G Numerics

The Numerics Annex addresses the particular needs of the numerically intensive computing community. Like the other specialized needs annexes, support of this annex is optional. The annex covers the following topics

- Various generic packages are provided for the manipulation of complex numbers including the computation of elementary functions and input-output.

- The annex specifies two modes of operation, a strict mode in which certain accuracy requirements must be met and the relaxed mode in which they need not be met. The accuracy requirements for the strict mode cover both arithmetic and the noncomplex elementary functions and random number generation of the core language.

- The models of floating point and fixed point arithmetic applicable to the strict mode are described; these differ from those of Ada 83.

- Various model attributes are defined which are applicable to the strict mode for floating point types; again these differ from Ada 83.

Note that since the elementary functions and random number generation are in the core language, they and their accuracy requirements are discussed elsewhere (see A.3). The majority of attributes (including the so-called "primitive function" attributes) are also defined in the core language.

Implementations conforming to the numerics annex should also support the package `Interfaces.Fortran`, which is discussed in B.4.

G.1 Complex Arithmetic

Several application areas depend on the use of complex arithmetic. Complex fast Fourier transforms are used, for example, in conjunction with radar and similar sensors; conformal mapping uses complex arithmetic in fluid-flow problems such as the analysis of velocity fields around airfoils; and electrical circuit analysis is classically modelled in terms of complex exponentials.

The Ada 95 facilities for complex arithmetic consist of the generic packages

```
Numerics.Generic_Complex_Types
Numerics.Generic_Complex_Elementary_Functions
Text_IO.Complex_IO
```

which are children of `Ada`.

G.1.1 Complex Types and Arithmetic Operations

When first designed, `Numerics.Generic_Complex_Types` was patterned after the version of the generic package, `Generic_Complex_Types`, that was then being developed in the SIGAda Numerics Working Group for proposal as an ISO standard for Ada 83. At that time, roughly mid-

1992, the latter defined a complex type as well as types for vectors and matrices of complex components, together with a large set of scalar, vector, and matrix operations on those types. A decision was made to abbreviate the Ada 95 package by omitting the vector and matrix types and operations. One reason was that such types and operations were largely self-evident, so that little real help would be provided by defining them in the language. Another reason was that a future version of Ada might add enhancements for array manipulation and so it would be inappropriate to lock in such operations prematurely.

The initial design for `Numerics.Generic_Complex_Types` also inherited a rather pedestrian approach to defining the complex type from the same proposed standard. The Ada 9X Distinguished Reviewers recommended a different approach that enabled the writing of expressions whose appearance closely matches that of the standard mathematical notation for complex arithmetic. The idea was to define not just a complex type, but also a pure imaginary type `Imaginary` and a constant `i` of that type; operations could then easily be defined to allow one to write expressions such as

```
    3.0 + 5.0*i   -- of type Complex
```

which has the feel of a complex literal.

Another advantage of this approach is that by providing mixed mode operations between complex and imaginary as well as between complex and real, it is possible to avoid unnecessary arithmetic operations on degenerate components. Moreover, avoiding such unnecessary operations is crucial in environments featuring IEEE arithmetic [IEC 89], where signed infinities can arise and can be used in meaningful ways.

(Ada 95 does not support infinite values, but in an implementation in which the `Machine_Overflows` attribute is `False`, an overflow or a division by zero yields an implementation-defined result, which could well be an infinite value. Thus a future Ada binding to IEEE arithmetic could capitalize on exactly that opportunity by providing semantics of arithmetic with infinities as in [IEC 89].)

To see how avoiding unnecessary operations provides more than just a gain in efficiency, consider the multiplication of a complex value $x + iy$ by a pure imaginary value iv. The result of the multiplication should, of course, be $-vy + ivx$. Without a pure imaginary type, we have to represent the pure imaginary value as the complex value $0.0 + iv$ and perform a full complex multiplication, yielding $(0.0x-vy) + i(vx+0.0y)$. This, of course, reduces to the same value as before, unless x or y is an infinity.

However, if x is infinity, y is finite, and v is nonzero (say, positive), the result should be $-vy + i$ ¥, but instead we get $NaN + i$ ¥, since multiplication of zero and infinity yields a NaN ("Not-a-Number") in IEEE arithmetic, and NaNs propagate through addition. See [Kahan 91].

A similar situation can be demonstrated for the multiplication of a complex value and a pure real value, but in that case we expect to have such a mixed-mode operation, and if we use it the generation of the NaN is avoided.

Another subtle problem occurs when the imaginary value iv is added to the complex value $x + iy$. The result, of course, should be $x + i(y+v)$. Without an imaginary type, we have to represent the imaginary value as the complex value $0.0 + iv$ and perform a full complex addition, yielding $(x+0.0) + i(y+v)$. The problem here [Kahan 91] is that if x is a negative zero, the real component of the result of the full complex addition will have the wrong sign; it will be a positive zero instead of the expected negative zero. This phenomenon, also a consequence of the rules of IEEE arithmetic, can and does occur in existing Ada 83 implementations, since it does not require an extension permitting infinities.

In both cases, the pure imaginary type gives the programmer the same opportunity to avoid problems in mixed complex/imaginary arithmetic as in mixed complex/real arithmetic.

With the inclusion of a pure imaginary type and mixed complex/imaginary operations, the generic complex types package in Ada 95 could have diverged from the proposed standard under development in the SIGAda NumWG. This was averted, however, when the working group changed its proposed standard to agree with the Ada 95 version. It also removed the vector and matrix types and operations from its generic complex types package and put them in a separate package. And so, `Numerics.Generic_Complex_Types` is just a slight variation of the generic package proposed for standardization for Ada 83. (The differences have to do with the use of `Real'Base`, rather than just `Real`, as the subtype mark for the components of the complex type and for the parameter and result subtypes of some of the operations, `Real` being the name of the generic formal parameter. This capability is lacking in Ada 83.)

The type `Complex` defined by `Numerics.Generic_Complex_Types` is a visible record type thus

```
type Complex is
   record
      Re, Im: Real'Base;
   end record;
```

corresponding to the cartesian representation of a complex value. We have made the type visible to allow one to write complex "literals" using aggregate notation as in some other languages (and to ensure efficiency). The cartesian representation was chosen over a polar representation to avoid canonicalization problems and because it is normal practice. An explicit choice of representation is required in any case to give meaning to the accuracy requirements. Operations are provided, however, to compute the modulus (length) and argument (angle) of a complex value and to construct a complex value from a given modulus and argument, so that it is easy to convert between the built-in cartesian representation and a polar representation, if needed.

It is perhaps unusual that the components of the complex type are of the unconstrained subtype of the generic formal parameter, `Real'Base`, rather than just `Real`, but this is intended to increase the chances of being able to deliver the result computed by a complex arithmetic operation even when their operands belong to some restricted domain. This provides behavior analogous to that of the elementary functions (see A.3.1), which also yield results in the unconstrained subtype of the relevant generic formal parameter. It is also similar to the behavior of the predefined arithmetic operations, which yield results of an unconstrained subtype, even when their operands are of a constrained subtype. A consequence is that we cannot create complex types with constrained components, but that does not seem so severe in view of the fact that applications of complex arithmetic typically have a naturally circular domain, rather than a rectangular domain.

The type `Imaginary`, on the other hand, is private, its full type being derived from `Real'Base` thus

```
type Imaginary is new Real'Base;
```

Making it private prevents the implicit conversion of a real literal to the type `Imaginary`, which would be available if `Imaginary` were visibly derived from `Real'Base`. This avoids various ambiguous expressions and enables overload resolution to work properly. It has the additional advantage of suppressing the implicit declaration of multiplying operators for the `Imaginary` type, which would incorrectly yield a result of the type `Imaginary`, when it should be `Real'Base`. Operations with the correct result type such as

```
function "*" (Left, Right: Imaginary) return Real'Base;
```

are, of course, explicitly declared. The same benefits could have been achieved by defining `Imaginary` as a visible record type with a single component, but the method chosen prevents the writing of expressions containing pure imaginary values as aggregates, whose meaning would not be intuitively obvious.

The imaginary constant `i` (and its equivalent, `j`, provided for the engineering community), has the value `1.0`, and so unoptimized expressions like `5.0*i` will have the proper numerical value. However, it is expected that compilers will optimize this and effectively convert the real literal `5.0` to the imaginary type. Similarly, an expression such as `3.0 + 5.0*i` can be optimized to perform no arithmetic at all since it is functionally equivalent to the aggregate `(3.0, 5.0)` of the type `Complex`.

Note that there are also constructor and selector functions such as `Compose_From_-Cartesian`. The following expressions are thus equivalent

```
X + i*Y                              -- using operators

(X, Y)                               -- using an aggregate

Compose_From_Cartesian(X, Y)         -- using the constructor
```

The constructor function has the merit that it can be used as a generic actual parameter, if it should be necessary.

Nongeneric equivalents of `Numerics.Generic_Complex_Types` are provided corresponding to instantiations with the predefined types `Float`, `Long_Float` and so on with names such as

```
Numerics.Complex_Types            -- for Float
Numerics.Long_Complex_Types       -- for Long_Float
```

This means that applications can effectively treat single-precision complex, double-precision complex, etc., as predefined types with the same convenience that is provided for the predefined floating point types (or, more importantly, so that independent libraries assuming the existence and availability of such types without the use of generics can be constructed and freely used in applications). The nongeneric forms also have the advantage that Fortran programmers migrating to Ada do not have to learn generics in order to use complex arithemtic.

Accuracy requirements are generally specified for the complex arithmetic operations only in the strict mode. Nevertheless, certain special cases are prescribed to give the exact result even in the relaxed mode, ensuring high quality at negligible implementation cost. Examples are where one operand is pure real or pure imaginary. (These prescribed results are likely to be achieved even without special attention by the implementation.) Accuracy requirements are not given for exponentiation of a complex operand by an integer, owing to the variety of implementations that are allowed (ranging from repeated complex multiplication to well-known operations on the polar representation).

Note finally that spurious overflows (those occurring during the computation of an intermediate result, when the final result, or its components, would not overflow) are not allowed. Thus, implementations of complex multiplication and division need to be somewhat more sophisticated than the textbook formulae for those operations.

G.1.2 Complex Elementary Functions

The package `Numerics.Generic_Complex_Elementary_Functions` differs from the corresponding proposed standard for Ada 83 by taking advantage of the formal package parameter facility of Ada 95. Thus it imports the one parameter which is an instance of `Numerics.Generic_Complex_Types` instead of the complex type and a long list of operations exported by such an instance.

In the Ada 83 version, the complex type has to be imported as a private type, and implementations of the complex elementary functions there have no choice but to use the imported selector functions, `Re` and `Im`, to extract the real and imaginary components of a complex value, and the imported constructor function, `Compose_From_Cartesian`, to assemble such components into a complex value. Implementations of the Ada 95 version see that type as the record type that it is, allowing more efficient composition and decomposition of complex values; they also have available the complete set of operations, not just the partial (albeit long enough) list of operations imported in the Ada 83 version.

Nonngeneric equivalents of `Numerics.Generic_Complex_Elementary_Functions` are also provided for each of the predefined floating point types.

The overloading of the `Exp` function for a pure imaginary parameter is provided to give the user an alternative way to construct the complex value having a given modulus and argument. Thus, we can write either

```
Compose_From_Polar(R, Theta)
```

or

```
R * Exp(i * Theta)
```

where the latter corresponds more naturally to the mathematical $Re^{i\theta}$.

The treatment of accuracy requirements and prescribed results for the complex elementary functions is analogous to that discussed above for the complex arithmetic operations. However, since the avoidance of spurious overflows is difficult and expensive to achieve in several of the functions, it is explicitly permitted, allowing those functions to be implemented in the obvious way. No accuracy requirement is imposed on complex exponentiation (the operator `"**"`) by a pure real, pure imaginary, or complex exponent, because the obvious implementation in terms of complex exponentials and logarithms yields poor accuracy in some parts of the domain, and better algorithms are not available.

G.1.3 Complex I/O

Complex I/O is performed using the procedures in the generic child package, `Text_IO.Complex_IO`. As with `Numerics.Generic_Complex_Elementary_Functions`, the user instantiates this generic package with an instance of `Numerics.Generic_Complex_Types`. (Note that nongeneric equivalents do not exist.)

A fundamental design decision underlying `Text_IO.Complex_IO` is that complex values are represented on the external medium in parenthesized aggregate notation, as in Fortran list-directed I/O. This is the format produced on output, and it is the format expected on input, except that the comma, the parentheses, or both may be omitted when the real and imaginary components are separated by appropriate white space. (This allows the reading of existing Fortran files containing complex data written by a variety of techniques.)

An implementation of `Text_IO.Complex_IO` can easily be built around an instance of `Text_IO.Float_IO` for `Real'Base`; the optional parentheses on input requires the use of the procedure `Text_IO.Look_Ahead`; see A.4.2.

`Text_IO.Complex_IO` defines similar `Get` and `Put` procedures to `Text_IO.Float_IO` with analogous semantics. The only somewhat arbitrary decision that we made concerns the mechanism for filling the target string in the case of output to a string. The question is where to put any extra blanks that are needed to fill the string. A rule like the one for `Text_IO.Float_IO.Put` to a string might read:

> *Outputs the value of the parameter* `Item` *to the given string, following the same rule as for output to a file, using a value for* `Fore` *such that the sequence of characters output exactly fills, or comes closest to filling, the string; in the latter case, the string is filled by inserting one extra blank immediately after the comma.*

Such a rule essentially allocates the available space equally to the real and imaginary components. But that is not desirable when the value of the `Exp` parameter is zero and the two components of the `Item` parameter have disparate magnitudes, so that the integer part of one requires more characters than the integer part of the other. To accommodate this case, we have chosen a rule, simple to implement, that left justifies the left parenthesis, real component, and comma and right justifies the imaginary component and right parenthesis. All the extra spaces are placed between the comma and the imaginary component; they are available to accommodate either component, should it be unusually long. If strings produced by this rule are eventually displayed in the output, the real and imaginary components will not line up in columns as well as with the previously cited rule, but it can be argued that output to a string is intended for further computation, rather than for display.

Early implementation experience indicated that it might also have been profitable to consider yet another rule for output to a string, namely, that all the components be right justified, with all the padding placed at the beginning of the string. This rule would be extremely easy to implement if `Text_IO.Float_IO.Put` to a string (which right justifies its output in the target string), had an additional out parameter which gave the index of the first non-blank character that it put into the output string.

G.2 Floating Point Machine Numbers

Many of the attributes of floating point types are defined with reference to a particular mathematical representation of rational numbers called the *canonical form*, which is defined, together with those attributes, in the Predefined Language Environment. The definitions clarify certain aspects of the floating point machine numbers. Several new representation-oriented attributes of floating point machine numbers are also defined, together with a group of functional attributes called the "primitive function" attributes.

G.2.1 Clarification of Existing Attributes

The machine numbers of a floating point type are somewhat informally defined as the values of the type that are capable of being represented to full accuracy in all unconstrained variables of the type. The intent is to exclude from the set of machine numbers any extra-precise numbers that might be held in extended registers in which they are generated as a consequence of performing arithmetic operations. In other words, it is the stored values that matter and not values generated as intermediate results.

The representation-oriented attributes `S'Machine_Mantissa`, `S'Machine_Emin`, and `S'Machine_Emax` of a floating point subtype `S` are defined in terms of bounds on the components of the canonical form. The attribute `S'Machine_Radix` is the radix of the hardware representation of the type and is used as the radix of the mantissa in the canonical form.

These definitions clarify that `S'Machine_Emin` is the minimum canonical-form exponent such that *all* numbers expressible in the canonical form, with that exponent, are indeed machine numbers. In other words, `S'Machine_Emin` is determined by the normalized floating point numbers only; the presence of IEEE denormalized numbers in the implementation does not affect (reduce) the value of `S'Machine_Emin`. A consequence of this definition of `S'Machine_Emin` is that the primitive function attribute `S'Exponent(X)` can yield a result whose value is less than that of `S'Machine_Emin` if `X` is denormalized.

The definitions also clarify that `S'Machine_Emax` is the canonical-form exponent of the machine number of largest magnitude whose negation is also a machine number; it is not the canonical-form exponent of the most negative number on radix-complement machines.

Alternative definitions for `S'Machine_Emin` and `S'Machine_Emax` were considered, namely, that they yield the minimum and maximum canonical-form exponents for which some combination of sign, exponent, and mantissa yields a machine number. This would have allowed denormalized numbers to be accommodated without relaxing the normalization requirement (see the next section) in the definition of the canonical form, and the result of `S'Exponent(X)` would have necessarily remained within the range `S'Machine_Emin .. S'Machine_Emax`, which is appealing. Nevertheless, it was judged to be too much of a departure from current practice and therefore too likely to cause compatibility problems.

G.2.2 Attributes Concerned With Denormalized Numbers and Signed Zeros

Many implementations of Ada do provide IEEE denormalized numbers and "gradual underflow", as defined in [IEC 89], even though the full capabilities of IEEE arithmetic are not provided and must await an Ada binding to IEEE arithmetic. (Denormalized numbers come for free with IEEE hardware chips and have always been consistent with the Ada model of floating point arithmetic; it would require extra code generation to suppress them.) Since denormalized numbers are capable of being stored in variables of an unconstrained floating point type, they are machine numbers. What characterizes the denormalized numbers is that they can be represented in the canonical form with an exponent of `S'Machine_Emin`, provided that the normalization requirement on the mantissa is relaxed. If every nonzero number expressible in this weakened canonical form is a machine number of the subtype `S`, then the new representation-oriented attribute, `S'Denorm`, is defined to have the value `True`.

Many implementations also provide IEEE signed zeros, which similarly come for free. The new representation-oriented attribute, `S'Signed_Zeros`, is `True` if signed zeros are provided by the implementation and used by the predefined floating point operations as specified in [IEC 89]. The idea behind a signed zero is that a zero resulting from a computation that underflows can retain the sign of the underflowing quantity, as if the zero represented an infinitesimal quantity instead; the sign of a zero quantity, interpreted as if it were infinitesimal, can then affect the outcome of certain arithmetic operations.

Moreover, various higher-level operations, such as the elementary functions, are defined to yield zeros with specified signs for particular parameter values when the `Signed_Zeros` attribute of the target type is `True`. And indeed, some such operations, for example, `Arctan` and `Arccot` produce different (nonzero) results in certain cases, depending on the sign of a zero parameter, when `Signed_Zeros` is `True`.

In some cases, no conventions exist yet for the sign of a zero result. We do not specify the sign in such cases, but instead require the implementation to document the sign it produces. Also, we have not carried over into the facilities for complex arithmetic and the complex elementary functions the specifications for the signs of zero results (or their components) developed by the SIGAda NumWG, largely because of their excessive complexity. Instead we merely suggest that implementations should attempt to provide sensible and consistent behavior in this regard (for example, by preserving the sign of a zero parameter component in a result component that behaves like an odd function).

G.2.3 The Primitive Function Attributes

A group of attributes of floating point types called the "primitive function" attributes is provided, in support of Requirement R11.1-A(1), to facilitate certain operations needed by the numerical specialists who develop mathematical software libraries such as the elementary functions.

These attributes are modelled on the various functions in the standard package `Generic_Primitive_Functions` for Ada 83 [ISO 94b], but made available in the language as attributes of a floating point subtype rather than as functions in a generic package. These attributes support

- error-free scaling by a power of the hardware radix,
- decomposition of a floating point quantity into its components (mantissa and exponent), and construction of a floating point quantity from such components,
- calculation of exact remainders, and various directed rounding operations.

All of these attributes yield the mathematically specified results, which are either machine numbers or have the accuracy of the parameters. For a general rationale for the design of the primitive functions, see [Dritz 91b]. (Some of the attributes have different names from the corresponding functions in [ISO 94b], since some of the names in the latter had already been used for other new, but unrelated, attributes.)

The casting of the primitive functions as attributes, rather than as functions in a generic package, befits their primitive nature and allows them to be used as components of static expressions, when their parameters are static.

The `Exponent` and `Fraction` attributes decompose a floating point number into its exponent and (signed) fraction parts. (These attributes, along with `Scaling` and `Leading_Part`, are useful in the argument reduction step of certain kinds of function approximation algorithms in high-quality mathematical software; `Scaling` and `Compose` are similarly useful in the final result assembly steps of such algorithms.)

`T'Exponent(X)` is defined in such a way that it gives an indication of the gross magnitude of `X`, even when `X` is denormalized. In particular, if `X` is repetitively scaled down by a power of the hardware radix, `T'Exponent(X)` will decrease by one and will continue to do so (on machines with denormalized numbers) even after `X` becomes denormalized. `T'Fraction(X)` will *not* change as `X` is scaled in this way, even when `X` becomes denormalized. To achieve this behavior, an implementation of `Exponent` must do more than just pick up the hardware exponent field and unbias it, and an implementation of `Fraction` must do more than just pick up the hardware fraction field; both attributes must be sensitive to denormalized numbers.

(Indeed, the presence or absence of the leading fraction digit is dependent on whether a number is denormalized or not, on IEEE hardware. Probably the most efficient solution to the problem is for the implementation of `Exponent` to scale the operand up by an appropriate fixed amount *k* sufficient to normalize it when the operand is in the denormalized range, as evidenced by its having the minimum exponent or by other means; extract and unbias the exponent field; and then subtract *k* to obtain the result. The implementation of `Fraction` can simply extract the fraction field after a similar scaling up when the operand is in the denormalized range, and then attach the appropriate fixed exponent; what the scaling accomplishes is the left shifting of the fraction field and the removal of its leading digit.)

The `Copy_Sign` attribute transfers the sign of one value to another. It is provided for those applications that require a sign to be propagated, even (on machines with signed zeros) when it originates on a zero; such a need cannot be met by the predefined comparison operators and various sign-changing operations (like **`abs`** and negation), because comparison ignores the sign of zero and therefore cannot be used to determine the sign of a zero. By the same token, the implementation of `Copy_Sign` must likewise use some other technique, such as direct transfer of the sign bit or some other examination of the `Sign` operand to determine its sign. An application can use the `Copy_Sign` attribute to determine the sign of a zero value, when required, by transferring that sign to a nonzero value and then comparing the latter to zero.

The `Remainder` attribute computes the remainder upon dividing its first floating point parameter by its second. It considers both parameters to be exact, and it delivers the exact remainder, even when the first parameter is many orders of magnitude larger than the second; for this reason, its implementation can be tricky. This attribute is useful in implementing the argument reduction step of algorithms for computing periodic functions (when the period is given exactly).

The function `T'Remainder(X, Y)` is defined so that the magnitude of the result is less than or equal to half the magnitude of `Y`; it may be negative, even when both parameters are positive. Note that the `Remainder` attribute cannot be considered to be a natural extension of the predefined **`rem`** operator for floating point operands (and, indeed, that is one reason why the functionality was not made available by overloading **`rem`**). To see this, observe that

```
    Float'Remainder(14.0, 5.0)    -- yields -1.0
```

whereas

```
    14 rem 5                      -- yields 4
```

and so are quite different. The rationale for defining `Remainder` this way is twofold: it exhibits the behavior preferred by numerical analysts (i.e., it yields a reduced argument of generally smaller magnitude), and it agrees with the IEEE rem operator. Indeed, the latter is implemented in hardware on some machines; when available, it should certainly be used instead of the painful alternative documented in [Dritz 91b].

The functionality of the `Successor` and `Predecessor` functions of [ISO 94b] is provided by extending the existing attributes `Succ` and `Pred` to floating point types. Note that `T'Succ(0.0)` returns the smallest positive number, which is a denormalized number if `T'Denorm` is `True` and a normalized number if `T'Denorm` is `False`; this is equivalent to the "fmin" derived constant of LIA-1 (Language Independent Arithmetic) [ISO 93]. (Most of the other constants and operations of LIA-1 are provided either as primitive functions or other attributes in Ada 95; those that are absent can be reliably defined in terms of existing attributes.)

G.3 Assignments to Variables of Unconstrained Numeric Types

Ada 83 did not make a distinction between unconstrained and constrained numeric subtypes. Any subtype `T` was considered constrained by the values given by `T'First` and `T'Last`; if no range constraint was declared for `T`, then `T'First = T'Base'First` and `T'Last = T'Base'Last`.

It was technically not possible for a variable of an unconstrained subtype to be assigned a value outside the range `T'Base'First .. T'Base'Last`. This prevented the optimization of leaving a value in an extended register beyond an assignment, fulfilling subsequent references to the target variable from the register. To guarantee that the new value of the target variable after an assignment is not outside the range `T'Base'First .. T'Base'Last`, it is necessary in Ada 83 either to store the register into the variable's assigned storage location (if such a store operation could signal violation of the range check by generating an overflow condition) or to compare the value in the register to the values of `T'Base'First` and `T'Base'Last`; if the range check succeeds, the value in the register can be used subsequently.

Ada 95 does not perform a range check on the value assigned to a variable of an unconstrained numeric subtype; consequently, such a target variable can acquire a value outside its base range (`T'Base'First .. T'Base'Last`). This allows the value to be retained in the register in which it was generated and never stored (if all subsequent references to the target variable can be fulfilled from the register) nor checked by comparison for inclusion in the base range.

A consequence of this change, which generally allows more efficient object code to be generated, is that an Ada 83 program that raised `Constraint_Error` for a range violation on assignment to a variable of an unconstrained numeric subtype may raise the exception later (at a different place) or not at all. The first possibility arises because it may be necessary to store the extended register in the storage format for any of several reasons at a place far removed from the original assignment. The opportunity to raise the exception at such remote places is provided by a new rule that allows `Constraint_Error` to be raised at the place where a variable is referenced (fetched), if its value is outside its base range.

G.4 Accuracy and Other Performance Issues

The Numerics Annex specifies the accuracy to be delivered by the elementary functions, the complex arithmetic operations, and the complex elementary functions, and the performance to be expected from the random number generator. It also specifies the accuracy expected of the predefined floating point and fixed point arithmetic operators. If the Numerics Annex is not implemented, the predefined arithmetic operations and the various numerical packages do not have to yield any particular language-defined accuracy or performance, except where specified outside the Numerics Annex.

Even though the accuracy and performance requirements of the Numerics Annex are realistic, strict conformance may, in certain implementations, come only at a price. One or more native floating point instructions may produce slightly anomalous behavior that cannot conform to the model without some kind of sacrifice; for example, the purported maximum precision may have to be reduced, or some operations may have to be simulated in software. Similarly, achieving the specified accuracy in the elementary functions (say) may mean abandoning less accurate but much faster versions available on a hardware chip. Thus, to allow the user to trade accuracy for other considerations, the Numerics Annex specifies a pair of modes, strict mode and relaxed mode, that may be selected by the user. The accuracy and other performance requirements apply only when the strict mode is selected; the relaxed mode exempts implementations from these requirements, exactly as if the Numerics Annex had not been implemented.

A program that is intended for wide distribution, and whose numerical performance is to be guaranteed and portable across implementations, must be compiled and linked in the strict mode. Its portability will clearly extend, in that case, only to implementations that support the Numerics Annex.

The language does not specify how the modes should be implemented. It is clear, however, that the choice of mode can affect code generation, the values of the model-oriented attributes, and the version of the numerical libraries that is used. In implementations that meet the requirements without any undue sacrifices, or that have nothing substantial to gain from their relaxation, the two modes may, in fact, be identical.

G.4.1 Floating Point Arithmetic and Attributes

The strict-mode accuracy requirements for predefined floating point arithmetic operations are based on the same kind of model that was used in Ada 83, but with several changes. The Ada 83 model of floating point arithmetic was a two-level adaptation of the "Brown Model" [Brown 81] and defined both model numbers and safe numbers. The Ada 95 model is closer to a one-level, classical Brown Model that defines only model numbers, although it innovates slightly in the treatment of the overflow threshold.

The existence of both model numbers and safe numbers in Ada 83 caused confusion which hopefully will not apply to Ada 95. Note however, that the model numbers in Ada 95 are conceptually closer to the safe numbers rather than the model numbers of Ada 83 in terms of their role in the accuracy requirements. Other problems with the Ada 83 model centered around inherent compromises in the way the Brown Model was adapted to Ada 83. These compromises are eliminated in Ada 95, and other improvements are made, by

- freeing the model numbers to have a mantissa length that depends only on the implementation's satisfaction of the accuracy requirements, rather than a quantized mantissa length;
- defining the model numbers to have the hardware radix, rather than a fixed radix of two;
- defining the model numbers to form an infinite set, and basing overflow considerations on the concept of a type's "safe range" rather than on the largest model number;
- separating range and precision considerations, rather than tying them together intimately via the infamous "4*B* Rule" [RM83 3.5.7]; and
- freeing the minimum exponent of the model numbers from a connection to the overflow threshold, allowing it to reflect underflow considerations only.

We will now consider these in detail.

Mantissa Length

The Ada 83 safe numbers have mantissa lengths that are a function of the `Digits` attribute of the underlying predefined type, giving them a quantized length chosen from the list (5, 8, 11, 15, 18, 21, 25, ...). Thus, on binary hardware having `T'Machine_Mantissa` equal to 24, which is a common mantissa length for the single-precision floating point hardware type, the last three bits of the machine representation exceed the precision of the safe numbers. As a consequence, even when the machine arithmetic is fully accurate (at the machine-number level), one cannot deduce that Ada arithmetic operations deliver full machine-number accuracy. By freeing the mantissa length from quantization, tighter accuracy claims will be provable on many machines. As an additional consequence of this change, in Ada 95 the two types declared as follows

```
    type T1 is digits D;
    type T2 is digits T1'Digits range T1'First .. T1'Last;
```

can be represented identically. This matches one's intuition, since the declaration of `T2` requests neither more precision nor more range than that of `T1`. In Ada 83, the chosen representations almost always differ, with `T2'Base'Digits` being greater than `T1'Base'Digits`, for reasons having nothing to do with hardware considerations. (Note that this artificial example is not intended to illustrate how one should declare two different types with the same representation.)

Radix

Using the hardware radix rather than a binary radix has two effects:

- It permits practical implementations on decimal hardware (which, though not currently of commercial significance for mainstream computers, is permitted by the radix-independent IEEE floating point arithmetic standard [IEEE 87]; is appealing for embedded computers in consumer electronics; and is used in at least one such application, a Hewlett-Packard calculator);

- On hexadecimal hardware, it allows more machine numbers to be classed as model numbers (and therefore to be proven to possess special properties, such as being exactly representable, contributing no error in certain arithmetic operations, and so on).

As an example of the latter effect, note that `T'Last` will become a model number on most hexadecimal machines. Also, on hexadecimal hardware, a 64-bit double-precision type having 14 hexadecimal (or 56 binary) digits in the hardware mantissa, as on many IBM machines, has safe numbers with a mantissa length of 51 binary bits in Ada 83, and thus no machine number of this type with more than 51 bits of significance is a safe number. In Ada 95, such a type would have a mantissa length of 14 hexadecimal digits, with the consequence that every machine number with 53 bits of significance is now a model number, as are some with even more.

Note that the type under discussion does not have Ada 83 safe numbers with 55 bits in the mantissa, even though that is the next possible quantized length and one which is less than that of the machine mantissa. This is because some machine numbers with 54 or 55 bits of significance do not yield exact results when divided by two and cannot therefore be safe numbers. This is a consequence of their hexadecimal normalization, and it gives rise to the phenomenon known as "wobbling precision": the hexadecimal exponent remains unchanged by the division, while the mantissa is shifted right, losing a bit at the low-order end.

Safe Range

Extending the model numbers to an infinite set is intended to fill a gap in Ada 83 wherein the results of arithmetic operations are not formally defined when they are outside the range of the safe numbers but an exception is not raised. Some of the reasons why this can happen are as follows:

- the quantization of mantissa lengths may force the bounds of the range of safe numbers to lie inside the actual hardware overflow threshold;
- arithmetic anomalies of one operation may require the attributes of model and safe numbers to be conservative, with the result that other operations exceed the minimum guaranteed performance;
- the provision and use of extended registers in some machines moves the overflow threshold of the registers used to hold arithmetic results well away from that of the storage representation;
- the positive and negative actual overflow thresholds may be different, as on radix-complement machines.

The change means, of course, that one can no longer say that the model numbers of a type are a subset of the machine numbers of the type. As a consequence we have introduced the concept of the "safe range" of a type in Ada 95. This is the subrange of the type's base range that is guaranteed to be free of overflow; in Ada 83 terms, this subrange was the range of the type's safe numbers. Thus, in Ada 95 the the model numbers of a type within the type's safe range are indeed a subset of the machine numbers of the type.

By continuing the model numbers beyond the safe range of a type, we can say that an operation whose result interval (defined, as usual, in terms of model numbers) transcends the safe range of the result type either raises `Constraint_Error`, signalling overflow, or delivers a value from that result interval (provided the `Machine_Overflows` attribute is `True` for the type). In Ada 83, the result when an exception was not raised was completely implementation defined in this case. Of course, it continues to be implementation defined when the `Machine_Overflows` attribute is `False` for the type.

Incidentally, the safe range of a type has bounds characterized by two independent attributes, `Safe_First` and `Safe_Last`. In Ada 83, the range of safe numbers of a type was necessarily symmetric, with its upper bound being given by the value of the `Safe_Large` attribute. This was necessary, because `Safe_Large` was itself defined in terms of `Safe_Emax`, which gave the maximum exponent of the safe numbers. Because the model numbers no longer have a finite range, we no longer talk about the maximum exponent of the model numbers, and in fact there is no longer such an atttribute. Allowing the safe range to be asymmetric accommodates radix-complement machines better than Ada 83; in fact, it removes another impediment to the identical representation of the types `T1` and `T2` in the example given earlier.

The 4B Rule

Separating range and precision considerations is equivalent to dropping the "4*B* Rule" as it applies to the predefined types. There is however a "4*D* Rule" which affects the implementation's implicit selection of an underlying representation for a user-declared floating point type lacking a range specification, providing *in that case* a guaranteed range tied to the requested precision.

The change in the application of the 4*B* Rule allows all hardware representations to be accommodated as predefined types with attributes that accurately characterize their properties. Such types are available for implicit selection by the implementation when their properties are compatible with the precision and range requested by the user; but they remain *unavailable* for implicit selection, in the absence of an explicit range specification, exactly as in Ada 83.

The 4*D* Rule says that the representation chosen for a floating point type declared with a decimal precision of *d*, but lacking a range specification, must provide a safe range of at least -10.0^{4d} .. 10.0^{4d}. If the type declaration includes a range specification, the safe range need only cover the specified range.

The 4*B* Rule was introduced in Ada 83 in order to define the model numbers of a type entirely as a function of a single parameter (the requested decimal precision). By its nature, the rule potentially precludes the implementation of Ada in some (hypothetical) environments; in other (actual) environments, it artificially penalizes some hardware types so strongly that they have only marginal utility as predefined types available for implicit selection and may end up being ignored by the vendor. Such matters are best left to the judgment of the marketplace and not dictated by the language. The particular minimum range required in Ada 83 (as a function of precision) is furthermore about twice that deemed minimally necessary for numeric applications [Brown 81].

Among implementations of Ada 83, the only predefined types whose characteristics are affected by the relaxation of the 4*B* Rule are DEC VAX D-format and IBM Extended Precision, both of which have a narrow exponent range in relation to their precision.

In the case of VAX D-format, even though the hardware type provides the equivalent of 16 decimal digits of precision, its narrow exponent range requires that the `Digits` attribute for this type be severely penalized and reported as 9 in Ada 83; the `Mantissa` attribute is similarly penalized and reported as 31, and the other model attributes follow suit. In Ada 95, in contrast, the `Digits` attribute of this predefined type would have a truthful value of 16, the `Model_Mantissa` attribute (corresponding to Ada 83's `Mantissa` attribute, but interpreted relative to the hardware radix rather than a fixed radix of two) would have a value of 56, and the other model-oriented attributes would accurately reflect the type's actual properties. A user-declared floating point type requesting more than 9 digits of precision does not select D-format as the underlying representation in Ada 83, but instead selects H-format; in Ada 95, it still cannot select D-format *if it lacks a range specification* (because of the effect of the new 4*D* Rule), but it *can* select D-format if it includes an explicit range specification with sufficiently small bounds.

The IBM Extended Precision hardware type has an actual decimal precision of 32, but the 4*B* Rule requires the value of its `Digits` attribute to be severely penalized and reported as 18 in Ada 83, only three more than that of the double-precision type. Supporting this type allows an Ada 83 implementation to increase `System.Max_Digits` from 15 to 18, a marginal gain and perhaps the reason why it is rarely supported. In Ada 95, on the other hand, such an implementation can support Extended Precision with a `Digits` attribute having a truthful value of 32, though `System.Max_Digits` must still be 18. Although a floating point type declaration lacking a range specification cannot request more than 18 digits on this machine, those including an explicit range specification with sufficiently small bounds can do so and can thereby select Extended Precision.

Note that the named number `System.Max_Base_Digits` has been added to Ada 95; it gives the maximum decimal precision that can be requested in a type declaration that includes a range specification. In IBM systems having the Extended Precision type, the value of this named number can be 32.

Minimum Exponent

Freeing the minimum exponent of the model numbers to reflect only underflow considerations removes another compromise made necessary in Ada 83 by defining the model numbers of a type in terms of a single parameter. The minimum exponent of the model or safe numbers of a type in Ada 83 is required to be the negation of the maximum exponent (thereby tying it implicitly both to the overflow threshold and, through the 4*B* Rule, to the precision of the model numbers).

One consequence of this is that Ada 83's range of safe numbers may need to be reduced simply to avoid having the smallest positive safe number lie inside the implementation's actual underflow threshold. Such a reduction gives yet another way of obtaining values outside the range of safe numbers without raising an exception. Another consequnce is that the smallest positive safe number may, on the other hand, have a value unnecessarily greater than the actual underflow threshold.

This change therefore allows more of the machine numbers to be model numbers, allowing sharper accuracy claims to be proved.

Machine and Model Numbers

Consideration was given to eliminating the model numbers and retaining only the machine numbers. While this would further simplify the semantics of floating point arithmetic, it would not eliminate the interval orientation of the accuracy requirements if variations in rounding mode from one implementation to another and the use of extended registers are both to be tolerated. It would simply substitute the machine numbers and intervals for the model numbers and intervals in those requirements, but their qualitative form would remain the same. However, rephrasing the accuracy requirements in terms of machine numbers and intervals cannot be realistically considered, since many platforms on which Ada has been implemented and might be implemented in the future could not conform to such stringent requirements.

If an implementation has clean arithmetic, its model numbers in the safe range will in fact coincide with its machine numbers, and an analysis of a program's behavior in terms of the model numbers will not only have the same qualitative form as it would have if the accuracy requirements were expressed in terms of machine numbers, but it will have the same quantitative implications as well. On the other hand, if an implementation lacks guard digits or has genuine anomalies, its model numbers in the safe range will be a subset of its machine numbers having less precision, a narrower exponent range, or both, and accuracy requirements expressed in the same qualitative form, albeit in terms of the machine numbers, would be unsatisfiable.

The values of the model-oriented attributes of a subtype `S` of a floating point type `T` are defined in terms of the model numbers and safe range of the type `T`, when the Numerics Annex is implemented; this is true even in the relaxed mode. (Some of these attributes have partially implementation-defined values if the Numerics Annex is not implemented.)

Although these attributes generally have counterparts in Ada 83, their names are new in Ada 95. The reason is that their values may be different in Ada 95. Clearly, `S'Model_Mantissa` and `S'Model_Emin` will have very different values on a nonbinary machine, since they are interpreted relative to the hardware radix, rather than a radix of two. (On a hexadecimal machine, each will have roughly a quarter of the value of the corresponding attribute in Ada 83.) `S'Model_Small` and `S'Model_Epsilon` will only change slightly, if at all, because various effects will tend to cancel each other out. In any case, the new names convert what would be upward inconsistencies into upward incompatibilities. We have recommended that implementations continue to provide the old attributes, as implementation-defined attributes, during a transition period, with compiler warning messages when they are used. An Ada 83 program using the `Safe_Small` or `Base'Epsilon` attributes should be able to substitute the `Model_Small` and `Model_Epsilon` attributes for an equivalent (and logically consistent) effect, but substitutions for the other attributes may require more careful case by case analysis.

It is instructive to consider how the fundamental model-oriented attributes and the `Digits` attribute of a predefined floating point type `P` are determined in Ada 95, when the Numerics Annex is implemented. The algorithm is as follows.

- Initially set `P'Model_Emin` to `P'Machine_Emin` and `P'Model_Mantissa` to `P'Machine_Mantissa`. This tentatively defines an infinite set of model numbers.

- If the accuracy requirements, defined in terms of the model numbers, are satisfied by every predefined arithmetic operation that is required to satisfy them, *when overflow does not occur*, then these are the final values of those attributes. Otherwise, if the machine lacks guard digits or exhibits precision anomalies independent of the exponent, reduce

P'Model_Mantissa by one until the accuracy requirements are satisfied (when overflow does not occur); if underflow occurs prematurely, increase P'Model_Emin by one until the accuracy requirements are satisfied near the underflow threshold. The final set of model numbers has now been determined.

- Let P'Safe_First be the smallest model number that is greater than P'First and similarly let P'Safe_Last be the largest model number that is less than P'Last. These tentatively define the safe range.

- If overflow is avoided throughout the safe range by every predefined arithmetic operation, then this is the final safe range. Otherwise, i.e. if overflow occurs prematurely, increase P'Safe_First and/or decrease P'Safe_Last by one model number until overflow is correctly avoided in the resulting safe range. The final safe range has now been determined.

- Finally, let P'Digits be the maximum value of *d* for which

  ```
  Ød*log(10.0)/log(P'Machine_Radix) + 1ø £ P'Model_Mantissa.
  ```

  This is relevant in the context of the selection of a representation for a user-declared floating point type, which must provide at least as many decimal digits of precision as are requested. If this condition is satisfied, the type's arithmetic operations will satisfy the accuracy requirements.

P'Model_Epsilon and P'Model_Small are defined in terms of other attributes by familiar formulae. The algorithm for Ada 83, which is not given here, is much more complex and subtle, with more couplings among the attributes.

G.4.2 Fixed Point Arithmetic and Attributes

The revision of the model of fixed point arithmetic focuses on two of the problems concerning fixed point types that have been identified in Ada 83:

- The model used to define the accuracy requirements for operations of fixed point types is much more complicated than it needs to be, and many of its freedoms have never been exploited. The accuracy achieved by operations of fixed point types in a given implementation is ultimately determined, in Ada 83, by the safe numbers of the type, just as for floating point types, and indeed the safe numbers can, and in some implementations do, have more precision than the model numbers. However, the model in Ada 83 allows the values of a real type (either fixed or float) to have arbitrarily greater precision than the safe numbers, and so to lie between safe numbers on the real number axis. Implementations of fixed point typically do not exploit this freedom. Thus, the opportunity to perturb an operand value within its operand interval, although allowed, does not arise in the case of fixed point, since the operands are safe numbers to begin with. In a similar way, the opportunity to select any result within the result interval is not exploited by current implementations, which we believe always produce a safe number; furthermore, in many cases (for some operations) the result interval contains just a single safe number anyway, given that the operands are safe numbers, and it ought to be more readily apparent that the result is exact in these cases.

- Support for fixed point types is patchy, due to the difficulty of dealing accurately with multiplications and divisions having "incompatible *smalls*" as well as fixed point

> multiplications, divisions, and conversions yielding a result of an integer or floating point type. Algorithms have been published in [Hilfinger 90], but these are somewhat complicated and do not quite cover all cases, leading to implementations that do not support representation clauses for `Small` and that, therefore, only support binary *smalls*.

The solution adopted in Ada 95 is to remove some of the freedoms of the interval-based accuracy requirements that have never been exploited and to relax the accuracy requirements so as to encourage wider support for fixed point. Applications that use binary scaling and/or carefully matched ("compatible") scale factors in multiplications and divisions, which is typical of sensor-based and other embedded applications, will see no loss of accuracy or efficiency.

A host of specialized requirements for information systems applications is addressed by the division of fixed point types into ordinary and decimal fixed point types. The facilities for the latter are to be found in the Information Systems Annex, see Chapter F.

The default *small* in Ada 95 is an implementation-defined power of two less than or equal to the *delta*, whereas in Ada 83 it was defined to be the largest power of two less than or equal to the delta. The purpose of this change is merely to allow implementations that previously used extra bits in the representation of a fixed point type for increased precision rather than for increased range, giving the safe numbers more precision than the model numbers, to continue to do so. An implementation that does so must, however, accept the minor incompatibility represented by the fact that the type's default *small* will differ from its value in Ada 83. Implementations that used extra bits for extra range have no reason to change their default choice of *small*, even though Ada 95 allows them to do so.

Note that the simplification of the accuracy requirements that apply in the strict mode, by expressing them directly in terms of integer multiples of the result type's *small* rather than in terms of model or safe intervals, removes the need for many of the attributes of fixed point types. However, it is recommended that implementations continue to provide these attributes as implementation-defined attributes during a transition period, with their Ada 83 values, and that implementations produce warning messages upon detecting their use.

The accuracy requirements for the adding operators and comparisons now simply say that the result is exact. This was always the case in Ada 83, assuming operands were always safe numbers, and yet it was not clear from the model-interval form of the accuracy requirements that comparison of fixed point quantities was, in practice, deterministic.

Other accuracy requirements are now expressed in terms of small sets of allowable results, called "perfect result sets" or "close result sets" depending on the amount of accuracy that it is practical to require. These sets comprise consecutive integer multiples of the result type's *small* (or of a "virtual" *small* of 1.0 in the case of multiplication or division giving an integer result type). In some cases, the sets contain a single such multiple or a pair of consecutive multiples; this translates into a requirement that the result be exact, if possible, but never off by more than one rounding error or truncation error. This occurs with fixed point multiplications and divisions in which the operand and result smalls are "compatible" meaning that the product or quotient of the operand *smalls* (depending on whether the operation is a multiplication or a division) is either an integer multiple of the result *small*, or vice versa.

These compatible cases cover much of the careful matching of types typically exhibited by sensor-based and other embedded applications, which are intended to produce exact results for multiplications and at-most-one-rounding-error results for divisions, with no extra code for scaling; they can produce the same results in Ada 95, and with the same efficient implementation. Our definition of "compatible" is more general than required just to cover those cases of careful matching of operand and result types, permitting some multiplications that require scaling of the result by at worst a single integer division, with an error no worse than one rounding error.

In cases where the *smalls* are incompatible, the accuracy requirements are relaxed, in support of Requirement R2.2-A(1); in fact, they are left implementation defined. Implementations need not go so far as to use the Hilfinger algorithms [Hilfinger 90], though they may of course do so. An Ada 95 implementation could, for instance, perform all necessary scaling on the result of a multiplication or division by a single integer multiplication or division (or shifting). That is, the efficiency for the cases of incompatible *smalls* need not be less than that for the cases of compatible *smalls*. This relaxation of the requirements is intended to encourage support for a wider range of *smalls*. Indeed, we considered making support for all *smalls* mandatory in the strict mode on the grounds that the weaker requirements removed all barriers to practical support for arbitrary *smalls*, but we rejected it because it would make many existing implementations (which could in all other respects satisfy the requirements of strict mode) instantly nonconforming.

Ada 95 allows an operand of fixed point multiplication or division to be a real literal, named number, or attribute. Since the value *v* of that operand can always be factored as an integer multiple of a compatible *small*, the operation must be performed with no more than one rounding error and will cost no more than one integer multiplication or division for scaling. That *v* can always be factored in this way follows from the fact that it, and the *smalls* of the other operand and the result, are necessarily all rational quantities.

The accuracy requirements for fixed point multiplication, division, and conversion to a floating point target are left implementation defined (except when the operands' *smalls* are powers of the target's machine radix) because the implementation techniques described in [Hilfinger 90] rely on the availability of several extra bits in typical floating point representations beyond those belonging to the Ada 83 safe numbers; with the revision of the floating point model, in particular the elimination of the quantization of the mantissa lengths of model numbers, those bits are now likely gone. Except when the operands' *smalls* are powers of the target's machine radix, requiring model-number accuracy for these operations would demand implementation techniques that are more exacting, expensive, and complicated than those in [Hilfinger 90], or it would result in penalizing the mantissa length of the model numbers of a floating point type just to recover those bits for this one relatively unimportant operation. An implementation may use the techniques in [Hilfinger 90] for fixed point multiplication, division, and conversion to a floating point target; the accuracy achieved will be exactly as in Ada 83, but will simply not be categorizable as model-number accuracy, unless the operands' *smalls* are powers of the target's hardware radix. Furthermore, in the latter case, even simpler algorithms are available.

G.4.3 Accuracy of the Numerics Packages

The Numerics Annex specifies the accuracy or other performance requirements that the mandatory elementary function and random number packages must satisfy in the strict mode. These are discussed in A.3 with the packages themselves.

G.5 Requirements Summary

The facilities of the Numerics Annex and the floating point attributes of the Predefined Language Environment Annex relate to the requirements in 11.1 (Floating Point).

The requirement

R11.1-A(1) — Standard Mathematics Packages

is met in part by the Complex types and related packages of the Numerics Annex and in part by the elementary functions and random numbers packages of the Predefined Language Environment Annex.

The study topic

S11.1-B(1) — Floating Point Facilities

is met by the numeric model presented in the Numerics Annex and by the floating point attributes provided in the Predefined Language Environment Annex.

H Safety and Security

For critical software, the key issue is *assurance of the application*, that is, gaining sufficient confidence in the application in order to authorize its use. The Ada 95 language contributes to this process by providing a language definition which minimizes potential insecurities and which thus facilitates independent program validation and verification. However, the size and richness of the language also raise some issues that need to be addressed if it is to be fully exploited for safety and security applications.

- As a high-level language, Ada 95 tries to leave implementation-oriented matters unspecified, but validation and verification of a system requires knowledge of these details.
- Although software development takes place in Ada 95, validation logically needs to be performed at the object code level; understanding the correspondence between source and object is therefore essential for the most critical applications.
- If the expressive power of the full language is not needed, there must be some way to ensure that a tailored version of the run-time system is used, without support for the unwanted features, thereby simplifying the process of validation and verification.

The Safety and Security Annex is designed to address these concerns. It should be noted that the prospective users of this annex form a small, specialized community who historically have been served by special contracts between large user organizations and specific vendors. However, such an approach can only satisfy the requirements of enterprises with significant resources. Since the Annex is part of the Ada 95 standard, "off-the-shelf" products should be able to satisfy the same requirements at substantially reduced costs. This will allow Ada 95 to be used with assurance by different application areas, including those outside the military sector such as medical electronics or electronic funds transfer. Over the period that the Ada 95 standard can be expected to be operative, the number of applications in such areas could rise quite steeply.

Relationship to Current Approaches

The UK Ministry of Defence standard for the procurement of safety-critical software [MoD 91] is based upon the use of mathematical specifications and formal proofs or rigorous arguments for showing compliance and has been effectively used in some environments. However, the complete application of this standard is often not feasible and hence other methods must be used, even within the defense context. Whatever approach is taken, the final form of the program is vital, since it is an analysis of the program itself which must provide the basis of most of the assurance procedures. This implies that the programming language used for the application is likely to have a key role. The Annex aids but does not require the application of mathematical specification techniques.

A mature standard in the safety-critical area which takes a different view to formal methods is the international avionics standard [DO-178B]. Here the emphasis is on design, analysis and test. Although there is little stated about programming languages, it is clear that any analysis will either depend upon the programming language or upon a corresponding analysis of the object code generated by the compiler. Quite rightly, the standard requires isolation from the correctness of the compiler for the most critical software, implying that one must either reason from the object code or else show that the object and source code are equivalent. The Annex provides facilities to aid in the analysis of the object code, including access to data values during testing, in order to ease validation.

In the context of safety, the requirements surrounding the application of computers to nuclear shut-down systems has been well documented [Archinoff 90]. In the same application area, the need to show that compiler errors can be detected is given in [Pavey 93].

In the security area, the general requirements are well documented in [DoD 85, ITSEC 91]. Although the latter document does imply some requirements on the programming language in use, they are at a level that is not really relevant to this Annex.

Dealing with Language Insecurities

To reason about a program requires that the structures used within the program be well-defined and properly implemented. However, almost all existing programming languages standardized by ISO are not defined in a mathematically precise form. Hence substantial care must be taken to ensure that the features of the language used are well-defined and that the program accords with the intention of the programmer.

Since programmers are fallible, languages which require checks for some forms of error are an advantage, especially if the design can allow the software to request the system to return to a safe state. Ada 83 is the only widely used language suitable for critical applications which requires such checking (and Ada 95 of course continues in this tradition). Many critical applications do not exploit this checking but demonstrate (perhaps by mathematical proof) that the checks could not fail. Undertaking this form of static checking is very expensive in staff time, and hence is not practical for less critical applications. A brief summary of the securities and insecurities of standard languages is shown in Figure H-1.

Standard Language	*Security Features*	*Insecurities*
Ada 83	Runtime checks required Pointer initialization Type-secure across packages Can recover from check failures	Access to unset scalars
Modula-2 (not yet an ISO standard)	Type-secure across modules Limited recovery from failures	Unset pointers (& scalars)
Pascal	Strongly typed	Runtime checks optional Unset pointers (& scalars)
C	(Additional tools: make and lint)	150 undefined "features" Runtime checking often not done
Fortran 77	Type checking No pointers	Default declarations No checking across routines

Figure H-1: Securities and Insecurities in Standard Languages

A comparison of programming languages for safety and security applications showed that a subset of Ada 83 was a good choice [Cullyer 91]. The subsequent maturity of Ada compilers, the better understanding of the language, and the provision of Ada-specific tools, makes the language the first choice for many applications. In contrast, the C language is deprecated in the IEC draft standard on safety software [IEC/SC65A 91] since it is very difficult to demonstrate that C code is restricted to the subset which is well-defined. Ada 95 fills a number of the insecurities noted in Ada 83 [Wichmann 89], and hence it provides the smallest area of uncertainty to the developer.

The C++ language is not listed in Figure H-1, since it is not yet standardized by ISO. There are substantial problems in the validation of C++ code, some of which arise due to the lack of an agreed standard. Moreover, since almost all C programs can be compiled by a C++ compiler, the inherent problems of the validation of C code are transferred to C++. In addition, compiler vendors are at a disadvantage compared to Ada in not having an internationally agreed and comprehensive validation suite for the language.

The insecurities may be avoided by suitably chosen subsets of Ada, such as that provided by SPARK [Carré 88]. The same subset could be used for Ada 83 and Ada 95, and may be exploited by the compiler through the pragma `Restrictions` described below. Note that the use of such subsets is not in conflict with Ada's traditional "no subsets, no supersets" policy, since for compiler validation purposes there must be a complete implementation of the language, and not simply a subset. The point is that any particular program will not use the full language, and if there are a set of features whose exclusion can result in software for which validation and verification is facilitated, then the implementation can enforce such exclusions in one mode of operation and can link in reduced versions of the run-time system.

Until now, language standards have not addressed the problem of the validation of the object code generated by a compiler. This is troublesome unless information is provided linking the source code to the object code. Such checking will be required for many years to come while there is a possibility of errors being introduced by a compiler. The alternative of having "trusted compilers" does not yet seem viable for the major languages.

The user can indicate exclusion of particular features in a partition by means of pragma `Restrictions`. For instance, the user can indicate that tasking is not needed, thus allowing the run-time system to be very much smaller and simpler. Similarly, the user can ensure that other language features are avoided. Through this facility, the language used on a particular system can be reduced to a quite small subset of Ada 95. This can facilitate the analysis of the source code, since not only can language features be avoided for which verification and validation are impractical, but one can be assured that these restrictions are enforced by the compiler.

One might argue in favor of a smaller language than Ada 95 for safety-critical systems, rather than a mechanism for indicating features to be excluded. However, past experience has shown that there is no consensus on what should be in such a language; applications differ, and agreement on a suitable subset is difficult since specific applications do require most features of Ada for convenient and maintainable coding.

After agreeing to the Ada 95 standard, WG9 discussed the problems in the validation and verification of Ada programs. Such activities could be substantially cheaper if tools could effectively analyze Ada source code to verify safety or security requirements. In practice, this is likely to require that the code conforms to some conventions, suh as being within a subset. These issues are to be investigated by the WG9 Verification Rapporteur Group (VRG).

H.1 Understanding Program Execution

A key issue for the language in critical applications is that of *understandable execution*. Ada 95 addresses this issue in several ways:

- Eliminating certain cases of Ada 83 erroneous execution, and replacing them by bounded errors;

- Adding an attribute to check for scalar data validity;

- Adding a pragma to cause otherwise uninitialized scalars to be set to values outside their nominal subtypes;

- Requiring documentation of implementation decisions.

H.1.1 The Valid Attribute

Although this feature is in the core language [RM95 13.9.2], it is discussed here since it is relevant to the safety and security applications. The `Valid` attribute allows the user to check whether the bit-pattern for a scalar object is valid with respect to the object's nominal subtype. A reason for including such a facility is that the other language-defined operations that could conceivably undertake the same function might not have the intended effect. The following example illustrates the issue.

```
declare
   I : Integer range 1 .. 10;  -- Uninitialized
   A : array (1 .. 10) of Float;
begin
   ...
   A(I) := 1.0;
   ...
end;
```

In Ada 83, users are sometimes surprised to learn that a compiler is permitted to remove the index check for `A(I)`. The reason that such an optimization is allowed is that a reference to an uninitialized scalar object in Ada 83 yields an *erroneous execution*, and therefore unspecified effects. Thus the compiler may apply the following logic:

- If the value of `I` happens to be within `1 .. 10`, then the check is unnecessary.

- On the other hand, if the value is not within `1 .. 10`, then since execution is erroneous any effect is allowed including using the out of range value as an index.

Perhaps even more surprising, the programmer cannot count on the following style to ensure that a check is carried out:

```
declare
   I : Integer range 1 .. 10;  -- Uninitialized
   A : array (1 .. 10) of Float;
begin
   ...
   if I in 1 .. 10 then
      A(I) := 1.0;
   else
      raise Bad_Index;
   end if;
end;
```

In this example the compiler may optimize the test

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

to true, using the same logic that led to the suppression of the index check in `A(I)`, namely that if the program's execution is not erroneous then the value of `I` will be within its declared subtype and hence the membership test can be omitted, and if the program's execution is erroneous then the effect is unspecified and thus again the test can be omitted.

Ironically, if the programmer had declared the variable `I` without a range constraint, then it is likely that the check would be performed (unless data flow analysis can show otherwise). The fact that including a range constraint with a scalar declaration might reduce the security of the code (by failing to raise a run-time exception) is against the Ada design philosophy.

This problem is addressed in several ways in Ada 95. First, a reference to an uninitialized scalar variable is no longer erroneous but rather a *bounded error*: the permitted effects are to yield a valid value (i.e., one within the variable's nominal subtype), to yield an invalid value (one within the variable's type but outside its nominal subtype), or to raise the exception `Program_Error`. This rule prevents the compiler from "reasoning from erroneousness". In the first example, unless the compiler has by default set the initial value of `I` to be within the range `1 .. 10` (which is not recommended, since it would mask errors), it will need either to raise `Program_Error` (because of the reference to an uninitialized object) or to generate a range check.

The second example will not have the problem of the **`in`** membership test being optimized away, but there is still the possibility that the reference to the uninitialized value of `I` will raise `Program_Error`, which is presumably not what the programmer intended. Moreover, to allow membership tests to be implemented simply and efficiently, the membership test only performs a range check and thus might not work as desired for enumeration types with "holes" in the representation.

These remaining issues have motivated the introduction of a simple, intuitive facility for checking that the value of a scalar object is within the object's nominal subtype. That is the purpose of the `Valid` attribute, which applies to any scalar object. As illustration, an alternative version of the second example, using `Valid` rather than the membership test, is as follows

```
declare
   I : Integer range 1 .. 10;   -- Uninitialized
   A : array (1 .. 10) of Float;
begin
   ...
   if I'Valid then
      A(I) := 1.0;
   else
      raise Bad_Index;
   end if;
end;
```

The purpose of the `Valid` attribute is to check the contents of a scalar object without formally reading its value. Using this attribute on an uninitialized object is not an error of any sort, and is guaranteed to either return `True` or `False` (it will never raise `Program_Error`), based on the actual contents of the object, rather than on what the optimizer might have assumed about the contents of the object based on some declaration.

Although the use of the `Valid` attribute for checking the validity of uninitialized data is somewhat contrived, other examples are more realistic, such as checking data from:

- An unchecked conversion;
- Calling procedure `Read` from `Sequential_IO` or `Direct_IO`;
- An object for which pragma `Import` has been specified;
- An object that has been assigned a value where checks have been suppressed.

The `Valid` attribute could potentially be applied to a wider range of types than that of scalars. Unfortunately, this extension is not easy to define with the rigor that should be expected. For instance, what action should an implementation perform to attempt to determine if a bit-pattern of an access type is a valid value? If the attribute *did* have a larger scope but with unspecified semantics, then its use on critical systems would require an implementation-defined specification, checked by analysis of the object code produce by the compiler. This complexity did not seem justified.

If the attribute is to be used on a record read from a file that was written by a COBOL program, it is important that the Ada program can check that the value is meaningful rather than execute code based upon the premise that the value is legal. In the context of safety-critical applications, such alien data is likely to be provided by some external device. Such data could well be a composite value, in which case the attribute must be applied to the scalar components. Checking on non-scalar components or for potential gaps between components cannot be undertaken with `Valid`.

It is not necessarily logically sound to apply `Valid` to a composite type, due to the presence of components which are undefined, as in the following example

```
type Stack_Data is array (1 .. 100) of Character range 'A' .. 'Z';

type Stack is
   record
      Index : Integer range 0 .. 100;
      Data  : Stack_Data;
   end record;
```

Only those elements of the array up to the position of `Index` need to be checked for validity, but there is no way for a general definition to capture such semantics.

In formulating the design of the `Valid` attribute, we considered several alternatives. One was to have it as a scalar subtype attribute function, applied to the object of that subtype whose validity was to be checked; for example,

```
declare
   subtype S is Integer range 1 .. 10;
   I : S;
begin
   ...
   if S'Valid(I) then ... else ... end if;
   ...
end;
```

However, this would present semantic difficulties since calling a function causes evaluation of the actual parameters; the main idea behind the `Valid` attribute, however, is that it should not read the value of the object, since this might raise an exception.

Another approach we considered was to have the attribute as a function associated with a "target" scalar subtype (either as an attribute or through generic instantiation) applied to a value of any (other) "source" scalar subtype. The idea is to check a source bit pattern (say an `Integer` value) to see if it is a valid value for the target subtype. As an example, if `Enum` is an enumeration type with "holes" (that is, it has a representation clause with non-contiguous integer values), and `V` is an `Integer` value, then `Enum'Valid(V)` would return `True` if `V` has one of the identified values, and `False` otherwise. The idea is to check validity *before* the source bits are copied to the target object.

One problem with such an approach, however, is that it raises some semantic and implementation questions with respect to the expected sizes of both the source value and the target subtype; this issue does not arise with the notation `X'Valid` since the object `X` is not itself evaluated. Another problem is that it is rather unwieldy in the common case where validity needs to be checked for data that is being read from an external device into a record, and where the record fields have scalar subtypes. In such a case it is simplest and most efficient to read the data into the record first, and then to check validity. This works conveniently using the `Valid` attribute applied to a scalar object

```
declare
   type Rec is
      record
         A : Character range 'A' .. 'Z';
         B : Integer range 1 .. 10;
      end record;
   R : Rec;
begin
   Read(R);
   if not R.A'Valid then ... end if;
   if not R.B'Valid then ... end if;
   ...
end;
```

With the alternative style, it would be necessary to have a record type with fields corresponding to the full types (`Character` and `Integer`), to read the data into an object of this record type, check validity of the fields, and then copy it into `R` so that constraint checks can be enforced on subsequent assignments to these fields.

As a result of analysis of these design alternatives, we decided on the approach where validity is realized through an attribute that is applicable to a scalar object, rather than an attribute function associated with a target scalar subtype.

H.1.2 Abnormal Values

A value can be *abnormal* instead of having an invalid representation [RM95 13.9.1]. From the point of view of safety and security, such abnormal values are a potential disaster, since they can give rise to erroneous execution — the opposite of predictable execution which the Annex strives to provide.

In general, it is not possible for a scalar value to be abnormal, and in any case, the user can take precautions against scalar values with an invalid representation by suitable use of the `Valid` attribute. It might be possible for an implementation to obtain an abnormal floating point value if a signalling NaN was produced in which no trap-handler was provided, since access to the value would produce unpredictable results.

(A signalling NaN is a bit pattern used in place of a floating point number in systems that support IEC 559; see [IEC 89]. Access to such a bit pattern will cause an interrupt, if the processor state is set correctly. If the interrupt is not then handled, disaster could ensue. In fact, signalling NaNs could be used to detect unset floating point values with very little overhead on some machines, although existing Ada systems do not appear to do this.)

Abnormal values can arise from an abort statement interrupting an assignment operation, or interactions with the environment external to the Ada program. Typically, the data type would be composite with a complex internal structure which can be placed in a state that the Ada system cannot subsequently handle. Task objects and types with discriminants are potential candidates for types which can have abnormal values. Vendors providing support for this Annex should be able to indicate if and how such value can arise (unless they require the use of the `Restrictions` pragma to effectively exclude such values).

Abnormal values can also arise if a language defined check would fail but the check has been suppressed. Suppressing checks can obviously lead to problems, since any storage could be overwritten making it generally impossible for the Ada run-time system to retain control.

H.1.3 The Pragma Normalize_Scalars

The use of an undefined scalar is a very common programming error which must be detected in critical systems (see [Wichmann 92] for a discussion of some of the subtleties of this issue in connection with an earlier draft of the Ada 9X mapping). As observed above, the Ada 95 rule treating this as a bounded error rather than an erroneous execution will inhibit certain compiler optimizations that would make this kind of error difficult to detect. However, it does not prevent the compiler from giving an in-range "default" initial value to otherwise uninitialized scalars, which would also make it difficult to find errors.

In the light of these considerations, Ada 95 supplies the configuration pragma `Normalize_Scalars`, which serves to ensure that elaboration of the declaration of an otherwise uninitialized scalar, sets the object to an invalid value if such a value exists. If no such invalid value exists for a scalar object, then the implementation needs to identify this (for example on the program listing) so that the programmer is alerted to ensure that the object is assigned before it is referenced. In such cases, the program will have a predictable (but not necessarily portable) value and the implementation needs to document the in-range value taken.

The name `Normalize_Scalars` reflects the intent of the pragma. A "normal" value for a scalar object is either valid (if within the object's nominal subtype) or invalid (if outside). Since scalar initializations induced by the pragma might or might not be invalid, "normalize" is an appropriate description. In general, an invalid value will be outside the object's nominal subtype, but there are also cases where it is possible for the implementation to produce an invalid value even when the nominal subtype has the same range as the type. For example, suppose that an implementation of the type `Boolean` reserves 8 bits for objects that are not in packed arrays or in records with representation clauses, with `16#00#` corresponding to `False` and `16#01#` to `True`. In the presence of pragma `Normalize_Scalars`, an otherwise uninitialized `Boolean` variable will be set to an invalid value, which is neither `0` nor `1`.

Some interactions with other pragmas need to be understood by prospective users. First, if a scalar object is the argument to pragma `Import`, then its (probable lack of) initialization is not affected by pragma `Normalize_Scalars`. This is reasonable, since an imported data item is under the control of foreign language code that is not subject to Ada semantics. Note that if an Ada scalar variable corresponds to a memory-mapped I/O location, then any implicit initialization could have an unwanted effect. This can be avoided by importing the scalar variable.

Another interaction is with pragma `Restrictions`. If a system is being developed in which exception handling is absent, then the use of pragma `Normalize_Scalars` is inappropriate, and even dangerous. With the pragma `Restrictions(No_Exceptions)` in effect, there is no object code generated to perform constraint checks. Clearly, referencing an out-of-range scalar would then result in an unpredictable effect.

H.1.4 Documentation of Implementation Decisions

One aspect of predictability is to understand the behavior of a program in situations identified by the language rules as either bounded errors or unspecified effects. Thus the implementation needs to document these behaviors, either as part of a listing or tool-processable output, or (if a general rule) as independent documentation. Some specific requirements are now discussed.

Parameter Passing Mechanism

Some parameters can be passed by reference or by copy. A different mechanism could even be chosen for two calls of the same subprogram. Incorrect programs can be affected by the choice and hence safety and security applications need to check that either this error has not been made or that the effect will be acceptable. The simplest solution is for the compiler to indicate the choice made. If the choice is a simple static one (say, always by reference, except for entry parameters which are always by copy), then this could be stated once in the compiler documentation, otherwise the object code listing (or appropriate tool) should indicate the choice. In fact, some compilers have a complex algorithm which varies from call to call, especially for slices. The complexity of this algorithm is not relevant to the issue, merely that the choices made should be clear.

Storage Management

Many safety critical applications are in the form of an infinite loop. It is important that this loop should not permanently consume storage. Therefore it must be possible, by reviewing the object code, to ensure that this does not happen. In the security context, it is important that storage used to contain classified information does not leak via the storage allocation system. It is possible that this requirement can be met by proposals for a user-defined allocation and de-allocation of storage — in which case the run-time system may be less critical. There is a parameter to the `Restrictions` pragma to avoid storage leaks.

Of course, even if there is no net consumption of storage within a program, and if the storage is not allocated and de-allocated via a stack, it will be necessary to show that storage fragmentation does not undermine the system. For this reason, the algorithm used by the run-time system is important and must be fully documented.

For time-critical applications, additional constraints could be required on the run-time routines, such as having a tight upper bound in execution time. This requirement is not specified here, since it will be application specific and many systems avoid the use of the heap and hence are unlikely to have a problem.

Evaluation of Numeric Expressions

Problems can arise if the evaluation of numeric expressions involves extended range or extra precision. Counter-intuitive results can be produced with floating point expressions when combined with equality tests, which is quite common on IEEE systems. Hence the vendor should document the approach taken, independent of any annotation of the object code, so that any potential confusion can be avoided. This implies that for any specific expression, the range and precision with which it is computed should be clear from the documentation (see [RM95 H.2(2)]).

The evaluation of the exponentiate operator is not defined exactly in [RM95 4.5.6(11..12)]. Apart from the association of the multiplications (which only makes a marginal difference to the result), performing the division (for negative exponents) at the end or initially make a significant difference to the occurrence of overflow.

Adherence of an Ada implementation to the Language Independent Arithmetic standard [ISO 93] would be appropriate in the context of this annex, since that standard requires that the numeric operations used are precisely defined.

H.2 Reviewable Object Code

The relevant features supplied by the Annex are the pragmas `Reviewable` and `Inspection_Point`.

H.2.1 The Pragma Reviewable

Due to the well-known fact that all compilers have bugs, it is the conventional wisdom of the safety critical community to avoid assuming that the generated object code is automatically correct. For instance, the approach taken in the avionics standard DO-178B is one of Design, Review and Test [DO-178B]. As far as practical, the review and test activities are undertaken at the object code level. Indeed, if the reliability requirements of Class 1 flight critical software (the most critical) are to be attained, every attempt must be made to detect errors induced by a compiler. This is expensive, but unavoidable given current technology.

The pragma `Reviewable` applies to a partition so that the compiler can generate code to match special documentation thus permitting independent review of the object code. The following specific requirements apply.

Elaboration Order for Library Units

Since the elaboration order may have a visible effect, it is essential that the chosen ordering be indicated by the implementation in forms convenient both for human readers (such as in a program listing) and for processing by automated tools. An example of such a tool is a static analyzer that determines the absence of accesses to undefined variables.

Object Code

It cannot be assumed that the vendor will provide every tool needed for validation and verification. In any case, complete reliance upon such tools may not be acceptable. Hence it should be possible to extract the object code in a form that can be easily machine processed. An example of a tool is one which provides maximal times for instruction sequences so that time-critical software can be shown to meet deadlines.

The wording in the Annex on the requirement of producing output suitable for tools uses the word "should" rather than the conventional (and stronger) "shall". This is because it may not be possible to check compliance with the requirement objectively. Since there are no standard data interchange formats for such information, there is no means of giving a precise description of what is "suitable".

Object Lifetime Analysis

The implementation needs to allow the user to determine which objects are assigned to which registers, and the lifetimes of those assignments.

An important aspect of code generation is the assignment of registers. The most general register assignment algorithm is known to be NP complete and hence it is quite unreasonable for compiler documentation to detail such an algorithm (especially since it may be proprietary). However, the result of the algorithm for the specific safety/security application is to be provided. The area allocated to any object is to be specified so that an object can be accessed, either via the register or via main memory for the entire lifetime of the object. Compilers typically produce code to address internal information as well as the information directly related to objects declared by the program. This issue is not specified in the Annex, since it is unclear how it can be specified in a testable form.

Initialization Analysis

For each reference to a scalar object, the implementation needs to identify whether the object is either "known to be initialized" or "possibly uninitialized". Note that pragma `Normalize_Scalars` does not affect this analysis.

Since the access to unset scalars can lead to severe errors, and compilers already perform some of the analysis required, the purpose of this requirement is to provide the information to aid validation and verification. In the case of "possibly uninitialized", the information would depend upon the strength of the analysis performed by the compiler, and hence different compilers (or even the same compiler under different options) could not be expected to give identical information.

Machine Instructions Used

For the most critical applications, it is necessary to check that the machine instructions required by an application are correctly handled by the processor hardware. Microcode faults in processor chips are not uncommon and therefore such checking may be needed [Wichmann 93]. A list of the used instructions aids the checking since unused instructions need not be checked. It would be helpful to identify instructions only used in the run-time system, but this is not essential.

For checking timing constraints, a user needs to consider only the instructions listed.

Relationship between Source and Object Code

Code sequences derived entirely from one Ada statement (or declaration) must be indicated as such. In those cases in which a code sequence is derived from a single Ada statement, this statement should be identified. Due to some optimizations, it could be that this identification is difficult. In such cases, some optimizations could be disabled when the pragma `Reviewable` is in force, rather than enhancing the compiler to meet the requirements with full optimization. In this area, a tool could be much more useful for independent validation and verification rather than an annotated listing.

Some compilers provide information based upon line numbers rather than Ada statements. For the purposes of this annex, it can be assumed that there is only one statement per line. For a single statement, several code sequences could be involved, especially if instruction scheduling merges the code from more than one statement. Addressing instructions derived from more than one statement would not have to be identified as such.

If an Ada statement results in no object code, then a positive indication of removal is required, rather than a mere absence of object code from a statement.

The user may need to compute the storage requirements for a program so that the absence of the `Storage_Error` exception can be checked. For subprograms containing only statically sized objects, an implementation should indicate the size of the stack frame required.

Exception Analysis

The implementation must indicate where compiler-generated run-time checks occur in the object code, and must also provide a method of determining which exceptions can be raised by any statement.

The handling of the predefined exceptions is problematic. Exception sites need not be in the same compilation unit as the handlers that service them. Some method is needed to indicate, explicitly in the object code, the actual locations at which exceptions are raised and handled. Some mechanism should be available, either through source or object code analysis, that permits the analysis of the program to determine which handler is used for each site that can raise an exception and to identify sites for which no handler is supplied. It would probably be most useful if this was in the form of a tool, rather than tables which required detailed analysis of each case. An example of a tool to undertake exception analysis is given by [Schaefer 93].

Since exceptions raised by predefined operations are not explicitly indicated in the source, and since the implementation is allowed some freedom in choosing actual execution order, this facility is best supported at the object code level. Even if a vendor does not choose to perform such an analysis, the information necessary to perform it should be made available to the user. For a detailed analysis of the issues involved, see Chapter 2 of the report on Formal Studies of Ada 9X [DoD 92].

Analysis of Run-Time System Components

Clearly, the fact that a compiler generates a call to an out-of-line routine does not obviate the need for reviewing the object code of the called routine. Hence the same requirements for reviewing the object code must apply to the run-time system components.

H.2.2 The Pragma Inspection_Point

A point in the program text can be marked as an *inspection point* through a pragma of the same name, optionally identifying a set of objects whose values are to be available. At the corresponding point(s) in the object code, the vendor is required to provide a means of determining the values of the specified objects, or, if none was specified, then a means of determining the values of all live objects. This implies that the object code can be analyzed with special tools so that properties of the code and object values can be verified, independently of the source code. In theory, full mathematical verification could be undertaken, although this implies that the specification of the application is available in a suitable form.

This proposal arose out of the discussion from the special meeting held in April 1993 [Brosgol 93] attended by experts associated with producing safety critical systems. The idea is to break down a program into code sections separated by inspection points to facilitate validation of the code. This idea is new, although the concept of "break-points" in assembly language debugging is clearly similar.

Note that a single occurrence of pragma `Inspection_Point` in the source text may correspond to several inspection points in the object code; for example, if the pragma appears in an inlined subprogram, a generic unit, or a loop that has been "unrolled" by the optimizer.

There are, not surprisingly, some interactions between the pragma and optimizations. Since a user will in general examine the values of inspectable objects when execution is suspended at an inspection point, it is essential that compilers not perform "dead code elimination" on prior assignments to such objects. On the other hand, disabling all optimization is too extreme and in fact is unnecessary. Thus the compiler is allowed to store inspectable objects in registers; the implementation needs to provide sufficient information to make this mapping known to the user. The compiler is also allowed to move expressions (including those which could raise predefined exceptions) over inspection points.

The main design decision in connection with inspection points was whether to provide a single pragma, or to separate it into two: one that identifies the inspectable objects, and the other that identifies the points in program execution where currently live inspectable objects could be inspected. An advantage of the two-pragma approach is separation of concerns, and the ability to specify that, say, a variable declared in a package body is inspectable outside the package without needing to have all live objects inspectable. (Note that, with the single pragma approach the name of a variable declared in a package body is inaccessible outside the package and hence cannot be used as an argument to pragma `Inspection_Point`. Thus in the single-pragma approach, if the user needs to be able to inspect such a variable, pragma `Inspection_Point` with no arguments needs to be provided, which makes all objects inspectable and thus may inhibit some optimizations.)

In the end, the choice of a single-pragma approach was based on anticipated usage of the functionality provided, and in particular on the desire to avoid tedious source code changes and recompilations during software development. That is, the exact set of objects that need to be inspected might not be apparent at the outset. With the two-pragma approach, a decision to identify an additional variable will require a source code modification and may induce significant recompilation costs depending on where the variable is declared. With the single-pragma approach, the default is to have all variables inspectable and hence this is not a problem.

In some respects the issue is similar to the decision taken in Ada that when a package is *with*ed, by default all entities declared in the package's visible part are available in the *with*ing unit. An alternative approach would have the latter specify exactly those entities that are needed. Although seemingly producing a narrower and more specific interface, this would in practice yield long lists of needed entities that no one would read, and programmers would end up using a notation that made all visible entities available. In the case of inspection points, the anticipated typical usage is to have all objects inspectable; in those contexts where the programmer knows that only a limited set is of interest, a specific list can be provided as argument to pragma `Inspection_Point`.

Pragma `Inspection_Point` with a specific list of names provides some of the capabilities of an assertion facility without the need of an additional language feature. For example, if one writes

```
    pragma Inspection_Point(Alpha, Beta);
```

then, when the program is executed under the control of an appropriate monitor / debugger, it may be suspended at the corresponding point in the object code. Information is available at that point to examine `Alpha` and `Beta`. Therefore an assertion, say that `Alpha < Beta`, can be checked at that point. Note that no change to the generated code is required, which would be an issue if the assert capability were to be provided via an alternative mechanism. Variables such as `Alpha` and `Beta` must be in scope. Also, if such `Inspection_Point` pragmas are added at several points in a program, it may be possible to formally verify that the object code between the pragmas performs the operations in the source code.

H.3 Safety and Security Restrictions

A key technique that those developing critical software adopt is that of restricting the language constructs used. For instance, if tasking is not used, then the validation process is much simpler, since certain kinds of programming errors specific to tasking (such as deadlock, race conditions, and so on) cannot arise, and, moreover, the run-time system does not need to include any tasking support.

Although the term "subset" often has negative connotations, since in the past uncontrolled subsets for other languages have led to major portability problems, in the context of safety and security applications the use of subsets is essential. The issue is then how to satisfy these requirements without sacrificing the obvious significant advantages that Ada has enjoyed as a single-language standard. Interestingly, the current revision to the COBOL standard is going to a single-language model, in contrast to the language modules approach of the past, partly because of the success that Ada 83 has had with program portability.

The approach adopted is for the user to indicate excluded features as arguments to pragma `Restrictions`. The default behavior is for a compilation unit (or more generally a partition) to be rejected if it uses a feature identified in the pragma. Hence an Ada 95 compiler may enforce usage subsets in the manner required by [MoD 91], thus avoiding the potential risk of manual checking for adherence to restrictions. Moreover, an implementation is permitted to "bundle" restrictions, since otherwise compiler vendors would need to support 2^N versions of the run-time system, where N is the number of possible arguments to the pragma. Thus the pragma should not be seen as a way for precisely defining a subset, but as a framework which a vendor can exploit. As an example, consider the safety critical system produced by Alsys and described in [Brygier 93]. Here, an Ada 83 subset is defined which has a zero-byte run-time system, called CSMART. This subset is vendor-specific since it is oriented around the structure of the Alsys run-time system. An Ada 95 version of this system could be developed which is based on language-defined parameters to the `Restrictions` pragma.

The set of restrictions identified in the Safety and Security Annex is a representative set, but a compiler implementation may extend it. For example, in the SPARK system [Carré 88] the requirement is to use only those language features to which formal proof tools can be applied. Hence features such as generics are excluded, even though the implementation issues are straightforward.

Other analysis tools [Rex 88] impose similar restrictions on the source language. A vendor intending to support Ada 95 in those environments may thus add further restrictions to those defined in the Annex.

Although the default behavior in the presence of pragma `Restrictions` is to reject a compilation unit (or partition) if a restricted feature is used, the implementation may have a "full language" mode of operation where the use of a restricted feature elicits a warning message versus a fatal error. This could be useful in some environments, and it helps address the compiler validation concerns that might otherwise surround an implementation of a subset.

Possible scenarios showing uses of the `Restrictions` pragma are given in the following two examples.

Application to a Safety System

Vendor A produces a compiler and an implementation of the Safety and Security Annex, targeted to safety applications which use a standard similar to [DO-178B]. To simplify the production of the compiler and ensure the documentation aspects of reviewable object code are met, they *require* the use of the arguments `No_Protected_Types`, `No_Allocators`, `No_Dispatch`, `No_Delay`, `No_Exceptions` and `Max_Tasks = 0` when the pragma `Reviewable` is applied to a partition.

The user of this system has chosen this compiler because of the option above, knowing that the object code produced has a very simple structure, and that therefore the source code and object code are easily related. The user understands that since checking code does not appear in the object code, it is essential for this application to ensure that the code is indeed exception-free. To this end, a program analysis tool is being used. The pragma `Normalize_Scalars` is not used.

To ensure that some language features are not used which would cause problems for the program analysis tool, additional parameters are specified by the user to the `Restrictions` pragma as follows: `No_Floating_Point`, `No_Fixed_Point`, `No_Access_Subprograms`. In other words, the design requirements of not using these features are enforced by the compiler by means of the pragma.

In fact, the program analysis tool cannot handle `Unchecked_Conversion`. However, this restriction cannot be imposed by use of the `Restrictions` pragma since the application does require its use. In consequence, the use of `Unchecked_Conversion` is confined to one package which is not analyzed by the tool. This package uses the attribute `Valid` to ensure that the raw data will not subsequently cause an unbounded error.

Application to a Security System

Vendor B produces a compiler specifically for the security community who produce systems complying with the Orange Book and ITSEC [DoD 85, ITSEC 91]. Here, the full language is supported by the vendor when pragma `Reviewable` is enforced, since some applications require almost all of the language.

The user chooses this compiler since the Annex is supported with the full language. Tested packages which have formed part of other systems are being imported into their applications, and therefore the imposition of the restrictions is not usually feasible.

Again, the user has tools to analyze the Ada source code to validate both the security kernel and the main application code for adherence to the security policy. Since the kernel and the application code are all in one partition, it is only possible to use the pragma `Restrictions` for those features not used in any part of the system. For instance, `Unchecked_Conversion` is used in the kernel but should not appear in the application code, and hence this rule cannot be enforced by the `Restrictions` pragma. On the other hand, the declaration of access-to-subprogram types is not required at all, and hence this restriction is checked by the pragma.

Since the full language is supported by the vendor, the documentation provided to adhere to the pragma `Reviewable` is quite complex. To aid the review of the object code, the vendor provides a special tool, based upon the debugger, to ease the process of independent validation and verification. In particular, the tool can be used to locate the object code arising from a specific statement or declaration. This facility depends on the removal of optimizations that would be applied in the absence of pragma `Reviewable`. The user decides to employ the pragma `Normalize_Scalars` to reduce the risk of a covert channel and also to aid the detection of programming errors. (In principle, program components could communicate information via unset scalar values, thus establishing a secret or covert communication channel.)

Hence, in this example, the pragma `Restrictions` has a modest effect upon the compiler and its method of use.

Other Applications

The `Restrictions` pragma could clearly be used in other contexts apart from safety and security. For instance, in the context of teaching Ada, it might be convenient to ensure students are using just those features which would be allowed when developing high integrity software. Since the pragma is a configuration pragma, it should be simple to set up an Ada compilation system so that the student does not need to use pragma `Restrictions` explicitly.

Some high performance applications could also use the pragma to ensure an appropriately tailored run-time system is used.

H.4 Validation against the Annex

The majority of the Ada 95 standard consists of specific features for which the conventional validation process works well. Corresponding to each feature, a number of Ada test programs can be written for which the outcome can be stated and checked, usually within the test program itself. In contrast, the essence of the Safety and Security Annex is not language features but high assurance for the application program. Thus it is clear that validation of implementations to this Annex is different from both the other Annexes and also from Ada 83. The problem of devising objective tests needs to be considered carefully in the context of the high assurance required, not just adherence to the standard.

H.5 Issues outside the Scope of the Standard

The Safety and Security community have concerns which cannot be addressed by the Ada 95 standard itself. These issues are recorded here for completeness and to demonstrate that the standard was not the correct place to have them resolved.

The general requirement is for a "high assurance" compiler and run-time system. However, it is not possible to characterize this by means of requirements which are consistent with a language standard. The philosophy adopted by the Annex is to require information so that a user can judge whether a specific compiler meets the (subjective) requirements of an application.

Some of the issues which arise here are as follows:

- Version control for compilers and run-time;

Each version of the compiler and/or run-time system must be distinguished by a version number. It is convenient if it is easy to trace back from the binary to the corresponding source code.

- Validation of compiler in non-standard modes;

  The compiler may well be used in a mode of operation which does not support the full language. In such a case, documentation is required (specified in the Annex). However, it is often convenient to provide information on the processing of the validation suite in this mode (not required by the Annex). There is a significant change from Ada 83 here, since a mode of operation in which some language features are not supported can be provided by means of the `Restrictions` pragma.

- Security of compiler against Orange book or ITSEC;

  If a compiler is being used in a security context, it would be necessary to know that it conformed to the relevant requirements [DoD 85, ITSEC 91].

- Compiler support;

  Critical projects are likely to require support from the vendor over an extended period.

- Evaluation;

  The Ada Compiler Evaluation Service provides additional tests to measure aspects such as processing speed which cannot be specified in the standard. Special tools are also available to stress test compilers in order to give further confidence in their quality; see [Austin 91] and [Elsom 91].

- Certification of a specific system;

  It would be convenient if a specific Ada run-time system could be formally certified. This would give additional confidence in a system, and would allow a vendor to provide an identical platform for several market areas. However, it is unclear how such a service would operate.

In short, the requirements for critical systems go beyond those aspects which are covered by a conventional standard, and hence those which can be satisfied by the ordinary validation process.

H.6 Requirements Summary

The facilities of the Safety and Security Annex relate generally to the requirements in 9.1 (Predictability of Execution), 9.2 (Certifiability), and 9.3 (Enforcement of Safety-Critical Programming Practices).

More specifically, the study topic

S9.1-A(1) — Determining Implementation Choices

is met by the `Normalize_Scalars` pragma and by the requirements for documentation of implementation decisions.

The requirement

R9.1-A(2) — Ensuring Canonical Application of Operations is met by the pragma `Inspection_Point` and by the provision of `Off` as an argument to the pragma `Optimize`.

The requirement

R9.2-A(1) — Generating Easily Checked Code

is met by the `Reviewable` and `Inspection_Point` pragmas.

The requirement

R9.3-A(1) — Allow Additional Compile-Time Restrictions

is met by the pragma `Restrictions`.

Part Four

The Appendices

This fourth part comprises three appendices which summarize various aspects of the relationship between Ada 83 and Ada 95. Appendix X covers the incompatabilities of which there are few of real significance. Appendix Y gives the main changes betwen the Committee Draft, the Draft International Standard and the final International Standard; it shows that these changes are few and hence that the final language has had the benefit of considerable stability throughout the review periods. Appendix Z is a brief summary of the mapping between the original Requirements and the sections of this rationale where they are addressed; it concludes that Ada 95 meets the Requirements.

Appendix X Upward Compatibility

A major design goal of Ada 95 was to avoid or at least minimize the need for modifying the existing base of Ada 83 software to make it compatible with Ada 95. This involves not only pursuing upward compatibility but also preserving implementation-dependent behavior that can currently be relied upon. In common with the experience of revising other language standards, it is infeasible to guarantee 100% compatibility.

Other languages have been more or less successful in meeting this goal. For example, COBOL 83 has been very successful in achieving upward compatibility with COBOL 74. Nevertheless some minor incompatibilities were introduced which affect existing programs. For example, IS_ALPHABETIC, accepts upper and lower case in the new COBOL standard. The transition from C to ANSI C and from there to C++ has also caused incompatibilities, for example C++ requires all procedure definitions in old-style C to be modified.

In the design of Ada 95, a very conservative approach has been adopted. The few incompatibilities that exist can be dealt with in a simple mechanical way. For example, the introduction of a small number of reserved words requires their replacement in any program using them as identifiers. Extensive surveys of existing code show few programs to be affected. Most of the other incompatibilities involve obscure or pathological programming styles which are expected to appear very infrequently in existing code.

The great majority of programs will not be significantly affected by these changes — the most likely incompatibilities being automatically detected at compilation time. Moreover, tools are being developed to aid in the reliable detection of any problems and thereby smooth the process of transition.

Only three incompatibilities are considered likely to occur in normal programs. They are as follows:

New reserved words — In Ada 95, six new reserved words have been added to the language.

Type `Character` has 256 positions — In Ada 95, the type `Character` has 256 positions. In Ada 83, it had 128 positions.

Unconstrained generic parameters — In Ada 95, different syntax must be used in a generic formal parameter to allow unconstrained actual parameters.

The following further two incompatibilities might occur in normal programs but are less likely:

Library package bodies illegal if not required — In Ada 95, it is illegal to provide a body for a library package that does not require one.

`Numeric_Error` renames `Constraint_Error` — In Ada 95, the declaration for `Numeric_Error` has been changed to a renaming of `Constraint_Error`.

These incompatibilities usually cause a legal Ada 83 program to be an illegal Ada 95 program and hence are detected at compile time. They are described in more detail in the ensuing sections. In each case we give an example of the incompatibility, an indication of how it can be avoided in existing Ada 83 programs and the possibility of its automatic detection and removal.

The remaining incompatibilities which are considered very unlikely to occur in normal programs are briefly considered in X.6.

The reader should note that we say that an incompatibility is consistent if the worst that can happen is that a legal Ada 83 program becomes illegal in Ada 95 and thus fails to compile. An incompatibility is said to be inconsistent if the program can remain legal but have a different meaning.

X.1 Reserved Words

Six new reserved words are introduced in Ada 95: **`abstract`**, **`aliased`**, **`protected`**, **`requeue`**, **`tagged`**, and **`until`**.

Two alternatives to new reserved words were considered: a new concept of unreserved keywords or the use of combinations of existing reserved words. Neither of these options was considered preferable to the transitory inconvenience caused by the introduction of the new reserved words.

An Ada 83 program that uses any of these words as identifiers is an illegal Ada 95 program. For example, the following fragment of Ada 83 will fail to compile in Ada 95 because it uses two of the new reserved words

```
Protected: Boolean := False;

procedure Requeue(The_Activity: Activity; On_Queue: Queue);
```

Avoidance is clearly straightforward — avoid use of these six words as identifiers. Detection of the incompatibility is also straightforward. Automatic correction is problematic — to ensure that a name change is valid requires significant analysis especially if the identifier is the name of a library unit, or occurs in a package specification for which use clauses occur.

X.2 Type Character

In Ada 95, the type `Character` has 256 positions. In Ada 83, it had 128 positions.

Although suitable for English-speaking nations, a character type based on ASCII is inappropriate for most of Europe. ISO has defined a number of 256 character standards such as Latin-1 and Latin-2. This change to the language thus accommodates non-English speaking nations.

An Ada 83 program could be an illegal Ada 95 program if it has a case statement or an array indexed by `Character`, but it could be a legal Ada 95 program with different semantics if it relies on the position number or value of `Character'Last`. For example

```
type Char_Kind is (Numeric, Alphabetic, Other);

Kind_Array: array (Character) of Char_Kind :=          -- (1)
              ('0' .. '9' => Numeric,
               'A' .. 'Z' | 'a' .. 'z' => Alphabetic,
               others => Other);

case Char is                                            -- (2)
   when Character'Val(0) .. Character'Val(63) => ...
   when Character'Val(64) .. Character'Val(127) => ...
end case;

I: Integer := Character'Pos(Character'Last);           -- (3)
```

Declaration (1) is legal in Ada 95 but probably does not achieve the desired effect. Statement (2) is illegal in Ada 95 and will be detected at compilation. Statement (3) illustrates a situation where the program will still execute but have a different effect in Ada 95 (it is inconsistent).

As it is likely that allowing for 256 characters is outside the scope of the original requirement for the program concerned, avoidance is not really the issue — a review of the requirements is necessary.

The inconsistency illustrated by the third example can be avoided by not depending on the position or value of `Character'Last`. Avoiding the other incompatibilities avoids the real issue of how the extra 128 characters are to be handled. Unless uniform behavior is acceptable for these extra characters, use of an others choice, whilst ensuring a legal (but bad style) Ada 95 program might cause unacceptable behavior.

Detection of the consistent incompatibilities is straightforward; detection that an inconsistency may arise is possible. Manual correction is necessary to determine whether the required semantics of the program are those defined by Ada 95.

Finally, it should be noted that the ISO Working Group with responsibility for maintaining the Ada standard, has decreed that this change can be introduced into Ada 83 compilers, so this will increasingly become an Ada 83 portability issue as more implementations support 256 characters.

X.3 Library Package Bodies

In Ada 95, library unit packages are allowed to have a body only if required by language rules. This avoids a nasty and not so rare error.

In Ada 83, a body need only be provided for a package that really needed one, such as where the specification contains subprogram or task declarations. If a body was provided for a library package that did not need a body (for performing initializations for example), then if the package specification was subsequently changed, the body became obsolete. However, since it was optional, subsequent builds incorporating the package would not incorporate the body, unless it was manually recompiled. This obviously affects packages, for example, that only declare types, constants and/or exceptions, a very common occurrence. As a trivial example in Ada 83 consider

```
package Optional_Body is
   Global_Variable: Integer;
end Optional_Body;

-----------------------------------------

with Integer_Function;
package body Optional_Body is
begin
   Global_Variable := Integer_Function;
end Optional_Body;
```

The solution adopted in Ada 95 is to allow a body for a library unit package only when one is required by some language rule; the above example is therefore illegal in Ada 95. However, the pragma `Elaborate_Body` can be used to cause a body to be required.

Given the non-uniform functionality of program libraries and sublibraries, it is probably wise not to try to automatically detect, let alone correct, this incompatibility.

X.4 Indefinite Generic Parameters

Ada 95 provides new syntax for a generic formal private type to indicate that the actual subtype is allowed to be indefinite. The old syntax is retained, but the meaning is changed to require definite actual parameters.

In Ada 83, no indication was given in a generic formal type declaration as to whether the actual needed to be definite, for example because the body declared an uninitialized variable for the type. It was thus possible for a legal instantiation to become illegal if the body was changed.

An Ada 83 program, where an indefinite type is used as a generic actual parameter is an illegal Ada 95 program. For example the following legal Ada 83 program is illegal in Ada 95

```
generic
   type Element_Type is private;
package Stack is ...

------------------------------------------------------

with Stack;
package String_Stack is new Stack(Element_Type => String);
```

There is no way to avoid this incompatibility but an Ada 83 program can be annotated with an appropriate comment, thus

```
generic
   type Element_Type is private;  -- !! (<>) in Ada 95
package Stack ...
```

Detection of the incompatibility is straightforward. Manual correction is necessary to determine whether restricting the actual to being definite is acceptable.

It is interesting to note that some predefined library units in Ada 83 used this feature and so are changed. Examples are `Unchecked_Conversion` and `Unchecked_Deallocation` and also `Sequential_IO`.

Finally, it should be noted that the ISO Working Group has recommended that Ada 83 compilers be allowed to accept the new syntax in order to simplify transition.

X.5 Numeric Error

In Ada 95, the exception `Numeric_Error` is declared in the package `Standard` as a renaming of `Constraint_Error`.

The checks that could cause `Numeric_Error` to be raised in Ada 83 have all been reworded to cause `Constraint_Error` to be raised instead. Indeed, this change has been sanctioned by the Ada Rapporteur Group and encouraged in existing Ada 83 implementations.

However, the alternative of completely removing `Numeric_Error` was rejected because it would naturally have caused an incompatibility in programs using the construction

```
when Numeric_Error | Constraint_Error => Some_Action;
```

which is the currently recommended way of avoiding the confusion between `Numeric_Error` and `Constraint_Error` in Ada 83.

This construction is still allowed in Ada 95 because of an additional rule that permits an exception to be mentioned more than once in the same handler.

Programs which do have distinct separate handlers for `Numeric_Error` and `Constraint_Error` such as

```
exception
   when Constraint_Error => Action_1;
   when Numeric_Error => Action_2;
end;
```

are illegal in Ada 95. Moreover, an inconsistency will arise if a frame has an explicit handler for `Numeric_Error` or `Constraint_Error` but not both as in the following

```
exception
   when Constraint_Error => Action_1;
   when others => Action_2;
end;
```

since `Numeric_Error` will be caught by the first handler in Ada 95 but by the second in Ada 83.

Detection of the incompatibility is straightforward but manual correction will be necessary in cases where `Numeric_Error` is treated differently.

X.6 Other Incompatibilities

It is considered that other incompatibilities will be unlikely to occur in normal programs — the Ada 83 semantics being known only to the most erudite of Ada programmers — and so only a brief description seems appropriate in this document. In the following summary, they are grouped according to whether they result in a legal Ada 95 program but with different semantics; whether they would be detectable by an Ada 95 compiler and so on.

X.6.1 Unlikely Inconsistencies

These incompatibilities might cause a change in the runtime behavior, but they are not thought likely to occur in normal programs.

Derived type inherits all operations of parent — In Ada 95 a derived type inherits all its parent's primitive operations previously declared in the same declarative part. In Ada 83, it did not.

Floating point types may have less precision — the chosen representation for a floating point type may have less precision in Ada 95 for hardware with a non-binary radix.

Fixed point types may have less precision — the chosen representation for a fixed point type may have less precision in Ada 95. This is related to the next item.

Default *Small* for fixed point types — In Ada 83, the default value of *Small* was defined to be the largest power of two not exceeding `S'Delta`. In Ada 95, it is allowed to be a smaller power of two.

Rounding from real to integer is deterministic — Rounding is defined in Ada 95 as away from zero if the real number is midway between two integers.

Evaluation order of defaulted generic actual parameters — The order of evaluation of defaulted generic actuals is arbitrary in Ada 95.

Static expressions evaluated exactly — Static expressions are always evaluated exactly in Ada 95. In Ada 83 this was only required in a static context.

X.6.2 Unlikely Incompatibilities

These incompatibilities cause a legal Ada 83 program to be an illegal Ada 95 program and hence are detectable at compile time. They are considered to be unlikely in normal programs.

Bad pragmas illegal — In Ada 95, a pragma with an error in its arguments makes the compilation illegal. In Ada 83, it was ignored.

`S'Base` not defined for composite subtypes — In Ada 95, `S'Base` is not defined for a composite subtype `S`.

`Wide_Character` shares all character literals — As a result of adding types `Wide_Character` and `Wide_String` to package `Standard`, Ada 95 character literals are always overloaded and Ada 95 string literals are always overloaded.

Definition of freezing tightened — In Ada 95, range constraints on a type after its declaration and in occurrences in pragmas freeze the representation (are treated as forcing occurrences). In Ada 83 they were not treated as forcing occurrences.

Static matching of subtypes — In Ada 95, matching of subtypes is now performed statically instead of at runtime (as in Ada 83) in array conversions and generic instantiations.

Illegal to use value of deferred constant — In Ada 95 it is illegal to use the value of a deferred constant before it is set. In Ada 83 it was erroneous.

Explicit constraints in uninitialized allocators designating access types — in Ada 95 such constraints are illegal; in Ada 83 they were ignord.

Exceptions in static expressions illegal — in Ada 95, it is illegal to raise an exception in a static expression; in Ada 83 it made the expression non-static.

Preference for universal numeric operators — In Ada 95, the overload resolution rules have been changed to simplify them. As a consequence certain pathological Ada 83 programs become illegal.

Assume worst when checking generic bodies — Ada 83 generic contract model violations have been overcome in Ada 95 by assuming the worst case in a generic body.

New identifiers added to package `System` — New identifiers in package `System` may introduce illegalities into a unit having a use clause for `System`.

`Append_Mode` added to `File_Mode` enumeration — In Ada 95, subtype `File_Mode` in packages `Sequential_IO` and `Text_IO` has an extra literal, `Append_Mode`.

New identifiers added to package `Text_IO` — New identifiers in package `Ada.Text_IO` may introduce illegalities into a unit having a use clause for `Text_IO`.

New identifiers added to package `Standard` — New identifiers in package `Standard` may clash with existing use-visible identifiers.

Functions returning local variables containing tasks — In Ada 95 it is illegal or raises `Program_Error` if a function with a result type with a task subcomponent returns a local variable. In Ada 83 it was erroneous to return a variable containing a local task.

Illegal to change representation of types containing tasks — In Ada 95, it is illegal to give a representation item for a derived type containing a task.

Character literals always visible — In Ada 95, character literals are visible everywhere. In Ada 83 they followed the usual rules of visibility.

X.6.3 Implementation Dependent Incompatibilities

These incompatibilities only arise with some implementations. They occur either as a result of tightening up Ada semantics or where an Ada 83 implementation has used an identifier now predefined in Ada 95. In the latter case, an inconsistency could occur if the Ada 83 use of the identifier is compatible with the Ada 95 use, though this is unlikely.

Real attributes replaced — The Ada 83 attributes for a real subtype `S` (such as `S'Mantissa`) have been replaced by Ada 95 attributes defined in the Numerics Annex.

Certain pragmas removed — Some pragmas (including `Interface` and `Shared`) have been removed from the language and `Priority` has been moved to the Real-Time Systems annex.

New pragmas defined — The names of new pragmas may clash with implementation-defined pragmas.

New attributes defined — The names of new attributes may clash with implementation-defined attributes.

New library units defined — The names of new (language-defined) library units may clash with user-defined or implementation-defined library units.

X.6.4 Error Incompatibilities

These incompatibilities only occur in programs containing runtime errors, either detectable (an exception is raised) or undetectable (the execution is erroneous).

Exceeding `'First` or `'Last` of an unconstrained floating point type — In Ada 95, the `'First` and `'Last` of a floating point type declared without a range constraint are treated as minimum bounds and may be exceeded without causing `Constraint_Error`.

Dependent compatibility checks performed on object declaration — In Ada 95, dependent compatibility checks are performed on object declaration. In Ada 83, they were performed on subtype declaration.

Implicit array subtype conversion — Ada 95 allows sliding in more situations than did Ada 83, so `Constraint_Error` might not be raised as in Ada 83.

Lower bound of catenation changed for constrained array types — In Ada 95, the lower bound of the result of catenation for a constrained array type is defined to be `'First` of the index subtype. In Ada 83, the lower bound of the result was `'First` of the left operand.

Raising `Time_Error` deferred — In Ada 95, raising `Time_Error` can be deferred until `Split` or `Year` is called, or might not be raised at all. In Ada 83, it is raised on `"+"` or `"-"`.

Data format for `Get` — In Ada 95, `Get` for real types accepts a wider range of formats which would raise `Data_Error` in Ada 83. Leading and trailing zeros and the radix point are not required.

X.7 Conclusion

This appendix has outlined the incompatibilities between Ada 83 and Ada 95. As we have seen, the small number that are likely to occur in practice are easily overcome. The remainder are unlikely to be encountered in normal programs but have been mentioned for completeness. For further details the reader should consult the comprehensive discussion in [Taylor 95] upon which this discussion has been based.

In conclusion, it is clear that there are unlikely to be significant transition issues for the vast majority of Ada 83 programs. Ada 95 has been carefully designed to minimize incompatibilities while meeting the overall goals of the requirements.

Appendix Y Revisions To Drafts

The final International Standard for Ada 95 incorporates a number of changes to the Committee Draft [CD 93] of September 1993 (version 4.0 of RM9X) and the Draft International Standard [DIS 94] of June 1994 (version 5.0). These were made in response to formal comments made by the ISO members as part of the ballots on these drafts and to informal comments made by individual reviewers.

Although many of the changes are of an editorial nature several are of significance to the normal user. The more important changes are outlined in this appendix for the convenience of readers familiar with the drafts. Unless otherwise mentioned changes are with respect to the CD; if a change has occurred since the DIS then this is explicitly mentioned. A reference to that section of the rationale containing further discussion of the topic is given where appropriate.

The organization of the standard has been rearranged into a more logical order. The most significant to users familiar with Ada 83 is that chapter 14 on input-output has been moved into the annex on the predefined environment where it logically belongs. The annexes have themselves been reordered so that the mandatory annexes on the predefined environment and interfacing to other languages come first, followed by the six specialized needs annexes, the obsolescent features and then finally the non-normative annexes summarizing attributes and so on.

Y.1 Core Language

Trailing underlines

Trailing underlines are not allowed in identifiers whereas they were in the Committee Draft. Reversion to the Ada 83 rule was deemed appropriate because allowing just trailing underlines did not achieve the flexibility desired for wide compatibility with other languages such as C. Permitting leading underlines and multiple embedded underlines would have given greater compatibility but was considered unacceptable given the strength of concern for readability of program text. (2.1)

Modular types

Modular types are no longer described in terms of principal values and secondary values; they just have a value. A consequence is that conversion to and from integer types always preserves the numerical value or raises `Constraint_Error`. Wraparound on conversion no longer occurs. (3.3.2)

Extension aggregates

The ancestor part can now be a subtype name as an alternative to an expression. This enables an extension aggregate to be written even when the ancestor is abstract such as in the case of controlled types. (3.6.1, 7.4)

Controlled types

The package `Ada.Finalization` is restructured. The types `Controlled` and `Limited_Controlled` are no longer derived from types in the package `System.Implementation` (which no longer exists) but are simply abstract private types. The previous problem with writing aggregates for types derived from abstract types is now overcome by the new form of extension aggregate mentioned above.

The procedure `Split` is now called `Adjust`. The procedures `Adjust` and `Finalize` are no longer abstract but have null bodies like `Initialize`. (7.4)

Task storage size

The new pragma `Storage_Size` permits setting the storage size for individual tasks of a task type. This pragma is placed in the task specification and could thus depend on the value of a task discriminant. It replaces the use of an attribute definition clause for setting `Storage_Size` which gave the same attribute value to all tasks of the type. (9.6)

Children of generic units

It is no longer necessary for a child of a generic unit to be instantiated as a child of an instantiation of its parent. This requirement of the CD and DIS caused problems for many applications and a child can now be instantiated anywhere provided the generic child is visible. (10.1.3)

Exception occurrences

The package `Ada.Exceptions` is significantly restructured. The generic child `Ada.Exceptions.Messages` has been deleted. The ability to attach a user message to the raising of an exception can now be done more flexibly using the procedure `Raise_Occurrence` and the new attribute `Identity`. A major advantage is that exceptions so raised do not all have the same name `Exception_With_Message`.

The type `Exception_Occurrence` is now limited so that occurrences cannot be directly assigned. Exceptions can now be saved by a procedure and function `Save_Occurrence`. This approach overcomes implementation problems associated with the size of saved information. (11.2)

Access in generic bodies

The `Access` attribute can now be applied to objects in generic bodies when the access type is external. The associated accessibility check is dynamic and raises `Program_Error` if it fails. This gives greater flexibility in the use of generics. Note that the `Access` attribute still cannot be applied to subprograms in generic bodies when the access type is external. (12.3)

Alignment and Size attributes

The rules regarding these attributes have been somewhat changed. They can now only be applied to first subtypes (and objects) and not to all subtypes. Furthermore the `Address` must be a multiple of the `Alignment`. (13.1)

Y.2 Predefined Environment

Package Characters

This has been slightly restructured. The classification and conversion functions are now in a child package `Characters.Handling` and the package `Characters` is itself empty (other than the pragma `Pure`). The reason for this change is so that the child `Characters.Latin_1` can be used without introducing any unnecessary executable code from its parent. A related change is that the package `Standard.ASCII` is now obsolescent; programmers are expected to use `Characters.Latin_1` instead. (A.1)

Import and Export

The pragmas `Import` and `Export` now have a fourth parameter. The third parameter now gives the name of the entity in the other language and the fourth parameter gives the link name. (B.1)

Text_IO

A number of improvements have been made to `Text_IO`.

The concept of an error output stream has been added in line with facilities in many operating systems. Subprograms enable the error stream to be manipulated in a manner similar to the default output stream.

The functions `Current_Input`, `Current_Output` and `Current_Error` are overloaded with versions returning an access value. This enables the current stream to be preserved for later use in a more flexible manner.

The procedure `Get_Immediate` provides immediate non-buffered and non-blocking input; this is useful for interactive applications.

The procedure `Look_Ahead` returns the next character without removing it; this enables the user to write procedures which behave in a similar manner to the predefined procedures `Get` for integer and real types.

The procedure `Get` for real types will now accept a literal in more liberal formats; leading and trailing digits around the radix point are not required and indeed the point itself may be omitted. This enables data produced by programs written in languages such as Fortran to be processed directly.

The procedure `Flush` is added; this outputs the contents of the current buffer.

Nongeneric packages equivalent to instantiations of `Integer_IO` and `Float_IO` with the predefined types have been added since the DIS. These will be found of considerable benefit for teaching Ada since simple input-output of numbers can now be performed without the introduction of genericity. (A.4)

Command line

The package `Ada.Command_Line` enables a program to access the commands and parameters of the command line interpreter if any. It also enables a program to set its result status. (A.5)

Random number generator

The package `Ada.Numerics.Random_Numbers` has been considerably restructured and renamed as `Ada.Numerics.Float_Random`. The additional generic package `Ada.Numerics.-Discrete_Random` produces streams of random discrete values. (A.3)

Y.3 Specialized Needs Annexes

Edited output

The package `Ada.Text_IO.Pictures` is now called `Ada.Text_IO.Editing`. The description has been recast to avoid dependence on the COBOL standard. (F.2)

Appendix Z Requirements

This appendix lists the various requirements and study topics discussed in the Requirements document [DoD 90] and generally indicates how they have been met (by refererence to other parts of this rationale) or else notes why they proved to be inappropriate.

Z.1 Analysis

The requirements are listed here in exactly the order of the requirements document; against each requirement is a list of the one or more Requirement Summary sections of this rationale containing an indication of how the requirement has been met.

Note that a detailed analysis of how the requirements themselves relate back to the original Revision Requests is contained in the Requirements Rationale [DoD 91].

General

R2.1-A(1) — Incorporate Approved Commentaries	1.5
R2.1-A(2) — Review Other Presentation Suggestions	1.5
R2.1-B(1) — Maintain Format of Existing Standard	1.5
R2.1-C(1) — Machine-Readable Version of the Standard	1.5
R2.2-A(1) — Reduce Deterrents to Efficiency	3.11, 8.5, 9.8
R2.2-B(1) — Understandability	1.5, 3.11, 8.5
R2.2-C(1) — Minimize Special-Case Restrictions	6.4, 8.5, 11.5
S2.3-A(1) — Improve Early Detection of Errors	1.5, 3.11
R2.3-A(2) — Limit Consequences of Erroneous Executions	1.5
R2.4-A(1) — Minimize Implementation Dependencies	3.11

International Users

R3.1-A(1) — Base Character Set	3.11
R3.1-A(2) — Extended Graphic Literals	3.11
R3.1-A(3) — Extended Character Set Support	3.11
R3.1-A(4) — Extended Comment Syntax	2.4

S3.1-A(5) — Extended Identifier Syntax	2.4

Programming Paradigms

S4.1-A(1) — Subprograms as Objects	3.11, 4.8
R4.1-B(1) — Passing Subprograms as Parameters	3.11
R4.1-B(2) — Pragma INTERFACE	3.11, B.5
R4.2-A(1) — Allocation and Reclamation of Storage	13.7
S4.2-A(2) — Preservation of Abstraction	7.5
S4.3-A(1) — Reducing the Need for Recompilation	4.8, 10.6
S4.3-B(1) — Programming by Specialization/Extension	4.8, 10.6
S4.3-C(1) — Enhanced Library Support	10.6
S4.4-A(1) — Generic Formal Parameters	12.8
R4.4-B(1) — Dependence of Instantiations on Bodies	12.8
S4.4-B(2) — Tighten the "Contract Model"	12.8
R4.4-C(1) — Generic Code Sharing	12.8
R4.5-A(1) — Accessing an Exception Name	11.5
R4.6-A(1) — Interactive TEXT_IO	A.6
R4.6-B(1) — Additional Input/Output Functions	A.6

Real-Time

R5.1-A(1) — Elapsed Time Measurement	9.8, D.13
R5.1-B(1) — Precise Periodic Execution	9.8, D.13
R5.1-C(1) — Detection of Missed Deadlines	9.8
R5.2-A(1) — Alternative Scheduling Algorithms	9.8, D.13
R5.2-A(2) — Common Real-time Paradigms	9.8
R5.3-A(1) — Asynchronous Transfer of Control	9.8
R5.4-A(1) — Non-Blocking Communication	9.8
S5.4-B(1) — Asynchronous Multicast	9.8

System Programming

R6.1-A(1) — Unsigned Integer Operations	3.11
R6.2-A(1) — Data Interoperability	13.7
R6.3-A(1) — Interrupt Servicing	9.8, C.7
R6.3-A(2) — Interrupt Binding	9.8, C.7
R6.4-A(1) — Access Values Designating Global Objects	3.11
S6.4-B(1) — Low-Level Pointer Operations	3.11, 13.7

Parallel Processing

R7.1-A(1) — Control of Shared Memory	C.7
S7.2-A(1) — Managing Large Numbers of Tasks	9.8
S7.3-A(1) — Statement Level Parallelism	9.8
S7.4-A(1) — Configuration of Parallel Programs	9.8

No specific standard constructs for vector (SIMD) machines have been introduced; however the rules regarding exceptions have been changed so that vendors are able to provide optimizations through pragmas as discussed in 9.8.

Distributed Processing

R8.1-A(1) — Facilitating Software Distribution	10.6, E.8
R8.2-A(1) — Dynamic Reconfiguration	10.6, E.8

Safety-Critical and Trusted

S9.1-A(1) — Determining Implementation Choices	H.6
R9.1-A(2) — Ensuring Canonical Application of Operations	H.6
R9.2-A(1) — Generating Easily Checked Code	H.6
R9.3-A(1) — Allow Additional Compile-Time Restrictions	H.6

Information Systems

R10.1-A(1) — Decimal-Based Types	3.11, F.3

S10.1-A(2) — Specification of Decimal Representation	F.3
S10.2-A(1) — Alternate Character Set Support	F.3
S10.3-A(1) — Interfacing with Data Base Systems	B.5, F.3
S10.4-A(1) — Varying-Length String Package	A.6
S10.4-A(2) — String Manipulation Functions	A.6, F.3

Scientific and Mathematical

R11.1-A(1) — Standard Mathematics Packages	A.6, G.5
S11.1-B(1) — Floating Point Facilities	G.5
S11.2-A(1) — Array Representation	B.5

Z.2 Conclusion

The above analysis shows that all the formal Requirements have been thoroughly met and it is only the Study Topics for parallel processing where compromises have been made.

We can therefore conclude that Ada 95 clearly meets the spirit of the Requirements as expressed in [DoD 90].

References

[1003.1 90] The Institute of Electrical and Electronics Engineers. *Portable Operating System Interface (POSIX) — Part 1: System Application Program Interface (API) [C Language].* The Institute of Electrical and Electronics Engineers, 1990.

[1003.4 93] The Institute of Electrical and Electronics Engineers. *Portable Operating System Interface (POSIX) — Part 1: Amendment 1: Realtime Extensions [C Language].* POSIX P1003.4/D14. March, 1993.

[1003.4a 93] The Institute of Electrical and Electronics Engineers. *Portable Operating System Interface (POSIX) — Part 1: Amendment 2: Threads Extensions [C Language].* POSIX P1003.4a/D7. April, 1993.

[1003.5 92] The Institute of Electrical and Electronics Engineers. *POSIX Ada Language Interfaces, Part 1 Binding for System Application Program Interface (API).* The Institute of Electrical and Electronics Engineers, 1992.

[AARM] *Annotated Ada 95 Reference Manual, Version 6.0.* Intermetrics Inc, 1995.

[Abadi 91] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. "Dynamic Typing in a Statically Typed Language". *Transactions on Programming Languages and Systems* 13(2), April 1991.

[ANSI 83] *Reference Manual for the Ada Programming Language*. ANSI/MIL-Std-1815a edition, 1983.

[Archinoff 90] G. H. Archinoff, R. J. Hohendorf, A. Wassyng, B. Quigley and M. R. Borsch. "Verification of the Shutdown System Software at the Darlington Nuclear Generating Station". In *International Conference on Control & Instrumentation in Nuclear Installations.* May, 1990. Glasgow.

[Atkinson 88] C. Atkinson, T. Moreton, and A. Natali (editors). *Ada for Distributed Systems.* The Ada Companion Series, Cambridge University Press, 1988.

[Austin 91] S. M. Austin, D. R. Wilkins, and B. A. Wichmann. "An Ada Test Generator". In *Tri-Ada 1991 Proceedings*, ACM SIGAda, 1991.

[Baker 89] T. Baker. "Time Issues Working Group". In *Proceedings of the First International Workshop on Real-Time Ada Issues,* pages 119-135. Software Engineering Institute and United States Office of Naval Research, Association for Computing Machinery, New York, NY, June, 1989. Appeared in ACM SIGAda *Ada Letters*, Vol. X, No..4.

[Baker 91] T. Baker. "Stack-Based Scheduling of Realtime Processes". *The Real-Time Systems Journal* 3(1): 67-100, March, 1991.

[Bal 92] H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum. "Orca: A Language for Parallel Programming of Distributed Systems". *IEEE Transactions on Software Engineering*, 18(3): 190-205, March 1992.

[Bardin 89] B. M. Bardin and C. J. Thompson. "Composable Ada Software Components and the Re-Export Paradigm". ACM SIGAda *Ada Letters* VIII(1), 1989.

[Barnes 76] J. G. P. Barnes. *RTL/2 Design and Philosophy.* Heyden, 1976.

[Barnes 82] J. G. P. Barnes. *Programming in Ada.* Addison-Wesley, London, 1982.

[Barnes 95] J. G. P. Barnes. *Accessibility Rules OK!* Ada Letters XV(1), 1995.

[Birtwistle 73] G. M. Birtwistle, O-J. Dahl, B. Myhrhaug, and K. Nygaard. *SIMULA Begin.* Auerbach, Philadelphia, PA, 1973.

[Blum 86] L. Blum, M. Blum and M. Shub. "A Simple Unpredictable Pseudo-Random Number Generator". *SIAM Journal of Computing* 15(2):364-383, 1986.

[Booch 86] G. E. Booch. "Object-Oriented Development". *IEEE Transactions on Software Engineering* SE-12(2), February 1986.

[Booch 87] G. E. Booch. *Software Components with Ada.* Benjamin/Cummings, Menlo Park, CA, 1987.

[Brinch-Hansen 73] P. Brinch-Hansen. *Concurrent Programming Concepts.* Computing Surveys 5(4): 224-245, 1973.

[Brosgol 89] B. Brosgol. "Is Ada Object-Oriented?" *ALSYS News*. Fall, 1989.

[Brosgol 92] B. M. Brosgol, D. E. Emery and R. I. Eachus. "Decimal Arithmetic in Ada". In *Proceedings of the Ada-Europe International Conference*, Amsterdam, June 1-5, 1992.

[Brosgol 93] B. M. Brosgol. *Minutes of Special Meeting — Security & Safety Annex.* Unnumbered Ada 9X Project Document, 1993.

[Brown 81] W. S. Brown. "A simple but realistic model of floating-point computation". *Transactions on Mathematical Software* 7(4): 445-480, December, 1981.

[Brygier 93] J. Brygier and M. Richard-Foy. "Ada Run Time System Certification for Avionics Applications". In *Proceedings of the Ada-Europe International Conference.* June, 1993. Paris, France.

[Budd 91] T. Budd. *An Introduction to Object-Oriented Programming*. Addison-Wesley, 1991.

[Burger 87] T. M. Burger and K. W. Nielsen. "An Assessment of the Overhead Associated With Tasking Facilities and Tasking Paradigms in Ada". ACM SIGAda *Ada Letters* VII(1): 49-58, 1987.

[Burns 87] A. Burns, A. M. Lister, and A. Wellings. *A Review of Ada Tasking.* Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1987.

[Burns 89]	A. Burns and A. Wellings. *Real-Time Systems and their Programming Languages.* International Computer Science Series, Addison-Wesley, Reading, MA, 1989.
[Cardelli 85]	L. Cardelli and P. Wegner. "On Understanding Types, Data Abstraction, and Polymorphism". *ACM Computing Surveys* 17(4): 471-522, December 1985.
[Carré 88]	B. A. Carré and T. J. Jennings. *SPARK — The SPADE Ada Kernel.* Technical Report, University of Southampton, March, 1988.
[CD 93]	*Ada 9X Reference Manual, Version 4.0 (Committee Draft).* Intermetrics Inc, 1993.
[Coffman 73]	Coffman and Denning. *Operating Systems Theory.* Prentice-Hall, Englewood Cliffs, New Jersey, 1973.
[Cohen 90]	S. Cohen. *Ada Support for Software Reuse.* Technical Report SEI-90-SR-16, Software Engineering Institute, October 1990.
[Cullyer 91]	W. J. Cullyer, S. J. Goodenough and B. A. Wichmann. "The Choice of Computer Languages in Safety-Critical Systems". *Software Engineering Journal* 6(2): 51-58, March, 1991.
[Dahl 72]	O-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare. *Structured Programming.* Academic Press, London, 1972, p111.
[Dewar 90a]	R. B. K. Dewar. *Shared Variables and Ada 9X Issues.* Special Report SEI-90-SR-1, Software Engineering Institute, January 1990.
[Dewar 90b]	R. B. K. Dewar. *The Fixed-Point Facility in Ada.* Technical Report Special Report SEI-90-SR-2, Software Engineering Institute, Pittsburgh, PA, February, 1990.
[Dijkstra 72]	E. W. Dijkstra. "Notes on Structured Programming". In O-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare. *Structured Programming.* Academic Press, London, 1972.
[DIS 94]	*Ada 9X Reference Manual, Version 5.0 (Draft International Standard).* Intermetrics Inc, 1994.
[DO-178B]	Issued in the USA by the Requirements and Technical Concepts for Aviation (document RTCA SC167/D0-178B) and in Europe by the European Organization for Civil Aviation Electronics (EUROCAE document ED-12B).
[DoD 78]	Defense Advanced Research Projects Agency. *Department of Defense Requirements for High Order Computer Programming Languages — STEELMAN.* USDoD, Arlington, Virginia, 1978.
[DoD 85]	*Trusted Computer Systems Evaluation Criteria.* DOD 5200.28-STD edition, Department of Defense, 1985. [DoD 92] Formal Studies of Ada 9X, Formal Definition Report Ada 9X Project Report edition, Department of Defense, 1992.

[DoD 88] *Ada Board's Recommended Ada 9X Strategy*. Office of the Under Secretary of Defense for Acquisition, Washington, D.C., 1988.

[DoD 89a] *Ada 9X Project Plan*. Office of the Under Secretary of Defense for Acquisition, Washington, D.C., 1989.

[DoD 89b] *Common Ada Programming Support Environment (APSE) Interface Set (CAIS) (Revision A) edition*. MIL-STD-1838A. United States Department of Defense, 1989.

[DoD 90] *Ada 9X Requirements*. Office of the Under Secretary of Defense for Acquisition, Washington, D.C., 1990.

[DoD 91] *Ada 9X Requirements Rationale*. Office of the Under Secretary of Defense for Acquisition, Washington, D.C., 1991.

[DoD 92] *Formal Studies of Ada 9X*, Formal Definition Report. Ada 9X Project Report edition, Department of Defense, 1992.

[Dritz 91a] K. W. Dritz. "Rationale for the Proposed Standard for a Generic Package of Elementary Functions for Ada". ACM SIGAda *Ada Letters* XI(7): 47-65, Fall, 1991.

[Dritz 91b] K. W. Dritz. "Rationale for the Proposed Standard for a Generic Package of Primitive Functions for Ada". ACM SIGAda *Ada Letters* XI(7): 82-90, Fall, 1991.

[Eachus 92] Personal communication from Robert Eachus, 1992.

[Ellis 90] M. A. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, Reading, MA, 1990.

[Elrad 88] T. Elrad. "Comprehensive Scheduling Controls for Ada Tasking". In *Proceedings of the Second International Workshop on Real-Time Ada Issues*, pages 12-19. Ada UK and United States Air Force Office of Scientific Research, Association for Computing Machinery, New York, NY, June 1988. Appeared in ACM SIGAda *Ada Letters*, VIII(7).

[Elsom 91] K. C. Elsom. *Grow: an APSE Stress Tensor.* DRA Maritime Division, Portsmouth, 1991.

[Emery 91] Personal communication from David Emery, 1991.

[Fowler 89] K. J. Fowler. *A Study of Implementation-Dependent Pragmas and Attributes in Ada.* Technical Report Ada 9X Project Report, Software Engineering Institute, November 1989.

[Goldberg 83] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, Reading, MA, 1983.

[Guimaraes 91] N. Guimaraes. "Building Generic User Interface Tools: An Experience with Multiple Inheritance". In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, 1991.

[Guttag 77] J. Guttag. "Abstract Data Types and the Development of Data Structures". *Communications of the ACM* 20(6): 396-404, June 1977.

[Hilfinger 90] P. N. Hilfinger. *Implementing Ada Fixed-point Types Having Arbitrary Scales.* Technical Report Report No. UCB/CSD 90/#582, University of California, Berkeley, CA, June, 1990.

[Hilzer 92] R. C. Hilzer. "Synchronization of the Producer/Consumer Problem using Semaphores, Monitors, and the Ada Rendezvous". *ACM Operating Systems Review*, 26(3), July 1992.

[Hoare 73] C. A. R. Hoare. "Towards a Theory of Parallel Programming". In C.A.R. Hoare and R.H. Perrott (editor), *Operating Systems Techniques.* Academic Press, New York, 1973.

[Hoare 74] C. A. R. Hoare. "Monitors — An Operating Systems Structuring Concept". *Communications of the ACM* 17(10), pp 549-557, October 1974.

[Hoare 78] C. A. R. Hoare. "Communicating Sequential Processes". *Communications of the ACM* 21(8): 666-677, August 1978.

[IBFW 86] J. D. Ichbiah, J. G. P. Barnes, R. J. Firth, and M. Woodger. *Rationale for the Design of the Ada Programming Language.* Reprinted by Cambridge University Press, 1991.

[IEC 89] *Binary Floating-Point Arithmetic for Microprocessor Systems.* IEC 559:1989 edition, IEC, 1989.

[IEC/SC65A 91] *Software for Computers in the Application of Industrial Safety-Related Systems (Draft).* IEC/SC65A/(Secretariat 122), 1991.

[IEEE 87] *Standard for Radix-Independent Floating-Point Arithmetic.* ANSI/IEEE Std. 854-1987 edition, 1987.

[IEEE 92] *POSIX Ada Language Interfaces — Part 1: Binding for System Application Program Interface.* Std 1003.5, IEEE, New York, 1992.

[Ingalls 86] D. H. H. Ingalls. "A Simple Technique for Handling Multiple Polymorphism". In *ACM OOPSLA'86 Conference Proceedings*, 1986.

[ISO 87] International Standards Organization. *Reference Manual for the Ada Programming Language.* ISO/8652-1987, 1987.

[ISO 90] International Standards Organization. *Memorandum of Understanding between the Ada 9X Project Office and ISO-IEC/JTC 1/SC 22/WG 9 Ada.* ISO-IEC/JTC 1/SC 22 N844, 1990.

[ISO 91] International Standards Organization. *Generic Package of Elementary Functions for Ada.* ISO/JTC 1 DIS 11430.

[ISO 92] *Database Language SQL.* Document ISO/IEC 9075:1992 edition, International Organization for Standardization (ISO), 1992.

[ISO 93] *Information Technology—Language Independent Arithmetic— Part 1: Integer and Floating Point Arithmetic.* DIS 10967-1:1993 edition, ISO/IEC, 1993.

[ISO 94a] International Standards Organization. *Generic Package of Elementary Functions for Ada*. ISO-IEC/JTC 1 11430:1994.

[ISO 94b] International Standards Organization. *Generic Package of Primitive Functions for Ada*. ISO-IEC/JTC 1 11729:1994.

[ISO 95] International Standards Organization. *Reference Manual for the Ada Programming Language.* ISO/8652-1995, 1995.

[ISO WG9 93] AI 00866/03, *The Latin-1 character set is used in source code and literals.* ISO/IEC JTC1/SC22 WG9 Ada, June, 1993.

[ITSEC 91] *Information Technology Security Evaluation Criteria.* Version 1.2 edition, Provisional Harmonised Criteria, 1991. UK contact point: CESG Room 2/0805, Fiddlers Green Lane, Cheltenham, Glos, GL52 5AJ.

[Kahan 87] W. Kahan. "Branch Cuts for Complex Elementary Functions, or Much Ado About Nothing's Sign Bit". In *The State of the Art in Numerical Analysis*. Clarendon Press, 1987, Chapter 7.

[Kahan 91] W. Kahan and J. W. Thomas. *Augmenting a Programming Language with Complex Arithmetic.* Technical Report UCB/CSD 91/667, Univ. of Calif. at Berkeley, December, 1991.

[Klein 93] M. H. Klein, T. Ralya, B. Pollak, R. Obenza, and M. G. Harbour. *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems.* Klewer Academic Publishers, 1993.

[Knuth 81] D. E. Knuth. *The Art of Computer Programming. Volume 2: Semi-numerical Algorithms*. Addison-Wesley, 1981.

[LaLonde 89] W. R. LaLonde. "Designing Families of Data Types using Exemplars". *ACM Transactions on Programming Languages and Systems* 11(2), April 1989.

[L'Ecuyer 88] P. L'Ecuyer. "Efficient and Portable Combined Random Number Generators". *Communications of the ACM* 31(6):742-749, 774; 1988.

[Lehoczky 86] J. P. Lehoczky and L. Sha. "Performance of Real-Time Bus Scheduling Algorithms". *ACM Performance Evaluation Review*, Special Issue 14(1): 44-53, May, 1986.

[Lewis 69] P. A. Lewis, A. S. Goodman, and J. M. Miller. " Pseudo-Random Number Generator for the System/360" *IBM System Journal* 8(2):136-146, 1969.

[Liskov 77] B. Liskov, A. Snyder, R. Atkinson, and C. Schaffert. "Abstraction Mechanisms in CLU". *Communications of the ACM* 20(8): 564-576, August 1977.

[Marsaglia 91] G. Marsaglia and A. Zaman. "A New Class of Random Number Generators". *Annals of Applied Probability* 1(3):462-480, 1991.

[Meyer 88] B. Meyer. *Object-Oriented Software Construction.* Prentice Hall, Englewood Cliffs, NJ, 1988.

[MoD 91] *The Procurement of Safety Critical Software in Defence Equipment (Part 1: Requirements; Part 2: Guidance).* Interim Defence Standard 00-55 edition, Ministry of Defence, 1991.

[Nelson 91] G. Nelson (ed). *Systems Programming with Modula-3.* Prentice Hall, Englewood Cliffs, NJ, 1991.

[Pavey 93] D. J. Pavey and L. A. Winsborrow. "Demonstrating Equivalence of Source Code and PROM Contents". *Computer Journal* 36(7): 654-667, 1993.

[Plauger 92] P. J. Plauger. *The Standard C Library.* Prentice Hall, Englewood Cliffs NJ, 1992.

[Rex 88] *MALPAS User Guide* Release 4.1, RTP/4009/UG, Issue 3 edition, Rex, Thompson and Partners Ltd., 1988.

[RM83] See [ANSI 83].

[RM95] See [ISO 95].

[Sales 92] R. Sales. *Currency Sign Enhancements.* Technical Report X3J4/WR-684, ANSI X3J4, 1992. COBOL Full Revision Working Paper.

[Schaeffer 93] C. F. Schaeffer and G. N. Bundy. "Static Analysis of Exception Handling in Ada". *Software Practice & Experience* 23(10):1157-1174, 1993.

[Schaffert 86] C. Schaffert, T. Cooper, B. Bullis, M. Kilian, and C. Wilpolt. *An* "Introduction to Trellis/Owl". In *ACM OOPSLA'86 Proceedings.* Portland, OR, 1986.

[Seidewitz 91] E. Seidewitz. "Object-Oriented Programming Through Type Extension in Ada 9X". ACM SIGAda *Ada Letters* XI(2), March/April 1991.

[Sha 90a] L. Sha, and J. B. Goodenough. "Real-Time Scheduling Theory and Ada". *IEEE Computer* 23(4): 53-62, April, 1990.

[Sha 90b] L. Sha, R. Rajkumar, and J. P. Lehoczky. "Priority Inheritance Protocols — An Approach to Real-Time Synchronization". *IEEE Transactions on Computers* C-39(9), September, 1990.

[Stroustrup 91] B. Stroustrup. *The C++ Programming Language, 2nd Ed.* Addison-Wesley, Reading, MA, 1991.

[Taft 93] S. Tucker Taft. "Ada 9X: From Abstraction-Oriented to Object-Oriented". In *ACM OOPSLA'93 Conference Proceedings*, 1993.

[Tang 91]	P. T. P. Tang. "A Portable Generic Elementary Function Package in Ada and an Accurate Test Suite". ACM SIGAda *Ada Letters* XI(7): 180-216, Fall, 1991.
[Taylor 95]	W. J. Taylor. *Ada 95 Compatibility Guide, Version 6.0.* Transition Technology, January 1995.
[Wellings 84]	A. J. Wellings, D. Keeffe, and G. M. Tomlinson. "A Problem with Ada and Resource Allocation". ACM SIGAda *Ada Letters* III(4): 112-123, 1984.
[Wichmann 82]	B. A. Wichmann and I. D. Hill. "An Efficient and Portable Pseudo-Random Number Generator". *Applied Statistics* 31:188-190, 1982.
[Wichmann 89]	B. A. Wichmann. *Insecurities in the Ada programming language.* Technical Report DITC 137/89, NPL, January, 1989.
[Wichmann 92]	B. A. Wichmann. *Undefined scalars in Ada 9X.* March, 1992.
[Wichmann 93]	B. A. Wichmann. "Microprocessor design faults". *Microprocessors and Microsystems* 17(7):399-401, 1993.
[Wirth 77]	N. Wirth. "Modula: A Language for Modular Multiprogramming". *Software Practice & Experience*, January 1977, pp 3-35.
[Wirth 88]	N. Wirth. "Type Extensions". *ACM Transactions on Programming Languages and Systems* 10(2): 204-214, April 1988.

Index

The entries in this index refer to the section and not to the page number. In identifying an appropriate entry the reader is reminded that references into Chapter II (of part One) are likely to give a general introduction to a new feature, those into Chapter III (of part One) are likely to briefly summarize a feature in the context of the whole language, whereas those in Parts Two and Three (identified by chapters 1 to 13 and A to H respectively) will give further detail of specific aspects of a feature. Furthermore Appendix X concerns incompatibilities with Ada 83 and Appendix Y concerns changes since the Committee Draft and the Draft International Standard.