2

Ada 9X Project Report

Ada 9X Project
Requirements Workshop

June 1989

Office of the Under Secretary of Defense for Acquisition

Washington, D. C. 20301

EXECUTIVE SUMMARY

During the week of 22 May 1989, the Ada 9X Project Requirements Workshop was held in Destin, Florida. The purpose of this workshop was to provide a forum for identifying and articulating revision requirements for the Ada language standard. Workshop recommendations will be further analyzed by the Ada 9X Project Requirements Team. Seventy people from DoD, industry and academia, representing eight countries, participated in five working groups. The working groups and the chairman for each were:

- Trusted Systems and Verification - John McHugh, Computational Logic, Inc.
- Software Engineering in the Large - Olivier Roubine, SYSECA Logiciel
- Real-Time Embedded Systems - Marlow Henne, Sverdrup Technology, TEAS Group
- Parallel/Distributed Systems - R. Kent Power, Boeing Military Airplanes
- Information Systems - Eugen Vasilescu, Grumman Data Systems

Chris Anderson, the Ada 9X Project Manager, presented an overview of the project which is sponsored by the Ada Joint Program Office (AJPO) and managed by the Ada 9X Project Office at the Air Force Armament Laboratory (AFATL), Eglin Air Force Base, Florida. The overall goal of the Ada 9X Project is to revise ANSI/MIL-STD-1815A to reflect current essential requirements with minimum negative impact and maximum positive impact to the Ada community. Ms. Anderson stressed the importance of international coordination during the Ada 9X revision process in order to retain Ada's multi-standard status. (KR)

Dr. David A. Fisher, author of STEELMAN, the original Ada requirements document, gave the keynote address and set the stage for the workshop. In the address, Dr. Fisher talked about the development of the original requirements, the requirements within Steelman that have not been met by the current language, features of Ada for which there were no requirements, and his views on how the Ada 9X process should proceed.

The working groups met for the next 3 1/2 days, giving brief progress reports at the end of each day. On the last day, each chair presented the results of his working group to all the workshop attendees. What follows is a summary of these presentations. A complete description of the discussions that took place in each working group and the final recommended requirements can be found in Sections 2 through 6 of this document. It should be noted that the recommendations of each group do not necessarily reflect the opinions of all the workshop attendees.

Trusted Systems and Verification

This working group focused on sources of unpredictable program behavior allowed by the current Ada definition and on solutions that would facilitate formal mathematical reasoning about program behavior. Applications of interest to this group included those that require high assurance of system security, safety, reliability, and other operational characteristics, where unpredictable software clearly cannot be tolerated. Two issues

☑
☐
☐
Codes
d/or
al
A-1

were so heavily debated that the Ada 9X Project Manager tasked the Institute for Defense Analyses (IDA) to organize two study teams: one on secondary standards and another on formal semantics.

Sources of unpredictable program behavior discussed included:

- Language constructs that were left undefined in the current standard – intentionally; by omission, incompleteness, or conflicting statements; or by the vagueness of English
- Nondeterminism, which allows multiple behaviors – e.g., tasking, implementation choices, and optimization
- Performance variability – dramatic differences across implementations
- Run-time environment and machine architecture variability – e.g., bindings to external run-time services, machine interfaces, and machine arithmetic

The following recommendations for solving these problems were made:

- Eliminate all arbitrary (and, therefore, unnecessary) uncertainties about the meaning of Ada 9X programs – identify and justify all elements of the language that permit unpredictable, nondeterministic, or implementation dependent program behavior
- Specify formal static semantics for Ada 9X that rigorously and unambiguously define the rules for legal programs – include this definition as part of the standard
- Develop formal dynamic semantics that rigorously and unambiguously define the run-time behavior of programs – plan to make this definition a part of future standards
- Require implementations to attempt to detect all cases of unsound programming – issue compilation warnings or raise exceptions at run time for incorrect order dependencies, aliasing, use of uninitialized variables, and unsynchronized use of shared variables
- Require complete documentation of all implementation choices and their effects on program behavior – allow programmers to specify desired implementation techniques and issue compilation warnings if they cannot be accommodated

Software Engineering in the Large

The Software Engineering in the Large Working Group concentrated on five topics: internationalization, program library issues, life cycle issues, reusability and portability. The discussion of internationalization resulted the following requirement:

- The language shall support the use of characters other than those in the ISO seven-bit coded character set (ISO standard 646) at least, as values of character and string, in character literals, and in comments. In particular, the language shall support at least the use of the ISO eight-bit single byte coded character set ISO standard 8859-1.

In the area of program library issues, the group felt it was important to leave open all possibilities of comprehensive project support environments and that the current language reference manual (LRM) was constraining and insufficient in this area. The discussion of life cycle issues resulted in the following requirements:

- It should be possible to define entities with the same external behavior and different implementations.
- It should be possible to define new entities whose behavior is adapted from that of existing ones by the addition or modification of properties or operations.
- When declaring a subprogram body, it should be possible to indicate when a subprogram is only an alternate name for an existing one with the same parameter modes and types.
- There should be no constraint on the elaboration of library units (and their bodies) other than the requirement to have such units elaborated (in a consistent order) before the first reference is made to them.
- The language should attempt to provide a mechanism allowing a user to (optionally) indicate when elaboration of a given unit must occur.
- There should be a mechanism whereby the visibility of certain library units may be restricted to a given set of program units.

The discussions on reusability resulted in the development of four requirements:

- The mechanism for generic parameters should be as uniform and comprehensive as possible.
- If the equality operation is defined on all component types of a composite type, then it should also be defined on the composite type.
- All cases of operation overloading should be treated uniformly.
- It should be possible to indicate which properties of a private type are visible and which are not.

The final issue discussed by the group was portability which resulted in the following requirements:

- Generic bodies should be treated exactly like other unit bodies for the purpose of separate compilation.
- The implicit use, in certain language constructs such as loop variables, array and string indices and exponentiation, of a type whose representation (and base type) may differ from one implementation to another is undesirable.

The following requirements were seen by the group as applicable to both reusability and portability:

- Non-uniformities in the treatment of generic units should be removed: in particular, all language constructs should have consistent rules whether used inside non-generic program units or inside generic units.
- The possibility of using the attribute 'BASE as a type indication (e.g., in type declarations) is desirable.

- Language rules regarding memory allocation strategies should be tightened.

Real-Time Embedded Systems

The purpose of the Real-Time Embedded Systems Session was to review existing issues and identify new issues related to using Ada for real-time embedded systems and generate requirements based on these issues. The group decided to initially review existing real-time issues that had been identified by other working groups and organizations (e.g., Ada Run Time Environment Working Group, Joint Integrated Avionics Working Group, etc.) and determine if they were valid issues for real-time embedded systems implemented in Ada. After generating a list of known Ada real-time issues, group members submitted new issues for discussion and potential consideration for language requirements.

The group placed existing and new real-time issues into one of 17 categories. Due to the amount of time and number of people in the working group, only a subset of new issues submitted were considered for developing requirements. The group submitted 9 requirements that addressed the following issues:

- interrupts - Two requirements were generated with respect to this topic: enable a task type to wait on different interrupts and provide a mechanism for backloging interrupts.
- exceptions - Two requirements were generated with respect to this topic: reliable capability to suppress checks (i.e. division_check, overflow_check, and elaboration_check): and; the ability to determine the cause of a storage_error(i.e. evaluation of an allocator fails or exhaustion of stack space).
- hardware-level bit manipulation - Many real-time applications require efficient operations on word-size bit vectors (e.g., testing bits in status words). There exists a need for a mechanism which supports operations such as shift, rotate, and finding the first bit efficiently for word length bit strings.
- fixed point - Fixed point additive operations do not have predictable results (due to arbitrary, non-exact representation of 'small). There exists a need for a capability of specifying fixed point model numbers exactly, for all rational numbers.
- shared memory with external agent(s) - When memory is shared with an external agent (i.e. independent program, device register, etc.), there is no way to express the need for every read and write to occur (i.e. that the memory can be arbitrarily and independently changed or read).
- integer types - Ada integer types are inadequate for some real-time systems applications that use algorithms such as discrete Fast Fourier Transform (FFT), random number generators, etc. The group discussed supporting a type which does not cause numeric_error or constraint_error but wraps around.
- copy and reference parameters - The group discussed the need to explicitly specify both call-by-copy and call-by-reference semantics for parameters of subprograms. Without this capability, program behavior could be unpredictable.

Parallel/Distributed Systems

The Parallel/Distributed System Working Group concentrated on life cycle costs, maintainability, reuse, and portabily of systems implemented on distributed or parallel architectures. The overriding issue is whether projects in the future will have language support for implementing applications in Ada on distributed and parallel architectures, or whether project-unique, extra-lingual solutions will continue to be created, as is the current practice.

Nineteen different topics were discussed in this working group: twelve requirements were developed:

- The language shall not preclude the distribution of a single Ada program across a homogeneous distributed or parallel architecture.
- The language shall not preclude the distribution of a single Ada program across a heterogeneous parallel or distributed architecture.
- The language shall not preclude partitioning of a single Ada program in a distributed or parallel system.
- The language shall support the explicit management of the partitioning of a single Ada program.
- The language shall support the allocation of a partitioned single Ada program. The intent is to support both dynamic and static allocation.
- The language shall support fault tolerance and configurability.
- The language shall provide an exact definition of semantics, including failure semantics, for all types of rendezvous.
- The language shall:
    a. Not prohibit scheduling by context, which may be dynamic.
    b. Provide mechanisms for scheduling by multiple characteristics, including user-defined characteristics.
    c. Support different scheduling paradigms in different parts of the system.
- The language shall provide efficient mutual exclusion, and consistent semantics for such exclusion.
- The language shall provide explicit control for the location of objects and storage allocation for objects.
- In a distributed system, the language shall not require the same perception of time at all points in the system.
- The language shall:
    a. Allow a raised exception to be identified, regardless of whether the raised exception is in scope.
    b. Allow the immediate context of where a raised exception was raised to be identified.

Information Systems

The Information Systems Working Group focused on information systems, including command and control. The salient issues discussed included the lack of and need for secondary standards when interfacing with commercial products which require binding with other standards, and the lack of explicit language mechanisms for decimal numeric processing. The application domain, reference to Steelman and workarounds were discussed for each of these issues. Based on these discussions, the Information Systems Working Group developed eight requirements. These include:

- An open process shall be established for creating and maintaining secondary standards.
- Ada should support essential commonly used mathematical and statistical functions.
- The I/O in the Ada language should be improved and extended to include required capabilities.
- Ada shall interface or bind to existing or emerging standards.
- Ada should include exact decimal representation and associated operations.
- Ada should have the ability to specify binary coded decimal representation of fixed point types.
- There should be an Ada language mechanism for passing Ada subprograms and task entries to a non-Ada process.
- There should be a method to perform necessary processing when entities come into and go out of existence.

This workshop was one of many proposed sources of input that the Ada 9X Project Office plans to utilize during the requirements gathering phase. The results of this workshop will be analyzed by the Ada 9X Project Requirements Team over the next 12 months. The results of this workshop will also be presented at the first Ada 9X Project Public Forum on 30 June 1989.

TABLE OF CONTENTS

1. INTRODUCTION

This document provides an overview of the results of the Ada 9X Project Requirements Workshop. This section provides the background of the Ada 9X Project, summarizes the workshop process, discusses the impact of the workshop on the Ada 9X process, and summarizes the keynote address. The remainder of the document presents the reports from each of the working groups.

1.1 Background

In the 1970s, recognizing the need to better manage its software, the Department of Defense (DoD) sponsored the development of a single high order language, Ada. The Ada language became a DoD standard in December 1980, an American National Standards Institute (ANSI) standard in February 1983, and an International Standards Organization (ISO) standard in March 1987. Both DoD and ANSI procedures require that action be taken periodically to reaffirm, revise, or withdraw a standard. To meet these procedural requirements, the Ada Joint Program Office (AJPO) notified ANSI of its intent to revise the standard and established the Ada 9X Program Office at the Air Force Armament Laboratory, Eglin Air Force Base, in October 1988. As part of the revision phase, the Ada 9X Project Office sponsored the Ada 9X Project Requirements Workshop in Destin, Florida in May 1989. The results of this workshop constitute an important source of input for the requirements phase.

1.2 Purpose of Workshop

The purpose of the Ada 9X Project Requirements Workshop was to provide a forum for identifying and articulating revision requirements for the Ada language standard.

Workshop attendees were split into five working groups:

- Trusted Systems and Verification

  This working group focused on problems of unpredictable program behavior and the use of formal methods in assuring security, safety, and reliability in critical software systems. Issues included: undefined language features, nondeterminism, implementation variability, effects of optimization, unsound programming, formal semantics, and assertions.

- Software Engineering in the Large

  This working group focused on development and maintenance of large, complex systems.

- Real-Time Embedded Systems

  This working group focused on real-time embedded systems issues including, but not limited to, scheduling, timing, error recovery, and tasking.

- Distributed/Parallel Systems

  This working group focused on distributed/parallel systems issues including, but not limited to, shared variables, scheduling, tasking, and communications.

- Information Systems

  This working group focused on military and commercial information systems including command and control. Relevant issues included, but were not limited to, I/O, modeling and simulation, and business and scientific numerical processing.

Participation in the Ada 9X Project Requirements Workshop was by invitation. Interested individuals were selected based on their present and past experience in the Ada community and position statements which focused on significant issues regarding requirements for the revision of the standard. The DoD, industry, academia, and the international community were represented.

1.3 Impact of the Workshop

The Ada 9X Project requirements phase has been opened to receive the recommendations from many sources. The Ada 9X Project Requirements Workshop was one of the sources by which requirements for revision of the language will enter the requirements development process. Revision requirements were based upon consideration of Ada community problems presented during the workshop with contributions from a broad base of Ada users in Europe and the U.S. These recommended requirements will be analyzed further by the Ada 9X Project Requirements Team over the next 12 months. The evaluation of requirements from other sources will also be considered [e.g. revision requests from the public and findings from focused studies on formal methods, secondary standards and real-time technical issues]. As the Ada 9X Project Requirements Team develops the Ada 9X requirements, there will be periodic reviews by the Ada 9X Project Distinguished Reviewers, the Ada 9X Project Government Advisory Group, ISO WG9, Ada Europe, and the Ada community at large. In the spirit of this openness, the Ada 9X Project Public Forum on 30 June 1989 will present the results of the workshop.

1.4 Ada 9X Project Office Presentation

Ms. Christine Anderson, the Ada 9X Project Manager, presented an overview of the project. The overall goal of the Ada 9X Project is to revise ANSI/MIL-STD-1815A to reflect current essential requirements with minimum negative impact and maximum positive impact to the Ada community. The adoption of the revised standard by DoD, ANSI, ISO, and the National Institute for Standards and Technology (NIST) is the ultimate objective of the revision process.

The revised standard must be transitioned into usage. Transition includes updating the Ada Compiler Validation Capability (ACVC)/Ada Compiler Evaluation Capability (ACEC), formulating a transition policy, developing an education and training program, and developing a long-term language maintenance plan. Thus, the Ada 9X process is divided into three major phases: revision, standardization, and transition. These phases include activities associated with revision request collection, language revision, standardization, transition policy, education/training, compiler validation and long-term language maintenance.

Ms. Anderson went on to identify the various groups and organizations that would support the Ada 9X Project. A Government Advisory Group has been established with representatives from government organizations spanning a wide range of user interests. This group is responsible for reporting the technical issues and concerns of their organization, providing policy and procedural advice, reporting on the status of the Ada 9X Project to their organizations, and developing an Ada 9X Transition Plan. The Ada 9X Project Office is in the process of establishing a Requirements Team to develop the requirements and supporting justification based on the approved Ada commentaries and the Ada 9X Project revision requests. Ms. Anderson indicated that members of the Requirements Team will be announced in mid-June. Also, the Distinguished Reviewers, Ada experts representing various groups in the Ada community, will be announced in mid-June. Other teams planned include:

- Mapping Team - map the requirements into language specific issues with several recommended solutions
- Revision Team - make the actual changes to the ANSI-MIL-STD-1815A based on the Requirements and Mapping Documents
- Implementation Teams - modify existing validated Ada compilers to reflect the Ada 9X changes and provide feedback

Ms. Anderson concluded her talk be thanking each participant for their support of the Ada 9X Project and stressing the importance of the workshop to the Ada 9X Project requirements phase.

1.5 Keynote Presentation

Dr. David A. Fisher, one of the principal authors of STEELMAN, was the keynote speaker for the workshop. Since the purpose of the Ada 9X Project is to revise the language that was designed to meet STEELMAN requirements, Dr. Fisher provided a historical perspective on the process of developing these requirements and his current view of these original requirements and their implementation in the language standard undergoing revision.

The first major theme of the talk was that the original and driving motivation for the Ada Program has not been consistently understood; specifically, the motivation for DoD (and, indeed, for a larger community) to standardize on a particular programming language was essentially economic– that is, it is cheaper to obtain a certain level of quality if investments in tools and expertise are focused on a single high-level language (or, at most, a very small number) as opposed to diluting investment across a large number of high-level and assembly languages. Second, it was known that previous attempts to establish a common language within the DoD embedded applications failed primarily for political, rather than technical, reasons. Thus, the Ada process was structured to build the largest possible constituency while still maintaining the secondary goal and moral obligation to produce the best possible technical product. To this end, a process was developed that was very different from previous DoD language efforts and consisted of the following aspects:

- language requirements came from the user community;
- requirements were iteratively refined with reviews by the services, by vendors, and by the research community;

- the effort was supported by both the OSD and by the services;
- an open, top-down approach was used; and
- the design was constrained to use mature programming language mechanisms as opposed to mechanisms still in the research phase.

Dr. Fisher indicated that he believed that the political goal of building a constituency had been largely achieved by this process. He further indicated that the economic benefits that had been the primary motivation were now becoming concrete with the recent group of compilation systems, although these benefits are occurring substantially later than had originally been envisioned. Dr. Fisher believes that the strongest requirement for the Ada 9X project is to do nothing that will jeopardize fulfillment of the economic objectives.

One obvious way in which Ada could fail to meet its economic objectives would be if the intended users of the language perceived that there is a significant technical risk associated with its use. Having said this, Dr. Fisher then proceeded to discuss the various STEELMAN requirements not fulfilled by Ada and various Ada facilities that are not directly traceable to STEELMAN requirements. Seven requirements are not fulfilled by the current language:

- avoid unnecessary complexity,
- no arbitrary restrictions,
- warning messages,
- no restrictions on user types,
- marking shared variables,
- completely and unambiguously defined, and
- the language shall be formally defined.

Of these, the last five were among the topics discussed by the Trusted Systems and Verification Working group. The first topic on the list led rather naturally to an enumeration of Ada features that were not called out in STEELMAN:

- named parameters,
- default parameters,
- package private part,
- restrictions on with,
- subunits,
- derived types,
- use and use visibility rules, and
- static expressions.

Dr. Fisher believes that a number of these features add unnecessary complexity to the language and that this complexity accounts, in some measure, for the slower than expected construction of appropriate programming environments for the language.

Dr. Fisher closed his talk by reiterating his views on how the revision process should proceed. He felt that there is a strong need to preserve the DOD's investment by working within the current language structure and correcting obvious language deficiencies but without adding any major new features.

representation specification. The Ada 9X language definition cannot be expected to give precise semantics for this operation. The programmer must appeal to the specification of the peripheral hardware device to verify such a program. That is, in order to predict the effects of each use of an "unsafe" operation, details of its implementation by the specific compiler and target system, must be precisely known.

All sources of unintentional undefinedness are considered to be defects in the language definition. That such defects exist in a natural language definition should not be surprising.[1] Some of these defects have been resolved by the language maintenance committee (ISO WG9 Ada Rapporteur Group). Others appear to have been resolved less openly by compiler validation tests. For defects that remain, there are no posted warnings for programmers and no assurances that interpretations used in implementations will remain valid.

1. Application Domain: All applications, depending on the criticality of correct system operation and the degree of assurance required.

2. Reference to Steelman: Page 4, paragraph 1H. "Complete definition. The language shall be completely and unambiguously defined. To the extent that a formal definition assists in achieving the above goals (i.e., all of Section 1), the language shall be formally defined." Also, page 21, paragraph 13A, "Defining documents. The language shall have a complete and unambiguous defining document. It should be possible to predict the possible actions of any syntactically correct program from the language definition."

3. Current Workarounds: Although some of these problems have been resolved by approved Ada Issues that interpret the reference manual, many problems still remain. Programmers often resort to writing small test programs to determine how a feature is implemented, and hope their real programs continue to work the same way.

B. Nondeterminism. Nondeterminism refers to aspects of the language that allow several possible outcomes such as the choice made among several open alternatives in a select statement. Nondeterminism becomes a problem when the number of possible different results (observable external behavior and internal states) grows too large to reason about. Sources of nondeterminism discussed included:

- Intentional nondeterminism – namely tasking
- Implementation choices
- Effects of optimization

1. See for example: I.D. Hill, "Wouldn't it be nice if we could write computer programs in ordinary English – or would it?", *The Computer Bulletin*, June 1972, pp. 306-312.

In addition to tasking, all implementation dependent aspects of the language must be treated as sources of nondeterminism. For example, the current standard does not specify a particular order for evaluating procedure parameters nor for checking constraints on values returned. Since parameter evaluation may have side effects, and both parameter evaluation and returned value constraint checking may raise exceptions, programmers must anticipate all possible orders of evaluation. Optimizations, which allow reordering of operations and statements, add further to these complications.

1. Application Domain: All applications.

2. Reference to Steelman: Page 3, paragraph 1B. "Reliability. The language should aid the development of reliable programs." Also, paragraph 1E: "Simplicity. The language should not contain unnecessary complexity." and page 4, paragraph 1G: "Machine independence. The design of the language should strive for machine independence."

3. Current Workarounds: Relying completely on the characteristics of a particular compiler and run-time system allows one to reason about, test, and verify programs at a reasonable cost. Program life-span, portability, and maintainability are compromised, however, because any change in the compiler or run-time system could invalidate any of the assumptions made. Compiler developers are loath to document implementation choices and strategies because use of this information in applications would limit their freedom to change their strategies in improving their compilers.

C. Optimization. Allowable program optimizations include reordering the evaluation of subexpressions. A program may produce different results due to side effects and the order in which exceptions are detected. The uncertainty of the program state in optimized computations makes program behavior unpredictable and necessitates conservative recovery strategies.

Optimizations may produce unpredictable results even in simple programs. In Section 11.6 of the current standard the following legal optimization is described, where an expression is moved out of a loop:

```
declare
   N: INTEGER;
begin
   N := 0;                   -- (1)
   for J in 1..10 loop
      N := N + J**A(K);      -- A and K are global variables
   end loop;
   PUT(N);
exception
   when others => PUT("Some error arose");  PUT(N);
end;
```

The evaluation of A(K) may be performed before the assignment statement (1), even if this evaluation can raise an exception. This makes the initialization of N unpredictable and limits the actions that can be taken for recovery in the exception handler.

1. Application Domain: All applications.
2. Reference to Steelman: Page 19, paragraph 11F. "Optimization. ... Except for the amount of time and space required during execution, approximate values beyond the specified precision, the order in which exceptions are detected, and the occurrence of side effects within an expression, optimization shall not alter the semantics of correct programs."
3. Current Workarounds: One workaround is to evaluate subexpressions independently within "begin ... end" blocks to eliminate the uncertainty of which subexpression raises which exception. This approach is entirely unsatisfactory, however, because it introduces extra variables and clutters otherwise simple programs.

D. Performance variability. Language features such as procedure calls, task rendezvous, and storage management have been found to vary widely in time and space performance across implementations. These variations are not merely problems of poor implementation strategies. There appear to be no reasonably efficient implementations of these features for certain target environments.

1. Application Domain: All Applications.
2. Reference to Steelman: Page 3, paragraph 1D. "Efficiency. The language design should aid the production of efficient object programs. Constructs that have unexpectedly expensive implementations should be recognizable by translators and users."
3. Current Workarounds: Experimentation with constructs within a given compiler. Choice of compiler to suit particular implementation needs.

E. Run-time environment and machine architecture variability. Programs may depend excessively on characteristics of the target run-time environment and machine architecture, which significantly inhibits program portability and software reuse. Specific problems include:

- Bindings to external system services and software – e.g., GKS, POSIX, SQL
- Interrupts and special purpose I/O devices – Ada's representation specifications provide a mechanism for such interfaces. The semantics of such interfaces need better description in Ada 9X.
- Real-time clock – the concept of time in Ada is not well- defined. This manifests itself in the nature of delay statements and timed entry calls, the resolution of function CLOCK, and the difference between DURATION'SMALL and SYSTEM.TICK.

- Machine arithmetic – the definition of machine arithmetic in Ada is extremely imprecise and leads to variability in the results of arithmetic computations. Problems include the lack of precise rounding rules and rules for conversion between floating-point and fixed-point numbers.

1. Application Domain: All applications.
2. Reference to Steelman: Page 4, paragraph 1G: "Machine independence. The design of the language should strive for machine independence." Also, page 19, paragraph 11A: "Data representation.", and 11E: "Interface to other languages."
3. Current Workarounds: Run-time environment- and machine architecture-dependent programs.

F. Lack of a formal definition. There is no official, complete, consistent, and unambiguous definition of Ada that can

- Always give definitive answers to language questions, and
- Support rigorous mathematical analysis of program behavior.

Many questions about the rules and meaning of programs have been raised. Over 800 language questions have been formally submitted to the official language committee (ISO WG9 Ada Rapporteur Group) for interpretation.

Projects have requested waivers on the basis of the absence of a formal definition and verification tools for Ada. One argument that was overheard was: "We can't verify Ada programs, so we'll use C instead." And that was before there was an official standard for C, let alone a formal definition!

1. Application Domain: This problem relates primarily to life-critical and trusted applications for which formal analysis and mathematical proofs of program properties are essential.
2. Reference to Steelman: Page 4, paragraph 1H: "Complete definition. The language shall be completely and unambiguously defined." Also, page 21, paragraph 13A: "Defining documents. The language shall have a complete and unambiguous defining document."
3. Current Workarounds: Unofficial formal definitions of subsets of Ada have been defined and used for verification. This method is unsatisfactory because:
    - Subsets tend to be very restrictive – for example, they may assume that exceptions are never raised
    - Mechanisms for annotation differ
    - Assumptions made about language rules and semantics may not match those implemented by compilers

    A more common approach is to determine the behavior of particular implementations based on studying or testing compiled code. This is not very successful, however. Compilers do not always produce identical object code for particular constructs and none of this analysis extends to other implementations or even new releases of the same compiler.

G. Lack of assertions. Although assertions were required by Steelman, they were not included in Ada. This omission impedes the use of machine processable, formal specifications for communicating the intended meanings of programs in a precise way.

Ada programs typically do not completely convey the programmer's intentions or understanding of the problem. Formal annotations such as assertions provide a means to augment the program source code with machine processable information that has potential use, for example, in optimization, program verification, and timing analysis. For example, a programmer cannot adequately define the type EVEN_INTEGER in Ada, because there is no way to express or automatically check that all such values are indeed even. Assertions allow such intentions to be expressed in the program text.

1. Application Domain: All applications.
2. Reference to Steelman: Page 18, paragraph 10F: "Assertions. It shall be possible to include assertions in programs."
3. Current Workarounds: Common approaches include: writing assertions in comments, inserting explicit Boolean checks in programs, and using separate preprocessors to analyze and translate assertions.

III. Requirements and Justification

This section identifies the requirements for language changes recommended by this working group. For each recommendation, there is a statement of the requirement, a proposed language change for achieving it, arguments justifying the change, and any minority views that were expressed.

The thrust of these recommendations is to precisely define Ada 9X and to shift more responsibility for detecting program errors to compilers and run-time systems. The group felt that a formal definition of the language, if developed as an integral part of the language revision process, would facilitate the process and improve the quality of the end result. A formal definition would also help application developers – as well as language lawyers, compiler developers, and compiler validation testers – adapt to the revised language. The concerns of imprecise language definition and unpredictable program behavior were shared by all of the workshop participants and can be seen in other sections of this report.

A. Predictable program behavior.

1. Statement of Requirement: All sources of arbitrary and, therefore, unnecessary uncertainties about the meaning of Ada 9X programs shall be eliminated from the language. All elements of the language that permit unpredictable, nondeterministic, or implementation dependent program behavior shall be identified and justified.
2. Proposed Language Changes: The two principal changes required are to tighten up the language definition and to reduce the number and range of

allowable implementation choices. For implementation choices that remain, the language should provide either:

- A means for the programmer to specify the desired implementation technique, or
- A means for the compiler to tell the programmer what technique has been used.

3. Justification for Change: These changes will enable software developers to provide higher assurance of the safety, security, and reliability of the software they produce and the systems it controls.

4. Minority Views: None.

B. Documentation of implementation choices.

1. Statement of Requirement: Implementations shall fully document all implementation choices and their effects on program behavior. (Although documentation must remain consistent with the implementation, no implementation should be bound or expected to maintain the same choices in different products or in product updates or revisions.)

2. Proposed Language Changes: Implementation choices should be minimized per the requirement to reduce unpredictable program behavior.

3. Justification for Change: Full documentation of implementation choices will remove the guesswork in determining program behavior for any particular language implementation.

4. Minority Views: None.

C. Control of implementation techniques.

1. Statement of Requirement: The language shall allow specification of desired implementation techniques such as parameter passing mechanisms. Compilers must issue warnings if specified techniques cannot be accommodated.

2. Proposed Language Changes: The envisioned solution to this requirement is a form of representation specification for implementation techniques.

3. Justification for Change: This change would allow programmers to document, in a formal way, program dependencies on particular implementation techniques.

4. Minority Views: None.

D. Detection of unsound programming.

1. Statement of Requirement: Implementations shall attempt to detect all cases of unsound programming and issue compilation warnings or raise exceptions at run time for incorrect order dependencies, aliasing, use of uninitialized or undefined variables, and unsynchronized use of shared

variables.

2. Proposed Language Changes: This requirement is intended to change a large class of program errors from "erroneous" to one of the following classifications:

   - Illegal – if the error can always be detected at compile time
   - Legal but raises an exception – if the error may not be detected at compile time but can always be detected at run time
   - Legal but ambiguous, with one of a small number of known possible effects – if the error may not be detected at all

   An example of an ambiguous program is one whose results depend on the implementation's parameter passing mechanism. Ambiguous programs are clearly undesireable but their behavior can be predicted. This would be a significant improvement over the current state where the effects of erroneous programs are completely undefined. This change will only affect erroneous programs. Standard exceptions for errors detected during program execution will be needed.

3. Justification for Change: Leaving the recognition and avoidance of unsound programming practices up to programmers, code reviews, and program testing is not sufficiently reliable for high-assurance systems. An inherently safer language is needed for such applications.

   In general, the detection of many of these errors is undecidable. In practice, however, many common errors can be detected. Warnings of potential errors can also be reported. Of course, detecting errors at compile or link time is preferable to raising exceptions at run time.

4. Minority Views: None.

E. Predictable effects of optimization.

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

2. Proposed Language Changes: The language must restrict optimizations that reorder computations so that exceptions and side effects are not allowed to occur in an arbitrary order. The rules for when and where such variations can occur should be simple and straightforward.

   Special consideration may be required for implementations that target parallel or pipelined machine architectures where "hardware optimizations" violate tighter language rules. These special cases, however, should not compromise the safety of the language.

3. Justification for Change: The current language allows complete freedom in the order of expression and parameter evaluation, and arbitrary reorderings and code motion for optimization. The results of every

possible combination must be verified for high assurance applications, which is intractable for real programs.

4. Minority Views: None.

F. Formal definition of static semantics.

1. Statement of Requirement: The standard shall include a formal specification of the context-sensitive language rules, called the static semantics, that apply to all legal programs. Tools to support the use of this definition to answer questions about the legality or meaning of programs should accompany the definition to make it accessible to general Ada practitioners.

2. Proposed Language Changes: This recommendation does not require any language changes, although changes may be indicated in the process of defining the language rules formally.

3. Justification for Change: Of 111 approved Ada Issues, 63 were clarifications of the static semantics of the language. A formal definition would eliminate the problems of imprecision that arise in natural language definitions.

4. Minority Views: None.

G. Formal definition of dynamic semantics.

1. Statement of Requirement: Formal dynamic semantics that rigorously and unambiguously define the run-time behavior of programs should be developed. This definition should be included in future language standards.

2. Proposed Language Changes: This recommendation does not require any language changes, although changes may be indicated in the process of defining the behavior of programs formally.

3. Justification for Change: A formal definition of the language is necessary for formal verification of program properties.

4. Minority Views: None.

H. Assertions.

1. Statement of Requirement: No requirement for assertions in Ada 9X was advanced. Reasons included:

   - A useful assertion language must be considerably more powerful than Ada's Boolean expressions. For example, it must be able to express quantification over non-scalar values and over classes of objects. It must also be able to express relations among program components such as types, functions, and procedures that cannot be captured without major extensions.

   - The use of assertions for formal analysis and verification presumes the existence of a formal definition of the language's dynamic semantics.

Because the necessary language extensions were not easily bounded and because a complete formal dynamic semantics of Ada 9X is not expected to be produced immediately, the majority felt that a requirement for assertions in Ada 9X would be premature. The group unanimously agreed, however, that nothing should be placed in the language to preclude the later introduction of assertions.

2. Proposed Language Changes: None.

3. Justification for Change: N/A.

4. Minority Views: A minority opinion was that useful assertions can be expressed with only minor language extensions and that tools that translate such assertions into run-time checks (e.g., the Anna preprocessor) are already in use. The minority recommended that an Anna-like capability be added to the language.

IV. List of Working Group Members

— John McHugh, Chairman, Computational Logic, Inc. - USA
— Reginald Meeson, Facilitator, IDA - USA
— William Todd Barborek, Booz-Allen & Hamilton, Inc. - USA
— Paul M. Cohen, Martin Marietta Information and Communications Systems - USA
— William B. Easton, Peregrine Systems, Inc. - USA
— Clarence "Jay" Ferguson, National Computer Security Center - USA
— David Guaspari, Odyssey Research Associates - USA
— Franco Mazzanti, Instituto di Elaborazione dell'Informazione - Italy
— David Preston, IIT Research Center - USA
— Brian Siritzky, Multiflow Computer, Inc. - USA
— Michael K. Smith, Computational Logic, Inc. - USA
— Brian Wichmann, National Physical Laboratory - UK

3. SOFTWARE ENGINEERING IN THE LARGE WORKING GROUP

I. Scope of the Working Group

The general concern of the group was the use of the language for the development and evolution of various kinds of systems, with an emphasis on large and complex system (e.g., online traffic control system or space platform muti-function system) development and maintenance.

A. The areas selected for consideration were:

- Internationalization: The ability of the language to support the development of applications by and for non-English language users.
- Life-cycle of Large Systems: The ability of the language to effectively support the development and maintenance of large and complex systems; this includes the ability to express designs arrived at by modern design methods, the ability to support various life-cycle approaches to the development of software (including the traditional "waterfall" model as well as more incremental approaches); the ability to support and facilitate the maintenance and evolution of complex systems, including long-lived systems with stringent operational requirements; and, more generally, the ability to provide substantial productivity improvements over the complete software life-cycle.
- Program library: The notion of the program library and, more generally, the rules governing separate compilation play an important role in the development of complex software systems, and may also affect the way software components can be reused.
- Reusability: The ability of the language to support both the development of reusable software components, and their effective reuse in the largest number of cases.
- Portability: This area covers language constructs and rules (or lack thereof) that may hinder portability; portability issues are considered both from the program point of view (i.e. will the program execute correctly on a different target?) and from the project point of view (i.e. how easy is it to migrate to a different compilation system?).

The issues that were excluded from consideration by this working group were:

1. non-portability stemming from ambiguities or permissiveness in the language definition (covered in depth in the Trusted Systems/Verification Working Group);
2. language support for developing bindings to other standards (addressed by the Information Systems Working Group); and,
3. needs for secondary standards (also addressed by the Information Systems Working Group).

B. The five subject areas selected by the working group as being within their domain were further examined to identify the salient issues within each area. The major internationalization issue is that the language must be able to support applications that manipulate characters other than ASCII: currently, character and string values and literals, as well as TEXT_IO, are defined only for the type CHARACTER. A second issue in internationalization is the relationship between the canonical sorting order of character code and the "natural" sorting order in various alphabets. Issues related to the development and maintenance of large systems concern shortcomings in the language that may result in other languages being preferred to Ada, especially as new industrial languages have now emerged that seem to be gaining rapidly in popularity (e.g., C++). Similarly, it can be argued, based on the flurry of recent publications in this area, that many developers would compensate for perceived deficiences by developing inadequate and non-standard extensions. Four major issues identified in this area were:

- the package concept alone is insufficient to express complex designs, where a system consists of several subsystems, each one made of a large number of packages;
- the language makes it very difficult to write software systems with requirements for continuous operation;
- maintenance of existing systems may require a large number of modifications; which allow for more flexible approaches;
- current language rules make it difficult or illegal to take advantage of certain architectures, in particular where these support alternative binding strategies.

C. The general issue for a program library is the simplistic approach to the notion of program library presented in the Ada LRM (Language Reference Manual), which ignores the general context of large system developments (in this particular case, large meaning more than a handful of compilation units, and/or more than just one programmer). Most compiler vendors provide additional functionalities, but in different and often incompatible ways, so as to make project portability even harder. A secondary issue is the extreme difficulty that one would have to distribute software components other than in source form, especially to a different vendor environment. Although the language is quite good at supporting the development of reusable software components, especially through the generic facilities, certain restrictions have been identified as limiting the potential for exploiting such generic units. These restrictions fall generally under the category of non-uniformity, in that any difference in the treatment of similar features often leads to difficulties in reusing some code in conjunction with such features. The two main issues identified under portability are:

- the freedom left to implementations to impose limitations on the separate compilation of generic bodies and generic instantiations (although this is a specific example of an undesired "implementation freedom", this one is seen as affecting portability without affecting the semantics of the program, and has therefore been retained for consideration here rather than deferred to the Trusted Systems/Verification Working Group);

- the pervasiveness of the predefined type INTEGER.

With a few exceptions, most of the issues addressed do not correspond to situations where the current version of the language makes it impossible to write a program; however, most of the cases correspond to opportunities to improve the economics of large system development, potentially by large factors. It is therefore felt that the positive impacts of the recommended evolutions in this area must not be judged solely on technical terms, but, more importantly, in terms of economical payoffs which should not be discounted or ignored.

Although some possible language modifications are indicated, they are given more as a clarification of the intent of the stated requirements than as a recommendation for a definite technical evolution. In certain cases, requirements are only expressed as desires for improvement: the group could not make any specific recommendation as to what language change would represent an improvement.

More generally, it was felt that the major purpose of this exercise was to provide useful input to the requirements team, for them to understand the concerns shared by a certain category of users, and to rely on their skills and competence to take these views into consideration to produce a consistent set of precise requirements for consideration during the requirements mapping phase. To all of them we wish the best of luck.

II. Specific Problems

This section discusses each specific problem, the application domain effected by the problem, references the STEELMAN requirements, and presents workaround solutions, when applicable.

A. **Support for Multinational Character Sets**: Characters other that the ISO seven-bit coded character set are not allowed in programs; more precisely, extended character sets can be used in user-defined types, but in this case, they cannot appear with their desired representation as character or string literals, and the predefined package TEXT_IO does not operate on them.

It is important to note that failure to address this need may cause Ada 9X to be unsuitable for acceptance as an ISO standard.

1. Application domain: Any application that must input or output characters in a language other than English; more generally, any program intended to be read and exploited by non-English speaking personnel. In such cases, there are needs to:

   - manipulate character and string values (including literals) that are meaningful in other languages;

- perform input and output operations on such characters and *strings*;
- insert comments written in other languages;
- write programs that are meaningful to non-English speaking nationals, e.g., by allowing foreign characters in identifiers, or even an alternate representation of keywords;
- perform comparisons and ordering operations that are meaningful in a given alphabet.

One can also mention the special case of applications for multinational use, where different character sets must be used simultaneously.

2. Reference to Steelman: The current limitation can be traced directly to *Steelman 2A:*

"Every source program shall also have a representation that uses only the following 55 character subset of the ASCII graphics ... Each additional graphic (i.e., one in the full set but not the the 55 character set ...)"

One can point out that the requirement for program representation expressed in the 55 character set was placed out of concern for compatibility with TTY model 33 terminals, which many museums would be now proud to own.

3. Current Workaround: Only a very partial workaround is possible, which consists in defining a new type, together with its input/output operations. Such a type would nevertheless not allow foreign characters in character and string literals, so it would be generally unacceptable.

B. **Support for Life Cycle of Large Systems**: Five specific problems associated with software development and maintenance life cycle phases of large systems were addressed by this working group as potential sources of requirements for language revision.

B.1 *Support for the Design of Large Systems:* When designing the architecture of large software systems, a number of methods follow a hierarchical approach, where a system is first decomposed into higher-level components (or subsystems), each one to be later refined in smaller units. It is generally desirable to introduce the abstract interfaces of higher-level components, for use by other subsystems, without concern for the later refinements. Even though the interfaces *will eventually be partitioned into smaller components*, one would like to retain in the code the original structure of the hierarchical design.

It is currently somewhat difficult to map this hierarchical structure into Ada units: several alternatives are possible, none too satisfactory. A first possibility is to break the higher-level specification into smaller ones, each one corresponding to individual package specifications; in this case, the resulting code no longer reflects the hierarchical design, making maintenance more difficult. A second possibility is to introduce package specifications that are nested within a top-level package; in this case, an extra level of naming is introduced, and, more importantly, the possibility of separately compiling the nested package specifications has been lost. Finally, one can introduce library units corresponding to the lower-level decomposition, and reference them in the body of the higher-level component, thereby losing the restrictions on access by other units that are gained by the use of a package body.

1. Application Domain: This problem is of concern to all the developers of large systems, and especially those developed in conjunction with hierarchical design methodologies.

2. Reference to Steelman: None.

3. Current Workaround: Although it is always possible to write the desired code in the end, some aspect (be it readability, flexibility, or safety) will be compromised. Currently, the preferred approach would be to use library units to represent the lower-level decomposition, and to rely on an outside tool to enforce visibility control on these units. However, this approach leads to an additional problem, in that one would still like to introduce packages to represent the higher-level subsystems, whose declarations and operations should really be implemented as namesakes for those in the lower units: since such a mapping is to be introduced only in the body, it cannot be provided through renaming declarations, and may thus lead to unnecessary indirect calls, with a possible loss in efficiency (the pragma INLINE is not necessarily a panacea in this respect).

B.2 *Systems with Requirements for Continuous Operation and Systems with Multiple Redundancy:* Ada is currently very far from providing adequate support for the implementation of systems with strong requirements for flexibility (see Application Domain). Such systems may offer different ways to execute a given request with the choice of how to handle the request delayed until execution time. The general problem in such systems is that one cannot determine in advance which specific hardware or software component will be used to handle a given request, as components offering the same services can exist in different forms, either simultaneously to provide increased capacity or reliability, or over time, as the system is upgraded on- line.

1. Application Domain:

   - Systems with replication of components, where the same service can be provided indifferently by components with different internal properties; specific examples could be printing services on a large installation, or telephone switching systems where capacity is achieved by a large replication of possibly dissimilar hardware components. The problem in such cases is to be able to determine dynamically which specific component will actually provide the service.

- Systems with requirements for long-lived, continuous operation, such as telephone exchanges, airline reservation systems, nuclear plant and other industrial control systems, defense warning systems, space station software, etc.: such systems must be maintained, with components modified and new components added, without turning the system down to bring up another version. In certain situations, for economic or other reasons (weight, power consumption, ...), it may be undesirable to provide this facility through stand-by backup systems.

2. Reference to Steelman: None.

3. Current Workaround: Coding examples to circumvent these issues have been proposed in the literature (see for example;[2] [3] [4] the solutions rely however on very complex tasking interactions, sacrificing run-time efficiency, program readability, reliability and maintainability.

   It should also be noted that because of the elaboration rules for library units, an implementation that would allow delayed binding of library units would be illegal.

B.3 *Support for Incremental System Development:* Departing from traditional "waterfall" approaches to software development, incremental strategies consist in designing, building and testing a subset of the desired functionalities, and adding more functionalities gradually afterwards. Although there is no inherent impossibility to apply such a strategy with Ada, the language provides no particular support for it.

A language solution may be possible which substantially reduce the cost associated with the incremental software development approach. In particular, one can identify two kinds of such language features. On one hand, it may be very helpful to be able to execute "incomplete" programs, i.e., programs where some features may be declared, but never invoked, and for which no body has been supplied by the user. LRM 10.5(1) gives a very precise definition of the elaboration of library units and of their bodies; as a result, most implementations will reject attempts to link or execute incomplete programs. On the other hand, incremental approaches call for step-by-step developments, where new features are added gradually; languages that allow such additions to be expressed separately have proven to be far superior for such developments.

2. Data Abstraction in Programming Languages, by J. Bishop, Addison Wesley International Computer Science Series, 1986.

3. Designing Large Real-Time Systems with Ada, by K. Nielsel and K. Shumate, Multiscience Press, McGraw Hill, 1988.

4. Programming Large and Flexible Systems in Ada, by O. Roubine, in Proc. International Conference on Ada, Paris, May 1985, Cambridge University Press (Ada Companion Series).

1. Application Domain:
    - large systems
    - systems with unstable and evolving specifications.
2. Reference to Steelman: None.
3. Current Workaround: The only workaround is to write "null" bodies instead of the deferred ones. Tools could be provided to automatically generate such alternate bodies, but such tools do not seem to have reached an industrial stage yet. In any case, while tools may allow the testing of incomplete programs, they do not address the problem of the amount of modifications that have to be made to add new functionalities.

B.4 *Support for System Maintenance:* The costliest part of system maintenance is generally the addition of new, unanticipated functionalities to an operational system. Such an evolution generally implies:

- adding new data type definitions,
- adding more information to existing data types,
- adding new operations,
- modifying existing operations to adapt to alterations made to data type representation.

In the current version of the language, such evolution would often imply modifications to type declarations, addition of new operations in the package that contains a type declaration, and potential modifications to all units that use the affected type. Although this is generally taken as a fact of life, various newer language approaches would largely alleviate this task by allowing modifications or refinements of existing data types to be introduced in units other than those containing the original declarations.

1. Application Domain: Any system, in operational use, that has to be maintained.
2. Reference to Steelman: None
3. Current Workaround: Maintenance can be done (hopefully) but it will cost more!

B.5 *Support for Alternative System Architectures:* Certain machine or system architectures offer facilities that cannot be effectively exploited by Ada implementations. This is especially the case for systems that depart from the traditional compile-link-execute approach. Such systems can be:

- systems that permit dynamic binding of program fragments during execution (it should be noted that this technique has been known for quite a number of years, and will be found in newer versions of the UNIX(TM) operating system);

- systems combining ROM and RAM memory, where certain initialized data are stored in ROM and are accessed every time the program is executed, without being elaborated each time around;
- more generally, systems with persistent object stores.

LRM 10.5(1), which specifies exactly when library units are to be elaborated, is overly constraining in this respect.

1. Application Domain:
    - development environments,
    - embedded systems,
    - systems with persistent object stores.
2. Reference to Steelman: None.
3. Current Workaround: The only workaround known to the group would be to write those parts of the programs that must be elaborated earlier or later in other languages, and use a pragma INTERFACE.

C. **Program Library Functionalities:** The current description of the notion of program library (LRM 10.4) is unnecessarily simplistic and constraining. More specifically, the notion of a monolithic library file is incompatible with the actual needs of any non-trivial program development. Those needs include:

- the possibility of sharing units between different libraries,
- the possibility of managing different versions of the same unit,
- the availability of other operations on the library, such as importing a unit, or moving or deleting a unit,
- the possibility of obtaining information on the current structure and contents of a library.

In response to customer demands, most compiler vendors currently provide their own extensions to the notion of program library, in non-compatible ways. As a result, the cost of transporting a project from one environment to another is often dominated by the efforts that have to be spent to modify the project structure to adapt it to a different library organization.

Furthermore, in the more complex systems, different hardware targets are used together, requiring the simultaneous use of different compilation systems from different vendors. Incompatibilities in the library management of the various vendors introduce additional expenditures.

Finally, some implementations make it close to impossible to obtain information on the current structure of the program library in a way that can be easily exploited by programs, thus hindering the development of complementary tools and the integration in project support environments.

1. Application Domain:
   - any non-trivial program development effort
   - projects that must be ported to different development environments
   - projects that use heterogeneous hardware targets
   - software engineering tools and project support environments
2. Reference to Steelman: Steelman 12A is not very precise on the issue, but mentions:

   "The library shall be structured to allow entries to be associated with particular applications, projects, and users."

   This may be viewed as an indication that the library concept was intended to address the problem of large team developments.
3. Current Workaround: There is no easy workaround. A developer wishing to provide a vendor-independent environment would have to define his/her own library management system, duplicating that of the compiler, and to provide bridges to automatically generate the correct library structures for the different compilers.

D. **Reusability**

D.1 *Anomaly in the Separate Compilation of Subunits:* Subunits of the same compilation unit must have different simple names (LRM 10.2(5)). As a result, if overloaded subprograms are used in a package, only one can be separately compiled.

This restriction is probably aimed at simplifying the implementation, but it places an undesired burden on the developer, with the result of making overloading often impractical.

1. Application Domain: Any application.
2. Reference to Steelman: None.
3. Current Workaround: Do not use overloading (or do not compile separately overloaded subprograms).

D.2 *Lack of Generality in Generic Parameters:* Generic units constitute the foremost language construct in support of reusability. However, it is felt that the current rules for what can be a generic parameter are somewhat restrictive. Specific examples include:

- impossibility to have an entry as a generic formal parameter: although an entry may be passed as a procedure, no conditional or timed entry call can be written inside the generic body;
- impossibility to have task types as generic formal parameters (other than passing them as limited private types);

- impossibility to have generic units be themselves generic formal parameters: such a feature would be useful to allow composition of abstract type constructors, such as being able to obtain a list of stacks (of anything) out of a generic stack package and a generic list package;
- possibilities of illegal instantiations of a generic that has been successfully compiled: such a situation could happen, for instance, if a generic body contains a declaration of a variable of a generic formal type parameter that has been declared as private, and if an instantiation uses as an actual parameter for this type a record type with discriminants with no default value. It is interesting to note that if the instantiation is compiled before the body, then one would expect the compilation of the body to report an error.
- Impossibility to pass exceptions as parameters to generic units.

1. Application Domain:
   - large programs,
   - reusable components.
2. Reference to Steelman The current limitations go against Steelman 12D:

   "An actual generic parameter may be any defined identifier (including those for variables, functions, procedures, processes, and types), or the value of any expression."

   In the context of Ada, this requirement would also call for packages, generic units, exceptions, to be allowed as parameters ("any defined identifier").
3. Current Workaround: In general, there is no easy workaround. In certain cases, it may be possible to write different generic units to provide the same facilities with different forms of parameters (e.g., one version with a private formal type parameter, and one with a limited private type instead).

D.3 *Restrictions on the Manipulation of Private Types:* Private types constitute an appealing feature to define abstract data types, both in reusable components, and to be used in conjunction with reusable components. However, the only control that the programmer has on what properties are made visible is the choice of making a private type limited or not.

Firstly, this is seen as too restrictive, as it does not allow one, for example, to indicate that a type is numeric (thus allowing the visibility of numeric constants). Secondly, the coupling between assignment and equality is often very constraining: in many cases, assignment may make perfect sense, but not equality (see in particular the TEXTHANDLER package in LRM 7.6), or conversely. When equality would make sense (from a logical point of view), it is possible to redefine it, but this is no for assignment. A number of special rules apply to the redefinition o ality: for example, it does not automatically extend to composite types, or it induces an implicit redefinition of inequality.

Finally, it can be observed that there is a need for a perfect symmetry between the expression of properties of a private type that can be made visible outside a package, and the properties of a generic formal type parameter that can be imported within a generic unit.

1. Application Domain:
   - reusable software,
   - any program.
2. Reference to Steelman Steelman 3.5B states:

   "In particular, it shall be possible to prevent external reference to any declaration within the encapsulation, including automatically defined type conversions and equality."

   With a strict interpretation, this requirement is met in the current version of the language. However, it is not clear that the intent was to call for an "all or nothing" situation.
3. Current Workaround: In most cases, the workaround consists in defining explicit operations corresponding to all the properties that must be exported. This is generally workable in the case of operations, except that subprograms introduced in the place of attributes (e.g., 'SUCC, 'PRED) cannot be used consistently (with private types) in generics.

   This approach leads to the introduction of many new subprograms that are new names for existing properties of a type, but renaming declarations cannot be used because the properties of the private type are not visible; the solution would be to allow renaming declarations to appear where a body is expected, instead of where the specification is expected.

   Of course, another workaround is not to use private types at all!

E. Portability

E.1 *Permissiveness in the Separate Compilation of Generic Units:* LRM 10.3(9) allows some flexibility in the implementation of separate compilation when generic units are involved; this permissiveness has been confirmed in Ada Issue AI 00408 ("Effect of compiling generic unit bodies separately"). As a result, some implementations may require generic bodies to be compiled together with their specification, or before the compilation of any instantiation, consequently forcing large amounts of recompilation during development.

This lax rule often means trouble when programs have to be ported from one compiler to the other, as files must be reorganized. It would be preferable to have a consistent rule enforced on all implementations, even if it corresponds to the most restrictive case (although the group unanimously felt that special rules should be removed, and generic units treated exactly like any other unit for the purpose of separate compilation).

1. Application Domain:
   - Systems which must be ported to different Ada implementations.
2. Reference to Steelman: None.
3. Current Workaround: This is not a blocking problem, as the various files constituting a program can always be reorganized; one can even use a tool that rebuilds a new compilation order for a given compiler.

E.2 *Non-Localization of Machine Dependencies:* Currently, machine dependencies, such as representation and address clauses, are spread throughout the code of the application, making it difficult and time-consuming to modify them when porting the program to another machine.

Life would be easier if such dependencies could be localized in specific units, or in specific sections of program units.

1. Application Domain:
   - Systems which must be used on different computer systems.
2. Reference to Steelman: None.
3. Current Workaround: The problem can be alleviated by the use of a good quality syntax-directed editor. Another way is to mark the machine dependencies with characteristic comments.

E.3 *Prevalent Use of the Type INTEGER:* It is well known that, for portability purpose, one should avoid using the predefined numeric types when declaring numeric types or subtypes in a program.

However, the predefined type INTEGER plays a special role

- with loop variables,
- with array indices (especially the pre-defined type STRING),
- with the exponentiation operator.

As a consequence, the unwary programmer may write correct programs that will be hard to port to implementations with a different representation for INTEGER.

1. Application Domain:
   - Systems which must be used on different computer systems.

2. Reference to Steelman: Steelman does not address the issue of predefined vs. user-defined numeric types.

3. Current Workaround: The obvious workaround would be to either declare loop variables with an explicit type, or use explicit conversions when loop variables, string indices or exponents are used, but is it realistic to expect programmers to do it? Without a "style checker" that could flag all such uses, the community will inevitably continue to write (unknowingly) non-portable programs.

E.4 *Non-Uniform Treatment of Language Constructs inside Generic Units:* Some language constructs cannot be used in the same way in non-generic program units and inside generic units, in particular when they depend on a generic formal type. For example

- attributes of a generic formal type can be used, but they are considered non-static. Consequently, it is impossible to declare a numeric type whose properties are based on those of the parameter.
- The expression governing a case statement cannot be of a generic formal type (LRM 5.4(3)).

Such non-uniformities cause difficulties when trying to make a unit reusable by transforming it into a generic, or when trying to exploit the parametrization.

1. Application Domain:
   - Reusable software.
2. Reference to Steelman: None
3. *Current Workaround:* None

III. Requirements and Justification

This section identifies the requirements for language change recommended by the working group. Minority views within this working group are also provided along with proposed language changes to help clarify the intended improvement. Generally, these requirements represent the consensus of this working group which may differ from the views of workshop attendees who were not members of this working group.

A. Support for Multinational Character Sets (Problem IIA)

1. Statement of Requirement:

   A.1 - "The language shall support the use of characters other than those in the ISO seven-bit coded character set (ISO standard 646) at least as values of character and string, in character literals, and in comments. In particular, the language shall support at least the use of the ISO eight-bit single byte coded character set ISO standard 8859-1 (Latin Alphabet 1). I/O operations shall be provided for all character sets supported."

A.2 - "There is no specific requirement for extended characters (i.e., those not in the ASCII character set, with their standard graphical representation) to be allowed in keywords and identifiers."

2. Proposed Language Changes: One way to fulfill requirement a is to change the predefined type CHARACTER to ISO standard 8859-1, together with corresponding evolutions of TEXT_IO, and with a canonical sorting order reflecting the internal codes used for these characters. This minimal solution is felt to be acceptable by the majority of the group. It does not preclude more comprehensive approaches such as, e.g., allowing user- definable character and string literals.

3. Justification for Change: Support for the ISO eight-bit coded character set(s) is likely to be required for acceptance of the language as an ISO standard, although the exact requirements placed by ISO should be tracked carefully. This requirement addresses problem II.A.

   It is felt that allowing extended characters in identifiers (let alone keywords) would be highly detrimental to the readability and "exportability" of programs, as some of the extended characters used may have a printable graphical representation in some countries, but not in some other, resulting in programs whose real meaning can be different from what the reader would see.

4. Minority Views:

   a. there should be a requirement for the language to allow user definitions of other character sets (including multiple-byte sets) and the use of those in character literals, string literals, and I/O, although such definitions may require specific actions from the user, such as, e.g., instantiations of generic units.

   b. Instead of requirement A.2, requirement A.1 should be modified to include extended characters in identifiers.

      (It can be noted that requirement A.1 uses the expression at least, potentially allowing more than single-byte coded character sets, or the use of extended characters in places other than comments, and character and string literals.)

B. Life Cycle Development and Maintenance (Problems identified in IIB)

The requirements for this area provide support for flexibility and maintainability of large systems. In the following, we use the term entity to designate any element of a program that is defined by an external behavior expressed in terms of visible properties and operations; an entity has an associated implementation (representation of data, and/or associated body). An entity may be a type, an object, a unit, a generic unit, etc..

B.1 - Support for Development and Maintenance of Large Systems

1. Statement of Requirement:

   B.1.1 - "It should be possible to define entities with the same external behavior and different implementations, and to use them interchangeably."

   B.1.2 - "It should be possible to define new entities whose behavior is adapted from that of existing ones by the addition or modification of properties or operations, in such a way that

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

2. Proposed Language Changes: It is fairly difficult to present here a comprehensive set of rules that would implement the above requirements. It is pointed out that specific inspiration may be found in modern, so-called object-oriented languages (although these generally provide a lot more than what is called for here).

   An example of a seemingly promising direction would be the introduction of package types (in a form similar to task types), and the possibility of introducing package subtypes that may contain additional operations (or may redeclare existing ones). This would essentially satisfy requirement B.1.2. A slight extension, the notion of a "virtual package type" (akin to the notion of virtual class in Eiffel) would fully meet requirement B.1.1 (a virtual package type consists of a package type specification for which no corresponding body is given, although subtypes of this type can be declared, each having a different body; no object of a virtual type can be created, but the type can be used in the declaration of formal parameters or of access types). Given facilities of this nature, the notion of subprograms as parameters (an often vocalized request) would only be a special case.

3. Justification for Change: Requirement B.1.1 addresses the problem of large systems with some form of replication, as well as the dynamic selection aspect necessary to solve the problem of upgrading systems through continuous operations.

   Requirement B.1.2 is felt to be an adequate answer to several of the problems identified , in particular

   - ease of maintenance,
   - support for incremental development

   The necessary technology has been developed in several existing languages (C++, Eiffel, CLOS, SmallTalk), and its effectiveness has been largely demonstrated.

The group did not feel the need for more complicated aspects of object-oriented programming, and in particular the dynamic resolution of operation designators, which adds more flexibility at the expense of readability and understandability, and often leads to less efficient implementations.

The current language features fall far short from providing the kind of functionality called for here: generic units allow multiple instances of entities with similar behavior, but not with different implementations; furthermore, different instantiations of the same generic unit cannot be used interchangeably.

Through derived types, it is possible to add new operations on a type, but not to change its representation, e.g., by adding new fields. In addition, objects of a derived type require an explicit conversion to be used as objects of the parent type, which may possibly be too restrictive.

Finally, it is felt that the economic benefits of such facilities are so significant that failure to meet these requirements may substantially reduce the audience of the language outside the U.S. DoD community.

B.2 - (Non) Requirements on the Elaboration of Library Units

1. Statement of Requirement:

   B.2.1 - "The language should place no constraint on the elaboration of library units (and their bodies), other than the requirement that a given program unit must have been elaborated before the execution of any reference to it."

   B.2.2 - "The language should attempt to provide a mechanism allowing a user to (optionally) indicate when elaboration of a given unit must occur."

2. Proposed Language Changes: Requirement B.2.1 can probably be met by modifying LRM 10.5(1) to replace the first sentence ("Before the execution of a main program, all library units needed by the main program are elaborated, as well as the corresponding unit bodies") by something less constraining.

   Requirement B.2.2 can probably be met already by the pragma ELABORATE.

3. Justification for Change: Requirement B.2.1 is intended to allow implementations to make full use of dynamic binding facilities that can be found in certain systems, or conversely, to allow library units to be elaborated once, although a program using them may be elaborated several times. This addresses problem II.F, and is also germane to two problems in II.B (i.e., systems with requirements for continuous operations and support for incremental development).

Requirement B.2.2 is a *reminder* that some users with a higher concern for precise control over the execution of a program may prefer to be able to force the elaboration to occur at predictable moments (this may in particular be the case in some classes of real-time systems).

B.3 - Renaming Declarations

1. Statement of Requirement:

   B.3.1 - "When declaring a subprogram body, it should be possible to indicate that that subprogram is only an alternate name for an existing one with the same parameter modes and types."

2. Proposed Language Changes: It seems sufficient (and it seems absolutely necessary) to allow renaming declarations to appear instead of a body, e.g., by changing the syntax in LRM 3.9(2) to

   body ::= properbody | bodystub | renamingdeclaration

3. Justification for Change: The intention is to allow the actual mapping of operations declared in a package specification to be deferred to the package body. This is believed to correspond to sound methodological practice, in particular in large system developments where implementation choices should be expressed independently of (and later than) the design of the interfaces. The feature would also be useful to export properties of a private type by renaming them as subprograms.

   From an implementation point of view, there seems to be no particular difficulty, as it is sufficient to make the two corresponding names (that of the new subprogram and that of the renamed one) aliased external references.

B.4 - Visibility Control on Library Units

1. Statement of Requirement:

   B.4.1 - "There is a need for a mechanism whereby the visibility of certain library units may be restricted to a specified set of program units."

2. Proposed Language Changes: This requirement need not necessarily be fulfilled by language constructs; library facilities might be an adequate answer.

   From a language standpoint, what is actually needed is a way to define "super-packages" that represent the interface of complex components, and that can be implemented by a set of Ada packages that can each be compiled separately whilst not being accessible directly from outer units.

3. Justification for Change: This requirement is intended to alleviate most of the problems encountered when designing large systems, using hierarchical methods (problem II.B).

C. Program Library Issues (Problems identified in II.C)

1. Statement of Requirement:

   C.1 - "The language shall not contain any specific indication implying a particular implementation of the library management system."

   C.2 - "The language should define all the aspects of the program library that may affect the order or necessity of compilations and recompilations, and only those aspects. Such aspects include in particular (without limitation) the effect of importing a unit in a library, or of removing a unit from a library."

   C.3 - "Implementations shall not prohibit other tools to be developed, that interact with the compilation system and program library."

   C.4 - "The language shall allow overloaded subprograms and operators declared in the same compilation unit to be compiled separately."

2. Proposed Language Changes: The major change implied by these requirements is the deletion of the notion of "library file" as defined in LRM 10.4(1 and 2). Requirement C.2 is a request for more details to be given in LRM 10.4(4).

   A preferred explanation would be in terms of a compilation context, referring to the compilation information concerning a consistent set of units, and that is made accessible to a compiler during the translation of another unit. The way in which such a context can be specified when invoking a translator need not be described.

   Within a context, all the units must be consistent, meaning that they reflect the dependency rules expressed in the language; also, for a given separately compiled program unit, there may be at most one unit in a context. This does not mean that a full-fledged program library must have such restrictions, but rather that it should be possible to extract an appropriate compilation context out of a program library.

3. Justification for Change: As pointed out in II.C, the simplistic view of the LRM does not correspond to the actual needs of the user, as evidenced by the fact that most industrial implementations provide additional mechanisms.

   Among the desired functionalities, a "decent" library management system should allow at least:

   - program libraries to be partitioned in smaller sets (not necessarily complete from the point of view of dependencies);
   - portions of program libraries to be shared between several libraries;

- program libraries that contain more than one version of a given unit.

Another need, expressed by requirement C.3 , corresponds to the openness that is expected from compiler vendors in order to allow users to integrate the compilers in comprehensive software engineering environments. A specific example is the coupling with a configuration management system. A minimal requirement is to have a programmatic interface to the program library, allowing tools to extract information (such as dependencies between units) stored in the program library.

Defining such a programmatic interface in a secondary standard was considered to present a very high potential benefit.

4. Minority Views: Whereas the above requirements correspond to minimal constraints on the language definition and on the implementors, it was felt by some that the language ought to be more precise in mandating some extra functionalities in the program library.

Minority View (a): "The language should define a minimal set of functionalities that must be supported by any implementation, although it should not specify how these functionalities are to be supported."

This minority view is only concerned with the fact that a minimal set of functionalities is necessary to make the language effectively usable in practice (it is clear that this working group was not concerned with the use of Ada for personal computing).

Minority View (b): "The language should define a minimal set of standard library mechanisms that must be supported by all implementations."

This minority view differs from the previous one in that it requires all compilers to provide the same functionalities in the same way.

A list of desired functionalities for a library management system should include facilities for

- structuring a library into collection of units (i.e., sublibraries);
- specifying, for each compilation, the relevant compilation context by selecting the appropriate sublibraries;
- creating and deleting sublibraries, or adding existing sublibraries to a library, in a way that preserves consistency (i.e., there should be no unit left in a sublibrary depending on a unit from a deleted sublibrary);
- providing complementary means for identifying units in addition to the Ada unit name, so as to allow multiple versions of a given Ada unit to exist simultaneously; the system should also provide for selecting the desired version when specifying a context;

- moving or copying units among sublibraries, and deleting units, while preserving consistency;
- sharing units or sublibraries between different libraries;
- obtaining information, including identification of library, sublibrary, *unit identification*, *compilation time*, identification of other units it depends on, list of dependent units, etc.; such information should be accessible by a program written in Ada as well as by the user;
- allowing controlled parallel access to a library.

D. Reusability

D.1 - Generic Units

1. Statement of Requirement:

   D.1 - "The *mechanisms for defining and passing generic parameters* should be as uniform as possible. In particular,

   - any kind of construct that can be declared in a program unit should be allowed as a generic parameter (this includes, e.g., exceptions, tasks, packages, entries, and other generic units);
   - any operation on a type (whether explicit or implicit) that can be exported from a package should also be allowed as a generic parameter."

2. Proposed Language Changes: There is no specific change that is being proposed, other than those directly implied by the requirement. It is indeed the intention that *all constructs should be treated on an equal* basis.

   From the point of view of which specific characteristics of a type should be allowed as generic parameters, an non-exhaustive list is given below:

   - *the fact that a type is numeric* (without having to indicate whether it is an integer, fixed-point or floating-point type), thus allowing arithmetic operations and numeric constants to be used;
   - the fact that a type is a record type, with a minimal set of fields (this may be used to provide through generics the added generality of structural type equivalence, without the related inconveniences).

   Whatever the properties, there seems to be an opportunity for a language unification between those properties of a type that can be passed to a generic unit, and those properties of a private type that can be made visible outside the enclosing package (see requirement p below).

3. Justification for Change: This requirement essentially addresses some of the reusability issues described in II.D (Anomaly in the separate compilation of subunits).

D.2 - Overloading of Predefined Operations

1. Statement of Requirement:

   D.2.1 - "All cases of operator overloading should obey exactly the same rules."

   D.2.2 - "If equality is defined on all components of a composite type, whether predefined or redefined, then equality should also be defined on the composite type, as the conjunction of the component-wise comparisons. (If, as a result of other requirements, it is also possible to redefine assignment, then the same rule should apply to component-wise assignment)."

2. Proposed Language Changes: Requirement D.2.1 is intended to lead to a unification, one possibility being to treat "=" like other operators: it can be overloaded, redefined, and should not necessarily imply redefinition of "/="; the alternate possibility would be to force the redefinition of relational operators, e.g., "<", to implicitly redefine the complementary operator (in this case ">=").

   Requirement D.2.2 calls for an obvious addition to the language.

3. Justification for Change: See problem II.D.3 (Restrictions on the Manipulation of Private Types).

D.3 - Properties of Private Types

1. Statement of Requirement:

   D.3 - "It should be possible to indicate (with a finer grain of control) which properties of a private type are visible to other units, and which are not."

2. Proposed Language Changes: There is no specific language change suggested; the kinds of properties that one would like to export have been discussed in II.J; they could include, e.g., numericness, discreteness (so that 'SUCC and 'PRED would be available), or the fact that a private type is nevertheless known to be a task (entries would not be visible, but abort could be).

   Examples of possible syntax could be

   ```
   type FOO is (<>);     -- discrete type
   task type FOO is limited private;     -- private task type
   ```

3. Justification for Change: Finer control over the properties of private types gives the possibility of writing generic units whose domain of

applicability is as large as possible, and to make such units usable in conjunction with the largest possible number of private types (see problem II.D.3. Restrictions on the Manipulation of Private Types).

E. Portability (Problems identified in IIE)

E.1 - Separate Compilation of Generic Units

1. Statement of Requirement:

   E.1 - "Generic bodies should be treated exactly like other unit bodies for the purpose of separate compilation:

   - it should always be possible to separately compile the bodies of generic units when they are library units, or when they are directly inside other compilation units;
   - there should be no extra dependence between a generic body and the units that contain instantiations of that generic;
   - optional dependencies may be introduced for the purpose of optimization, but the "non-optimized" case should always be supported."

2. Proposed Language Changes: The intended changes are directly formulated in the requirement.

3. Justification for Change: The special rule concerning separate compilation of generic bodies is seen as detrimental to portability, to the transformation of software components into generally reusable ones (i.e., making existing program fragments generic), and to the use of generics in the development of large systems (see II.E.1, Permissiveness in the Separate Compilation of Generic Units).

   It is realized that this new requirement places a certain burden on compiler writers, but the existence of some high-quality industrial implementations that have no specific restriction on the separate compilation of generics is seen as feasibility.

E.2 - Use of the Predefined Type INTEGER

1. Statement of Requirement:

   E.2 - "The implicit use, in certain language constructs such as loop variables, array and string indices and exponentiation, of a type whose representation (and base type) may differ from one implementation to another is undesirable."

2. Proposed Language Changes: This requirement has been intentionally stated in the least constraining manner possible.

   There are several ways to remedy this problem, including making the use of untyped expressions illegal, forcing the type INTEGER to have the same representation on all machines, or deciding to use another type (e.g., universal integer) in the cases listed above. Except for a minority view presented below, there is no particular feeling one way or another.

3. Justification for Change: See II.E.3 (Prevalent Use of Type INTEGER).

4. Minority View: "The language should define the set of predefined numeric types, together with their exact characteristics."

   This view considers that a 32-bit INTEGER is largely acceptable on all machines (the counter-argument being that this is probably true today, but not necessarily in ten years).

F. Miscellaneous Requirements for Improving Portability and Reusability

1. Statement of Requirement:

   F.1 - "Non-uniformities in the treatment of generic units should be removed: in particular, all language constructs should have consistent rules whether used inside non-generic program units or inside generic units."

   F.2 - "The possibility of using the attribute 'BASE as a type indication (e.g., in type declarations) is desirable."

   F.3 - "Language rules regarding memory allocation strategies should be tightened."

2. Proposed Language Changes: This set of requirements is only intended to indicate possibilities for improvements. It is difficult to suggest a language change without a broad look at all the implications (this comment actually holds for most of the language changes already suggested).

3. Justification for Change: All these requirements attempt at correcting language aspects that hinder portability of programs or the development of reusable software.

   Requirement F.1 addresses the problem described in II.E.4 (Non-Uniform Treatment of Language Constructs Inside Generic Units). The inconsistencies mentioned there may make it difficult to evolve a non-generic unit into a generic one for improved reusability.

   Requirement F.2 aims at improving portability among implementations where predefined types have different representations.

   Requirement F.3 is only an indication of the difficulties in making an effective use of dynamic memory allocation in a portable way: since it is generally not known whether an implementation effectively supports garbage collection or unchecked deallocation, it is fairly difficult to predict the behavior of programs that rely on one strategy or the other when transporting them to different installations.

IV. List of Working Group Members

— Olivier Roubine, Chairman, THOMSON/SYSECA - France
— Audrey Hook, Facilitator, IDA - USA
— Paul Baker, CTA - USA
— John Barnes, Alsys - UK
— Tom Bolick, Air Force, HQ AFSC - USA
— Phyllis Crill, Magnavox - USA
— Ernesto Guerrieri, SofTech - USA
— Stephan Heilbrunner, IABG - FRG
— Rich Hilliard, MITRE - USA
— Michael Holloway, NASA - USA
— Randal Leavitt, PRIOR - Canada
— Steven Litvintchouk, MITRE - USA
— Lennart Mansson, Telelogic - Sweden
— Bob Mathis, Contel - USA
— David Pogge, Navy, Naval Weapons Center - USA
— Mats Weber, Ecole Polytechnique Federale de Lausanne - Switzerland

4. REAL-TIME EMBEDDED SYSTEMS WORKING GROUP

I. Scope of the Working Group

Considering the extensive work by others in the real-time embedded systems domain and the limited amount of time for working group analysis, this working group focused on identifying problem areas that had not already been addressed for further work. The working group considered requirements provided from the following sources:

- Revision Requests already submitted to the Ada 9X Project Office
- Report of the Ada-Europe Ada 9X Working Group
- ARTEWG
- The Joint Integrated Avionics Working Group (JIAWG)

Each problem/revision request applicable to real-time embedded systems was discussed and cataloged into one of sixteen areas of concern. During the discussion, the working group added relevant issues to the catalog. After discussions by the whole group, individual subgroups were formed to further study one of eight major concern areas. As a result, the following specific problems (II A through II J) were refined and recorded.

It was interesting to note that the real-time embedded systems area seems to be more homogeneous than might be expected. After very active and sometimes animated discussions on approximately 80 issues, the whole working group reached generally unanimous opinions on the specific problems. Also the group was in general agreement with the reports from the Ada-Europe Ada 9X working group, the ARTEWG, and the JIAWG.

A. Domain of interest: Real-time mission critical embedded systems used in strategic or tactical military systems. Application domains include military systems in which both functional and temporal correctness are essential. A few examples of military systems that generally fit into this domain are: command and control , flight control, robotics, and space shuttle.

B. Salient Issues: Major issues include language deficiencies that inhibit the use of Ada for implementing military real-time embedded systems. Issues of interest are: pre-elaboration, memory control, scheduling, asynchronous control, synchronization methods, timing abstractions, run-time kernel interface, optimizations, exceptions, fixed point and unsigned arithmetic, hardware-level bit manipulation, and external interfaces.

II. Specific Problems

A. Storage Error Problem. There is no method in the language by which a program can determine whether STORAGE_ERROR is raised because insufficient space remains for an object created by an allocation or because insufficient space remains for a locally declared object, temporary value, or subprogram/entry call.

1. Application Domain: Real-time embedded systems that make heavy use of dynamic allocation.

2. Reference to Steelman: Paragraph 10.b requires that an error be detected when "requesting a resource (such as stack or heap storage) when an insufficient quantity remains." There is no clear requirement that lack of stack storage and lack of heap storage be distinguishable.

3. Current Workaround: There is no current execution-time workaround.

B. Reliably Turning off Runtime Checks. Some real-time and embedded applications need to turn off certain runtime checks, to guarantee that exceptions will not be raised by certain operations. The pragma SUPPRESS in Ada83 does not solve this problem, since it is allowed to have no effect. This is not acceptable.

Two kinds of situations where this feature is required have been reported. One of these involves numeric computations that produce formally "incorrect" but tolerable results, and the other involves violations of sequential elaboration order that are known to be safe.

Some very time-critical applications, such as signal processing, not only cannot afford the cost of runtime checks, but also cannot permit the cost of handling certain exceptions that might be detected by hardware. For example, transient hardware (e.g. sensor) faults may result in out-of-range data, which in turn may lead to numeric errors, such as overflow or division by zero. Such systems can be designed to tolerate occasional incorrect calculations or to check for output validity in a later less time-critical phase, but they cannot tolerate long delays such as would be imposed by pre-checking that the data cannot cause a check to fail, or raising an exception, handling it, and restarting the computation. The desired response to numeric and constraint errors may therefore be to continue with the computation.

This requirement could conceivably be met by changing the Ada exception recovery model to allow immediate resumption of a computation after an exception. However, the exception would also need to be very fast. It is probably sufficient to simply ignore the exception.

Another example of a situation where it may be necessary to ignore an apparent numeric error is in bit unpacking operations, which is typically found in real-time communications. The only efficient way to get the effect of a left-shift operation in Ada is multiplication, but then this can cause overflow. Such an overflow is not an error, since it is the programmer's intent to discard the high-order bits. Another example is where modular arithmetic is desired.

The second kind of check that may need to be suppressed for correct operation is elaboration-check. The basic reason for elaboration rules is to insure that references are not made to data objects before they have been allocated and initialized. They are based on the assumption that a subprogram or task may access any object that precedes the subprogram or task body in elaboration order. Clearly, this is always true.

This pessimistic assumption can be a problem with start-up (bootstrapping) of complex systems, especially in systems involving hardware device-drivers or where the division into packages is primarily for manageable components. It is very difficult, or perhaps impossible, to design such systems so that all packages can be elaborated in a linear order, without a subprogram or task ever being executed before the corresponding body is elaborated.

1. Application Domain: This is primarily of interest in the real-time embedded systems where execution time is critical.
2. Reference to Steelman: Steelman 10G states "it shall be possible during translation to suppress individually the execution time detection of exceptions within a given scope."
3. Current Workarounds: There are none within the Ada language.

C. Problems with Address Specifications. The intended effect of the address clause is ambiguous in Ada83; moreover since real-time embedded applications have several different address specification requirements, this one mechanism cannot satisfy them all, no matter how it is interpreted.

It is not clear whether the 'ADDRESS attribute of a subprogram returns the address of the first instruction to be executed (entry-point address), or the first (lowest) location in memory occupied by the subprogram. The use of an address to identify a source of interrupts is inconsistent with most machine architectures. Therefore supporting "address clauses" for entries requires an implementation to treat addresses used in this context as encodings of other types of information, or pointers to descriptors. In this way the use of address clauses to specify a binding between a source of hardware interrupts and a task entry is inconsistent with other uses of anything related to the entry.

1. Application Domain: These problems arise primarily in real-time systems embedded in or interfacing with other hardware or programs.
2. Reference to Steelman (and others): Steelman 11A partially states this requirement, but is too imprecise.
3. Current Workarounds: None.

D. Control over Reference vs. Copying in Parameters Transmission. Ada83 provides no way to predict or control whether subprogram parameters are passed by reference or by copying. Such control over parameter passing may be required for:

- interface to external subprograms that require call by reference or call by value semantics, and such that the information in the INTERFACE pragma is not sufficient to determine the parameter passing method (by reference or by copying);
- modifying an OUT or IN OUT parameter in an exception handler before reraising the exception (by reference);
- insuring that an OUT or IN OUT parameter is not changed when an exception is propagated from the subprogram (by copying);

- implementing operations on memory-mapped objects (by reference);
- protecting against aliasing (by copying);
- interface to atomic update operations, such as test-and-set and compare-and-swap, especially on shared memory multiprocessor machines.

1. Application Domain: This comes from the real-time embedded problem domain, but applies also to writing other programs where reliability and predictable behavior are important.
2. Reference to Steelman: Requirement 7F says IN OUT parameters "enable access and assignment throughout execution or only upon call and prior to any exit". Ambiguities exists as to whether the programmer should be allowed to specify which of these two alternatives is taken by the compiler.

   Requirement 7I requires avoiding aliasing. Ada83 not only fails to do this, but does not even permit a programmer to voluntarily avoid aliasing by passing parameters by copying (see item 5 above).
3. Current Workaround: There is no acceptable workaround. The two partial workarounds are implementation-dependent.

   With some compilers it may be possible to use the 'ADDRESS attribute and unchecked conversion between address and access types to obtain the effect of call by reference, but this is not defined by the Ada language.

   With some compilers it may be possible to achieve the effect of call by copying through the use of explicit local temporary variables. This is cumbersome, and if the compiler happens to pass parameters by copying anyway, is will be wasteful. There also seems to be some risk that a compiler may eliminate such variables through optimization, so that the effect could still be that of call-by-reference.

E. Task type interrupts. There is no way to make objects of a task type wait on different interrupts because the interrupt address clause is in the specification. Therefore, task types cannot be used for a set of interrupt handlers.

1. Applications Domain: Real-time embedded systems.
2. Reference to Steelman: None.
3. Current Workaround: Two approaches were discussed: 1) use of generics by passing an address (there was some questions about the feasibility of this approach); and 2) recursive generation of tasks (however, this approach was quite messy and unreliable).

F. Backlogging Interrupts. The LRM allows the implementor to interpret an interrupt as "an ordinary entry call, a timed entry call, or a conditional entrycall." This is a permission for an implementor to discard interrupts.

1. Application Domain: Real-time systems where some external entities communicate with the system by way of interrupts. The application systems include avionics, command and control, missile, shipboard systems, etc.

2. Reference to Steelman: None.

3. Current Workaround: There is no known workaround short of changing the runtime system. Even if the user provided an assembly language routine to backlog the interrupts, there would be no way to get control at the appropriate time to send the interrupt to the runtime system.

G. Bit Operations. In the real-time problem domain there are many requirements for efficient operations on word sized bit vectors. Some of these applications are:

- Testing bits in status words
- Unpacking hardware defined data structures
- Encoding and decoding fixed and variable length codes (such as gray code)
- Scanning for flags (find next set bit)

Although Ada currently has primitives to provide minimal support for these applications, it is difficult to write code that is efficient or portable, and impossible to write Ada code that is both. In addition, since the actual algorithm is hidden by the contortions required, the generated machine code is often easier to read than the Ada source code.

There are three interlocking problems here: Lack of sets (see Steelman 3.4), no portable way to easily assign static values to bit arrays, and lack of Ada constructs which map to the hardware primitives.

1. Application Domain: Embedded systems where the software controls the interface (data or control) to external hardware.

2. Reference to Steelman: 3.4 Sets

   "3-4A. Bit Strings (i.e., Set Types). It shall be possible to define types whose elements are one-dimensional Boolean arrays represented in maximally packed form (i.e., whose elements are sets).

   3-4B. Bit String Operations. Set construction, membership (i.e., subscription), set equivalence and nonequivalence, and also complement, intersection, union, and symmetric difference (i.e., component-by-component negation, conjunction, inclusive disjunction, and exclusive disjunction respectively) operations shall be defined automatically for each set type."

3. Current workaround: None. While it is possible to write code that "works," in applications with significant amounts of bit manipulation, the usual solution is to use some other language such as C or assembly.

H. Volatile Memory Sharing. Many embedded computer systems use shared memory to interface with the hardware or other programs. Typical examples are memory mapped device registers, memory with special side effects, and memory shared with external programs. In these cases the Ada program may write to or read from a particular location several times before program action could change the value. In normal situations an optimizing compiler would remove the extra steps that appear meaningless, however if an external entity is reading from or writing to that location, the optimization will cause a loss of data.

The need for "atomicity" (or the need to permit non-atomic operations) is not clear. Those arguing for atomicity feel that many shared memory situations require assurance that the object is updated or read consistently, independent of external agent operations. This is especially true for scalars (those that have hardware support for atomic read and atomic write).

Those arguing against atomicity state that shared memory objects are often quite large. The essential requirement is to make sure a read or write occurs when written; atomicity is handled by alternate mechanisms. In many cases atomicity can be replaced by some weaker requirement, such as ensuring that an array is updated from lower to higher addresses.

The best middle ground appears to be providing some mechanism to ensure atomicity of object read/update independently. If atomicity is bound with volatility, objects which can be atomically read/updated must be implementation (hardware architecture) defined.

1. Applications Domain: Embedded systems that use memory mapped interfacing with hardware or other programs.
2. Reference to Steelman: N/A
3. Current Workaround: Workarounds include using machine code or interfaces to other languages so that each read or write involves a procedure or function call. This workaround leads to code that is difficult to read and maintain.

I. Fixed Point Accuracy.

1. Application Domain: Certain embedded applications require a guarantee that fixed- point values be exactly limited to user-specified model numbers so that results of additive (and integer multiplicative) values are predictable.

   For example, if a device divides a circle into exactly 100 sections, then it is desirable to have a fixed-point type that has values that are only multiples of 360/100. The result would allow tests for absolute equality rather than testing with T'SAFE_SMALL.

   With current Ada semantics, specification of a fixed-point type (or even of 'SMALL) does not ensure that model numbers are limited to integral multiples of 'SMALL.

2. Reference to Steelman: 3-1G. Fixed Point Scale. "The scale or step size (i.e., the minimal representable difference between values) of each fixed point variable must be specified in programs and be determinable during translation. Scales shall not be restricted to powers of two."

3. Current Workarounds: Elaborate scaling schemes that approximate fixed point operations.

J. Unsigned Types without Overflow. Ada Integer types are inadequate for many algorithms such as discrete FFT, random number generators, checksums, and hashing.

1. Application Domain: Real-time embedded system applications such as signal processing, navigation, communications, etc.

2. Reference to Steelman: None.

3. Current Workarounds: Use a larger type and MOD. However, the larger type is not always available, and the calculation is much slower. Note that it is precisely access to the hardware multiply (32*32->32) without overflow that is needed.

III. Requirements and Justification

A. Storage Error Problem

1. Statement of Requirement: At minimum, the standard shall specify a way for programs to distinguish between exhaustion of local storage due to an attempt to create locally declared objects, temporary variables, or subprogram/entry calls and exhaustion of storage used for objects designed by values of access types.

For the general requirement, the language shall provide a finer-grained exception facility than is currently provided. At a minimum, the following exceptional conditions shall be distinguishable:

a. Exhaustion of local storage due to an attempt to create locally declared objects, temporary variables, or subprogram/ entry calling frames;

b. Exhaustion of storage used for objects designated by values of an access type (regardless of whether a representation clause has specified a collection size for the access type); and

c. Each condition represented by one of the checks allowed as parameters to program SUPPRESS.

2. Proposed Language Changes:

a. For the specific requirement, add two predefined exceptions (e.g., LOCAL_STORAGE_ERROR and ALLOCATOR_ERROR). LOCAL_STORAGE_ERROR is raised when creation of a declared object, temporary variables, or subprogram/entry call frame cannot be completed because storage is insufficient.

ALLOCATOR_ERROR is raised when an attempt to create an object designated by a value of an access type cannot be completed because storage is insufficient. If a sequence of handlers includes a handler for STORAGE_ERROR, then no handler for either LOCAL_STORAGE_ERROR or ALLOCATOR_ERROR is permitted. In this case, the handler for STORAGE_ERROR is involved whenever either LOCAL_STORAGE_ERROR or ALLOCATOR_ERROR is raised in the associated sequence of statements.

b. For the general requirement, provide a function or set of functions that return details about the circumstances under which the current exception was raised. At a minimum, these details shall include the following:

   — name of exception.

   — name of the innermost named construct enclosing the point at which the exception occurred.

   — circumstances of the exception expressed so as to distinguish among the cases given in the requirement.

   Provide a hierarchy of exceptions, analogous to a hierarchy of subtypes. The base is EXCEPTION, and each predefined exception is a subexception. Subexceptions of each predefined exception are also defined. Handlers for an exception and for one of its subexceptions must not be given in the same sequence of handlers. A handler for an exception is involved whenever that exception or one of its subexceptions is raised in the associated sequence of statements. A user-defined exception may have subexceptions. The suggested syntax is illustrated by the following:

```
STACK_ERROR: exception;
STACK_OVERFLOW,
STACK_UNDERFLOW: exception in STACK_ERROR;
```

3. Justification for Change: In an embedded system making heavy use of dynamic allocation, exhaustion of storage for locally declared objects, temporary variables, and calling frames for subprograms and entries may require that current task be stopped and a degraded mode of execution be started. On the other hand, exhaustion of storage for objects designated by values of access types may best be handled by waiting until storage becomes available or by deallocating some objects. Unfortunately, no method exists by which programs can differentiate the two cases.

The impact on current compiler technology and computer technology is nil; the required information is already available. The proposed language change will not make any existing programs illegal. Its apparent awkwardness may affect its acceptance by the software engineering community and the standards organizations.

Implementation of this proposal is clearly within the capabilities of current technology. It adds complexity from the user's view, as well as begging the question of specifying the form and content of the desired information. The resulting perception of complexity could impact acceptance by users and standards organization. No existing programs would be rendered illegal.

This proposal is the cleanest and most general solution. Its impact on compiler and computer technology is the same as for the previous proposed. The clarity and symmetry of this solution should encourage its acceptance by the software engineering community and the standardization organization. No existing programs would be rendered illegal.

4. Minority Views: None.

B. Reliably Turning off Runtime Checks

1. Statement of Requirement: A way (not necessarily a pragma) shall be provided for a programmer to specify that certain checks either not be performed, or at least must not cause an exception to be raised, within a given region of program text. A compiler shall not be allowed to ignore such a specification. That is, it shall reject the program if it cannot comply.

   It is sufficient to be able to ignore checks only for major regions of program text, such as package specifications and units with declarative parts.

   This feature does not need to be available for all checks. Specific checks where it appears to be needed include DIVISION_CHECK, OVERFLOW_CHECK, and ELABORATION_CHECK. It does not make any sense for some other checks, such as ACCESS_CHECK and STORAGE_CHECK. Whether is has any value for other checks is unclear.

2. Proposed Language Changes: A new form of representation clause could be added:

   for <check_name> IGNORE;

   The set of check_names supported can be restricted by the language to include only those for which continuation might be safe, as discussed above.

3. Justification for Change: This requirement is essential in a small but very critical application domain, and has no workaround. It only enables a programmer to require a compiler to do what the SUPPRESS pragma already permits a compiler to do. That is, it does not introduce any new unsafe feature to the language, but simply makes the behavior

of a program more predicatable.

There is no special implementation difficulty. For some machine architectures it may be difficult or impossible to turn off checks performed by hardware, but the Ada runtime system can then use a software switch in the hardware trap handler, that controls whether the trap causes an exception to be raised. The extra cost of entering the trap handler, checking this switch, and then resuming the computation will still be much less than the cost of entering the trap handler, raising the exception, propagating it to an exception handler, executing the exception handler, exiting the frame, reentering the frame, and restarting the computation (if this is possible at all).

There should be no negative impact on the software engineering community if the language is changed. An implementation can still support their own old solutions (for backward compatibility.)

The change is compatible with Ada83.

4. Minority Views: None.

C. Problems with Address Specifications

1. Statement of Requirement:

   a. The effect of an address clause shall be more precisely defined.

   b. A way shall be provided to specify the address that the compiler shall use to refer to an object when reading and writing it, and as the subprogram entry-point when calling a subprogram. The compiler shall not generate code to initialize such an object if it is declared as a constant. It shall be possible to specify an address determined at run time or via a generic formal parameter. Specifying the same address for two objects shall cause them to share memory locations (implying that if they have the same type they will have the same value).

   c. A way shall be provided to specify the address that the compiler shall use to call a subprogram. The compiler shall provide code for the subprogram if and only if the pragma INTERFACE is not specified for it. For any Ada subprogram declared as a library unit or immediately within the specification of a library package, a way shall be provided to obtain an address such that if this address is specified for a second subprogram calls to the second shall have the same effect as calls to the first subprogram.

   d. A way shall be provided to specify where in memory an object or code unit generated by the compiler should be located. It shall be possible to optionally either specify the exact address or to specify a logical category of storage and let the compiler choose the exact address. It shall be possible to modify such an address specification for an entity without modifying the Ada source code of the compilation unit containing the entity. This may be via a post-compilation specification, within or without the Ada language, whose form may be dependent on the compiler or target machine architecture. If such an object has a specified value, the compiler is required to provide that the storage at the specified address is initialized to this value. Similarly, for a program units the compiler is required to provide that the code for the unit is loaded at the specified address.

e. An Ada language implementation shall be permitted to define a private type in package SYSTEM which may be used as an alternative to SYSTEM.ADDRESS in specifying bindings between interrupts and task entries.

2. Proposed Language Changes: The following, together, would partially solve these problems:

   a. Allow the INTERFACE pragma for objects, with the meaning that the compiler should treat the object as external, and should not provide an initial value (even if one is specified in the program). However, the compiler may use an initial value in optimizations, if such a value is specified.

   b. Require the INTERFACE pragma to be supported for at least the language Ada (meaning the same compiler).

   c. Define a new way of optionally specifying certain objects and program units as having specific addresses, or as belonging to certain named logical storage categories, without specifying their exact addresses, and require that an implementation provide a way of specifying separately (after compilation of the affected units) the specific storage device (or portion thereof) to which each named category is allocated.

   d. Introduce a private type (e.g. SYSTEM.INTERRUPT_ID), which would be *implementation-defined*. Require an implementation to provide values of this type for: (1) all standard sources of interrupts in the target machine architecture not handled by the Ada runtime system or O.S. (e.g. constants of type INTERRUPT_ID); (2) user application/configuration defined sources of interrupts; (3) pseudo-interrupts raised by the Ada runtime system in response to hardware interrupts handled by the Ada runtime system in cases where the runtime system action is not defined (when there may be several causes for the hardware interrupt, only some of which are meaningful to the Ada runtime system).

   e. Require that the address given in the Ada83 form of address clause and the value of 'ADDRESS be the address which should be used by the compiler to read/write and object, and to call a subprogram.

3. Justification for Change: Ada users who require these capabilities will negotiate with compiler suppliers for implementation-specific solutions.

Code will be compiler-specific, and different styles of programming will develop. Some users may decide not to use Ada, since it may seem purer or more convenient to use a language for which no special new compiler features need to be negotiated.

These changes are compatible with Ada83, and do not require new compiler technology.

The notion of storage category may require further explanation. A system may have several kinds of memory, with significantly distinct attributes. These include:

- different memory banks, which may be interleaved to permit concurrent access without interference
- memories of different speeds (e.g. cache, fast local memory, global memory with fast local shadow, memory accessed via a bus)
- shared vs. per-processor memory
- ROM vs RAM

It does not make sense for the language to specify a standard storage set of such storage attributes, since these will need to vary over different memory configurations and machine architectures.

4. Minority Views: There were no minority views about there being problems with the address clause in Ada83. But, there was some uncertainty over whether the compiler should initialize external or memory-mapped variables for which addresses are specified (it being clear that it should not initialize constants in this case).

   A minority felt that the library unit and subunit structure already provides sufficiently fine granularity to permit an implementation to support specifying storage attributes or classes at link-time, without any further help in the source code.

   There was not agreement on the syntactic form of storage attribute/class specifications, though there appeared to be consensus that some way should be provided for the application to name sets of things that should go together, and then bind attributes or locations to these named sets at link-time.

D. Control over Reference vs Copying in Parameters Transmission

1. Statement of Requirement: A way shall be provided for a programmer to specify whether each subprogram parameter is to be passed by copying or by reference. The default shall be the Ada83 semantics.

2. Proposed Language Changes: The keyword such as NEW could be used to signify a parameter should be passed by copying, and a keyword such as ACCESS could be used to signify a parameter should be passed by reference. For example,

procedure UPDATE(X: access in out INTEGER; Y: new in INTEGER);

3. Justification for Change: This provides a capability required to solve several problems for which there is no reliable workaround. It does not require any new compiler technology; it simply allows the user to specify something that is presently left up to the compiler. It is entirely upward-compatible with Ada83.

4. Minority Views: None.

E. Task Type Interrupts

1. Statement of Requirement: A safe way is required to dynamically change interrupt association to task entries.

2. Proposed Language Changes: Add the following capabilities:

```
CONNECT_INTERRUPT(ENTRY_NAME,
   INTERRUPT_ADDRESS);

DISCONNECT_INTERRUPT(ENTRY_NAME);
```

3. Justification for Change: The justification for the proposal is based on the importance of this capability to the applications domain. Support for these capabilities were not discussed due to time constraints. Also, the impact of the software engineering community and current computer technology was not discussed due to time constraints.

4. Minority views: Agreement was reached on the need for the general capabilities, but considerable discussion was held on exact capabilities required.

F. Backlogging Interrupts

1. Statement of Requirement: There is a requirement for the runtime system not to lose interrupts.

2. Proposed Language Changes:

   a. One solution is for the runtime system to maintain a chain of interrupt-event records for each type of interrupt that needs to be backlogged. A pragma could indicate which interrupts these are. The backlogging would have to be indefinitely deep, or its depth would have to be specifiable by the user.

   b. Another solution is to provide a pragma by which a task is made to run at a user-specified interrupt level. But if the task does not run at the highest level, interrupts could still be lost in the above case 2.A. If run at the highest level, then it would lock out all higher priority activity. Either way, tasks that wait or interrupts would have to be broken into two tasks to minimize the time that other processing is disabled, causing additional rendezvous overhead.

It is noted that by virtue of the second case, an interrupt cannot be modeled as an entry call, as the LRM states. This is because hardware may properly make another call before the previous one goes through.

3. Justification for Change: The loss of a critical system interrupt could result in a catastrophic event, such as loss of life. The LRM 13.5.1 states that an interrupt may be implemented as "an ordinary entry call, a timed entry call, or a conditional entrycall . . . depending on the implementation." This allows a runtime system to discard the interrupt if the associated task is not waiting at the interrupt entry when the interrupt occurs.

   However, there are common, unavoidable situations where the task will not always be ready for the interrupt:

   a. The interrupt may be unsolicited – not the result of something that the task does, but a signal of unsynchronized input. For example, a signaling of a message from a different CPU. Therefore, the interrupt can occur at any time and can re-occur arbitrarily soon.

      Forcing the sender of the interrupt to wait for a ready signal from the receiving task is not always a solution because the sending device may not be controllable (e.g., RS-232 input) or too much data must be moved too fast to allow the sender to wait.

   b. Even if the interrupt is the result of an action by the receiving task:

      i. The response could occur before the accept because the receiving task is preempted by a task connected to a higher priority interrupt.

      ii. The interrupt can have multiple, independent causes; that is, the interrupt line is shared among devices because there are too many different interrupts for the CPU or the interrupt controller to distinguish.

4. Minority Views: Several group members were concerned about forcing all implementations to support an interrupt backlogging mechanism. It was suggested that this mechanism should be an option supported by the vendor.

G. Bit Operations

1. Statement of Requirement: There should be a mechanism that supports bit operations to include shift (right and left), rotate, set and test a bit, and find first bit efficiently for word length bit strings.

2. Proposed Language Changes: Require a (generic) package for bit manipulations. Significant effort should be placed in defining specifications to allow for portability to other machines of the same word length.

3. Justification for Change: Since real-time embedded systems almost always interface to the system in which they are embedded by way of bit-parallel transfers, the efficient and transportable handling of bits is extremely important.

   Implementation of this change is well within the state of the compiler art. Meeting this requirement will also allow more readable and understandable code. Standard handling of bits will also simplify training and implementation.

   By using a required package as opposed to revising the language itself. the impact on the standardization process should be minimal.

4. Minority Views: Bit arrays and bit operations should be included within the language itself.

H. Volatile Memory Sharing

The Ada language shall provide a reliable mechanism for sharing data with agents external to the program.

1. Statement of Requirement: The Ada language needs a way to indicate that certain objects are arbitrarily volatile with respect to the Ada program, so each read and write in the source code must be mapped to a single read and write of the memory that represents the object. Where feasible, an implementation shall provide atomic read and write operations, and shall provide some mechanism to warn of situations that will be non-atomic.

2. Proposed Language Changes: Two proposed language changes were recommended for this requirement.

   a. A new Pragma - Pragma VOLATILE (<type name>). Type name indicates that representations and operations on all objects of the type are to be computed using "volatile" semantics.

   b. The second proposed change was a representation clause (separate), for <type name> use VOLATILE.

3. Justification for Change: Such volatile memory sharing occurs with surprising frequency in embedded systems. Often the use of such interfaces pervades an entire program and cannot be easily encapsulated to a small set of code. Pragma SHARED does not provide this feature and should not be used for it. Pragma SHARED is used for sharing data within an Ada program between tasks; its semantics are (correctly) weaker than that needed for VOLATILE memory.

   The problem with the proposed representation specification is that it creates a new syntax and gives special meaning to VOLATILE in that context (possibly to other words as well). For the proposed Pragma VOLATILE begins to encroach on the semantic meaning of the program, something pragmas are not supposed to do.

The group advocates this change since the current Pragma Shared is deficient for control of object which may be asynchronously referenced by external agents (i.e. hardware or software). This was based on many embedded applications using device registers as memory, memory shared with hardware devices, and memory shared with separate programs.

Applications need to ensure that for certain objects, each occurrence of the object on the right hand side causes a single read of the memory location assigned to the object and each occurrence of the object on a left hand side causes a single write of the object's memory location. Also, the order of occurrence shall not be changed.

The workarounds for this feature are so cumbersome as to lead to rejection of Ada, less reliable code, or both.

4. Minority Views: None.

I. Fixed Point Accuracy.

1. Statement of Requirement: The language shall permit specification of fixed-point model numbers exactly, for all rational numbers.

2. Proposed Language Changes: Add a subclass of fixed point type that require that a delta (or similar notation) represents an exact value rather than a value relative to the specified precision. The expected implementation might map this software change as integer with the compiler providing the scaling.

3. Justification for Change:

   a. This requirement is already an Ada commentary. (AI-00146)

   b. A more formal expression of need is given:

```
    Type F is delta D range L..H;
    for F 'SMALL use D;
    X,Y: F;
    I: integer;

begin

    X: = < any possible value of F >;
    I: = INTEGER (X/F (F 'SMALL));
    Y: = I*F (F 'SMALL);
```

   Then, Y must be exactly the same value as X (a stronger equality than Ada equality).

   c. There is a separate issue of needing Ada to permit "efficient" fixed-point multiplication for frequent embedded cases. This is being independently researched.

d. There is NO implied underlying representation for these values. For representation specific requirements, reliable conversion to and from integer can be made using F 'SMALL and integer representations used to get into any needed bit layout.

J. Unsigned Types without Overflow

1. Statement of Requirement: There must be a type which does not cause NUMERIC_ERROR (or CONSTRAINT_ERROR) but wraps around. Such a type with range 0..2*INTEGER'LAST - 1 is necessary, and additional ranges should be provided.

2. Proposed Language Changes: Add a new family of discrete predefined numeric types.

3. Justification for Change: There are many important numeric algorithms which cannot currently be conveniently expressed in standard Ada. The normal current workaround is to write the code in some other language and use pragma INTERFACE, but even this is problematical, since there is no legitimate way to declare the parameter types. (The normal workaround is to declare the parameters to be of type Integer.)

   The current standard implicitly requires that unsigned representations be used for enumeration types. In addition most major compiler vendors have support of some sort for unsigned numeric types (for interface to system run-time routines and for address calculation), but are not permitted by the current interpretations of the standard to make it available in usable form to users.

   However, it is important to note that unsigned types should not be allowed to match generic formal parameters of the form:

   ```
   type FOO is range <>;
   ```

   but should be allowed to match:

   ```
   type BAR is (<>);
   ```

   since the former could cause difficulties for many implementations.

   Almost all current generation machines currently support all of these operations in hardware, and the ones that don't are usually missing the corresponding integer operations also. For example, the new Intel 80860 has NO integer divide instruction.

   This change would require extra ACVC tests.

4. Minority Views: Some people would like to have an unsigned type with overflow checking with a range 0..System.MAX_INT * 2 + 1. However many of these people would actually prefer to have both types available.

IV. List of Working Group Members

- — Marlow Henne, Chairman, Sverdrup Technology - USA
- — Jim Baldo. Facilitator, IDA - USA
- — Ted Baker, Florida State University - USA
- — Phil Brashear, SofTech - USA
- — George Buchanan, IIT Research - USA
- — David Cobb, McDonnell Douglas Missile Systems - USA
- — Steven Deller, Verdix - USA
- — Robert Eachus, MITRE Corporation - USA
- — Tzilla Elrad, Illinois Institute of Technology - USA
- — John Goodenough, Software Engineering Institute - USA
- — John Hamilton, FAA - USA
- — Toomas Kaer, Ericsson Radar Electronics AB - Sweden
- — Ken Littlejohn, Air Force, AFWAL - USA
- — Jim Mason, McDonnell Douglas Missile Systems - USA
- — Mike Mills, Air Force, ASD - USA
- — Roger Racine, The Charles Stark Draper Labs - USA
- — Jim Schnelker, General Dynamics - USA
- — Pete Vaccaro, Air Force, HQAFSC - USA
- — Steve Wilson, Air Force, ASD - USA

5. PARALLEL/DISTRIBUTED SYSTEMS WORKING GROUP

I. Scope of the Working Group

A. Domain of Interest: The parallel/distributed systems working group (described as the working group or group in the remainder of the report) decided that its domain of interest included the use of Ada in parallel and distributed processing architectures. Signal processing was not explicitly considered.

B. Salient Issues: A recurring theme in this report is that current projects are both required to use Ada, and to implement a distributed or parallel processing architecture. The overriding issue is whether projects in the future will have language support for implementing applications in Ada on distributed or parallel architectures, or whether project-unique, extra-linguistic solutions will continue to be created, as is the current practice. Lower life cycle costs, and enhancement of maintainability, reuse, and portability are of concern with this issue.

II. Specific Problems

The working group decided that most of the problems it would attempt to cover fell into the categories of one Ada program spanning a distributed or parallel architecture, and of multiple Ada programs used in a distributed or parallel architecture. Nineteen areas were identified.

A. Types of distributed/parallel architectures.

1. Application Domain: The application domain is all projects which intend to use Ada on a distributed or parallel processing architecture.

2. Reference to Steelman: 9B implies that no architecture should be excluded from consideration; it also implies that no language construct should explicitly support a particular type of architecture ("Processes shall have consistent semantics whether implemented on multicomputers, multiprocessors, or with interleaved execution on a single processor").

3. Current Workaround: Extra-linguistic, project-unique workarounds are currently used to implement Ada applications in distributed or parallel processing architectures.

B. Heterogeneous processing architectures.

1. Application Domain: The application domain is all projects using Ada on heterogeneous processing architectures.

2. Reference to Steelman: No Steelman requirement explicitly considers heterogeneous processing architectures; 9B, however, does not rule such architectures out ("Processes shall have consistent semantics whether implemented on multicomputers, multiprocessors, or with interleaved execution on a single processor").

3. Current Workaround: Extra-linguistic, project-unique workarounds are currently used to implement Ada applications in heterogeneous processing architectures.

C. *Pre-partitioning versus post-partitioning of a single Ada program.*

1. Application Domain: The application domain is all projects using a single Ada program in a distributed or parallel processing architecture.

2. Reference to Steelman: 9A ("It shall be possible to define parallel processes") calls out the definition of parallel processes, but no Steelman requirement addresses the allocation of parallel processes to processors.

3. Current Workaround: Both pre-partitioned and post-partitioned approaches are being implemented in an extra-linguistic way.

D. Fault tolerance and survivability.

A fault tolerance approach within the language is not what is desired; the tools to construct appropriate fault tolerance approaches are needed. Many of the subsequent problem areas deal with such tools.

1. Application Domain: The application domain is all projects implementing fault tolerant Ada software in a distributed or parallel environment.

2. Reference to Steelman: No Steelman requirement explicitly addresses fault tolerance or survivability.

3. Current Workaround: There are no workarounds within the language; many approaches exist outside the language, using capabilities described in other problem areas.

E. Dynamic configuration control.

As with fault tolerance, a language-specified approach to dynamic configuration control is not desired. Instead, the tools to construct appropriate dynamic configurability approaches are needed.

1. Application Domain: The application domain is all projects which must implement Ada code dynamically in a distributed or parallel processing architecture.

2. Reference to Steelman: No Steelman requirement explicitly addresses dynamic control of configurations. 9A refers to initiation of processes within the scope of the process definition, but this is not a complete set of requirements for dynamic control of configuration.

3. Current Workaround: There are no workarounds within the language; many projects implement dynamic configuration control outside the language, using capabilities described in other problem areas.

F. Intertask communication, particularly with a program which spans multiple processors.

This problem was being covered by the real-time issues working group; this group thought it had a stake in the area as well, to support fault tolerance and dynamic configurability.

1. Application Domain: The application domain is all projects which must implement Ada code dynamically in a distributed or parallel processing architecture.
2. Reference to Steelman: 9H (passing data), 9I (signaling), and 9J (waiting) all address aspects of intertask communication. Very little of this is directly implemented in 1815A.
3. Current Workaround: Only the rendezvous construct is given within the language; many vendor or project-unique solutions as well are used to implement communication in a distributed or parallel architecture.

G. Adaptive scheduling.

Adaptive scheduling was being covered by the real-time issues working group; this group thought that it had a stake in this area as well, to support fault tolerance and dynamic configurability.

1. Application Domain: The application domain is all projects which must implement Ada code dynamically in a distributed or parallel processing architecture.
2. Reference to Steelman: 9D ("A process may alter its own priority.") 9G (Asynchronous termination), 9I (Signaling), and 9J (Waiting) imply a much greater control over scheduling than that given by the Ada tasking paradigm.
3. Current Workaround: much of the functionality needed in this area; obviously there are portability and reusability difficulties.

H. Memory Management.

This is another area in which the group felt it had a stake. Access types and shared variables were originally mentioned.

1. Application Domain: The application domain is all projects which must implement Ada code dynamically in a distributed or parallel processing architecture.
2. Reference to Steelman: 9C addresses mutual exclusion ("It shall be possible efficiently to perform mutual exclusion in programs.").
3. Current Workaround: Most users of Ada in distributed or parallel systems without shared memory use some communications approach to guarantee mutual exclusion between processors, and depend upon design methodology to keep types consistent between Ada programs.

I. Asynchronous Events.

The group thought that this was another area being handled by the real-time issues group in which it had a stake.

1. Application Domain: The application domain is all projects which must implement Ada code dynamically in a distributed or parallel processing architecture.
2. Reference to Steelman: 9G (Asynchronous Termination) covers some of the functionality envisioned for asynchronous events, although this requirement does not cover all asynchronous events that would be used for distributed or parallel systems.
3. Current Workaround: As with adaptive scheduling, many vendors for embedded targets provide the necessary functionality in a non-standard way.

J. Time in a distributed system.

This is another area handled by another group in which the distributed/parallel systems group felt it had an interest.

1. Application Domain: The application domain is all projects which must implement Ada code dynamically in a distributed or parallel processing architecture.
2. Reference to Steelman: 9E (Real Time) addresses the issue of time. However, it seems to implicitly assume a uni-processor environment; communication delays are not discussed.
3. Current Workaround: support, a uniform system clock is usually available. Such clocks often do not map well to package Calendar. In the absence of hardware support, or in widely distributed systems, time is handled in a unique way.

K. Resource Reclamation.

This is another area handled by the real-time group in which this group felt it had a stake.

1. Application Domain: The application domain is all projects which must implement Ada code dynamically in a distributed or parallel processing architecture.
2. Reference to Steelman: 3-3I ("An element of an indirect type shall remain allocated as long as it can [b]e referenced by the program."), and 3-5C ("Variables declared within an encapsulation ... shall remain allocated ... throughout the scope in which the encapsulation is instantiated") imply resource reclamation, but do not address such things as task control blocks or other implicit data structures which some implementations allow to survive past the lifetime of the objects.
3. Current Workaround: The use of dynamic constructs which lead to unreclaimed resources may be avoided.

L. Identification of raised exceptions regardless of scope, and information about the context of raised exceptions.

The concern centered around aid for debugging, and for requirements of some systems to identify all reasons for abnormal performance or termination of processing elements.

1. Application Domain: The application domain is all projects which must implement Ada code dynamically in a distributed or parallel processing architecture.

2. Reference to Steelman: No Steelman requirement addresses identifying exceptions out of scope.

3. Current Workaround: The propagation of exceptions out of scope may be avoided (which may be difficult to do with reusable software incorporated into a project). The use of exceptions to handle unexpected or fatal events may be avoided at the expense of a project-unique approach, which sacrifices reuse and portability. Since most embedded target compilation systems and run-time systems have necessary information about exceptions and suppress the visibility of the information, a number of vendor-unique ways of determining exception context have been implemented.

M. Identification of objects and threads of control.

This capability is seen as useful in survivability, fault tolerance, scheduling, and reconfiguration designs.

1. Application Domain: The application domain is all projects which must implement Ada code dynamically in a distributed or parallel processing architecture.

2. Reference to Steelman: No Steelman requirement addresses the ability of a task to identify itself or other tasks, or to identify objects.

3. Current Workaround: Task identification is often carried out in an ad hoc fashion which varies from vendor to vendor. Object identification must be done in an extra-linguistic way.

N. Multi-programming Semantics.

Currently, if a multiple Ada program approach is taken to implementing Ada in a distributed or parallel environment, each project must determine its own semantics for program interaction.

1. Application Domain: The application domain is all projects which must implement Ada code dynamically in a distributed or parallel processing architecture, and do so as multiple Ada programs.

2. Reference to Steelman: Multi-programming is outside the scope of Steelman.

3. Current Workaround: Multi-programming is outside the scope of 1815A; projects which use multi-programming implement their own semantics, usually with vendor support.

O. Secondary Standards (optional chapters).

In order to avoid burdening applications which have no need of support for distributed or parallel architectures, the group thought it might be appropriate for the specific requirements to support distributed and parallel systems to be contained within a secondary standard, or an optional chapter of the Ada Reference Manual.

1. Application Domain: The application domain is all users of Ada 9X.
2. Reference to Steelman: Secondary standards and optional chapters were not within the scope of Steelman.
3. Current Workaround: Not applicable.

P. Dynamic security and integrity.

In particular, implementing trusted computing bases in distributed or parallel processing systems was of concern.

1. Application Domain: The application domain is all projects which must implement trusted computing bases in a distributed or parallel processing architecture.
2. Reference to Steelman: No Steelman requirement addresses security and integrity.
3. Current Workaround: To the best of our knowledge, no trusted computing base to an orange book B3 level has been implemented on a distributed embedded processing architecture.

Q. Semantic support of determinism.

1. Application Domain: The application domain potentially is all projects using Ada on a distributed or parallel architecture which must undergo test and evaluation.
2. Reference to Steelman: 1H (Complete Definition) addresses the *supporting area of formal* definition of the language.
3. Current Workaround: There are no current workarounds to our knowledge regarding determinism.

R. Semantics of performance monitoring.

Information about a task's use of resources, particularly processor throughput, ought to be acquirable.

1. Application Domain: The application domain is distributed Ada programs which must monitor performance.

2. Reference to Steelman: 9E refers to acquiring part of the resource usage information (Real Time. "A process may have an accessible clock giving the cumulative processing time (*i.e.*, CPU time) for that process").

3. Current Workaround: Many projects using Ada in distributed or parallel systems implement unique ways of acquiring performance data.

S. Support for distributed software debugging.

1. Application Domain: The application domain is all projects using Ada on distributed or parallel architectures.

2. Reference to Steelman: No Steelman requirement explicitly addresses software debugging support for distributed or parallel systems.

3. Current Workaround: Most projects use project or vendor unique support for debugging their software on distributed or parallel architectures.

III. Requirements And Justification

Due to the number of items which the distributed/parallel systems working group addressed during the week, to the general lack of support for distributed systems in particular within Ada, and to time limitations, actual language changes were not addressed. Instead, the group determined requirements and discussed potential implementations.

No attempt was made to make the requirements independent. Indeed, in many places, requirements overlap.

A. Types of distributed or parallel architectures.

This requirement addresses problem area A.

1. Statement of Requirement: The language shall not preclude the distribution of a single Ada program across a homogeneous distributed or parallel architecture.

2. Proposed Language Changes: No explicit language changes were formulated.

3. Justification for Change: This is obviously of prime importance. Any language inhibition to distributing an Ada program across multiple processors drives projects which must do so into extra-linguistic multi-programming approaches, or extra-linguistic distribution of a single program. This obviously degrades portability, reuse, and life cycle costs.

4. Minority View: No minority views were recorded.

B. Heterogeneous Architectures.

This requirement addresses problem area B.

1. Statement of Requirement: The language shall not preclude the distribution of a single Ada program across a heterogeneous parallel or

distributed architecture.

2. Proposed Language Changes: No explicit language changes were formulated. However, both packages SYSTEM and STANDARD are affected by this requirement; the former because of hardware-unique items defined within it, such as type ADDRESS, and the latter because of type definitions such as INTEGER.

3. Justification for Change: To those projects which use heterogeneous architectures, precluding or inhibiting the distribution of a single Ada program drives them into extra-linguistic solutions. Impact to compilation systems (and to standardization processes) could be minimized if these requirements were implemented in an optional part of the language.

4. Minority Views: A minority view was that "should" ought to be used rather than "shall" in this requirement, in order to avoid compromising homogeneous architecture support.

C. Partitioning.

This requirement addresses problem areas C, D, and E.

1. Statement of Requirement:

   a. The language shall not preclude partitioning of a single Ada program in a distributed or parallel system.

   b. The language shall support the explicit management of the partitioning of a single Ada program.

   c. The language shall support the allocation of a partitioned single Ada program. The intent is to support both dynamic and static allocation.

2. Proposed Language Changes: No explicit language changes were formulated.

3. Justification for Change: Part a) of the requirement is intended to protect an existing quality of Ada.

   Part b) reflects the view of the group that while post-partitioning does not require additional semantics, pre-partitioning does. The idea of supporting virtual nodes within Ada repeatedly arose.

   Part c) generated the most controversy, and indeed passed by one vote. It was thought that such things as options similar to those specified in "A Catalogue of Interface Features and Options for the Ada Run Time Environment" (from the Ada Run Time Environment Working Group (ARTEWG) of the Special Interest Group - Ada (SIGAda)) or a secondary standard might insulate uninterested Ada users from being impacted.

   This is likely to have a large impact on compilation systems.

4. Minority Views: Minority view centered around item c). Among the criticisms were that it is:

   a. Premature (no standard in this area should be adopted prior to seeing how it works in a real system),

   b. Infringing on an important area where project-unique requirements are met, and where consequently one solution is not likely to be appropriate (the preference is for primitives to construct the appropriate allocation approach),

   c. Likely to entail a large run-time penalty, and

   d. Runs counter to the view that it is inappropriate for system resources to be managed from an application.

   If c) were interpreted to mean supplying the tools or primitives (as mentioned above) to achieve dynamic allocation, much of the opposition to it might diminish.

D. Support for Fault Tolerance and Dynamic Configurability.

This requirement addresses problem areas D and E.

1. Statement of Requirement: The language shall support fault tolerance and configurability by:

   a. Providing failure semantics for such constructs as abort, tasking attributes, and task dependencies, and

   b. Adding primitives which permit the construction of fault tolerance and configurability in a portable and reusable way.

2. Proposed Language Changes: No explicit language changes were formulated.

3. Justification for Change: There were four positions which the group could take regarding this issue.

   a. Do nothing - Solutions in this area would continue to be ad hoc.

   b. Clean up the semantics - ad hoc solutions would be easier to achieve.

   c. Provide "building blocks" - portability, reuse, and maintainability would be enhanced.

   d. Require fault tolerance - this was thought to be a potential rendezvous in the making.

   The group opted for 2 and 3. It seemed odd to the group that Ada, a language to enhance reliability, left such an important area subject to project or vendor unique solutions.

   Part a) in the requirement refers to such things as the abort construct requiring acknowledgment from dependent tasks (which might be impossible if the processor containing the task to be aborted has tailed). Under this circumstance, an abort "hangs." Tasking has instances where failure (such as within the accept body) leaves the calling task hanging. Such failures must be accommodated within the language to have portable fault tolerance and configurability designs.

Part b) refers to "building blocks" which can be used universally in constructing fault tolerant and configurable designs. Among the building blocks discussed in the group were user-defined task characteristics (a subject of the real-time issues group, and the JIAWG (Joint Integrated Avionics Working Group) Ada 9X revision requests), and elaboration control. For fault-tolerant embedded systems, it was felt that about 90% of current approaches could be made more portable and reusable with these two building blocks alone.

4. Minority Views: A minority view was recorded, which expressed concern that defining primitives may restrict flexibility in the construction of fault tolerant and configurable designs. The following points were made:

   No mechanism should be added to the language for fault tolerance before it has been shown to be successful in several real applications as an integrated feature of some Ada implementation. Even though many fault tolerance approaches are well known, their clean integration into Ada is unproven. A substantial amount of research is currently being done to study the use of distributed Ada utilizing radically different approaches. There is a concern that any approach selected in the near future may make it more difficult (or impossible) to support an approach which is considered preferable in the future. If a consensus can be obtained for such approaches, they could be defined by secondary standards.

E. Intertask Communication.

This requirement addresses problem areas D, E, and F.

1. Statement of Requirement: The language shall provide an exact definition of semantics, including failure semantics, for all types of rendezvous. Additionally, the language shall explicitly support:

   a. Asynchronous send without acknowledgment (with failure semantics),

   b. Asynchronous receive without acknowledgment (with failure semantics), and

   c. Remote procedure calls (with failure semantics).

2. Proposed Language Changes: No explicit changes were formulated.

3. Justification for Change: This area is seen as supporting fault tolerance, configurability, and transactions.

Currently, there are no semantics for the situation when a failure in a remote rendezvous occurs after acknowledging that a rendezvous could occur (*e.g.*, in the accept body). There are also no semantics for the situation in which communication delays in a remote rendezvous obscure the time of delay for a timed entry.

With regard to remote procedure calls, a recommendation for "exactly once" semantics was expressed; the syntax should be transparent to the programmer.

4. Minority Views: A minority view expressed reservations about increasing the cost of a rendezvous.

F. Adaptive Scheduling.

This requirement addresses problem areas D, E, and G.

1. Statement of Requirement: The language, to support distributed and parallel systems, shall:
   a. Not prohibit scheduling by context, which may be dynamic.
   b. Provide mechanisms for scheduling by multiple characteristics, including user-defined characteristics.
   c. Support different scheduling paradigms in different parts of the system.
2. Proposed Language Changes: No explicit changes were formulated.
3. Justification for Change: This requirement supports several areas of interest to the group, including fault tolerance, configurability, and multi-programming semantics. The ability to use scheduling algorithms as parameters for allocation decisions was attractive.

   Concern was expressed over potential penalties which might be incurred in the absence of using these features. An optional portion of the standard covering this area might be a way to avoid such penalties.

   The impact of this requirement upon compilation systems was thought to be potentially minor (depending on how the vendor implements priorities). Many vendors already support portions of this capability. Note that the different scheduling paradigms and characteristics govern only how tasks are placed on the ready-to-run queue, not on the behavior of the ready-to-run queue itself.
4. Minority Views: The following minority view is recorded.

   Adaptive Scheduling is a "Systems Issue" that should be clearly stated in a single specification (file). The scope rules of the Ada language do not permit static visibility to all of the objects that need scheduling control. Therefore, the language cannot be changed sufficiently to allow all of the relevant information to be specified in one place. A more expressive specification language should be used to describe the system scheduling behavior. This specification would be correlated with the program using a tool which would adapt the program or load modules to implement the semantics of the specification language.

There is opposition to having partial solutions to this problem because the language changes may impede the implementation and understanding of mechanisms that are later deemed to be superior. Furthermore, the lack of consensus currently within the distributed Ada community indicates that extreme caution must be used when trying to standardize this area.

The argument: "portability necessitates that these capabilities be included in the language" is certainly an valid issue. However, the likelihood is very high that implementation dependencies will be allowed in any features incorporated to support distributed scheduling. The portability of programs using these features will be only marginally more portable than a program that uses one of several approaches common in the industry. If or when common approaches become well defined, they provide excellent candidates for secondary standards.

G. Memory Management.

This requirement addresses problem area H.

1. Statement of Requirement:

   a. The language shall provide efficient mutual exclusion, and consistent semantics for such exclusion, in a program (including a program distributed throughout a distributed or parallel system).

   b. The language shall provide explicit control for the location of objects and storage allocation for objects by:

      i. Providing the ability to name, locate, and size specific regions of storage,

      ii. Constraining the location of objects within a named region,

      iii. Providing the ability to locate a storage allocation for objects to a predetermined location, and

      iv. Constraining the location of a storage allocation for objects within a named region.

2. Proposed Language Changes: No explicit language changes were formulated.

3. Justification for Change: Part a) is a restatement of Steelman requirement 9C (Shared Variables and Mutual Exclusion). Such current workarounds as machine code insertion, and using rendezvous to implement the desired behavior were not considered acceptable.

   Part b) concerns explicit heap control, particularly with respect to supporting shared variables created by allocators.

4. Minority Views: Reservations were expressed regarding having the type of information referenced in part b) scattered throughout the program.

H. Time in Distributed Systems.

This requirement addresses problem areas J and S.

1. Statement of Requirement: In a distributed system, the language shall not require the same perception of time at all points in the system.

2. Proposed Language Changes: No explicit changes were formulated.

3. Justification for Change: It was thought that a language requirement for the same perception of time throughout a distributed or parallel system would actually be a requirement on the underlying hardware, since the requirement would be unachievable in the absence of special hardware support.

   The issue was also raised about the perception of time when a program is distributed throughout a widely dispersed system, as, for example, with transcontinental networks, or even throughout the solar system (a space probe and its "home" computational support). In cases where communication delay at best (*i.e.*, the speed of light) will exceed an acceptable value of the smallest representation of time (*e.g.*, roughly 13 milliseconds for 2500 miles), it does not seem reasonable to require that this value be used for the resolution of calendar.clock.

4. Minority Views: A dissenting opinion was recorded, that consistent time should be required within a system.

I. Identification of Raised Exceptions.

This requirement addresses problem areas D, E, L, and S.

1. Statement of Requirement: The language shall:

   a. allow a raised exception to be identified, regardless of whether the raised exception is in scope.

   b. allow the immediate context of where a raised exception was raised to be identified.

2. Proposed Language Changes: No explicit change was formulated.

3. Justification for Change: In particular, many embedded military systems are not allowed to bring processing elements down without recording information to support later maintenance actions. While workarounds exist for identifying exceptions (*e.g.*, no propagation of unidentified exceptions), they tend to involve unacceptable maintenance, or unacceptable visibility of exceptions.

   "Context" was purposefully left vague, and might include such things as thread of control, level of recursion, processor identification, and the address where the exception was raised.

The group's opinion was that much potentially useful information is deliberately suppressed by compilation systems in order to conform to the current semantics of exceptions.

4. Minority Views: A minority view expressed concern for the lack of definition of "context." and for the burden this might place on the compilation system.

J. Identification of Threads of Control.

This requirement addresses problem areas D, E, F, G, L, M, R, and S.

1. Statement of Requirement: The language shall provide accessible unique identification of threads of control for a distributed or parallel system.
2. Proposed Language Changes: No explicit changes were formulated.
3. Justification for Change: The real-time issues group was considered to have primary responsibility in this area. From a parallel or distributed systems view, this is thought of as a tool to accomplish objectives in other areas such as fault tolerance, configurability, or debugging.
4. Minority Views: An opinion was expressed that this is not appropriate to put in the language.

K. Multi-Programming Semantics.

While this section does not state a requirement (see below), problem areas D, E, and N are addressed.

1. Statement of Requirement: No requirement was formulated.
2. Proposed Language Changes: No language changes were formulated.
3. Justification for Change: The group discussed at length current ad hoc solutions to multi-programming semantics. and explored how these did not apply to the general case. The most general case seems to be multiple Ada programs each spanning multiple processors in a parallel or distributed system in a non-exclusive way.

   The group finally agreed that current major efforts in Ada require the use of multiple programs, and that an effort is appropriate to try to meet at least needs as they are understood today.

   The group recommended that the following be adopted:

   To support multi-programming semantics for Ada 9X, multi-programming should be added to the list of complex issues identified in the 9X process. Input from current Ada efforts in multi-programming should be solicited. This effort should investigate the dynamic introduction of new programs with scheduling parameters.

   The above recommendation was arrived at from consideration of the other requirements this group has defined. In particular, user-defined task characteristics, system-unique exception identification and exception contexts, asynchronous modes of task communication, remote procedure calls, and time perception and management are considered important in successfully implementing current extra-linguistic solutions.

L. The following areas identified in section II were not explicitly discussed by the group. Some areas, however, are covered at least to some degree in the requirements listed in section III.

1. Problem area I, Asynchronous Events. The group saw nothing to add from what the real-time issues group was producing during the week.
2. Problem Area K, Resource Reclamation. The group was satisfied with what the real-time issues group was coming up with in the course of the week.
3. Problem Area O, Secondary standards/optional chapters. This issue was not discussed because the information systems group was working the issue.
4. Problem Area P, Dynamic Security and Integrity. This issue was deferred due to lack of time. The entire week could easily have been spent debating orange book requirements.
5. Problem Area Q, Semantic Support of Determinism. This issue was not discussed because the trusted systems/verification group was handling the issue.
6. Problem Area R, Semantics of Performance Monitoring. This issue was deferred due to lack of time. However, the requirements in section III do provide some support for performance monitoring.
7. Problem Area S, Support for Distributed Software Debugging. The group felt that many of the other requirements established by the group aided in this area. For example, identification of exceptions and context and identification of threads of control provide language support for debugging.

IV. List Of Working Group Members.

The members of the Distributed/Parallel Systems Working Group included:

— Kent Power, Chairman, Boeing Military Airplanes - USA
— Cy Ardoin, Facilitator, IDA - USA
— Doug Dunlop, Intermetrics - USA
— Judy Edwards, General Dynamics - USA
— Tom Griest, LabTek Corporation - USA
— Peter Hoffman, Calspan Corporation - USA
— Charles McKay, University of Houston, Clear Lake - USA
— Steve Michell, Prior Data Sciences - Canada
— James Silver, Indiana University - Purdue University at Fort Wayne - USA
— Dave Smith, McDonnell Aircraft Company - USA

— William Taylor. Ferranti Computer Systems - UK

6. INFORMATION SYSTEMS WORKING GROUP

I. Scope of the Working Group

This working group was concerned with information systems used in the military or business environments for strategic or tactical planning, decision making, monitoring operations, and for operations such as transaction processing and maintaining databases. The salient issues addressed by the working group during the week were:

- Lack of, and need for, secondary standards regarding the problem of interfacing Ada with commercial products and existing standards;
- lack of language features to support secondary standards; and
- lack of explicit language mechanisms for decimal numeric processing.

II. Specific Problems

Eight specific problem areas were addressed during the workshop. This section specifies each individual problem; points out the application domain related to that problem; where appropriate. shows how the problem relates to the Steelman requirement(s); and discusses workarounds.

A. The problem is that although Ada is already large, it constantly needs new capabilities in specific application domains. Ada itself can be expected to change slowly, every decade or so, and cannot possibly anticipate its needs to co-exist and interface with new technology. This problem points to the need for a new class of Ada related standards: *Secondary Standards.*

1. Application Domain: The Information Systems community (database applications, payroll, logistics, funds transfer, contracting, financial accounting, tax accounting, budgetary planning, and many large government applications), C3I community, graphics, existing and future software environments.

2. Reference to Steelman: Page 22, paragraph 13G. "Software tools and application packages. The language should be designed to work in conjunction with a variety of useful software tools and application support packages. These will be developed as early as possible and include editors, interpreters, diagnostic aids, program analyzers, documentation aids, testing aids, software maintenance tools, optimizers, and application libraries. There will be a consistent user interface for these tools. Where practical, software tools and aids will be written in the language. Support for the design, implementation, distribution, and maintenance of translators, software tools and aids, and application libraries will be provided independently of the individual projects that use them."

3. Current Workaround: Currently the user depends on vendor supplied packages, run-time kludges, or writes his own ad-hoc packages.

Applications written for specific application domains have greatly diminished chances for reuse and portability, making the user, the application, and the programmer heavily dependent on current vendors and software environments.

B. The Ada language does not allow for consistent standards for math or statistical packages.

1. Application Domain: Primarily scientific and business applications developers, including the simulation/modeling area.

2. Reference to Steelman: Page 12, paragraph 12. "Library. There shall be an easily accessible library of generic definitions and separately translated units. ...Library entries may include ... application oriented software packages..."

3. Current Workarounds: Vendor packages, user written packages. There is no uniform naming conventions and too much duplication of effort.

C. Ada I/O packages are incomplete, and certain widely used I/O capabilities are not part of the Ada language.

1. Application Domain: Currently the Information Systems area makes use of I/O capabilities unavailable in Ada (such as Indexed I/O). These capabilities are readily available in other languages leaving Ada in a non-competitive position. The C3I area is in a similar situation with respect to Stream I/O and Terminal I/O.

2. Reference to Steelman: Page 16, Paragraph 8B. "User Level Input-Output. The language shall specify (i.e. give calling format and general semantics) a recommended set of user level input-output operations."

3. Current Workaround: By using the current Ada I/O, the user is assured of high-overhead, as well as simplistic and incomplete functionality. The alternative for the user is to use existing implementations of I/O capabilities supplied by vendors (non-portable), or write his own (error-prone and non-portable.)

D. The lack of consistent interface standards makes programs which must interface to existing standards non-portable and also makes the programmers non-portable.

1. Application Domain: The Information Systems, C3I, graphics and other areas requiring large scale software development in conjunction with existing standards (such as SQL, PHIGS, POSIX, and many others).

2. Reference to Steelman: Page 22, Paragraph 13G. "Software tools and application packages. The language should be designed to work in conjunction with a variety of useful software tools and application support packages. These will be developed as early as possible and include editors, interpreters, diagnostic aids, program analyzers, documentation aids, testing aids, software maintenance tools, optimizers, and application libraries. There will be a consistent user interface for these tools. Where practical, software tools and aids will be written in the language. Support for the design, implementation,

distribution, and maintenance of translators, software tools and aids. and application libraries will be provided independently of the individual projects that use them."

3. Current Workaround: The user is largely on his own, writing one-of-a-kind and fairly complex bridges to existing standards.

E. No mechanism exists for passing Ada subprograms or task entries to a non-Ada process.

1. Application Domain: Application systems which need to interface to standards that require gaining control of the Ada application. This environment is common in Information Systems, C2 and graphics areas.

2. Reference to Steelman: Page 22, Paragraph 13G. "Software tools and application packages. The language should be designed to work in conjunction with a variety of useful software tools and application support packages. These will be developed as early as possible and include editors, interpreters, diagnostic aids, program analyzers, documentation aids, testing aids, software maintenance tools. optimizers, and application libraries. There will be a consistent user interface for these tools. Where practical, software tools and aids will be written in the language. Support for the design, implementation, distribution, and maintenance of translators, software tools and aids, and application libraries will be provided independently of the individual projects that use them".

3. Current Workaround: Not possible on many vendor platforms. When possible, low-level assembler kludges and intimate knowledge of the run-time systems are used for passing Ada subprograms or task entries to a non-Ada process.

F. There is no mechanism to provide for the orderly and controlled initialization/termination of Ada entities.

1. Application Domain: Information Systems, C3I and other areas making use of re-usable components and abstractions. Information Systems and C3I applications required by the environment to perform clean up services.

2. Reference to Steelman: Page 3, Paragraph 1B. "Reliability. The language should aid the design and development of reliable programs. The language shall be designed to avoid error prone features and to maximize automatic detection of programming errors. The language shall require redundant, but not dupli .tive, specifications in programs...."

3. Current Workaround: The user is responsible for the cleanup, writing his own error-prone code.

G. Arithmetic operations on monetary values can not be represented as decimal arithmetic. Numeric literals in decimal may not be represented exactly by fixed point types.

1. Application Domain: Database applications, payroll, logistics, funds transfer, contracting, financial accounting, tax accounting, budgetary planning, and many large government applications (such as social security entitlements and budget formulation).

2. Reference to Steelman: Page 8, Exact Arithmetic Paragraph 3-1G. "Fixed Point Scale. The scale or step size (i.e. the minimal representable difference between values) of each fixed point variable must be specified in programs and be determinable during translation. Scales shall not be restricted to powers of two." This requirement should be rephrased to require exact representation of values in base ten. Also 3-1F: "Fixed point numbers shall be treated as exact numeric values".

3. Current Workaround: The user writes custom, error prone Ada packages.

H. New Ada applications lack interoperability with existing applications written in other standard HOL (3405.1) and standard DBMS (such as SQL). Ada does not support Binary Coded Decimal (BCD) representation of exact decimal.

   1. Application Domain: Database applications, payroll, logistics, funds transfer, contracting, financial accounting, tax accounting, budgetary planning, and many large government applications (such as social security entitlements and budget formulation).

   2. Reference to Steelman: Page 8, Exact Arithmetic Paragraph 3-1G. "Fixed Point Scale. The scale or step size (i.e. the minimal representable difference between values) of each fixed point variable must be specified in programs and be determinable during translation. Scales shall not be restricted to powers of two." This requirement should be rephrased to require exact representation of values in base ten. Also 3-1F: "Fixed point numbers shall be treated as exact numeric values"

   3. Current Workaround: Presently the user writes highly inefficien ıd non-portable Assembler level subroutines. Inefficiencies stem from:

      - scaling at runtime, instead of compile time
      - lack of <universal-fixed> for these values
      - operations (*,/) must be done in Max precision
      - converting BCD to binary is (1) very inefficient in Ada and (2) causes unnecessary overhead in many Information Systems applications.

III. Requirements and Justification

This section lists the new language requirements that the working group feels are necessary to meet the eight specific problems discussed in Section 6.2. First, the problem is restated. Four subsections follow for each specific problem: statement of the requirement, proposed language change(s), justification for the change(s) and any minority views that were expressed during the workshop sessions.

A. The problem is that Ada is already large, yet it constantly needs new capabilities in specific application domains. Ada itself can be expected to change slowly, every decade or so, and cannot possibly anticipate its needs to co-exist and interface with new technology. This problem points to the need for a new class of Ada related standards: *Secondary Standards*.

1. Statement of Requirement: An open process shall be established for creating and maintaining secondary standards, coordinated as necessary with national and international standards organizations.

   In this case, a secondary standard is characterized by:

   - a set of Ada specifications;
   - the semantics for that set of specifications; and
   - conformance criteria for validating an implementation of the set of specifications.

   The Ada community shall establish mechanisms to facilitate the interfacing of Ada to other established standards. Either the Ada 9X Project Office or the AJPO shall assume responsibility for establishing a secondary standard forum.

2. Proposed Language Changes: There are no specific LRM modifications to support this requirement. Future changes in specific application domains or software environments should not be affected by changes to the LRM. Although secondary standards are capabilities which are not part of the LRM, they do require standardization. These capabilities will prevent the language from exploding or ballooning.

3. Justification for Change: Currently, the Information Systems area makes use of specific capabilities unavailable in Ada (such as Indexed I/O), and relies heavily on existing standards (such as SQL, LU6.2) for which no standardized bridge from Ada is available. The graphics and C3I areas have similar needs with respect to standards (such as GKS or PHIGS) or math/stat capabilities. There will only be a few commercial large Ada systems implemented in these application domains unless Ada provides a uniform approach to existing standards and competitive capabilities. To summarize, secondary standards fulfill a number of requirements in the Information Systems arena:

   a. Extend the capabilities of the Ada language (e.g. I/O including possibly Chapter 14 as a secondary standard).

   b. Interface in a uniform manner with commercial products which implement existing standards.

   c. Perform essential functions of a specific, widespread application domain.

   The impact on compiler vendors will be beneficial because compiler vendors will not bear the burden of potentially large and radical changes to the LRM. In addition, compiler vendors will be able to offer competitive products in new markets represented by specific application domains. There is no negative impact on the software engineering community, as no existing code will have to be modified. There is considerable benefit to be derived because of the opportunity for much greater reuse and portability in future applications. This change should facilitate the standardization process because secondary standards will make Ada competitive in many application domains.

4. Minority Views: None

B. The Ada language does not allow for consistent standards for math or statistical packages.

   1. Statement of Requirement: There is a requirement to support essential commonly used mathematical and statistical functions in the Ada language. At this time, Ada lacks essential features to perform a full range of mathematical and statistical functions (square root, trigonometric functions, pseudo-random number generators, etc.)

   2. Proposed Language Changes: If a standard math/stat package is integrated into the Ada language, the changes will be extensive. Assuming a math/stat secondary standard, no changes to the LRM will be required.

   3. Justification for Change: It is self-evident that the Information Systems area and, in fact, all Ada users are potential beneficiaries of a math/stat secondary standard. A math/stat secondary standard will have little impact on current compiler technology. For vendors who already offer an equivalent package, one can expect mostly cosmetic changes. No changes are necessary to existing Ada programs. Wider use of the language would occur due to 'general purpose' functions being added.

      Future ISO or ANSI standards covering the same functionality can be reflected in math/stat secondary standards. One can envision several secondary st[illegible]s with increasing functionality and no noticeable disruption [illegible]a.

   4. Minority Views: None

C. Ada I/O packages are incomplete, and certain widely used I/O capabilities are not part of the Ada language.

   1. Statement of Requirement: The I/O in the Ada language should be improved and extended to include required capabilities. A uniform and portable approach to I/O is essential for Information Systems applications.

   2. Proposed Language Changes: Ada I/O should include improved TEXT_IO, DIRECT_IO, and SEQUENTIAL_IO. It should be extended to include Stream I/O, Indexed I/O, and Terminal I/O.

TEXT_IO should be improved to remove some machine dependencies such as 7 bit characters. DIRECT_IO must require support for instantiation with unconstrained record types. SEQUENTIAL_IO should include an APPEND operation. New Stream I/O, Indexed I/O and Terminal I/O capabilities should be defined and included as secondary standards. The listed I/O requirements are not presumed to be exhaustive. Other I/O capabilities may be added as their need becomes obvious. Ada I/O (LRM, Chapter 14) should be removed from the standard Ada LRM and included in secondary standards. A predefined package supporting capabilities common to all I/O (such as File_Exists, Can_Read, File_Form) should be included in the LRM. Future I/O requirements and changes should be controlled by a secondary standards body.

3. Justification for Change: I/O is required by most Information Systems and C3I applications. Ada is at a significant disadvantage compared to languages such as COBOL and PL/1 because of its poor and incomplete I/O. Current workarounds are error-prone and/or non-portable. The required I/O packages will make Ada a serious contender in these application domains. Ada will grow exceedingly large if the required I/O packages are made part of the LRM, even though one can be sure that new I/O requirements will become obvious in a short while. Assuming that I/O packages will be defined as secondary standards, the impact on vendors will be minimal.

   Consistent and uniform I/O packages across platforms will greatly increase the portability and reuse potential of new applications in the Information Systems and C3I areas. No existing Ada code will require modifications. The required changes to Ada will also facilitate the standardization process because it will make the language significantly more suitable for the intended application domains.

4. Minority Views: Two minority views were expressed:

   a. The ability to define standard I/O capabilities may be too difficult to be successful.

   b. Deferring I/O standards to a secondary body removes the control exerted by Ada 9X on Ada changes.

D. The lack of consistent interface standards makes programs which must interface to existing standards non-portable and also makes the programmers non-portable.

1. Statement of Requirement: Ada shall interface or bind to existing or emerging standards.

   Initially Ada must interface to X-Windows, POSIX, SQL, graphics (GKS, PHIGS), and communication protocols (LU6.2, IRDS). DoD has mandated the heavy use of standardized COTS products and other government agencies promote the use of these standards and COTS.

2. Proposed Language Changes: Ada shall provide bindings to SQL, X-Windows, POSIX, GKS, PHIGS, LU6.2, and IRDS. Since these bindings would make the LRM explode, they should be handled as secondary standards. This list of bindings to existing standards is not intended to be exhaustive.

3. Justification for Change The development of large mainframe Information Systems applications typically involve interfacing with COTS relational DBMS (such as SQL) and/or communication protocols (such as LU6.2). Ada application development in the Information Systems, C3I and other areas require interfacing with standards such as X-Windows, POSIX, GKS, PHIGS, LU6.2 and IRDS. Without a disciplined and uniform (standard) approach, any ad-hoc Ada interface to existing standards, if at all possible, will effectively negate Ada's advantages.

   Assuming that bindings to existing standards will be defined as secondary standards, the impact on vendors will be minimal. Bindings to secondary standards are essential for promoting portability and reuse in the Information Systems, C3I and other areas. No existing Ada code will require modifications. Also, the required bindings will facilitate the standardization process because it will make Ada significantly more suitable for the intended application domains.

4. Minority Views: The bindings contemplated cannot be accomplished since no organization currently has the necessary funding/manpower to carry out this requirement.

E. No mechanism exists for passing Ada subprograms or task entries to a non-Ada process.

1. Statement of Requirement: There shall be an Ada language mechanism for passing Ada subprograms and task entries to a non-Ada program.

2. Proposed Language Changes: The required language change(s) are complex and should be given more focused study within the Ada 9X Project.

3. Justification for Change: Existing Information Systems and command and control environments provide interfacing to non Ada programs such as X-Windows, PHIGS and DBMS. These non-Ada programs require gaining control of the Ada application. Application development environments and target environments such as X-Windows and PHIGS require the handling of multiple threads of control and control flow in both directions. Some DBMS have capabilities to operate asynchronously and require control of the Ada application and termination of operations.

   Compilers will be more constrained in implementation decisions, thus potentially impacting existing compilers. Compilers do not have a defined set of rules for where subprograms and entries are and how they are to be called; these must be defined in order to allow for this capability. Tasking run-time systems may need to be modified to avoid conflicts with the external multiple threads of control.

Having this capability will avoid the use of obscure workarounds when workarounds are available, and remove explicit knowledge of implementation details. This capability is not available without leaving the Ada system in the current language. The capability, where provided by existing systems, is not portable. As stated in Steelman [13G], "the language should be designed to work in conjunction with a variety of useful software tools and application support packages". This will facilitate the standardization process, because it removes a common complaint generated by the users in the Information Systems, C3I and CAD communities.

4. Minority Views: None

F. There is no mechanism to provide for the orderly and controlled initialization/termination of Ada entities.

1. Statement of Requirement: There shall be an explicit Ada mechanism to perform necessary processing when entities (such as data objects, packages and tasks) come into and go out of existence.

2. Proposed Language Changes: Language changes will be difficult for data objects, packages and tasks. Possible syntactic forms are:

```
  type <identifier> is <type_definition> [:=initialization_data]
   [final <subprogram_specification>] :

package body <identifier> is [<declarative_part>]
 [ begin <seq_of_statements>
 [ final <seq_of_statements>
 [ exception <exception_handler>
       {<exception_handler>} ]]]
 end [<identifier>];

task  body <identifier> is [<declarative_part>]
 [ begin <seq_of_statements>
 [ final <seq_of_statements>
 [ exception <exception_handler>
       {<exception_handler>} ]]]
 end [<identifier>];
```

The process of initialization will be performed at elaboration and finalization at de-elaboration.

3. Justification for Change: A variety of applications in the Information Systems, graphics and C3I areas require this capability. For instance, Information Systems environments (DBMS) need to know when applications are done. In the graphics area, PHIGS forces application programs to perform reclamation of storage allocated by PHIGS services. Real time C3 applications need to perform necessary cleanup

following software failures prior to recovery/restart activities.

For most entities, the impact on compilers will be minimal. In the case of tasks, the impact of the change by itself will be significant; however, if changes to the tasking model are made in conjunction with requirements generated by other areas, it may well turn out that the changes are not extensive. There may be an additional impact on Ada because of newly created keywords.

In the case of objects, compilers must already handle initialization when they are of record, access and task types, and must provide finalization for task objects. Thus, adding this feature in the general case is not too difficult. In the case of packages, compilers already handle elaboration; finalization ("de-elaboration") could be handled as an analog, therefore minimizing the impact. In the case of tasks, other likely language changes will probably force task runtime environments to be rewritten anyway; the additional impact will be minimal.

The added capability will enhance the creation of bullet proof reusable software. These capabilities also make the language more consistent. The ability to finalize will allow encapsulation to be fully handled within the unit. Currently clean up (deallocation) has to be manually done by the user of the encapsulation; this is error prone. Also, some designs are impossible because appropriate choices are not available, or the costs (efficiency in time/space) are prohibitive. These capabilities allow the logical separation of clients from servers. This change will also facilitate the standardization process because it will remove barriers to interoperability between Ada programs and other international and national standards.

4. Minority Views: None

G. Arithmetic operations on monetary values can not be represented as decimal arithmetic. Numeric literals in decimal may not be represented exactly by fixed point types.

1. Statement of Requirement: Ada shall support exact decimal representations and associated operations.

2. Proposed Language Changes: Add an attribute ('Radix) to fixed point types. The value of T'Radix is the base (or radix) of the machine representation of the type. Modify the definition of fixed point model numbers (LRM 3.5.9(6)) such that the value B is defined in terms of 'Radix digits (rather than binary digits). Modify the 'Mantissa attribute definition accordingly (this should allow 'Mantissa to return decimal digits). Otherwise, the definition of fixed point semantics should not change.

3. Justification for Change: Large scale financial and monetary applications will not be implemented in Ada unless this change is made. The change to the language is essential. Compilers must recognize the conditions for generating fixed decimal code or software to emulate

fixed decimal operations. They must also recognize the conditions for generating fixed decimal code or software to emulate fixed decimal operations. This change will simplify the implementation of business applications and will facilitate the use of Ada for large scale applications.

This change will enable Ada programmers to take advantage of fixed decimal hardware available on many general purpose computers. It should also facilitate the standardization process because it removes a common complaint generated by financial and monetary user communities. Cleaner semantics for exact decimal representations when using fixed point types will be provided, as well as non-binary representation of fixed point values.

4. Minority Views: Considerable discussion was raised over the question of overflow detection on intermediate calculations. Some participants felt that overflow of intermediate calculations must be made known to the executing program.

H. New Ada applications lack interoperability with existing applications written in other standard HOL (3405.1) and standard DBMS (such as SQL). Ada does not support Binary Coded Decimal (BCD) representation of exact decimal.

1. Statement of Requirement: Ada shall have the ability to specify Binary Coded Decimal (BCD) representation of fixed point types.

2. Proposed Language Changes: Add an attribute ('Radix) to fixed point types. The value of T'Radix is the base (or radix) of the machine representation of the type. Modify the definition of fixed point model numbers (LRM 3.5.9(6)) such that the value B is defined in terms of 'Radix digits (rather than binary digits). Modify the 'Mantissa attribute definition according (this should allow 'Mantissa to return decimal digits). Otherwise the definition of fixed point semantics should not change. In addition the following alternatives may be considered.

Solution 1: Pragma Decimal (<fixed point type mark>)

Solution 2: for <fixed point type typemark>'Radix use 10.

In both cases, the compiler must use decimal representation or may report a semantic error. (If the program executes, it must use decimal.)

3. Justification for Change: Large scale financial and monetary applications will not be implemented in Ada unless this change is made. The change to the language is essential.

Compilers must recognize the conditions for generating fixed decimal code or software to emulate fixed decimal operations. They must also recognize the conditions for generating fixed decimal code or software to emulate fixed decimal operations. This change will simplify the implementation of business applications and will facilitate the use of

Ada for large scale applications.

This change will enable Ada programmers to take advantage of fixed decimal hardware available on many general purpose computers. It will also enhance the interoperability with other DBMS standards and other DoD approved languages (such as COBOL).

4. Minority Views: Three minority views were expressed:

   a. BCD should not be supported at all.

   b. Compilers must use BCD when asked.

   c. Compilers may accept the request but not act on it (but must set 'Radix).

IV. List of Working Group Members

— Eugen Vasilescu, Chairman, Grumman Data Systems - USA
— Cathy McDonald, Facilitator, IDA - USA
— Benjamin M. Brosgol, Alsys - USA
— Randall L. Brukardt, R.R.Software - USA
— Marc Graham, SEI - USA
— Fred Hathorn, Army, SPOD/ISSC - USA
— Butch Higley, AIRMICS - USA
— Michael Iaeger, Air Force, Space and Warning System Center - USA
— David Moore, SYCON Corp. - USA
— Howard Stewart, Idaho National Engineering Lab - USA
— Dana Wilson, Cristie - UK

APPENDIX A

Ada 9X Project Requirements Workshop Agenda

WORKSHOP AGENDA
22-26 MAY 1989

Monday, 22 May 1989

0800-0900	Registration	
0900-1000	Welcome	Ms Chris Anderson *Ada 9X Project Manager*
	Guest Speaker Topic: STEELMAN	Dr David Fisher *Incremental Systems Corp.*
1000-1030	Break	
1030-1200	Orientation Introductions: Working Group Chairs Working Group Sessions Working Group Products	
1200-1330	Lunch (Group Function) Guest Speaker	Dr John Solomond *Director-Elect, AJPO*
1330-1400	Free Time	
1400-1530	Working Group Session	
1530-1600	Break	
1600-1730	Working Group Session	
1730-1800	Working Group Chair Reports (5 Minutes per Group) Trusted Systems/Verifications Software Eng. in the Large Real-Time Embedded Systems Parallel Systems Information Systems	
	Dinner (on your own)	

Tuesday, 23 May 1989

0830-1000	Working Group Session
1000-1030	Break
1030-1200	Working Group Session
1200-1400	Lunch and Free Time (on your own)
1400-1530	Working Group Session
1530-1600	Break
1600-1630	Working Group Chair Reports (5 Minutes per Group) Trusted Systems/Verifications Software Eng. in the Large Real-Time Embedded Systems Parallel Systems Information Systems
1900-	Dinner (Group Function) Eglin AFB Officer's Club Guest Speaker — Dr Brian Wichman, *National Physics Laboratory*, United Kingdom Topic: *Ada the Person*

Wednesday, 24 May 1989

0830-1000	Working Group Session
1000-1030	Break
1030-1200	Working Group Session
1200-1400	Lunch and free time (on your own)
1400-1530	Working Group Session
1530-1600	Break
1600-1630	Working Group Chair Reports (5 Minutes per Group) Trusted Systems/Verifications Software Eng. in the Large Real-Time Embedded Systems Parallel Systems Information Systems
	Dinner (on your own)

Thursday, 25 May 1989

0830-1000	Working Group Session
1000-1030	Break
1030-1200	Working Group Session
1200-1400	Lunch and free time (on your own)
1400-1530	Working Group Session
1530-1600	Break
1600-1630	Working Group Chair Reports
	(5 Minutes per Group)
	Trusted Systems/Verifications
	Software Eng. in the Large
	Real-Time Embedded Systems
	Parallel Systems
	Information Systems
	Dinner (on your own)

Friday, 26 May 1989

0830-1000	Working Group Reports
	(30 Minutes per Group)
1000-1030	Break
1030-1200	Working Group Reports
1200-1230	Concluding Reports
1230	Adjourn