirement R5.1-A(1) — Elapsed Time Measurement 27
Requirement R5.1-B(1) — Precise Periodic Execution 27
Requirement R5.1-C(1) — Detection of Missed Deadlines 28
5.2 Task Scheduling 28
Requirement R5.2-A(1) — Alternative Scheduling Algorithms 28
Requirement R5.2-A(2) — Common Real-Time Paradigms 29
5.3 Asynchronous Control of Execution 30
Requirement R5.3-A(1) — Asynchronous Transfer of Control 30
5.4 Asynchronous Communication 31
Requirement R5.4-A(1) — Non-Blocking Communication 31
Study Topic S5.4-B(1) — Asynchronous Multicast 32

6 Requirements for System Programming 33
6.1 Unsigned Integer Operations 33
Requirement R6.1-A(1) — Unsigned Integer Operations 33
6.2 Data Interoperability 33
Requirement R6.2-A(1) — Data Interoperability 33
6.3 Interrupts 33
Requirement R6.3-A(1) — Interrupt Servicing 33
Requirement R6.3-A(2) — Interrupt Binding 34
6.4 Dynamic References to Global Objects 34
Requirement R6.4-A(1) — Access Values Designating Global Objects . . 34
Study Topic S6.4-B(1) — Low-Level Pointer Operations 35

7 Requirements for Parallel Processing 37
7.1 Shared Memory 37
Requirement R7.1-A(1) — Control of Shared Memory 37
7.2 Massively Parallel Architectures 38
Study Topic S7.2-A(1) — Managing Large Numbers of Tasks 38
7.3 Vector Architectures 38

Study Topic S7.3-A(1) — Statement Level Parallelism 38
7.4 Configuration of Parallel Programs 39
Study Topic S7.4-A(1) — Configuration of Parallel Programs 39

8 Requirements for Distributed Processing 41
8.1 Distribution of Ada Applications 41
Requirement R8.1-A(1) — Facilitating Software Distribution 42
8.2 Dynamic Reconfiguration of Distributed Systems 42
Requirement R8.2-A(1) — Dynamic Reconfiguration 42

9 Requirements for Safety-Critical and Trusted Applications 45
9.1 Predictability of Execution 45
Study Topic S9.1-A(1) — Determining Implementation Choices 45
Requirement R9.1-A(2) — Ensuring Canonical Application of Operations 46
9.2 Certifiability 46
Requirement R9.2-A(1) — Generating Easily Checked Code 46
9.3 Enforcement of Safety-Critical Programming Practices 47
Requirement R9.3-A(1) — Allow Additional Compile-Time Restrictions 47

10 Requirements for Information Systems 49
10.1 Handling Currency Quantities for Information Systems 49
Requirement R10.1-A(1) — Decimal-Based Types 49
Study Topic S10.1-A(2) — Specification of Decimal Representation 49
10.2 Compatibility with Other Character Sets 50
Study Topic S10.2-A(1) — Alternate Character Set Support 50
10.3 Interfacing with Data Base Systems 50
Study Topic S10.3-A(1) — Interfacing with Data Base Systems 50
10.4 Common Functions 51
Study Topic S10.4-A(1) — Varying-Length String Package 51
Study Topic S10.4-A(2) — String Manipulation Functions 51

11 Requirements for Scientific and Mathematical Applications 53
11.1 Floating Point 53
Requirement R11.1-A(1) — Standard Mathematics Packages 53
Study Topic S11.1-B(1) — Floating Point Facilities 53
11.2 Representation of Arrays 53
Study Topic S11.2-A(1) — Array Representation 54

References 55

Appendix A: Efficiency, Simplicity, and Consistency Issues 57
A.1 Efficiency of Executed Code 57
A.1.1 Access to a Task Outside its Master 57
A.1.2 Null Ranges 57
A.2 Understandability 57

Table of Contents

A.2.1 Elaboration Order 57
A.2.2 Later Declarative Items 57
A.2.3 Visibility of Literals and Operations 57
A.2.4 Obsolete Optional Bodies 58
A.2.5 OTHERS Clause in Aggregates 58
A.3 Generality 58
A.3.1 IMAGE and VALUE for Real Types 58
A.3.2 Exception Handlers in Accept Statements 58
A.3.3 RANGE Attribute for Scalar Types 58
A.3.4 Permit ‘‘raise ... when <condition>’’ 58
A.3.5 STORAGE_SIZE for Task Objects 58
A.3.6 Explicit Type Conversions in Static Expressions 58
A.3.7 Use of a Subprogram Name in its Specification 59
A.3.8 Default Names for Generic Formal Parameters 59
A.3.9 Ability to Redefine ‘‘=’’ 59
A.3.10 Reading Out Parameters 59
A.3.11 Implicit Subtype Conversions 59
A.3.12 Negative Literals in Loops 59
A.3.13 Naming Syntactic Items 59
A.4 Usability of Ada 60
A.4.1 Completion of Subprogram Declarations 60
A.4.2 Completing Incomplete and Private Types by Subtype Declarations . . 60

Foreword

The revision of ANSI/MIL-STD-1815A, the Ada programming language, is being conducted by the Ada 9X Project Office located at the Air Force Armament Laboratory at Eglin AFB, Florida. The Ada 9X Project is sponsored by the Ada Joint Program Office in the Office of the Under Secretary of Defense for Acquisition. The Project was initiated in October 1988 with an invitation to the public to submit revision requests. Over 750 revision requests were submitted world-wide between October 1988 and October 1989. In addition, several international workshops and meetings were organized by the Ada 9X Project Office for the purpose of further refining and prioritizing user needs in order that they might better be addressed in the revision effort.

While many individuals have greatly contributed to the requirements process, one person deserves special recognition. John Goodenough, at the Software Engineering Institute (SEI), has made enormous contributions to the Ada 9X Project since its inception. John not only orchestrated the Ada 9X Complex Issues Studies but also led the team that successfully completed the Ada 9X Requirements. That Team consisted of John, Arthur Evans, Jr., also at the SEI, Benjamin Brosgol from Alsys, Inc., and Robert Dewar from New York University. They are to be congratulated for a difficult job superbly accomplished.

I would also like to thank participants of the initial phase of the requirements process for their dedication in collecting and analyzing the revision requests from the Ada community. This team, based at the Institute for Defense Analyses (IDA), consisted of Cy Ardoin (IDA), Paul Baker (CTA), Douglas Dunlop (Intermetrics), Audrey Hook (IDA), Joseph Linn (IDA), Catherine McDonald (IDA), Reginald Meeson, Jr. (IDA), Steven Michell (Prior Data Sciences: Canada), and Karl Nyberg (Grebyn Corp).

I would also like to thank the Ada 9X Project technical experts, the Distinguished Reviewers, who, under the very capable chairmanship of Erhard Ploedereder of Tartan Laboratories, performed many intense reviews of the material leading to the publication of this document. The Distinguished Reviewers are: G. Booch (Rational), B. Brosgol (Alsys), N. Cohen (IBM), R. Dewar (NYU), G. Dismukes (TeleSoft), D. Fisher (Incremental Systems), A. Gargaro (Computer Sciences), M. Gerhardt (ESL), J. Goodenough (SEI), E. Guerrieri (Boston University), T. Harrison (SPC), S. Heilbrunner (IABG: Germany), M. Henne (Sverdrup), P. Hilfinger (UC/Berkeley), Jean Ichbiah (Alsys: DR emeritus), M. Kamrad II (UNISYS), R. Landwehr (CCI: Germany), C. Lester (Portsmouth Polytechnic: UK), R. Leavitt (Prior Data Sciences: Canada), L. Månsson (Telelogic AB: Sweden), R. Mathis (Contel), M. Mills (US Air Force), C. Mitchell (DEC), E. Ploedereder (Tartan), D. Pogge (US Navy), K. Power (Boeing), O. Roubine (Consultant: France), T. Taft (Intermetrics), W. Taylor (Ferranti: UK), E. Vasilescu (GNV), and T. Wheeler (US Army).

Defining the requirements for the revision of Ada 83 has been a challenging task. The overall goal has been to balance the necessary changes for Ada's continued growth in the 1990s with the need for stability in terms of preserving the integrity of existing Ada software and tools. The requirements as stated in this document are intended to enhance Ada's usability and strengthen its position in the marketplace.

I wish to express my sincerest thanks to all of the Ada 9X Project Team members and to those of you in the Ada community who have so generously given of your time to attend workshops and special meetings, submit revision requests, and review and comment on interim reports.

Publication of this document is a major milestone but we still have a long way to go. I look forward to your continued participation and support in the remaining phases of the Project.

Christine M. Anderson
Ada 9X Project Manager

1 Introduction

The purpose of this document is to specify needs that are considered to be the appropriate focus of the Ada 9X revision effort and to identify revision requirements that are to be addressed by the Mapping/Revision Team.

1.1 Background

This document is the result of a collective effort to identify the most appropriate kinds of revisions that should be made to the Ada Programming Language (ANSI/MIL-STD-1815A-1983 and ISO 8652-1987). The effort began in January 1988 when the Ada Board was asked to prepare a recommendation to the Ada Joint Program Office (AJPO) on the most appropriate standardization process to use in developing a revised Ada standard, known as Ada 9X. The recommendation [4] was delivered in September 1988 to Virginia Castor, Director, Ada Joint Program Office, who subsequently established the Ada 9X Project for conducting the revision of the Ada Standards. Christine M. Anderson was appointed Project Manager in October 1988.

The Ada 9X effort consists of several phases. The output of the Requirements Definition Phase is this document, which specifies the revision needs to be addressed. The next phase is performed by the Mapping/Revision Team, which is responsible for defining changes to the standard that meet these requirements. The purpose of the Requirements Definition Phase is to identify and prioritize the revision requirements to be addressed by the Mapping/Revision Team.

The requirements definition effort and the mapping/revision efforts are both constrained by the overall objective of the Ada 9X effort [1]:

> Revise ANSI/MIL-STD-1815A-1983 to reflect current essential requirements with minimum negative impact and maximum positive impact to the Ada community.

A revision was undertaken because, according to the Ada Board, ‘‘some omissions, limitations, and minor errors have been discovered … since the ANSI standard was approved. Although a major revision does not appear to be necessary to correct these problems, some revision is warranted.’’ [4] The Ada Board further recommended that the scope of the revision be defined in a document that captures the high-level requirements to be satisfied by the revision. The Board then went on to say, ‘‘The objective of the proposed revision should be to select only those changes that improve the usability of the language while minimizing the disruptive effects of changing the standard.’’

1.2 Scope of the Revision

Several key issues affecting the scope of the Ada 9X revision are evident in the revision requests received from the general public [3] and in the results of various workshops and special meetings dealing with the revision (e.g., [2]):

- Dissimilar Needs for Change: Users from different application domains want different changes, and changes that are important to some domains are either unnecessary or detrimental to others. For example, support for decimal arithmetic is important if Ada is to satisfy the needs of information systems developers, but real-time and embedded systems programmers typically have no need for such numeric represen-

tations and would prefer that compiler vendors devote time and effort to supporting other specialized capabilities (often ones that are not critical to information systems).

- Expanding Ada's Domain of Use: There are some Ada users who feel strongly that if Ada 9X is not enhanced in certain areas (e.g., by providing support for distributed systems, object-oriented programming, or information systems), it will fail to attract enough new users to ensure the continued improvement and availability of compilers. In this event, neither the Department of Defense (DoD) nor Ada's non-DoD users would obtain the benefits of a strong competitive market.
- Extent of Change: Many in the Ada community are worried that if Ada 9X incorporates too many changes, Ada 9X compilers will be less efficient, less reliable, and more costly than existing compilers (at least initially). Indeed, the mere prospect of major changes may lead people to delay or reduce their commitment to use Ada because of fears about the length of time it will take for the revised language to stabilize.
- Unstandardized Tailoring of Implementations: Quite a few application-specific needs are being met today by providing implementation-defined (and hence, non-standard) pragmas, attributes, packages, support tools, etc. Of course, if compiler vendors cater to the needs of special classes of customers by providing specialized compiler support that is not under control of the standard, Ada programs become more vendor-dependent and the benefits of language standardization are increasingly lost. Changes that lead to more implementation uniformity would be helpful here, but increased uniformity is somewhat in conflict with the diversity of user needs and concerns about the potential scope of the revision.

We[1] believe that these concerns can best be met by providing *annexes* in the 9X standard that address specialized application needs. In particular, such annexes should be limited to the following kinds of specifications:

a) Specialized packages. Standard packages of subprograms are often an appropriate means of providing special capabilities needed in given application areas. For example, a package for supporting dynamic priorities might be provided for use by real-time applications, and a package supporting COBOL's ''picture'' format might be specified for use by information systems applications. Standardizing the interface for such capabilities in annexes should improve Ada's ability to meet specialized needs.

b) Pragmas and Attributes. In Ada 83, only a basic set of required pragmas and attributes is defined. In cases where a special pragma or attribute is the appropriate solution to a specialized need, then a relevant annex will prescribe its exact form and meaning.

c) Representation clauses. The extent to which Ada 83's representation clauses must be supported by all implementations is currently a subject of review by the Ada Rapporteur Group (the group responsible for recommending interpretations of the standard). When this review is completed, there will still be some representation clause behavior that is implementation-dependent. In Ada 9X, it is expected that annexes will impose further restrictions on representation clause support required for the application area addressed by the annex.

[1]The Requirements Team.

d) Performance specifications. Since inefficient support for certain uses of language constructs is tantamount to non-support as far as some applications are concerned, each annex is expected to address critical performance issues that cannot or need not be satisfied by all implementations.

In short, Ada 9X annexes will permit the standard to address specialized needs without requiring support by all compilers. This will improve the degree of uniformity that can be achieved as compared with the current standard because the annexes will impose restrictions on capabilities where the current standard imposes none. For example, an Information Systems annex might specify how representation clauses are to be used to support packed decimal representations and it might specify that at least 18 digits of precision are required. Such a requirement probably should not be imposed on all implementations. Consequently this requirement is not addressed at all in Ada 83 and the result is lack of uniform support for this capability even for implementations that are attempting to address the needs of Information Systems applications.

The annex concept should give greater freedom to the Mapping/Revision Team to address the needs of specialized application areas without having to be overly concerned that a specified capability is inappropriate for some community of users (since not every implementation will have to support the required capability).

A possible advantage of this approach is that it allows for incremental transition to an Ada 9X implementation that supports all the annexes — implementers can decide to concentrate initially on supporting just those annexes of greatest interest to their customer base. Of course, some implementers (e.g., those targeting the education market) may decide not to support any of the annexes, thereby allowing them to implement Ada 9X with less effort. The other advantage, of course, is that the annexes allow the specialized and critical needs of different user communities to be addressed explicitly in the standard.

The role of the annexes in compiler validation needs to be kept in mind during the revision. The Mapping/Revision Team should assume that validation policy can be developed to help ensure that the purpose of the specialized annexes is achieved. One possibility is that validation certificates would mention only those annexes that are fully supported by an implementation. Moreover, it is probably reasonable to allow an implementation to provide only partial support for additional annexes without claiming validation status for such support.[2] The details need to be worked out during the revision and depend to some extent on the nature of the specialized annexes.

Of course, the notion of annexes needs to be refined further during the revision. The Mapping/Revision Team must determine how many annexes are necessary (there should not be many) and their interrelationship (e.g., whether a specified pragma appears in more than one annex). In refining this concept, the Mapping/Revision Team should keep in mind that the main purpose of the annex notion is to allow Ada 9X to address a broader spectrum of user needs than might otherwise be possible, thereby expanding Ada's domain of use while respecting the incompatible nature of some required capabilities and the need to limit the overall extent of change.

[2]Some existing compilers might already support some of the capabilities required by a given annex.

1.3 Structure and Terminology

This document is organized to reflect groups of needs for improving the usability of Ada. Each chapter is devoted to a different group of needs. After specifying a revision need in user-oriented terms, we specify topics to be addressed by the Mapping/Revision Team. The topics are prioritized as either *Requirements* or *Study Topics* with the intent that the Mapping/Revision Team work primarily on satisfying the *Requirements*. Revisions that are not covered by either a Requirement or a Study Topic are considered outside the scope of the 9X revision effort.

It is expected that some revision needs will be satisfied by providing additional language features (syntax, semantics, attributes, or pragmas) while others may simply impose performance constraints on implementations. Having the revised standard address performance issues reflects the fact that the *way* in which a feature is supported is sometimes critical to its usability in particular applications. It is expected that performance issues of particular interest to different user communities will be addressed in separate annexes of the standard.

Generally we do not specify whether the solution to a particular requirement[3] should be given in one of the special annexes or in the main body of the standard. This decision is left to the Mapping/Revision Team because it depends on the nature of the solutions to the requirements. On the other hand, some requirements seem sufficiently specialized to merit being addressed in a special annex; we sometimes mention this in our discussion of a requirement to remind the reader that we realize the requirement could lead to solutions that are of little general interest or are inapplicable for programming some applications.

A user need or requirement may be appropriate for more than one application domain. If so, we generally state the need or requirement for only that domain considered to be the dominant source for the requirement (based on the results of the various workshops and written materials submitted during the Requirements Phase).

Not all user needs are reflected in the stated Requirements because it is not clear that all such needs can be satisfied with the requisite minimal impact on implementations. Also, for some needs it is unclear whether the benefits of making a change are worth the potential costs. Such needs are sometimes reflected in Study Topics when it was felt that it would be appropriate for the Mapping/Revision Team to give attention to the topic, but not at the cost of neglecting the more important topics stated as Requirements. Revisions addressing Study Topics must have a low negative impact.

Requests for language changes came from a variety of sources, including Revision Requests (RRs) submitted by the public [3], various Workshops and special meetings sponsored by the Ada 9X Project (e.g., [2], and Ada Commentaries [5]. A separate document, ‘‘Rationale for the Ada 9X Requirements,’’ will list all submitted RRs and will discuss how each is related to the Requirements and Study Topics presented here. A few RRs, workshop proceedings, Ada commentaries, and other sources are cited directly in this document when they provide additional explanation and background that could not be presented directly while keeping the document short. We

[3]The lack of an initial capital ‘‘R’’ means we are neutrally referring to one of the topics to be addressed by the Mapping/Revision Team without distinguishing whether it is classified as a Requirement or as a Study Topic.

also acknowledge RRs and other sources when they have been directly helpful in suggesting the wording of specific requirements or user needs. A revision request is cited as RR-dddd, which is the number of the RR that can be found in [3]. An approved Ada Commentary is cited as AI-ddddd, which is the number of the commentary as found in [5].

1.4 General Guidelines for the Mapping/Revision Team

The following guidelines provide overall guidance to the Mapping/Revision Team. The Mapping/Revision Team must make appropriate tradeoffs when the guidelines cannot be satisfied simultaneously.

G-1: **Minimum Disruption:** Changes shall minimize disruption to the Ada community while satisfying as many of the recognized needs as possible. In particular, this guideline covers the following criteria:

- **Implementation Ease:** The recommended changes shall be relatively easy to implement with existing compiler technology.
- **Execution Efficiency:** Changes to Ada shall not negatively affect object code efficiency.
- **User Need:** The extent of a change shall not be out of proportion with the degree of user need for the change.
- **Upward Compatibility:** In general, any legal Ada 83 program must also be a legal Ada 9X program and must have the same effect allowed or required by Ada 83. However, to improve the portability of Ada programs, and to satisfy other requirements, it may be necessary to eliminate some effects allowed by Ada 83 or to make some Ada 83 programs illegal. In these cases, the following guidelines shall apply:
  - No implementation-dependent behavior allowed in Ada 83 shall be excluded in Ada 9X if that behavior is actually supported by a significant number of implementations and is important to a significant number of users.
  - No Ada 83 program shall be illegal in Ada 9X unless it is relatively easy to correct the illegal program or it is expected that very few existing legal programs will be affected (since the programs are either pathological or erroneous).
  - No legal Ada 83 program shall have a different semantic effect in Ada 9X unless it is expected that extremely few existing programs will be affected (e.g., it might be acceptable to modify the optimization rules in ways that affect when the evaluation of an expression requires CONSTRAINT_ERROR or NUMERIC_ERROR to be raised).

  Any non-upward-compatible changes shall be justified by the Mapping/Revision Team and where possible, accompanied by an analysis of techniques for detecting possible incompatibilities in existing programs.

G-2: **Reliability/Safety/Maintainability:** Support for the reliability, safety, and long-term maintainability of large, complex systems shall take precedence over convenience of Ada coding. For example, this means greatly favoring statically checkable rules over rules checked at run-time, and favoring run-time checks over uncheckable rules (erroneous behavior).

1: Introduction

2 General Requirements

There are a number of revision requirements that almost go without saying, since the revision of any language standard must at least address the small defects that have been discovered during the use of the standard. In addition, the overall structure and style of the revised standard present fundamental issues that must be addressed by the requirements. These basic requirements are addressed here.

2.1 Presentation Requirements

User Need U2.1-A: Improve Wording to Reflect the Intended Meaning: A number of commentaries on the existing standard have been approved by the Ada Joint Program Office (AJPO) or by the ISO Working Group responsible for the standardization of Ada (ISO-IEC/JTC1/SC22/WG9). The commentaries serve to correct unintended implications of the wording of the current standard and to point out places where some users have had difficulty in understanding the standard's intent. To minimize different interpretations among Ada users and implementers, the wording of the standard should be revised, where necessary, to reflect these clarifications.

Requirement R2.1-A(1) — Incorporate Approved Commentaries: Ada 9X shall incorporate the approved Ada Commentaries [5] to the extent that they do not conflict with changes dictated by other requirements and to the extent that incorporation makes the wording of the standard clearer.

Discussion: WG9's Ada Rapporteur Group (ARG) has been developing a report (the *Ada Commentary Integration Document* (ACID)) that reflects the impact of the approved commentaries by suggesting revised wording for appropriate sections of the standard. ACID would be a good starting point for satisfying this requirement. Of course, some of the approved commentaries may be inconsistent with changes suggested by the Mapping/Revision Team to meet other requirements, and in this case, the commentary should not be incorporated into the revised standard. In addition, since the process for developing the commentaries was limited to maintenance changes, some of the clarifications proposed by the commentaries may not be appropriate given the latitude for change allowed by the Ada 9X revision process. In short, the approved commentaries should be reflected in the revised standard only when the intent of the commentary is consistent with the overall nature of the Ada 9X revisions.

Requirement R2.1-A(2) — Review Other Presentation Suggestions: Ada Commentaries in the ''Presentation'' class as well as the Revision Requests shall be examined for suggested wording improvements. Appropriate suggestions shall be incorporated into the revised standard.

Discussion: Commentaries in the ''Presentation'' class have not generally been formally approved by the Ada Rapporteur Group and WG9, but some of them make useful suggestions for improving the presentation of the standard. Similarly, some revision requests point out areas where the clarity of the wording can perhaps be improved.

User Need U2.1-B: Maintain Continuity with the Existing Standard: Since most users of Ada 83 are familiar with the structure and format of the existing standard, it is important that the revised standard retain that structure and format.

Requirement R2.1-B(1) — Maintain Format of Existing Standard: The Ada 9X standard shall be similar in style and structure to the existing standard. Chapter and section headings and their order shall be retained to the largest extent possible. The presentation style shall also be retained.

Discussion: The structure and style of the Ada 83 standard must be retained by Ada 9X in order to preserve continuity with Ada 83. Maintaining the same document structure and style will also help to minimize the impact of changes on programmers.

User Need U2.1-C: Derivative Use of the Standard: Compiler vendors and other organizations have found it helpful to provide annotated versions of the standard that reflect vendor implementation decisions or approved commentaries. Providing a machine-readable version of the standard would facilitate such use.

Requirement R2.1-C(1) — Machine-Readable Version of the Standard: A machine-readable version of the Ada 9X standard shall be made available using a widely supported formatting language.

Discussion: Providing a version of the standard that contains formatting commands will make it easier to incorporate its content into data bases. In addition, some implementers and other organizations have produced annotated versions of the current standard that are helpful to users. This is easier to do if the standard is made available with machine-readable formatting codes.

2.2 Efficiency, Simplicity, and Consistency

User Need U2.2-A: Minimize Compilation and Execution Costs: The language must permit efficient compilation and execution of useful combinations of language features. Appendix Section A.1 on page 57 includes some of the cases that shall be considered.

Requirement R2.2-A(1) — Reduce Deterrents to Efficiency: Combinations of language features shall not incur compilation and execution costs (in time and space) that are significantly increased primarily because of extremely unlikely interactions with other language features.

Discussion: For example, AI-00867 notes that the ability to access a task outside its master makes it difficult for an implementation to decide when it can release the task's space, especially when the task is returned as part of a composite object. Similar cases need to be identified and resolved as part of the Ada 9X revision.

User Need U2.2-B: Ease of Learning: Language rules should be easy to teach and easy to remember.

Requirement R2.2-B(1) — Understandability: Rules that have proven to be confusing or error-prone to users shall be reviewed and eliminated when possible. Appendix Section A.2 on page 57 includes some of the rules that shall be considered for revision.

User Need U2.2-C: Generality: Sensible uses and combinations of language constructs should be allowed.

Requirement R2.2-C(1) — Minimize Special-Case Restrictions: To the extent consistent with Requirement R2.2-B(1), Ada 9X shall minimize exceptions to general rules specifying how lan-

guage constructs interact. Special uses and combinations of language constructs shall be eliminated. Appendix Sections A.3 on page 58 and A.4 on page 60 list some of the existing rules that shall be considered for revision.

Discussion: Experience with Ada 83 has revealed a number of opportunities for simplifying, generalizing, or otherwise revising the language rules so as to avoid unusually large compilation or execution costs and to make the language easier to learn. The above Needs and Requirements direct the Mapping/Revision Team to reexamine the rationale for the original language rules in such areas. If the reasons are no longer applicable, appropriate corrections should be made, consistent with the general guidelines **G-1** and **G-2**.

It should be noted that the above requirements directly conflict with each other. Minimizing costs of unused capabilities is typically satisfied by having special case restrictions, thus violating User Need 2.2-B (Ease of Learning) and User Need 2.2-C (Generality). User Need 2.2-B can be satisfied by making the language very general and orthogonal, risking inefficiencies that would violate User Need 2.2-A, or by simplifying and restricting the language, which could violate User Need 2.2-C. The Mapping/Revision Team must find an appropriate balance among these conflicting requirements.

2.3 Error Detection

User Need U2.3-A: Minimize Consequences of Programmer Error: Violations of language rules should be detected by pre-execution checks whenever practical. When such checks are not practical, violations should be detected by run-time checks, so exception handlers can recover from the error, or so programs can be abandoned in a safe and predictable manner. Whenever practical, violations should not place a program in an undefined state with unpredictable consequences.

Study Topic S2.3-A(1) — Improve Early Detection of Errors: If an implementation is able to detect that certain run-time program errors are present (e.g., that execution will necessarily be erroneous or that initialization of a declared object will necessarily raise an exception), the implementation shall provide an option for rejecting the compilation (that is, not adding it to the program library). A similar option shall also be provided when a restriction associated with a pragma has been violated.

Discussion: Implementations can sometimes detect at compile-time that the execution of a program will be erroneous or will raise a predefined exception such as CONSTRAINT_ERROR or STORAGE_ERROR. Such situations, especially those that can be easily detected at compile-time, are almost always programmer errors. In general, programmers want as much help in detecting possible problems as they can get. Moreover, they want to be able to count on the nature of such help as they port applications among compilers. Ada 9X can help achieve these goals by specifying more clearly that implementations are allowed to reject compilations containing probable run-time errors of the kind specified in the requirement. This recognition of errors also applies to the use of certain pragmas, e.g., pragmas that inform an implementation of the programmer's intent to avoid the use of certain language features (e.g., see Requirement R9.3-A(1) on page 47).

Of course, implementations today are allowed to issue ‘‘warning’’ messages in these situations, and they can provide an optional mode of operation in which certain kinds of warnings cause compilations to be rejected, but this potentially useful behavior is not uniformly provided. For some applications (such as safety critical and trusted system applications), a special annex might specify that implementations must definitely detect certain kinds of these situations, requiring that a compilation be rejected if such situations are detected.

On the other hand, there are some compilations that should not be rejected even though they appear to contain these kinds of errors. For example, a program might appear to reference an uninitialized variable but the programmer may know that the variable is given a valid value when the system is loaded. In such cases, there must be some standard way for programmers to ensure that the compilation is not rejected.

This requirement is classified as a ‘‘Study Topic’’ because it is not completely clear how far the Ada 9X standard should or can go in specifying situations that *must* be detected. Such a list might be application-dependent, e.g., trusted and safety-critical systems might want to impose more severe constraints on error detection than would be considered cost-effective for other application areas. Moreover, not all such errors can be easily detected at compile-time. Requiring that all compilers detect certain errors of this kind could impose an unacceptable implementation burden on some compilers, e.g., inexpensive compilers that are intended for the education market.

Requirement R2.3-A(2) — Limit Consequences of Erroneous Executions: Run-time violations of language rules shall have narrowly defined consequences. Whenever practical, a violation shall result in the raising of an exception. When the raising of an exception is not practical, the possible consequences of the violation shall be few, confined to limited aspects of the computation state, and defined in the standard. This definition shall be in terms of those aspects of the computation state addressed elsewhere in the standard. To the extent possible, catastrophic consequences shall be excluded.

Discussion: In Ada 83 most rules are enforced either by compile-time or run-time checks. For a small number of rules, neither compile-time nor run-time checks are practical. In general, violation of such rules can have unpredictable consequences. Execution is said to become ‘‘erroneous’’ when such a rule is violated.

The consequences of erroneous execution can be devastating. For example, an arbitrary storage location can be overwritten when assigning to an array component if the array index is known to have an appropriate range but has an undefined value. In such a case, Ada 83 does not require that a range check be performed because an implementation is allowed to assume that the index has an acceptable value. Ada 9X could require that a range check be performed in such cases (except when RANGE_CHECK has been suppressed), but the net effect of such a rule might be to discourage implementations from removing truly unnecessary range checks, so the solution in this case is not obvious.

The possibility of devastating consequences is more easily eliminated in other cases. For example, execution is rendered erroneous by a subprogram call whose effect depends on the parameter-passing mechanism, but for any reasonable implementation only a small number of precisely defined effects are possible — the call proceeds normally with each parameter passed either by copy or by reference. Such a rule change would tighten the language while probably having little or no effect on implementations.

In short, the problem addressed by the requirement is a difficult one. The situations giving rise to erroneous program executions were given close attention in the original design. Nonetheless, it is important that these situations be re-examined in the Ada 9X revision.

2.4 Controlling Implementation-Dependent Choices

User Need U2.4-A: Uniformity of Compiler Behavior: Because Ada is intended to be a language that enhances the portability of software, Ada programs should behave in the same way for different implementations. In some cases, Ada 83 has deliberately permitted implementation freedom to reflect expected variations in user needs as well as in target and operating system environments. For example, the range of the subtype PRIORITY is not specified since the intention is that this range reflect both user needs and operating systems capabilities. There is a need to minimize these implementation-dependent choices, since in practice they cause difficulty in porting programs.

Requirement R2.4-A(1) — Minimize Implementation Dependences: Implementation dependences permitted in Ada 9X shall be justified by considerations of target architecture variations and different needs of applications. Ada 9X shall specify performance parameters and constraints affecting the use of critical constructs, although, since such constraints are typically application dependent, they should usually be specified in annexes.

Discussion: In some cases, Ada 83 permits implementation-dependent behavior in anticipation of operating system and target variability, but experience has shown that more uniformity can be achieved. WG9's Uniformity Rapporteur Group (URG) has been systematically investigating such variations and is in the process of producing uniformity recommendations. For example, Ada 83 specifies that the status of unclosed files at program termination is undefined, but the URG recommends that external files be closed on program termination. In some cases, it is not possible to completely eliminate variations among implementations, but their effect can be minimized. For example, the behavior of the I/O packages has proven to be more implementation dependent than seems necessary (e.g., with respect to the ability to open the same external file with different data types or the behavior of TEXT_IO when applied to terminals).

Another source of implementation dependence is in the efficiency with which certain constructs and combinations of constructs are supported by an implementation. If an application's use of time or space is critically dependent on how an implementation supports a particular construct, then application programmers should be able to determine or control the performance associated with their use of the construct. Such constraints have typically not been a part of a language standard, but given their importance in fostering the usability of Ada implementations and the portability of Ada programs, it is appropriate for Ada 9X to attempt improvements in this area.

2: General Requirements

3 Requirements for International Users

Ada is an international language. However Ada 83 does not adequately address the need for handling non-English character sets. In addition, the Ada 83 syntax is inconvenient for use by non-English speaking programmers. These issues were identified during the ISO standardization process for Ada 83 and a commitment was made by ISO-IEC/JTC1/SC22/WG9, the ISO group responsible for the Ada standard, that the character set issue would be addressed when the ISO standard was next revised. Since a goal of the Ada 9X effort is to maintain Ada as an international standard, this ISO commitment must be addressed.

3.1 International Character Sets

User Need U3.1-A: International Character Set: The character set of Ada 9X must be convenient for international use. In particular:

1. It must be convenient to write applications that deal with character sets of languages other than English.
2. Ada must be convenient to use for programmers whose native language is not English.
3. Despite differences in national character sets, there must be no country-specific variations affecting Ada's syntax and lexical rules; these rules must continue to be completely defined by the standard.

Discussion: The Ada 83 character set represents a rather small subset of the graphic symbols used in natural languages throughout the world. This fact complicates writing programs and handling character and string data in non-English-speaking communities. Enhancement of Ada's character set will help extend the world-wide use of Ada.

It follows from item 1 of the User Need that the alphabet of character and string literals must include additional characters and that there must be functionality equivalent to the present TEXT_IO package for such alphabets.

A minimum response to item 2, already reflected in commentary AI-00339, is that non-English characters must be permitted in comments; it is desirable that they also be permitted in identifiers. However, various meetings with members of the international community have indicated that the translation of reserved words is not considered to be a requirement, and indeed such a translation would be difficult to achieve in a manner that did not conflict with item 3.

Item 3 reflects the need to guard against variations in Ada rules that could make it difficult to process Ada code (such as standard packages) using compilers produced in different countries. In particular, despite the needed extension of Ada's character set, any Ada lexical element accepted by one Ada compiler must be accepted by all.

Requirement R3.1-A(1) — Base Character Set: The Ada 9X type STANDARD.CHARACTER shall be based on ISO 8859 [22], an 8-bit standard for character representation, in the same way that the Ada 83 character set is based on the ISO 646 [21], a 7-bit standard.

Discussion: This requirement introduces the potential for incompatibility with existing programs (for example, in constructs such as case statements and array aggregates that might require

choices for every value of type CHARACTER), but it has been widely recognized that an 8-bit standard is more appropriate as the base character set. ISO 8859 actually defines a number of different 8-bit character sets, and the Ada 9X design must take this into account.

It is intended that the semantics of the predefined ordering operators for STRINGs continue to be based on the ordering of the elements of the (extended) type CHARACTER, even though this ordering will not be ''lexicographic'' with respect to graphic symbols (e.g., 'ê' will not be sorted next to 'e', since the underlying character codes are not adjacent).

Requirement R3.1-A(2) — Extended Graphic Literals: Ada 9X shall permit the use of character and string literals containing non-English characters, including those from international character sets with more than 256 graphic symbols.

Requirement R3.1-A(3) — Extended Character Set Support: Ada 9X shall provide input/output facilities and other operations for extended character sets (specifically, for international character sets with more than 256 graphic symbols). The operations shall be comparable to those provided for the current base character set.

Discussion: The two requirements above address the issue of very large character sets, in particular, those of Asian countries such as Japan, China, and Korea. Ada 9X must provide a way to support these and other international languages. Where possible, Ada 9X should be consistent with ISO standards that are developing in this area.

These requirements should not be interpreted as mandating a method of displaying extended graphic symbols in source text. Instead, the point is that Ada 9X must support character types that require more than eight bits for their representation (typically 16 or 32), and must support the corresponding string types, file operations, etc. The Ada 9X report on character set issues [9] discusses these points in more detail.

Requirement R3.1-A(4) — Extended Comment Syntax: The use of international character sets (including those with more than 256 characters) shall continue to be permitted in comments in Ada 9X source code.

Discussion: The inclusion of extended characters in comments raises no conceptual problems since this is allowed in Ada 83 (see AI-00339).

Study Topic S3.1-A(5) — Extended Identifier Syntax: Ada 9X shall permit the use of extended character sets in identifiers while continuing to keep Ada's syntax and lexical rules independent of source code representation.

Discussion: The identifier issue is somewhat complex. Among issues to be considered is the applicability of the upper-lower case equivalence rule. The various parts of the ISO 8859 standard present difficulties here. For example, a character code that represents a letter in one part of ISO 8859 may represent a non-alphabetic graphic symbol in another part.[4] A general solution providing completely satisfactory use of identifiers for all languages may be unacceptably complex. One suggestion is that at least Latin-1 be permitted, which fully meets the needs of all

[4]For example, the character code representing the Greek letter χ in Part 7 of ISO 8859 represents the ÷ symbol in Part 1 (Latin-1).

Western European countries. It is reasonable to move at least this far — the question of how much further to go needs study.

It is highly desirable that the Ada 9X solutions in this area be compatible with the recommendations of WG9's Character Rapporteur Group (CRG), since the CRG's recommendations are likely to be used as guidelines for accommodating international character sets in existing Ada 83 implementations.

In considering possible extensions of identifier syntax, it should be kept in mind that Ada 83 does not specify the physical form of source programs, and this approach could be continued in Ada 9X — the standard does not need to specify how to convert alternative source code representations into the representation accepted by a given compiler. Separate standards for representing source code may be desirable since methods for representing text are not well standardized, but such standards are outside the scope of the Ada 9X revision effort.

3: Requirements for International Users

4 Support for Programming Paradigms

This Chapter deals with a variety of issues related to expressive power — the ability to use Ada in a way that is consistent with modern software engineering practice and that meets the needs inherent in programming a broad variety of applications. Some of the needs expressed in this Chapter derive from experience in using Ada and some are derived from experience in using other languages.

4.1 Subprograms

User Need U4.1-A: Dynamic Subprogram Selection: One common programming idiom involves deciding at run time which subprogram is to be applied to a particular parameter list. This style of programming arises naturally in a number of situations, e.g.

- when processing input messages and the action to be taken depends partly on the content of certain fields in a message;
- when traversing a data structure and different actions are to be taken depending on the values found at each point in the structure;
- when associating actions with user-interface buttons. The action to be taken can depend on the current input mode. When the mode changes, the set of actions to be taken is easily and quickly changed simply by pointing to a different table of subprograms. Moreover, the user sometimes determines what subprogram is to be associated with a given button, mouse key, etc. This means the action to be taken in response to some selection criterion is not known at compile time.
- when needing to change the effect of a group of subroutines without passing the same subprogram to each subroutine as a parameter. This can occur when specifying a ‘‘trap’’ procedure that is to be called when an exceptional situation occurs and the routine detecting the situation doesn’t want to raise an exception (e.g., because the routine is written in another language or because the trap procedure is expected to take a recovery action that allows processing to continue at the point where the exception situation was detected). Such a need has arisen in defining Ada bindings to the IEEE floating point standard and to SQL (e.g., see [11]).

In addition, some programs can be updated more easily and reliably if the subprograms to be selected are stored separately from the selection code; updating a table of subprograms that are called in certain situations can be less error-prone than modifying the selection code itself and promotes better program modularity. This ‘‘table-driven’’ programming style is a common paradigm that should be supported directly by Ada.

Study Topic S4.1-A(1) — Subprograms as Objects: Ada 9X should provide:

1. an easily implemented and efficient mechanism for dynamically selecting a subprogram that is to be called with a particular argument list;
2. a means of separating the set of subprograms that can be selected dynamically from the code that makes the selection.

Discussion: An obvious solution to this requirement is to provide subprogram types and variables.

This requirement is presented as a Study Topic because there are a number of well-known and difficult language design issues posed by such a solution (e.g., ensuring that non-local entities referenced by subprograms exist when the subprogram is called). In addition, there are problems unique to Ada such as possible interactions with overloading rules, the propagation of exceptions, and the notation for distinguishing between a reference to a parameterless function variable and a call on the value of the variable. Given the complexity of the design issues, it is appropriate to categorize this requirement as a Study Topic.

Of course, one way to select subprograms at run-time in Ada 83 is to use a case statement; each case alternative calls a different subprogram with the same argument list for each call. The case statement approach does not, however, satisfy item 2 of the Study Topic since the unit containing the case statement must be modified directly when changing the subprograms to be called. Direct modification is not necessary if the subprograms are contained in a data structure that is separate from the selection code.

This Study Topic interacts strongly with Study Topic S4.3-B(1) on page 21.

User Need U4.1-B: Interfacing with Non-Ada Subprograms: Ada programs must interface with non-Ada procedures and functions, including some that pass subprograms as parameters. Ada 9X should ensure greater uniformity in how implementations support such interfaces.

Requirement R4.1-B(1) — Passing Subprograms as Parameters: Ada 9X shall provide a uniform means of passing subprograms as parameters to non-Ada procedures and functions.

Discussion: Because the ability to pass subprograms as parameters to procedures and functions is provided by other widely used languages (particularly, FORTRAN and C), there exist important libraries of subprograms that make use of this capability (e.g., standard FORTRAN mathematics libraries, interfaces to windowing systems, database manager interfaces, etc.). Ada programs need the ability to call such ''foreign'' routines in a standard and reliable way.

The use of generic formal subprograms is not a solution to the problem of interfacing with non-Ada subprograms that require subprogram parameters since pragma INTERFACE cannot be applied to a generic unit. Consequently, less reliable workarounds must be used in implementations today that support this requirement (e.g., the subprogram parameter is declared as an access type and unchecked conversion is used to pass the address of the desired subprogram).

The ability to pass subprograms directly as parameters to any Ada procedure or function (not just to those that interface with foreign code) is a viable solution to this requirement that could be justified by the requirement for generality (see Requirement R2.2-C(1) on page 8). Moreover, such a capability would provide an easily implemented solution to one situation in which generic code sharing is required (see Requirement R4.4-C(1) on page 23). Finally, the need to manipulate subprograms as objects (see Study Topic S4.1-A(1) above) also leads naturally to a general ability to pass subprograms as parameters. However, the ability to pass subprograms as parameters does not pose as many complex language design issues as does the general ability to manipulate subprograms as objects.

Requirement R4.1-B(2) — Pragma INTERFACE: Ada 9X shall provide standard methods for specifying the link name of an externally defined subprogram and for specifying the parameter passing mechanism that is to be used. Other capabilities associated with pragma INTERFACE

that are commonly provided by implementations today shall also be considered for standardization (e.g., see [15]).

Discussion: There are a variety of implementation-dependent pragmas that in effect, extend the functionality of pragma INTERFACE. The most common extension is to define the link name of the external subprogram (since the name might not conform to Ada's syntax for subprogram names). Another, somewhat less common extension allows a programmer to specify the parameter passing mechanism. Extended support for pragma INTERFACE is one of the most common areas in which implementation-dependent pragmas have been created [15]. Ada 9X needs to standardize the means of providing this information.

4.2 Storage Management

User Need U4.2-A: Control of Storage Use: In many applications the programmer requires precise control over the allocation and deallocation of data storage. In the extreme (and realistic) case of a long-running application, allocation without reclamation is unacceptable, whether the storage is allocated explicitly by the programmer or implicitly by the run-time system.[5] In addition, the ability to provide specialized storage management algorithms is often essential when tuning an application's performance.

Requirement R4.2-A(1) — Allocation and Reclamation of Storage: Ada 9X shall allow an application to ensure that storage is allocated and reclaimed efficiently and predictably.

Discussion: In some applications, it is necessary to predict when storage is reclaimed and to ensure that the time taken to reclaim storage is not greater than some bound. In other applications, it is only important to be sure that storage will be reclaimed without needing to know exactly when or how long it will take. At the minimum, implementations should ensure against storage leaks, that is, the inability to reuse storage that no longer contains values of interest to a program.

Study Topic S4.2-A(2) — Preservation of Abstraction: To the extent possible, Ada 9X shall meet Requirement R4.2-A(1) in a manner that allows storage allocation and reclamation to be controlled reliably for user-defined data types.

Discussion: A programmer implementing an abstract data type in terms of dynamic data structures needs to be able to determine when memory used for values of that type may be reclaimed. At a minimum, the programmer should be able to gain control whenever storage is allocated and whenever a scope is deactivated. (Note that user-defined assignment and finalization appear to be needed here.)

Meeting this requirement will assist the programmer in controlling storage deallocation. In Ada 83, it can be difficult to insulate a user of a type from the decision to use dynamic data structures for its implementation unless garbage collection is provided; there is no way for the implementer of a type to determine when an abstract object is no longer needed. For time-critical applications,

[5] For example, some implementations allocate storage implicitly on the heap when an object's size can vary widely at run-time. Such storage must be subject to reclamation at an appropriate time.

storage allocation and reclamation actions must occur at predictable times and must be accomplished in a bounded amount of time.

4.3 Composition of Program Units

One of the principal design goals of Ada was to support the development of programs from separately compiled units while providing the same degree of checking as for a program compiled as a whole. Experience with Ada 83 has shown that Ada's rules for checking separately compiled units can sometimes make it difficult to minimize recompilation costs when programs are modified. Moreover, it is important to make it easier for programmers to compose programs by specializing and extending existing units. The needs and requirements expressed in this section address different aspects of these problems.

User Need U4.3-A: Reduced Recompilation: When a program unit is changed in a way that does not logically affect the behavior of other units, recompilation of these units should not be required or should be easy to avoid.

Discussion: From a user point of view the need is clear — changes that the programmer believes should have no effect on dependent units (e.g., the addition of new subprogram declarations to a package specification; a change to the initial value of a variable; the introduction of a new type declaration; a change to a subprogram body) should not require recompilation of other units. But Ada's rules make it difficult for a compiler to decide that a particular change will have no effect on dependent units: adding a new subprogram declaration could change the result of overloading resolution in dependent units; changing the initial value of a variable could lead to a change in the object code if a compiler performed optimizations based on its knowledge of the initial value; the introduction of a new type or object declaration could change the visibility of a name in a dependent unit; changing the body of a subprogram in a package usually means recompiling the whole package body, and it is not always easy to turn such subprograms into subunits because subunit names cannot be overloaded; changing the body of certain generic units could lead to recompilation of all units that instantiate the generic unit since in some cases, these instantiations could now be illegal. In short, Ada 83's recompilation rules, while ensuring that collections of units are consistent in their interdependences, have in practice made it complicated for implementations and programmers to avoid logically unnecessary recompilations.

Study Topic S4.3-A(1) — Reducing the Need for Recompilation: Ada 9X recompilation and related rules should be revised so it is easier for implementations to minimize the need for recompilation and for programmers to use program structures that reduce the need for recompilation.

Discussion: The Mapping/Revision Team should determine what changes would make it easier for implementations to avoid logically unnecessary recompilations. Some of these changes might affect how programmers use Ada. For example, allowing subunit names to be overloaded would allow programmers to use subunits more freely, helping to ensure that any change to a body will not result in recompilation of other units belonging to the same package body.

This requirement is categorized as a Study Topic because it is not clear to what extent Ada 9X can change Ada's recompilation rules without introducing possibly subtle semantic changes into existing Ada programs. Solutions to the next requirement, which deals with the need to adapt and

extend existing units without physically modifying them, may also serve to reduce the need for recompilation.

User Need U4.3-B: Programming by Adaptation of Existing Units: When adapting an existing package to new uses, programmers typically change just a few properties of the package. For example, the package might be adapted to new uses by adding several subprograms or by modifying the implementation of an existing subprogram. Similarly, an existing record type and its operations might be reused by extending its declaration. An increasing number of programmers want to program in this style.

Discussion: This User Need expresses one of the benefits of ''object-oriented'' programming, namely, the ability to adapt existing code to new uses by extending type declarations and adding new operations or modifying existing ones. Although Ada 83 supports this need to some extent, improvements are needed. (See [12] for a more detailed discussion of some of the difficulties encountered when attempting to use Ada 83 in this style.)

Study Topic S4.3-B(1) — Programming by Specialization/Extension: Ada 9X shall make it possible to define new declared entities whose properties are adapted from those of existing entities by the addition or modification of properties or operations in such a way that:

- the original entity's definition and implementation are not modified;
- the new entity (or instances thereof) can be used anywhere the original one could be, in exactly the same way.

Discussion: The statement of this requirement is taken from [2, p. 30]. The kinds of entities for which specialization and extension are required include types, subtypes, subprograms, packages, and generic units.

An important concept in software reuse is that a software component may be parameterized by a type. A parameterized component should work not only for a given type but also for any of a number of related types constructed from the original type. The advantage of constructing related types by specialization and extension of existing types is that there are broad conditions under which operations defined for the original type continue to be meaningful for the constructed types. This means that common underlying capabilities can be implemented and shared easily among related types in a layered fashion. When the construction of related types is combined with the ability to dynamically select a subprogram to be called (as specified in Study Topic S4.1-A(1) on page 17), the result can be a form of polymorphism.

This requirement is specified as a Study Topic mainly because of concern about the amount of change that might be required to support it effectively and the possible difficulty of integrating such changes smoothly into the existing language. Nonetheless, satisfying this requirement is considered to be important as a means of increasing Ada's ability to support software reuse.

Although the inspiration for the User Need and Study Topic arises from the so-called object-oriented languages, these generally provide more than is called for here. However, what is required here will go a long way toward improving Ada's support for software reuse.

User Need U4.3-C: Library Support: When developing large systems, developers need to exploit the program library in complex ways. For example, developers need to use multiple libraries in conjunction with version control systems, and they need the ability to partition a library so not all library packages are necessarily visible.

Study Topic S4.3-C(1) — Enhanced Library Support: Ada 9X shall make it easier for implementations to provide improved library management support and when possible, shall improve the uniformity with which such support is provided.

Discussion: The Ada concept of a program library exists primarily to ensure that separately compiled units can be combined with the same degree of checking as if they had been compiled as a single unit. In attempting to specify this goal, the standard uses words that sometimes seem to limit unnecessarily the services an implementation can provide to a user (see RR-0091, for example). On the other hand, implementations vary considerably in the library management services they provide. For example, some implementations support multiple libraries and can support multiple versions of units while others are quite restricted in their capabilities. These differences cause difficulties in porting Ada programs across compilers. Although Ada intended to provide wide implementation freedom in support for libraries, some improved standardization would be useful and may be possible.

This requirement is classified as a Study Topic because it is uncertain what library management functions are sufficiently proven that they should be specified at least in an Ada 9X annex.

4.4 Generics

There are three broad areas of user needs for generics. One is support for development of reusable components; this requires a facility with expressive power and general parameterizability. The second is separation of concerns between the users and the implementer of a generic unit. The third area is code space efficiency, sometimes known as code sharing.

There are conflicts among some of these needs, and Ada 83 offers one particular compromise. In the light of user experience, the Mapping/Revision Team should examine the tradeoffs that were made, identify which if any language improvements are required, and make the necessary revisions.

User Need U4.4-A: Support for Reusability: The design of a generic unit involves abstracting from particular instances to a general template, where the variability across the instances is reflected in the values passed as parameters at instantiations. A fundamental need, then, is for the language to supply sufficiently general generic parameterization to allow the definition of wide classes of reusable components such as data structures (lists, queues), graphics packages, etc.

Study Topic S4.4-A(1) — Generic Formal Parameters: Ada 9X should enhance generics to liberalize the uses of generic formal parameters within the template, and/or to allow more kinds of formal parameters, in the interest of improving the language's abstraction facilities.

Discussion: The so-called ''contract model'' (see below), coupled with requirements for staticness in a number of contexts in Ada 83, has resulted in situations where generics do not provide all the expressive power that would be desired. For example, it is not possible to use a generic formal **in** object or an attribute of a formal generic type where a static value is required. Relaxing the contract model or the requirements for staticness would make it easier to write certain useful kinds of generics (for example, a generic math package where the precision of a local type depends on the precision of formal type).

A related issue is generalization of the kinds of entities allowed as generic formal parameters. For example, if exceptions were allowed, exceptions raised by formal subprograms could be handled more appropriately within the generic unit (see RR-0383). However, a major reason for Ada 83's disallowance of formal generic exceptions is that such a facility could incur a run-time penalty by preventing an implementation from composing static tables for handlers. Such interactions need to be considered by the Mapping/Revision Team.

User Need U4.4-B: Independent Use and Implementation of Generics: A generic specification ought to serve as a ''contract'' between the users and implementer of a generic unit. An appropriate contract promotes *separation of concerns*, allowing a user to instantiate a generic unit without regard to the contents of the generic body and allowing an implementer to write a body without regard to the ways in which the unit will be instantiated. It also facilitates early detection of errors: if an instantiation is compiled before the generic body, then all semantic errors in the instantiation will be detected when the instantiation is compiled.

Requirement R4.4-B(1) — Dependence of Instantiations on Bodies: A user must have the ability to compile a generic instantiation before compiling the corresponding generic body. In those cases where a body is compiled before the instantiation, the user must have the ability to ensure the unit containing the instantiation is not dependent on the generic body.

Discussion: The ability to prevent the establishment of a compilation dependence is important in order to control the need for recompilation. If a dependence is established (e.g., if a generic unit is compiled inline at the point of instantiation), any modification of the generic body will generally require recompilation of all units containing an instantiation.

Study Topic S4.4-B(2) — Tighten the "Contract Model": Ada 9X should ensure that compatibility of a generic instantiation with the corresponding generic body is based solely on the compatibility of each with the generic specification. That is, there should be no cases where the legality of an instantiation depends on how a generic formal is used within the generic body.

Discussion: In a small number of situations, it is possible in Ada 83 to have the legality of a generic instantiation depend on how a generic formal parameter is used. One example is passing an unconstrained array type as an actual parameter to a formal private or limited private type. Such an instantiation is correct only if the generic template contains no declaration of a variable having that type, since it is illegal to declare a variable having an unconstrained array type. Tightening the contract model would eliminate such anomalies.

Note that the discussion of Study Topic S4.4-A(1) above suggested that the contract model should be relaxed rather than tightened, in order to satisfy other needs. This conflict with Study Topic S4.4-B(2) needs to be resolved by the Mapping/Revision Team after an analysis of the tradeoffs.

User Need U4.4-C: Code Space Compactness: A direct implementation of generic instantiation results in replicating the code for the expanded template. In some environments such code replication is undesirable, and it is preferable to have a single version of the template, with run-time parameterization to identify instantiation-specific information.

Requirement R4.4-C(1) — Generic Code Sharing: Ada 9X shall provide a mechanism (for example, a pragma) requesting the sharing or replication of generic code templates. An imple-

mentation may place restrictions on the kinds of generic units for which sharing will be performed.

Discussion: Several situations are candidates for code sharing. One is a generic template parameterized by subprogram (for example, a generic function for mathematical integration). This is the Ada 83 style for what in other languages is done by passing subprograms as run-time parameters; such an implementation is desired by many users. Another situation is a generic template taking, for example, a formal access-type parameter. Instantiations with types sharing the same machine representation can often share the same code. A third example is a generic body that depends only on certain attributes of a formal type, such as its 'SIZE attribute. This situation arises in cases such as block moves and I/O interfaces.

4.5 Exceptions

User Need U4.5-A: Exception Name: When an unexpected exception is raised, many applications require that at least its name be logged to help in tracing what went wrong.

Requirement R4.5-A(1) — Accessing an Exception Name: The simple name of the most recently raised exception shall be available as a string in an exception handler. If additional information (such as the name of the unit where the exception was raised) can be made available at little implementation cost, this would also be helpful.

Discussion: Nine revision requests, from a variety of sources, mentioned a need to have at least the name of a raised exception available in a standard way in an exception handler, and several implementations already support this capability. Ideally, additional information concerning the unit in which the exception was raised would also be made available. There was no strong demand for any additional information to be made available in a handler.

4.6 Input/Output

User Need U4.6-A: Interactive Text Input/Output: Applications typically require the ability to communicate with users via interactive devices.

Requirement R4.6-A(1) — Interactive TEXT_IO: Ada 9X shall improve the ability of Ada programs to communicate with users via interactive devices. Among the capabilities to be considered are:

- The ability to output text and receive input on the same line.
- Ensuring that column, line, and page numbers reflect use of the same device for input and for output.
- The ability to continue executing other tasks while one task is waiting for input.

Discussion: In principle, the current definition of TEXT_IO allows programs to output text and receive input on the same line (as illustrated by the example given in section 14.7 of the standard). But in practice, many TEXT_IO implementations buffer their output until a call to NEW_LINE is given, thereby making it impossible to query a user and receive input on the same line. Moreover, because text files cannot be opened in mode IN_OUT, the column, line, and page

numbers for input and output operations are maintained separately, and therefore are not useful when interacting with a single device. Finally, if a GET procedure is called in a task, some implementations do not allow any other task to execute until the GET procedure returns. This is often unsatisfactory for applications.

User Need U4.6-B: Various Input/Output Functions: Experience has indicated that there are a few simple additional I/O functions that would make the input/output packages more useful to users.

Requirement R4.6-B(1) — Additional Input/Output Functions: Ada 9X shall augment Ada's input/output functionality in ways that make the capabilities easier to use. In particular, the following functions shall be considered for inclusion in Ada 9X:

- The ability to check whether a file exists (without having to open it).
- A standard way to open a file for appending data to it.
- The ability to determine the maximum line length and page length for a given text file.
- The ability to read or write different types of data to the same external sequential file.

Discussion: The listed capabilities are among those that have been requested by Ada users. The ability to test for the existence of a file without having to open it is just a matter of providing a standard function for performing this useful operation. The ability to open a file so data can be appended to it is already allowed in Ada 83 (see AI-00278), but in an implementation-dependent manner. The ability to read and write different types of data to the same external file is required in many applications. In such files, the type of an item is often determined by the content of earlier items in the file. Such a file structure can be supported in Ada 83 by opening the same external file with different data types, but most implementations do not support such a use of ''shared'' files, and in any event, the behavior of such files is almost entirely implementation-dependent.

In short, the purpose of this requirement is to provide modest functional improvements to Ada's input/output capabilities.

4: Support for Programming Paradigms

5 Real-Time Requirements

This chapter deals with requirements posed by real-time applications, i.e., applications in which it is important to have precise control over *when* some action is taken. Other requirements of potential interest to the real-time community are addressed in Chapter 6, Requirements for System Programming (interrupt handling), in Chapter 7, Requirements for Parallel Processing (shared variables), and in Chapter 8, Requirements for Distributed Processing. Many real-time applications are safety-critical; safety-critical requirements are addressed in Chapter 9.

5.1 Time

User Need U5.1-A: Time Measurement: Real-time systems typically need to measure time in two distinct ways [8, p. 124]. Computations of time-based physical parameters such as velocity and acceleration depend on the measurement of *elapsed time*. The key property of elapsed time is that its value increases monotonically at a known rate (from some convenient starting point, e.g., mission start time).

On the other hand, for interfacing with external entities such as other processors, devices, and human operators, the *time of day* is needed. The time of day does not necessarily increase monotonically at a fixed rate, since it may have to be adjusted to account for time zones or leap seconds, or to synchronize with other clocks on a local area network. Such adjustments may cause the time-of-day value to retrogress, while retrogression is never allowed for measurements of elapsed time.

The precision of the two kinds of timing measurement may differ; a clock used to control and monitor elapsed time may be updated at a faster rate than the time-of-day clock when elapsed time measurements are used to control physical processes in tight feedback loops.

Requirement R5.1-A(1) — Elapsed Time Measurement: Ada 9X constructs that deal with time shall support two abstractions of time — monotonically increasing elapsed time and the time of day. The time of day abstraction shall be compatible with the Ada 83 package CALENDAR and may be maintained with different precision from that of elapsed time.

Discussion: The time defined in Ada's package CALENDAR is the time of day (see AI-00195). While real-time applications must sometimes be synchronized with the time of day, more frequently such applications need to use measurements of elapsed time. Baker [7, 8] discusses in more detail why both kinds of time measurement are needed.

User Need U5.1-B: Periodic Computation: Many real-time applications must perform certain actions at precise periodic intervals.

Requirement R5.1-B(1) — Precise Periodic Execution: Ada 9X shall provide a standard way to control the periodic execution of a sequence of statements with respect to elapsed time and the time of day. The method provided shall allow the period to be changed during the execution of a program and shall ensure that the statements are scheduled for execution at precisely the specified time.

Discussion: The current **delay** statement is not generally suitable for precise periodic scheduling because the computation of the necessary delay can be preempted at critical moments, leading to inaccurate periodic execution (see [7]).

This requirement is phrased neutrally with respect to whether precise periodic scheduling is provided by a pragma, a special call to the run-time system, or new syntax. In any case, performance constraints on implementations targeted to real-time applications will have to be specified to ensure that the specified solution is supported in usable fashion. Indirect solutions to this requirement could also be satisfactory. For example, the Third International Workshop on Real-Time Ada Issues [8] suggested that a **delay until** statement could be used to achieve this requirement. Another approach is being explored by the Ada Run-Time Environment Working Group (ARTEWG) with their Catalogue of Interface Features and Options (CIFO), which is currently under development.

User Need U5.1-C: Overrun Detection and Response: Real-time programmers need ways of specifying deadlines and ways of specifying what action is to be taken when a deadline is missed. Among the actions typically taken are: log the overrun and continue processing, stop the overrunning activity from continuing its execution (restarting it at an appropriate point and time), temporarily changing the rate of execution, etc.

Requirement R5.1-C(1) — Detection of Missed Deadlines: Ada 9X shall facilitate the detection of missed deadlines and the ability to initiate an appropriate response.

Discussion: Although watch-dog tasks can be used to detect overruns,[6] when an overrun is detected, it is sometimes important to quickly stop the activity that has missed its deadline so other processing can be started. In general, this requires an asynchronous transfer of control (see Requirement R5.3-A(1) on page 30).

5.2 Task Scheduling

User Need U5.2-A: User-Controlled Scheduling: Scheduling algorithms determine when a task is selected for execution. Scheduling decisions are associated with entry queues (selection of the next call to be serviced), selective waits (selection from among calls waiting at open alternatives), attempts at task synchronization via entry calls (how is the priority of the called task affected), the expiration of a delay, the priority, deadline, or execution time of a given task in relation to other tasks, etc. Specialized scheduling algorithms giving improved performance are sometimes needed to take advantage of an application's expected use of tasking constructs. Since the same scheduling algorithm is not suitable for every real-time application, application developers must be able to determine the algorithm to be used for a particular application.

Requirement R5.2-A(1) — Alternative Scheduling Algorithms: Ada 9X shall allow real-time programmers to determine the scheduling policy and algorithm to be used for a particular program.

[6] Such tasks execute timed waits. If an appropriate entry is not called before the delay expires, an overrun has occurred.

Discussion: This requirement intentionally avoids specifying how a scheduling policy will be specified by the user. One possibility is that Ada 9X will specify a set of standard scheduling policies plus a means of selecting which policy/algorithm to use for a given program. Another approach is for Ada 9X to specify low-level primitives or a standard run-time interface from which scheduling policies and algorithms can be built. In either case, scheduling alternatives that are not of general interest could be described in a real-time annex.

Many in the real-time community favor the low-level or standard interface approach because of the flexibility it can provide. But it is not yet clear that a standard interface or set of primitives can be exploited efficiently for all potential target machines — any given interface always seems to be biased in favor of some class of hardware architectures or some anticipated pattern of use which may not be right for a given application. Even compiler vendors sometimes find it difficult to stick to a self-imposed standard run-time interface when highly efficient performance or specialized optimizations are demanded.

The advantage of having Ada 9X specify some standard scheduling policies (e.g., in a real-time annex of the standard) is that the policies could provide a standard framework for designing and implementing real-time applications. The potential disadvantage, of course, is that applications requiring some other policy or some adjustment to a scheduling algorithm may be unable to use Ada tasks. Moreover, there is little agreement on what such policies must be. Nonetheless, specifying a few standard scheduling policies could help reduce the difficulty of porting real-time applications.

Given that Ada's current scheduling rules are relaxed to clearly allow the use of alternative scheduling algorithms, some feel that sufficient flexibility is provided simply by giving application developers access to vendor-supplied Ada run-time systems. This is not an entirely satisfactory solution since a given compiler vendor may not provide compilers for all the target configurations of interest to the application developer. In such a case, an application developer prefers to develop a portable executive that can be interfaced efficiently with different Ada compilers.

Requirement R5.2-A(2) — Common Real-Time Paradigms: Ada 9X shall ensure that common real-time programming paradigms can be implemented with predictably efficient performance. Among the paradigms to be considered are: controlling access to data shared among different tasks, resetting task priorities in response to a mode change, critical regions, and asynchronous communication (see User Need 5.4-A).

Discussion: The requirement mentions a number of real-time programming paradigms. Although the Ada tasking model can generally be used to support these programming requirements today, many in the real-time community feel that special language constructs are needed to express these paradigms clearly and to ensure they are implemented with appropriate efficiency. Others feel that special language features are not needed, that pragmas can be used to advise implementations that certain programming paradigms are associated with certain tasks, and that this information is sufficient to ensure acceptably efficient implementation support. But there is no disagreement that these paradigms must be supported with particular efficiency.

Some real-time programming paradigms are discussed in the Proceedings of the Third International Workshop on Real-Time Ada Issues [23]. For example, a monitor task (also called a

passive task) is a specialized tasking idiom that allows rendezvous to be performed for essentially the same cost as a procedure call. To achieve this performance, different implementations today impose different constraints on the structure and content of such tasks (e.g., the task body is required to consist of a single select statement in a single loop, and it can only call other passive tasks). Ada 9X could improve this situation by specifying standard restrictions that ensure no scheduler intervention is required to execute a rendezvous (assuming the entry call is not blocked) and by requiring that implementations document tasking performance parameters. On the other hand, given the frequent need for such operations, it might be best to provide a special language feature to support it.

Application developers also sometimes restrict their uses of tasking constructs in ways that permit special execution efficiency. For example, tasks in embedded real-time systems are often declared only at the library package level, meaning there is a fixed number of tasks and there are no dependent tasks. Under these conditions, the performance of various scheduling operations can be significantly improved. For example, task abortion and termination is simplified because there are no dependent tasks. The Mapping/Revision Team should consider whether such usage restrictions deserve standardization in a real-time annex.

5.3 Asynchronous Control of Execution

User Need U5.3-A: Asynchronous Control of Execution: There are many situations (e.g., mode change, dynamic reconfiguration, system restart) where it is useful to terminate one sort of processing and continue processing in a new place, based on the asynchronous occurrence of some event. For example, it is sometimes necessary to stop a process in response to a user interrupt because there is no longer any need for the activity and resources must be released to do something else (e.g., aborting an autopiloted landing). Similarly, it is sometimes necessary to stop a process when it has overrun a deadline.

Requirement R5.3-A(1) — Asynchronous Transfer of Control: Ada 9X shall allow the execution of a sequence of statements to be abandoned in order to execute a different sequence within the same task. The event causing this transfer might be, for example, an external interrupt from a timer or a call from another task. It shall be possible to abandon the execution quickly, even if execution is suspended at an entry call or an accept statement. In balancing the need for immediate response with the need for safe and predictable execution (e.g., the ability to release resources held by the abandoned code) (see Guideline **G-2**), it shall be possible to identify certain sections of code in which a request for asynchronous transfer of control will not be honored.

Discussion: Events requiring an asynchronous transfer of control can arise under conditions where they must be handled within strict time constraints. It must be possible to stop the computations of certain tasks quickly. Suspension is not enough because when a task resumes execution it must continue at a different execution statement. (For example, if an autopiloted landing is aborted, the responsible tasks should be restarted from some earlier point (with fresh data) the next time a landing is attempted.) In some cases, the transfer of control must be immediate. This requirement conflicts somewhat with other needs, e.g., the need to ensure that data structures hold valid values, the need to ensure that allocated storage is released when no longer needed, etc. The Mapping/Revision Team must find an appropriate way of satisfying these conflicting requirements.

The design of Ada in 1980 provided an asynchronous exception for asynchronous transfer of control. This mechanism was removed from the final design because various significant difficulties were discovered (see RR-0196 and [24] for a full discussion). The relevant difficulties have been mentioned in the above requirement to ensure that they are addressed in developing proposed solutions.

5.4 Asynchronous Communication

User Need U5.4-A: Asynchronous Message Passing: The ability to send data to a task without waiting for it to be received is an important and common paradigm in real-time programming. For example, device drivers in an embedded system must be able to send data to tasks without any scheduler intervention that might cause interrupts to be missed. Similarly, non-blocking operations are used when sending large amounts of information to relatively slow recording devices.

The message-passing paradigm is also appropriate for computer systems that do not have shared memory and for massively parallel systems. Finally, in some applications, messages with different priorities need to be sent to the same task; the messages need to be processed in priority order.

Discussion: Real-time programs could be easier to understand and maintain if the non-blocking communication paradigm were directly expressible in the language. Most real-time operating systems provide direct support for non-blocking communications (e.g., mailboxes), so real-time programmers are accustomed to having this capability available. Non-standard, implementation-dependent mechanisms are currently being used to support asynchronous communication needs. To reduce implementation dependencies and increase maintainability and portability, Ada 9X needs to specify standard asynchronous communication mechanisms.

Requirement R5.4-A(1) — Non-Blocking Communication: Ada 9X shall support non-blocking, asynchronous communication between tasks in an Ada program. In particular, from the viewpoint of the sending task, the maximum time to execute an asynchronous communication operation shall normally be independent of the state of the receiving task.

Discussion: Agent tasks that accept messages to be queued are the standard Ada construct for attempting to support non-blocking communications. Since all agent tasks have the same priority, service order is not guaranteed by Ada 83 rules. Moreover, if agent tasks are to be used for this purpose, the overhead that is apparently involved in creating and terminating them must be somehow be avoided, since fast service is especially important when an interrupt handling routine needs to store data very quickly and predictably in a buffer. For all these reasons, agent tasks have not proven to be a very satisfactory solution to this requirement.

From a programmer viewpoint, providing the capability with a standard package would probably be satisfactory, but Ada's current generic unit capabilities make the use of such a package awkward (different generic units are needed for asynchronous calls with different numbers of parameters; see [6]). It is possible that other Ada 9X changes to generics (see Section 4.4) would make it easier to provide an efficient message-passing package. On the other hand, considering these difficulties, direct support for a new language construct might prove to be the best solution.

User Need U5.4-B: Asynchronous Multicast: The ability to send messages to a group of tasks (without waiting for a reply) is sometimes necessary. Similarly, the ability of a receiver to determine its own eligibility to receive such a message is sometimes useful.

Study Topic S5.4-B(1) — Asynchronous Multicast: Ada 9X shall support asynchronous communication from a single sender to multiple receivers as well as the ability of potential receivers to control their eligibility to receive a message.

6 Requirements for System Programming

6.1 Unsigned Integer Operations

User Need U6.1-A: Unsigned Integers: Many coding algorithms (for example, checksums and cyclic redundancy checks) are formulated most naturally in terms of operations on unsigned integers. It should be possible to express such algorithms efficiently and directly in Ada.

Requirement R6.1-A(1) — Unsigned Integer Operations: Ada 9X shall provide efficient support for operations on unsigned integer data. In particular, unsigned arithmetic, logical operations, and shifting operations shall be supported.

Discussion: AI-00402 explains why unsigned integer types cannot be predefined types in Ada 83. AI-00597 and RR-0460 discuss the benefits of unsigned integer types and how they might be provided by careful interpretation of the current standard. However, it would be best if Ada 9X supported such types clearly and directly.

6.2 Data Interoperability

User Need U6.2-A: Data Interoperability: Many applications require the ability to interchange data with other programs and systems when the required format of the data is not under the control of the application.

Requirement R6.2-A(1) — Data Interoperability: Ada 9X shall provide a means to specify the exact representation of data in memory and on external files so that interoperability with other systems can be facilitated.

Discussion: Ada's representation clauses were intended to allow the complete specification of an object's representation. However, existing implementations do not always fully support representation clauses. The Mapping/Revision Team needs to ensure that methods for specifying data representations can and will be supported more uniformly.

6.3 Interrupts

User Need U6.3-A: Interrupt Handling: Interrupts typically need to be serviced with little or no run-time overhead. In addition, an association between interrupt-handling code and a source of interrupts must sometimes be changed at execution time.

Requirement R6.3-A(1) — Interrupt Servicing: Ada 9X shall ensure that an interrupt can be serviced with minimal (and predictable) run-time overhead. Ada 9X shall allow interrupts to be processed at an appropriate priority.[7]

[7]The appropriate priority will or could depend on the hardware architecture, the scheduling policy, and the application's view of the relative importance of processing the interrupt.

Discussion: Among the issues to be considered in satisfying this requirement are the ability to:

1. Permit the interrupt handling code to mask its triggering interrupt, e.g., while it is manipulating hardware that may cause the interrupt to occur or while it is not ready to accept or handle the interrupt.
2. Handle interrupts with minimal scheduler overhead when data must be passed to tasks executing at a lower priority or when the occurrence of an interrupt serves as a signal that some task can be released for execution.
3. Permit the handling of nested interrupts when the hardware architecture makes this feasible.

In addition, it is consistent with the requirements for user-controlled scheduling (Requirement R5.2-A(1)) that the priority associated with interrupt processing not be dictated by the language but instead be subject to the scheduling policy developed for a particular application, the hardware architecture, and the overall software design.

Ada implementations today that are aimed at embedded applications generally provide special implementation-defined capabilities to support interrupt handling [15]. Even when targeting the same processor, implementations from different vendors can vary widely in their support for interrupt handling code [15]. If possible, Ada 9X should bring more uniformity to how implementations support interrupt handling.

Requirement R6.3-A(2) — Interrupt Binding: Ada 9X shall allow the association between an interrupt and its interrupt-handling code to be modified during program execution.

Discussion: An interrupt binding may need to be changed dynamically for a variety of reasons, e.g., to recover from a hardware fault or to respond to an operational mode change. Ada 83 does not support such changes because the method for associating interrupts with interrupt handling code can only be invoked when a task is activated. There is no standard way of later changing the association, although implementations sometimes provide non-standard ways of doing so.

Ada 83 implies that the notation for specifying a memory location is the same as the notation for identifying an interrupt that is to be associated with handler code (since address clauses and the type SYSTEM.ADDRESS are used for both purposes). Although considerable flexibility is allowed by the language in the notation used for an ‘‘address’’ and hence, for specifying an interrupt binding, the Mapping/Revision Team should consider whether improvements are needed.

6.4 Dynamic References to Global Objects

User Need U6.4-A: Dynamic Access to Global Objects: In embedded applications, data may be pre-allocated by mechanisms other than the Ada program that needs to use the data (e.g., the data may be stored in read-only memory or may be created and loaded prior to program execution). The selection of which data to access may need to be determined at run-time (e.g., see RR-0018 and RR-0726).

Requirement R6.4-A(1) — Access Values Designating Global Objects: Ada 9X shall allow the use of access values that designate objects declared in library packages. Operations using such access values shall maintain the reliability and safety of Ada 83 access types.

Discussion: The ability to use access values to designate objects that have not been created by allocators is essential in some applications. Such designated objects are typically created and initialized by external mechanisms or are stored in read-only memory, so allocators are not a usable means of creating access values designating the objects. In addition, the selection of an access value must often be done at run-time, e.g., it must be possible to store such access values in data structures (such as arrays) that permit dynamic selection of an appropriate designated object.

Address clauses, the ADDRESS attribute, and unchecked conversion can often be used today to obtain access values that can be used in the desired manner, but such usage is more implementation-dependent than it needs to be.

User Need U6.4-B: Low-Level Manipulation of Access Values: Low level system programming, such as writing special-purpose storage allocation routines, require the ability to manipulate pointers, including the ability to obtain pointers to local objects and to perform address arithmetic.

Study Topic S6.4-B(1) — Low-Level Pointer Operations: Ada 9X shall provide low-level mechanisms for manipulating pointers.

Discussion: Ada 83 provides some capabilities for manipulating pointers via use of the ADDRESS attribute and unchecked conversion between access values and ADDRESS values. However, these techniques are potentially non-portable, although in practice, many implementations support these operations in a fairly portable fashion. WG9's Uniformity Rapporteur Group has been addressing these issues.

6: Requirements for System Programming

7 Requirements for Parallel Processing

The use of parallel computers, in which a single computer consists of a (possibly very large) number of identical processors, is an important architectural trend that must be accommodated by Ada 9X. Not only can we expect to see greater use of tightly coupled multi-processors for computationally intensive tasks, but individual processors will increasingly turn to chip-level parallelism as a path to increasing instruction throughput.

The tasking mechanism in Ada 83 provides an important starting point for building Ada programs for such parallel environments, but a number of shortcomings have been identified that makes it difficult to map Ada tasking programs to parallel machines. In other languages, such as C or FORTRAN, vendors have addressed these problems by adding specialized extensions, which are often non-portable. This approach is not desirable for Ada compilers.

An important part of the problem is that there is a wide variety of parallel computing architectures, differing in such regards as the availability and treatment of shared memory and the extent to which separate processing elements are distinct. On the one hand, vector machines allow for massive parallelism without separately identifiable threads of control, whereas MIMD machines provide separate threads of control much nearer to the Ada tasking model. There is no clear consensus on what constitutes the best design approach, and indeed a most likely scenario is that widely differing architectures will continue to prove applicable to different problem domains. This means that any solutions proposed for Ada 9X must be able to accommodate a variety of architectures.

7.1 Shared Memory

User Need U7.1-A: Control of Shared Memory: It must be possible to write Ada programs that efficiently map onto parallel machines with a range of architectures from systems with no shared memory to those where shared memory is available and must be efficiently accessible.

Requirement R7.1-A(1) — Control of Shared Memory: Ada 9X must permit an Ada programmer to control accesses to shared memory and to make full use of shared memory.

Discussion: In Ada 83, objects are shared by default. In general there is no way for a compiler to determine at compile time, or even at link time, which variables are actually shared (i.e., accessed by more than one task). In systems with no shared memory (such as hypercube architectures), shared memory must be simulated by message passing and is typically significantly more expensive than direct memory references. A programmer writing an Ada application for such a system must be able to minimize the use of shared variables and it must be possible for the compiler to determine which variables are shared.

In parallel systems that do provide shared memory, it is important that Ada programs be able to take full advantage of shared memory algorithms. As discussed in the Ada 9X Special Study on Shared Variables [13], the current facilities in Ada 83 are inadequate. For example, there is no provision for asynchronous shared access to elements of composite objects. Even in parallel systems with shared memory, it is typically the case that individual processing elements have local memory or caches; taking full advantage of such an architecture also requires the ability to clearly distinguish between shared and non-shared data.

It is not in general possible to write efficient programs for parallel systems that are completely unaware of the architecture. Although it will continue to be possible for any Ada 9X program to run on any computer supporting Ada 9X, maximally efficient execution often involves a choice of algorithms and layout that are architecture dependent. Requirement R7.1-A(1) specifies that Ada 9X must provide features that allow a program to clearly specify its requirements for using shared memory.

Shared memory also plays an important role in Ada applications intended to run on a single processor. Solutions to Requirement R7.1-A(1) should also provide for improved control of the use of shared memory on conventional processors.

7.2 Massively Parallel Architectures

User Need U7.2-A: Large Numbers of Tasks: If Ada tasks are used to describe threads of control, and if a program is run on a massively parallel system with many processors, then the resulting program must be able to handle a large number of tasks in an efficient manner.

Study Topic S7.2-A(1) — Managing Large Numbers of Tasks: Ada 9X must provide for the efficient creation, initialization, execution, and termination of large numbers of tasks.

Discussion: A critical requirement for efficiency of large scale parallel applications is that there be no ‘‘serial bottlenecks,’’ i.e., points at which the execution time depends on executing serial code whose execution time is dependent on the number of processors.

Ada 83 does provide many of the needed facilities to meet this requirement. Tasks can be initiated in parallel, and terminated in parallel (using the terminate alternative). However, there is no easy way to give tasks an identity so that the tasks that are initiated in parallel can work on separate parts of a problem without communicating with some master controlling task.

There are a number of possible approaches to solving this problem. One approach is to provide a task with the ability to directly determine its own identity (for example, its index in an array of tasks). Another approach is to provide some kind of mechanism for parameterizing tasks.

7.3 Vector Architectures

User Need U7.3-A: Support for Vector Architectures: Ada 9X compilers must be able to efficiently exploit vector machines and other architectures requiring statement level parallelism.

Study Topic S7.3-A(1) — Statement Level Parallelism: Ada 9X should accommodate compiler techniques for efficiently mapping sequences of Ada statements, including particularly appropriate loops, onto vector architectures.

Discussion: Although Ada permits the use of tasks for explicitly describing parallelism, this is much too heavy a mechanism for loop level parallelism. In other languages, approaches to this problem have involved automatic recognition of loop parallelism by compilers and the introduction of specialized loop constructs explicitly specifying parallel execution.

Compiler techniques for mapping sequential code onto vector architectures are well known. However, there is some difficulty in applying these techniques to Ada because of exception semantics. The relaxations permitted by section 11.6 of the Ada standard are insufficient to permit vectorization to the desired degree. For example, a simple loop that adds the elements of two vectors together cannot be vectorized since Ada requires that an exception occurring in the first loop prevent further iteration of the loop.

7.4 Configuration of Parallel Programs

User Need U7.4-A: Configuration of Parallel Programs: Control is needed over the mapping of tasks to processors in multi-processor machines in order to achieve maximum performance.

Study Topic S7.4-A(1) — Configuration of Parallel Programs: Ada 9X compilation systems should permit the programmer to optionally control the assignment of Ada tasks to separate processing elements in a multi-processor system.

Discussion: There are two situations in which this kind of control is required if programs are to achieve maximum performance.

First, in some architectures, the communication between separate processors is not uniformly efficient. Certain processors may be ‘‘nearer’’ to other processors, allowing more efficient access between them, or processors may be arranged into groups, with more efficient communication within a group than between processors in separate groups. In such situations, it should be possible, though not required, for the programmer to control the placement of Ada tasks to optimize inter-task communication.

Second, there may be more tasks than processors, in which case multiple tasks must be mapped onto a single processor. In such situations, it may typically be the case that communication between tasks on a single processor may be considerably more efficient. For example, such tasks may be able to efficiently access the local memory of a processor to give shared memory effects between one another, even though there is no system-wide shared memory. Again, such architectures require that the programmer exercise control over task placement.

Another factor to be considered is the effect of priorities. Depending on the architecture, the effect of priorities might be quite different on tasks residing on the same processor and on different processors. Similarly, whether tasks can migrate from one processor to another must be considered and, if necessary, controlled in designing an application.

7: Requirements for Parallel Processing

8 Requirements for Distributed Processing

The use of distributed multiple computers to implement large-scale resource-intensive applications has increased significantly since the adoption of Ada in 1983. In the future, this use is expected to proliferate as the advantages of distributed execution become economically attractive to more applications. Typically, distributed applications use dynamic configuration facilities to optimize the use of available computing resources. Furthermore, fault tolerance must be addressed, since an important motivation for the use of distributed systems is the construction of survivable systems that can continue operation even if one or more of the distributed components ceases to function. Applications for which distributed processing support is essential include command and control systems, fault-tolerant systems, trusted systems, safety-critical systems, and information systems.

To the extent that a distributed computer system is similar to a parallel computing system, the considerations in the preceding section on parallel processing apply. In particular, control over shared memory is even more important, since typical distributed systems have no shared memory and simulation by message passing can be prohibitively expensive. Similarly, the ability to partition tasks across processors is important if a single Ada program is to be distributed.

However, distributed processing introduces a number of additional complications that must be addressed. In particular, a distributed system may be heterogeneous, involving different processors and different operating systems, and it may also require dynamic configuration, in which parts of the system are replaced or modified as the program is running. The Ada model of binding together a single large application statically into one program may well not be applicable in such situations.

Currently, Ada 83 can be used to program applications for multiple computers. A variety of approaches are used in practice. Most commonly, an application is treated as an interacting group of independently executing Ada programs. Less frequently, an application is written as a single program whose parts are allocated across target architectures. In each approach, applications rely upon different degrees of extra-lingual support, thereby compromising dynamic control and safety among the program units comprising the application. Moreover, the lack of specific language support degrades portability.

Ideally, Ada 9X should specify an approach to distributed programming that minimally extends the language to accommodate both the logical and physical separation of an application for distributed execution. The extensions must not preclude the continued use of existing approaches.

8.1 Distribution of Ada Applications

User Need U8.1-A: Distributing an Application: It must be possible to structure the code for a given application so it can be executed using multiple separate computers. This requires the ability to partition code into parts that need to be executed on a single computer. After distributing the parts among separate computers, the resulting code must retain the same degree of type safety as for a single program.

Requirement R8.1-A(1) — Facilitating Software Distribution: Ada 9X shall facilitate the distribution of Ada code across a homogeneous distributed architecture. Enhancements to facilitate distribution might include, for example:

1. The specification of the exact semantics (behavior and timing, including failure modes) of timed and conditional entry calls, potentially remote subprogram calls, and exception propagation.
2. The specification of semantics to deal with hardware failure and recovery.
3. Treatment of the existence of independent clocks with possibly different timing precisions.

Discussion: Distributed Ada systems will become more prevalent in the future. One form of software for a distributed system is a single program that is partitioned so that parts of it can be distributed to separate processing elements (e.g., see [16]). The advantage of this approach is that type safety is automatically retained since the normal compile-time checks are performed.

A second form of distribution is to have multiple programs that run on separate computers, with perhaps more than one Ada program running on a given computer. In this case, a standard interface for inter-program communication and synchronization is needed. Many systems today use this model of distribution.

It is beyond the scope of Requirement R8.1-A(1) to deal with the distribution of a single program across heterogeneous systems. The problems introduced by multiple representations of data and by operating systems with incompatible semantics make it difficult to accommodate the normal Ada model that does not recognize such possible differences. In particular a problem arises in dealing with the system dependent definitions introduced in multiple versions of package SYSTEM.

The semantics of entry calls and subprogram calls are oriented towards non-distributed execution in that no provision is made for the existence of multiple clocks or for the possible failure of remote computers or communication links. These problems must be addressed if Ada 9X is to directly support distributed application programming.

8.2 Dynamic Reconfiguration of Distributed Systems

User Need U8.2-A: Configuring an Ada Application: It must be possible to control the allocation of code to the computers upon which an Ada application executes. This control includes specifying both the initial configuration and subsequent configurations so as to optimize the use of available computing resources.

Requirement R8.2-A(1) — Dynamic Reconfiguration: Ada 9X shall allow for the possibility of dynamic reconfiguration of a distributed applications. It shall be possible to replace or modify individual components of a distributed application without recompiling or restarting the entire application.

Discussion: The fundamental model of Ada 83 execution is to bind together a single large program, elaborate its library packages, and then elaborate the program. This model is somewhat at odds with the requirement to be able to dynamically reconfigure parts of an application while it

is running. Issues to be considered here are the pre-elaboration of packages, and also the possibility of state-free packages for describing interfaces between components of an application (e.g., see [16]).

A minimal solution to this requirement might be simply to rephrase certain of Ada's rules to allow for the kinds of reconfiguration activities that are, in practice, being supported today.

8: Requirements for Distributed Processing

9 Requirements for Safety-Critical and Trusted Applications

Safety-critical and trusted applications, including those that control aircraft, medical equipment, and nuclear reactors, often have special requirements arising from the need for predictable execution, certifiable code, and the enforcement of safe programming practices.

9.1 Predictability of Execution

User Need U9.1-A: Constraining the Possible Meanings of a Program: Developers of safety-critical and trusted applications need to be able to determine, by examination of program text, the precise effect of executing a program. When language rules allow implementations to make choices that affect program behavior, determining the behavior of a program is more difficult. Programmers of safety-critical and trusted applications need ways of controlling such choices or the ability to determine what choices will be made under given circumstances.

Study Topic S9.1-A(1) — Determining Implementation Choices: Wherever Ada 9X *explicitly* allows implementation-defined choices that affect program behavior, implementations shall be required either to document the choice that has been made (or the situations that control what choice is made) or Ada 9X shall provide a mechanism for controlling the choice.

Discussion: While Requirement R2.4-A(1) on page 11 calls for minimizing implementation dependences in Ada 9X, some implementation freedom is beneficial. Such freedom allows the language to accommodate a wide variety of target architectures and implementation techniques. For example, Ada today allows implementations to choose the range associated with the predefined INTEGER type, the parameter passing method to be used in subprogram and entry calls, and the value (if any) that is associated with uninitialized variables. Each of these implementation-dependent choices potentially affects the behavior of programs.

Given that some aspects of a program's behavior will continue to vary from one implementation to another, developers of safety-critical and trusted systems require at least the ability to predict how a program will behave under a *given* implementation. Some aspects of implementation-dependent behavior (such as the policy used for rounding real numbers midway between two whole numbers to integers) may be reportable in a document like Appendix F of the Ada 83 reference manual. Other aspects (such as the mechanism chosen for passing composite parameters) may have to be reported, upon request, on a case by case basis by the compiler. A safety-critical/trusted-system annex might specify constraints on such choices or standard methods for controlling some of them.

This requirement is stated as a Study Topic because it may not always be possible or reasonable to provide useful documentation or methods of controlling every explicitly allowed implementation choice. For example, Ada explicitly does not define the order in which subprogram arguments are evaluated. Study Topic S9.1-A(1) therefore says that the choice of a particular order of evaluation must either be documented or controlled. However, it could be argued that an optimizing implementation cannot usefully or easily document the algorithm it uses when evaluating the arguments of a subprogram call, and since programmers can easily avoid the use of arguments for which evaluation order has an effect, there may be little value in specifying a pragma that requires, say, left-to-right evaluation. Documenting or controlling the possible ef-

fects of erroneous executions may also be problematic in some cases. In short, although the principle of being able to determine the precise effect of program execution seems essential in safety-critical/trusted-system applications, it may prove to be difficult to ensure that all the necessary information and control is provided by Ada 9X. In any event, the generated code can always be examined to determine the precise effect of some portion of a program (see User Need 9.2-A, below).

Requirement R9.1-A(2) — Ensuring Canonical Application of Operations: Ada 9X shall provide a mechanism, applicable to a region of program text, for restricting any freedom otherwise allowed to reorder, replace, or remove actions involving predefined operations.

Discussion: The intent of this requirement is to allow a programmer to curtail the semantic effects currently allowed by section 11.6 of the standard. The currently allowed freedom to reorder operations, to use operations with increased numeric accuracy, and to suppress checks for predefined exceptions can be detrimental to the development of safety-critical and trusted systems because it enlarges the set of possible effects of program execution, thereby complicating formal or informal reasoning about program behavior.[8]

9.2 Certifiability

User Need U9.2-A: Validation of Generated Code: Developers of safety-critical and trusted systems do not rely on compilers to generate code that correctly reflects the intent of a high-level-language programmer. Moreover, the complexity of translating any reasonably high-level language into object code makes it unlikely that a program performing that translation could ever be completely relied on. It is common practice when writing safety-critical and trusted software in a high-level language to certify the correctness of the generated object code rather than that of the source code, treating the source code as documentation of the object code. Those who certify safety-critical and trusted systems by examining the generated code require a clear correspondence between the source code and the generated code.

Requirement R9.2-A(1) — Generating Easily Checked Code: Ada 9X shall provide a mechanism for advising a compiler that code should be generated in a style that allows it to be checked against the source text with reasonable effort.

Discussion: The style of code generated by a compiler is not a meaningful concept in the definition of a programming language. Therefore, there is no appropriate way to specify what it means for a compiler to obey advice about the style of generated code — obedience cannot be enforced. In this respect, the mechanism called for by Requirement R9.2-A(1) is like the Ada 83 pragma OPTIMIZE, which allows a programmer to indicate whether the preferred criterion for generating code is optimization of execution time or optimization of space. In essence, Requirement R9.2-A(1) identifies a third criterion — clarity of the relationship of the generated code to the source code.

[8] See AI-00315 for a discussion of some effects of section 11.6 that can be quite surprising to programmers.

There are legitimate reasons why a compiler might ignore such a request. For example, implementations targeted to RISC architectures must schedule instructions in unintuitive orders to exploit the full power of the machine. Even compilers for conventional architectures may be so strongly oriented towards the production of efficient code that optimization algorithms are not easily separated from normal code generation. On the other hand, the notion of what constitutes a clear relationship to source code can be affected by the tools that are available for analyzing this relationship. For example, a tool could reorganize the generated code to help display its relationship to the source text.

In the future, safety-critical and trusted software may be certified by other means, such as by validating intermediate-level code whose translation into machine language is straightforward enough that a translation program could be certified as trustworthy, e.g., see [10, 26, 27]. Nonetheless, validation of object code is likely to remain a common practice for some time to come.

9.3 Enforcement of Safety-Critical Programming Practices

User Need U9.3-A: Restricting the Use of Ada Features: Use of certain language features is often incompatible with the development and certification of safety-critical and trusted software. Often a requirement is imposed that safety-critical and trusted software be written in a ‘‘safe subset’’ of a programming language.

Discussion: There are a variety of reasons for excluding the use of a feature from a safe subset. For example, a feature might be excluded because it has nondeterministic behavior and the effect of safety-critical and trusted code must be predictable. Exclusion of a feature from a ‘‘safe subset’’ does not suggest that a feature does not have legitimate uses in other applications or that it should be excluded entirely from Ada 9X. However, developers of safety-critical and trusted code must be able to ensure that such code is written entirely within a prescribed subset.

Requirement R9.3-A(1) — Allow Additional Compile-Time Restrictions: Ada 9X shall allow a mode in which a compiler enforces adherence to coding practices beyond those imposed by the rules of the language.

Discussion: Ada 83 intended to allow such freedom to implementations, but the emphasis on no subsets (which was intended only to prevent the appearance of implementations unable to support the full language) has led some implementers and users to conclude that Ada 83 does *not* permit such modes. The point of this requirement is to ensure that this confusion is resolved in Ada 9X.

This requirement has utility beyond its use for safety-critical and trusted applications. For example, program developers might wish to forbid use of certain implementation-defined pragmas and attributes to enhance program portability.

9: Requirements for Safety-Critical and Trusted Applications

10 Requirements for Information Systems

Information systems typically need to deal with character data in a flexible manner and with large currency amounts. In addition, many information systems applications use non-ASCII character sets (often EBCDIC), and these alternate character sets must be fully supported in Ada without loss of efficiency.

10.1 Handling Currency Quantities for Information Systems

User Need U10.1-A: Support for Currency Quantities: Information systems programs that deal with currency amounts need to perform exact arithmetic, with control over rounding. At least 18 digits of decimal precision are needed.

Requirement R10.1-A(1) — Decimal-Based Types: Ada 9X shall support fixed point types whose value of *small* is a power of ten. Such types shall have at least 18 decimal digits of precision. It shall be possible to ensure that intermediate results for expressions of such types are not computed or stored with extra precision, and that rounding is under programmer control.

Discussion: The current fixed point model in Ada does not meet this requirement in a number of respects (see [14]):

- It currently appears that implementations need not support non-binary values of *small* since most validated implementations do not do so. The reason for this lack of support is, at least in part, due to implementation issues stemming from Ada's accuracy requirements for numeric computations, particularly for operations on fixed point types with incompatible values of *small* and for conversions of fixed point values to integer and floating point types (e.g., see RR-0592).
- Ada fixed point is oriented to its use as a replacement for floating point, where exact results are not required. Consequently, the model for fixed point is close to that of floating point, whereas the arithmetic facility required for information systems is closer to scaled integer arithmetic. In particular, neither stored values nor intermediate results should be permitted with extra precision in this context. Extra range is of course permitted (actually required) for intermediate results.
- Deterministic semantics with control over rounding are required for currency computations as opposed to model number nondeterminism.

Study Topic S10.1-A(2) — Specification of Decimal Representation: Ada 9X information system implementations shall provide a mechanism for specifying decimal representations for decimal-based fixed point types.

Discussion: Information systems applications typically make heavy use of decimal representations (both 8-bit character and 4-bit packed decimal) for decimal scaled data. There are three reasons for supporting this feature in Ada:

- On many architectures used for information systems applications, the instruction set directly supports arithmetic using these representations. Ada 9X should facilitate use of this hardware.
- It commonly arises in this type of application that quantities are manipulated in contexts requiring external decimal representations (for example, for input/output), so

the internal use of decimal representations can be efficient even in the absence of hardware support.

- Interfacing to COBOL (including files generated by COBOL programs) and to other related tools, including most data base managers, requires support for decimal representations.

The issue of whether representations of this type should be usable throughout a program, or more restricted (concentrating on the interface requirements) needs to be evaluated by the Mapping/ Revision Team.

10.2 Compatibility with Other Character Sets

User Need U10.2-A: Alternate Character Set Support: Many information systems applications must interface smoothly with applications based on character sets other than ASCII, particularly EBCDIC.

Study Topic S10.2-A(1) — Alternate Character Set Support: Ada 9X information-system implementations shall provide for the extension of the set of graphic symbols to obtain input/output facilities and IMAGE attributes for alternate character sets (specifically, for EBCDIC) comparable in both functionality and performance with the features provided for the built-in character set.

Discussion: It is expected that this requirement will be satisfied by capabilities specified in an information systems annex. This requirement is also related to the international requirements described in Section 3.1 on page 13.

10.3 Interfacing with Data Base Systems

User Need U10.3-A: Interfacing Ada Programs to DBMSs: Information systems programs written in Ada must often interface with Data Base Management Systems (DBMSs). Such DBMSs are typically designed to interface with COBOL programs.

Study Topic S10.3-A(1) — Interfacing with Data Base Systems: It should be relatively easy to write Ada 9X programs that interface smoothly with DBMSs.

Discussion: The intent here is not to specify a standard for interfacing with data base management systems from Ada, but instead to ensure that it is not unduly difficult to specify such standards. For example, existing non-Ada DBMS interfaces assume that the data base manager can call a specified ‘‘trap’’ procedure in exception situations. If Ada 9X supported subprograms as objects, it would be easy to specify how such subprograms could be passed to the DBMS as a trap procedure (e.g., see [11] for a discussion of this and other issues).

10.4 Common Functions

User Need U10.4-A: Standard Data Manipulation: The manipulation of varying-length strings and ‘‘picture-based’’ editing of strings is commonly required in information systems applications.

Study Topic S10.4-A(1) — Varying-Length String Package: Ada 9X shall provide a standard varying-length string package.

Discussion: The manipulation of varying-length strings is common in information system applications as well as in other applications. If Ada 9X specified a standard package for performing such manipulations, some implementers would find it easier to support the specified capabilities with considerable efficiency, perhaps even generating optimized machine code. On the other hand, it is not easy to agree on what standard functionality should be provided, nor is it easy to agree on how varying-length strings should be represented, since different representations are appropriate depending on how the strings are to be used, performance requirements, etc. Therefore this requirement is classified as a Study Topic.

Study Topic S10.4-A(2) — String Manipulation Functions: Ada 9X shall provide a standard package of general string manipulation functions such as a function for finding a pattern in a string, a conversion function, and functions for creating formatted data in the style of ‘‘picture’’ formats as specified in languages such as COBOL and PL/I.

Discussion: The presentation of values in ‘‘edited’’ format is a common requirement in information systems processing. It does not seem reasonable to support such a capability directly in Ada 9X, given the goals of the 9X revision effort. However, a generic package of editing functions could be specified as part of an information systems annex in order to ensure standard support for such string processing needs in information systems applications.

10: Requirements for Information Systems

11 Requirements for Scientific and Mathematical Applications

11.1 Floating Point

User Need U11.1-A: Common Mathematical Functions: Users need standard support for common elementary functions such as sine, cosine, square root, logarithm, etc. Developers of numerical software also need standard primitive subprograms that allow them to develop portable implementations of scientific and mathematical software.

Requirement R11.1-A(1) — Standard Mathematics Packages: Ada 9X shall provide a standard package of elementary functions such as square root, logarithm, sine, cosine, etc. In addition, Ada 9X shall provide a standard interface to subprograms that are useful in developing portable implementations of mathematical functions.

Discussion: A standard package of elementary functions is currently being considered for adoption as an ISO standard [17]. It would be sensible for Ada 9X to incorporate this standard by reference (assuming that it becomes an ISO standard; if not, it should be incorporated directly). Similarly, a package of primitive functions and procedures [18] is beginning the ISO standardization process and could also be referenced by the Ada 9X standard.

User Need U11.1-B: Floating Point Support: Numerical analysts require predictable use of a machine's floating point hardware.

Study Topic S11.1-B(1) — Floating Point Facilities: Ada 9X shall require that the results of floating point operations be predictable from the documentation provided for any given implementation. Ada 9X shall also allow implementations to fully support the IEEE floating point standard [20].

Discussion: The Ada 83 formal model for floating point uses nondeterminism to allow implementations sufficient freedom to conform to almost all floating point architectures. Subsequent experience has indicated that it is preferable to require deterministic results that are well-specified for any given architecture, with the language providing a framework for, and constraints on, these specifications. The proposed ISO Language Compatible Arithmetic Standard (LCAS) is an example of such a framework.

In addition, Ada 83 makes it difficult to accommodate certain features of the IEEE-754 standard, such as output of -0.0 and input of infinities. There are few such constraints, and they should be easy to relax.

11.2 Representation of Arrays

User Need U11.2-A: Ordering of Array Components in Memory: Programmers of numerically intensive computing applications need to be able to determine how a compiler has chosen to allocate array components in memory (row-major order or column-major) for at least two reasons: (1) to foster locality of reference in paged and cached systems and (2) to ensure that when interfacing with subprograms written in FORTRAN or C, successive array components in memory correspond with the expected index values.

11: Requirements for Scientific and Mathematical Applications

Study Topic S11.2-A(1) — Array Representation: Ada 9X shall allow programmers to determine whether arrays are organized in row-major or column-major order.

References

1. *Ada 9X Project Plan.* Office of the Under Secretary of Defense for Acquisition, Washington, D.C. 20301, 1989.

2. *Ada 9X Project Requirements Workshop.* Office of the Under Secretary of Defense for Acquisition, Washington, D.C. 20301, June 1989.

3. *Ada 9X Project Revision Request Report.* Office of the Under Secretary of Defense for Acquisition, Washington, D.C. 20301, 1990.

4. *Ada Board's Recommended Ada 9X Strategy.* Office of the Under Secretary of Defense for Acquisition, Washington, D.C. 20301, 1988.

5. *Approved Ada Language Commentaries.* Published by Grebyn Corporation, Vienna, VA. See also the up-to-date files kept by the Ada Information Clearinghouse; these files are also available on the AJPO machine (ajpo.sei.cmu.edu).

6. Real-Time Tasking Semantics Working Group. "Asynchronous Communication Between Tasks". *Ada Letters X*, 4 (Spring 1990), 35-39. Proceedings of the Third International Workshop on Real-Time Ada Issues.

7. Baker, T. "Fixing Some Time-Related Problems in Ada". *Ada Letters X*, 4 (Spring 1990), 136-143. Proceedings of the Third International Workshop on Real-Time Ada Issues.

8. Baker, T. "Timing Issues Working Group". *Ada Letters X*, 4 (Spring 1990), 119-135. Proceedings of the Third International Workshop on Real-Time Ada Issues.

9. Brender, R. F. Character Set Issues for Ada 9X. Software Engineering Institute, October, 1989. Ada 9X Project Report.

10. Carré, B. A. and Jennings, T. J. SPARK — The SPADE Ada Kernel. University of Southampton, England, March, 1989.

11. Chastek, G., Graham, M. H., and Zelesnik, G. The SQL Ada Module Description Language — SAMeDL. Tech. Rept. SEI-90-TR-26, Software Engineering Institute, November, 1990.

12. Cohen, S. Ada Support for Software Reuse. Software Engineering Institute, October, 1990. Ada 9X Project Report.

13. Dewar, R. B. K. Shared Variables and Ada 9X Issues. Software Engineering Institute, January, 1990. Ada 9X Project Report.

14. Dewar, R. B. K. The Fixed-Point Facility in Ada. Software Engineering Institute, January, 1990. Ada 9X Project Report.

15. Fowler, K. J. A Study of Implementation-Dependent Pragmas and Attributes in Ada. Software Engineering Institute, November, 1989. Ada 9X Project Report.

16. Gargaro, A. B., Goldsack, S. J., Volz, R. A., and Wellings, A. J. A Proposal to Support Reliable Distributed Systems in Ada 9X. Tech. Rept. 90-10, Computer Science Department, Texas A&M University, June, 1990.

17. *Generic Package of Elementary Functions for Ada.* ISO-IEC/JTC1/SC22/WG9 (Ada) Numerics Rapporteur Group, 1990. Being submitted as a Proposed Standard.

18. *Generic Package of Primitive Functions for Ada.* ISO-IEC/JTC1/SC22/WG9 (Ada) Numerics Rapporteur Group, 1990. Being submitted as a Proposed Standard.

References

19. Ichbiah, J. D., Barnes, J. G. P., Firth, R. J., and Woodger, M.. *Rationale for the Design of the Ada Programming Language.* Honeywell Systems and Research Center, Minneapolis, MN, 1986.

20. *IEEE Standard for Floating Point Arithmetic.* ANSI/IEEE Standard 754-1985 edition, 1985.

21. *7-Bit Coded Character Set for Information Processing Interchange.* ISO 646-1973 edition, 1973.

22. *8-bit Single-Byte Coded Character Sets.* ISO 8859-1987 edition, 1987.

23. Real-Time Tasking Semantics Working Group. "Real-Time Performance Standard". *Ada Letters X*, 4 (Spring 1990), 40-45. Proceedings of the Third International Workshop on Real-Time Ada Issues.

24. Quiggle, T. J. "Ramifications of Re-introducing Asynchronous Exceptions". *Ada Letters X*, 4 (Spring 1990), 25-31. Proceedings of the Third International Workshop on Real-Time Ada Issues.

25. *Ada 9X Revision Issues.* Release 2 edition, Office of the Under Secretary of Defense for Acquisition, Washington, D.C. 20301, 1990. Ada 9X Project Report.

26. Smith, M. K., Craigen, D., and Saaltink, M. The nanoAVA Definition. Tech. Rept. 21, Computational Logic Inc., June, 1988.

27. Wichmann, B. A. Low-Ada: An Ada validation tool. Tech. Rept. NPL Report DITC 144/89, National Physical Laboratory, August, 1989.

Appendix A: Efficiency, Simplicity, and Consistency Issues

Section 2.2 requests simplifications to Ada 83 but does not spell out specific problems to be addressed. Examples of such problems are given in this Appendix. These examples are to be considered suggestive rather than exhaustive.

A.1 Efficiency of Executed Code

This section lists some examples of Ada rules that might be changed to make it easier to generate efficient object code.

A.1.1. Access to a Task Outside its Master

In Ada 83 there is only one situation in which a task can be accessed from outside its master. AI-00867 points out that this consequence of Ada's rules poses a burden on implementations without any functional benefit to users.

A.1.2. Null Ranges

The difference between the upper and lower bound of a null range can exceed one. This possibility increases the overhead of index checks, particularly for machines with hardware index checking based on the lower bound and the length.

A.2 Understandability

User Need 2.2-B on page 8 calls for simplifying rules that appear to be confusing or error-prone. This section lists some of the rules that should be considered for improvement.

A.2.1. Elaboration Order

Ada's requirements for library unit elaboration and rules restricting access to an entity before its elaboration contribute to the language's apparent complexity from a user's viewpoint. RI-4017 [25] provides a thorough analysis of the problems these rules are attempting to solve and the problems that have been experienced with the rules, and lays a basis for possible improvements.

A.2.2. Later Declarative Items

Ada 83's distinction between basic declarative items and later declarative items was motivated mainly to prevent mixing large and small textual items [19, pp. 19-20]. Despite this motivation, the rule has proven to be confusing to users and should be reviewed to see if the intended benefits are indeed worth the costs.

A.2.3. Visibility of Literals and Operations

Ada 83 provides facilities to control the visibility of declarations. However, the rules do not provide visibility of literals and operations in certain situations where the type itself is visible. It is a common programming error to believe that being able to declare an object of a type means the literals and infix operations of the type are directly visible.

A.2.4. Obsolete Optional Bodies

Section 10.3(5) of the standard requires that an obsolete unit be recompiled (unless the implementation can deduce that compilation is not required). If a package body that is optional becomes obsolete, because of compilation of either its specification or of some other unit it depends on, the compilation system must delete it from the library. If the body is required by the logic of the application (even though it is ‘‘optional’’ by Ada’s rules), this behavior can cause a hard-to-find bug.

A.2.5. OTHERS Clause in Aggregates

The rules for use of OTHERS in array aggregates cause great confusion among programmers. See RR-0029, RR-0571 and RR-0605 for examples of the problems.

A.3 Generality

Requirement R2.2-C(1) on page 8 calls for minimizing special-case restrictions or omissions in Ada 9X. This section gives examples of restrictions that might be relaxed as well as examples of omissions that might be remedied.

A.3.1. IMAGE and VALUE for Real Types

The IMAGE and VALUE attributes can currently only be applied to integer and to enumeration types. Allowing these attributes to be applied to real types seems a natural generalization to most users.

A.3.2. Exception Handlers in Accept Statements

Accept statements look very much like subprogram bodies to users, but an exception handler cannot be written as part of the syntax of an accept statement.

A.3.3. RANGE Attribute for Scalar Types

It seems quite natural to users that they should be allowed to apply the RANGE attribute to a scalar type.

A.3.4. Permit “raise ... when <condition>”

For consistency with the syntax of the **exit** statement, the syntax of Ada 9X could allow a similar structure for the raise statement.

A.3.5. STORAGE_SIZE for Task Objects

Ada 83 allows a storage size specification only for a task type, but it would be reasonable to allow it to be specified for a single task as well.

A.3.6. Explicit Type Conversions in Static Expressions

An explicit conversion of a static expression is not itself static. This can lead to unexpected programming problems (see RR-0099 for examples). This restriction was introduced into Ada because of a pathological case — if one writes INTEGER(1.5) as a choice in a case statement, the legality of the case statement will depend on an implementation’s decision about how to round 1.5. Given the problems described in RR-0099, it is probably worthwhile to revisit this decision.

A.3.7. Use of a Subprogram Name in its Specification

Section 8.3(16) of the standard forbids use of of the identifier that is a subprogram's simple name in the specification or generic instantiation of that subprogram. AI-00370 discusses the motivation for this rule, but even so, the result of the rule is to forbid generic instantiations that are in fact quite useful, e.g.:

```
with PACKAGE_1;
package PACKAGE_2 is
     procedure PROCEDURE_1 is new PACKAGE_1.PROCEDURE_1;  -- illegal
end PACKAGE_2;
```

The second use of PROCEDURE_1 is illegal.

A.3.8. Default Names for Generic Formal Parameters

Ada 83 permits a default name or expression for all generic formal parameters except formal types.

A.3.9. Ability to Redefine "="

Although Ada 83 apparently restricts the ability to redefine the equality operator, in fact, it is possible to declare an equality operator for any type. Given that this is possible, it might be reasonable to remove the current restriction on direct redeclaration of the equality operator.

A.3.10. Reading Out Parameters

In general, a variable can be read after it has been assigned a value, but this is not true for **out** parameters. There was a lengthy discussion concerning the validity of this restriction during the initial Ada design. It might be worthwhile to revisit this discussion to see if the motivation for this special-case rule is still considered sound.

A.3.11. Implicit Subtype Conversions

The lack of an implicit array subtype conversion in record and array aggregates is a continuing source of surprise to programmers (e.g., see RR-0734 and RR-0749). While it is not clear that this rule can be revised in an acceptably upward-compatible way, its revision should be considered.

A.3.12. Negative Literals in Loops

New users are surprised to discover that constructs such as

```
for N in -1 .. 1 loop
```

are illegal. This restriction exists because of a pathological overloading resolution case discovered during Ada's design. The rationale for this rule should be reconsidered to see if the rule can be relaxed.

A.3.13. Naming Syntactic Items

Two features of Ada 83 help to make Ada programs easier to comprehend: the optional name permitted on loop and block statements, and optional identifiers at the end of various constructs. It is particularly helpful that the latter (if present) are required to match the name of the unit.

Similar labeling of other syntactic constructs is often required by coding standards and seems consistent with Ada's existing rules, but is not allowed by the language. The Mapping/Revision Team should consider whether it is worthwhile to permit additional kinds of statements (**if**, **case** and **select**) to be labeled with matching end names. It would also be reasonable to permit matching end names for record declarations.

A.4 Usability of Ada

This section is concerned with small changes to Ada that will ease its use.

A.4.1. Completion of Subprogram Declarations

Ada 83 requires that the body for a subprogram specification be supplied by a subprogram body. However, it is often desirable to define the implementation of a subprogram by renaming another subprogram or by generic instantiation; doing so is not permitted in Ada 83. Although this restriction can be circumvented by placing the renaming or instantiation in a package specification, doing so reveals an implementation detail that should be concealed in the body. RR-0470 gives good arguments for why this capability is useful.

A.4.2. Completing Incomplete and Private Types by Subtype Declarations

Ada 83 requires that an incomplete type declaration or a private type declaration be completed by a type declaration. Relaxing this rule by permitting a subtype declaration as a completion will simplify coding.