Guidance for the use of the Ada Programming Language in High Integrity Systems

Version 3.4
30 April, 1998

1. Introduction

As a society, we are increasingly reliant upon high integrity systems: for safety systems (such as fly-by-wire aircraft), for security systems (to protect digital information) or for financial systems (cash dispensers). As the complexity of these systems grows, so do the demands for improved techniques for their production.

These high integrity systems must be shown to be fully predictable in operation and have all the properties required of them. This can only be achieved by analysing the software, in addition to the use of conventional dynamic testing.

There is, currently, no mainstream high level language where all programs in that language are guaranteed to be predictable and analysable, so for all choices of implementation language it is essential to control the language features used by the application.

The Ada language is designed with specific mechanisms for controlling the use of certain aspects of the language. Furthermore,

1. The semantics of Ada programs are well-defined, even in error situations. Specifically, the effect of a program can be predicted from the language definition with few implementation dependencies or interactions between language features.
2. The strong typing within the language can be used to reduce the scope (and cost) of analysis to verify key properties.
3. The Ada language has been successfully used on many high integrity applications. This demonstrates that validated Ada compilers have the quality required for such applications.
4. Guidance can be provided to facilitate the use of the language and to encourage the development of tools for further verification.

Ada is therefore ideally suited for implementing high integrity software and this document provides guidance in the controls that are required on the use of Ada to ensure that programs are predictable and analysable.

There are, however, a number of different analysis techniques in use for high integrity software and this document is not prescriptive about which techniques to use. Furthermore, each analysis technique requires different controls on the use of the language. This document therefore identifies, for each technique, the controls over the use of the language that are needed for proper application of that technique.

It is the user's responsibility to select the analysis techniques for a particular application; this document can then be used to define the full set of controls necessary for using that set of techniques.

The guidance given here first specifies its scope, by reference to the safety and security standards to which critical applications may be written.

Section 3 then analyses the verification techniques that are applied in the development of critical systems. By this means, the regulatory rules of the standards for safety and security are abstracted to avoid the need to consider each such standard separately.

Section 4 addresses general issues concerning how computer languages must be constructed if programs written in that language are to be fully predictable. These issues are relevant to any restricted language defined through the application of this guidance.

Section 5 provides identification of a three-way classification system used for Ada language features. This classification is based upon the ease with which verification techniques can be applied to a program containing the feature. This classification is needed since while the majority of the core features in Ada assists verification, the use of certain features makes the resulting code difficult or impossible to analyse with the currently available program analysis tools and techniques.

Section 6 provides the main technical material of this guide by classifying Ada language features. Users of this guide can then determine which features of Ada are appropriate to use from the verification techniques that are to be employed. The assessment has shown that the vast majority of the Ada features lend themselves to effective use in the construction of high integrity systems.

The guide concludes by providing information to aid the choice of a suitable Ada compiler together with its associated runtime system.

A bibliography to relevant standards and guides is provided.

A comprehensive index is provided to ease the use of the guide.

1.1 Levels of criticality

Many of the Standards to which high integrity software is written use multiple levels to classify the criticality of the software components which make up the system. While the number and nature of the levels vary, the general approach is always the same: the higher the criticality of the system, the more verification techniques need to be used for its assurance. Table 1 relates the various levels of classification used in some well known International Standards.

Standard	Number of levels	Lowest Level	Highest Level
[DO-178B]	4	D	A
[IEC-61508]	4	Safety Integrity Level 1	Safety Integrity Level 4
[ITSEC]	7	E0	E6

Table 1: Levels of criticality in some Standards

This guidance places more emphasis upon the higher levels of criticality, for which the more demanding verification techniques are employed and for which Ada provides major benefits.

This guidance, however, does not directly use any such levels but focuses on the correlation between the features of the language and the verification techniques to be employed at the higher levels of criticality. The material in [ISO/IEC 15026], [ARP 4754] and [ARP 4761] may be useful in determining the criticality of a system if this is not covered by application-specific standards.

1.2 Readership

This report has been written for the following;

1. Those developing critical systems in Ada who have a working knowledge of the language including an understanding of good Ada style (as in [AQS]).
2. Regulators who need to approve critical systems written in Ada.
3. Vendors marketing Ada compilers and other tools designed for use on critical systems.
4. Those concerned with critical systems who wish to consider the advantages of using the Ada language.

2. Scope

This technical report provides guidance on the use of Ada when producing high integrity systems. In producing such applications it is usually the case that adherence to guidelines or standards has to be demonstrated to independent bodies. These guidelines or standards vary according to the application area, industrial sector or nature of the risk involved.

For safety applications, the international generic standard is [IEC-61508] of which part 3 is concerned with software.

For security systems, the multi-national generic assessment guide is [Common Criteria].

For sector-specific guidance and standards there are:

Airborne civil avionics: [DO-178B]

Nuclear power plants: [IEC 880]

Medical systems: [IEC 601-4]

Pharmaceutical: [GAMP]

For national/regional guidance and standards there are the following:

UK Defence: [DS 00-55]

European rail: [EN 50128]

European security: [ITSEC]

US nuclear: [NRC]

UK automotive: [MISRA]

US medical: [FDA]

US space: [NASA]

The above standards and guides are referred to as Standards in this technical report. The above list is not exhaustive but indicative of the type of Standard to which this technical report provides guidance.

The specific Standards above are not addressed individually but their requirements and recommendations have been analysed from which this guidance is synthesised.

2.1 Within the scope

These guidelines assume that a system is being developed in Ada to meet a standard listed above or one of a similar nature. The primary goal of these guidelines is to translate general requirements into Ada specific ones. For example, a general standard might require that dynamic testing provides evidence of the execution of all the statements in the code of the application. In the case of generics, this is interpreted by the guidelines to mean all instantiations of the generic should be executed.

These guidelines are intended to provide guidance only, and hence there are no ‘shalls’. However, these guidelines identify verification and validation issues which should be resolved and documented according to the sector-specific standards being employed.

The following topics are within the scope of this guidance:

- the choice of features of the language which aid verification and compliance to the Standards,
- identification of language features requiring additional verification steps,
- the use of tools to aid design and verification,
- issues concerning qualification of compilers for use on high integrity applications,
- tools which generate Ada code where the code is visible and potentially modifiable (hence the tool vendor needs to take these Guidelines into account).

2.2 Out of scope

The following topics are considered to be out of scope with respect to these guidelines.

- Domain-specific Standards,
- Application-specific issues,
- Hardware and system-specific issues,
- Human Factors,
- Tools which generate Ada but this code is invisible to the systems developer.

3. Verification Techniques

{remove Ada references & change to Ada & other high level languages}
Verification is the process of ensuring that the implementation satisfies the requirements, whereas validation is the process of ensuring that the requirements are correct and complete.

There are currently four processes required by standards to support the verification of software:

1. traceability,
2. review,
3. analysis, and
4. testing.

Each one of these is discussed below to highlight its relationship to the Ada programming language.

3.1 Traceability

Traceability is required to establish that the implementation is complete, and to identify new derived requirements. It occurs throughout the life-cycle e.g. there needs to be traceability from

- lower level (decomposed / derived) requirements to higher level requirements;
- test procedures to requirements, design, or code;
- object code to source code.

While this process is not language specific, certain attributes of design or coding styles can aid (or detract from) this process. For example, consider a single module that implements a single low level requirement, which has an associated single test procedure, then the method to support traceability is straightforward. On the other hand, if there is a many-to-many relationship between the various decomposed levels of software (because of design choices or test procedures), traceability can become very complicated. Deduction of completeness of implementation (without extraneous code) may therefore be difficult or impossible.

Additionally, the use of some of the more sophisticated language features of high-level languages that require extensive compiler generated code may detract from the straightforward translation into, and hence verification of, the object code.

3.2 Reviews

Reviews are an important part of the verification process. They can be carried out on requirements, design, code, test procedures, or analysis reports. Reviews are conducted by humans and may be undertaken ‘formally’ such as in a Fagan inspection or ‘informally’ such as in desk checks. Typically, reviews are done by an ‘independent’ person i.e. the producer of the artefact is different from the reviewer. This independence is a mandatory requirement of safety-critical software standards.

Coding standards and avoidance of certain language features of high-level languages are essential for critical systems in order to facilitate reviews. These aspects become important since the 'independent' code review may at times be conducted by an expert in the application domain who may not have detailed insight into language constructs and their interactions.

3.3 Analysis

These guidelines distinguish between analysis (i.e. static analysis) and testing (i.e. dynamic analysis). Analysis supplements testing to establish that the requirements are correctly implemented.

Analysis can be performed on requirements, design, or code; the major emphasis of these guidelines is the analysis of the design and code.

Described below are ten analysis methods which are required in different combinations by various standards.

1. Control Flow
2. Data Flow
3. Information Flow
4. Symbolic Execution
5. Formal Code Verification
6. Range Checking
7. Stack Usage
8. Timing Analysis
9. Other Memory Usage
10. Object Code Analysis

3.3.1 Control Flow Analysis

Control Flow Analysis is conducted in order to ensure that:

1. code is executed in the right sequence,
2. code is well structured, and
3. code execution terminates deterministically.

Call tree analysis, an example of one of the many Control Flow Analysis techniques available is used to verify that the sequencing stated by the design is correctly implemented. Also, call tree analysis can help detect direct and indirect recursion, which are prohibited be most high integrity standards. Also, if a system is partitioned into critical and non-critical parts, then the call tree analysis can confirm that the design rules for partitioning have been followed.

Ada supports and encourages well-structured code.

3.3.2 Data Flow Analysis

The objective of Data Flow Analysis is to show that there is no execution path in the software that would access a variable that has not been set a value. Data Flow Analysis uses the results of Control Flow Analysis in conjunction with the read or write access to variables to perform the analysis. Data Flow Analysis can also detect other code anomalies such as multiple writes without intervening reads.

The inter-module checking performed by an Ada compiler combined with the rules for parameter modes ensures that this analysis is tractable, when combined with suitable design rules.

3.3.3 Information Flow Analysis

Information Flow Analysis identifies how execution of a unit of code creates dependencies between the inputs to and outputs from that code. For example:

```
X:=A+B;
Y:=D-C;
if X>0 then
  Z:=(Y+1);
end if;
```

Here, X depends on A and B, Y depends on C and D, and Z depends on A, B, C, and D (and implicitly on its own initial value).

These dependencies can be verified against the dependencies in the specification to ensure that all the required dependencies are implemented and no incorrect ones are established. It can be performed either internal to a module (i.e. a procedure or a function), across modules, or across the entire software (or system). This analysis can be particularly appropriate for a critical output that can be traced back all the way to the inputs of the hardware/software interface.

3.3.4 Symbolic Execution

The objective of Symbolic Execution is to verify properties of a program by algebraic manipulation of the source text without requiring a formal specification. This technique is typically applied using tools that also undertake Control, Data and Information Flow Analysis.

Symbolic Execution is a technique where the program is 'executed' by substituting the left hand side with the right hand side in all assignment statements. Conditional branches are represented as conditions under which the relevant expression gives the values of the outputs from the inputs. To undertake this computation, it is assumed that no aliasing is taking place, i.e. two variables X and Y do not refer to the same entity, that there is no recursion, and that functions have no side-effects. Tools that provide support for symbolic execution may or may not check for these above conditions.

The information on static bounds of numeric values that Ada provides allows Symbolic Execution to demonstrate freedom from run-time errors in many situations. The algebraic expressions giving the output in terms of the input can also be compared (manually) with the specification of a subprogram to verify the code.

Using the fragment of Ada code which illustrated Information Flow Analysis gives:

```
A+B ≤ 0:
    X = A+B
    Y = D-C
    Z = undefined (initialised value)

A+B > 0:
    X = A+B
    Y = D-C
    Z = D-C+1
```

An example of this technique is to show, from the source text alone, that a subprogram is free from run-time errors (e.g. Range Checking). The conditions under which a run-time error would occur is given as an expression involving the input values, but these conditions should be mutually inconsistent; if not, the solution of the conditions will give values which would result in a run-time error.

3.3.5 Formal Code Verification

Formal code verification is the process of proving that the code of a program is correct with respect to the formal specification of its requirements. The objective is to explore all possible program executions which is infeasible by dynamic testing alone.

Each program unit is verified separately, against those parts of the specification that apply to it. For instance, formal code verification of a subprogram involves proving that its code is consistent with its formally-stated post-condition (specifying the intended relationships between variables on termination), given its pre-condition (specifying the conditions which must apply when the subprogram is called). A more restricted proof aimed at demonstrating a particular safety property can also be conducted.

The verification is usually performed in two stages:

1. Generation of verification conditions (VCs). These theorems are proof obligations, whose truth implies that if the pre-condition holds initially, and execution of the code terminates, then the post-condition holds on termination. These VCs are usually generated mechanically.

2. Proof of verification conditions. Machine assistance in the form of a suitable proof tool can be used to discharge verification conditions.

The process outlined above establishes *partial correctness*. To establish *total correctness* it is also necessary to prove *termination* of all loops when the stated pre-condition holds. Termination is usually demonstrated by exhibiting a *variant* expression for every loop, and showing that this expression gives a non-negative number that decreases on each iteration. *Termination conditions* can be generated and proved, similar to the generation and proof of verification conditions.

The value of formal code verification depends on the availability of a specification expressed in a suitable form such as results from formal specification methods. Formal methods involve the use of formal logic, discrete mathematics, and computer-readable languages to improve the specification of software.

Proof of absence of run-time errors
In some real-time safety-critical systems, occurrence of run-time errors is not acceptable. (An example is the flight-control system of a dynamically unstable aircraft, in which there would not be time to recover from such an error.) The techniques of formal code verification described above can be used to prove that (with appropriate language constraints) certain classes of run-time errors, e.g. range constraint violations, cannot arise in any execution.

To perform such verifications, the object type and variable declarations are used to construct pre-conditions on the ranges of initial values, and at each place in the source code where a run-time check would be produced, an assertion formally describing the check is generated. From these pre-conditions, assertions and the program code, verification conditions are mechanically produced.

These verifications (or ‘proof obligations’) are very numerous, but for the most part simple enough to be proved automatically. Full formal requirement specifications are not needed to apply this technology. Difficulties in constructing proofs may indicate the need for local use of ‘defensive programming’.

3.3.6 Range Checking

The objective of this analysis is to verify that the data values lie within the specified range as well as maintain specified accuracy. These forms of analysis include, but are not limited to,

1. overflow and underflow analysis,
2. rounding errors,
3. range checking, and
4. array bounds.

For discrete types, the static bounds placed upon variables often allows many cases to be checked automatically. When enumerated types are used instead of integer types, these checks are more effective. For real types, the need to show the absence of overflow is more demanding than the analysis of operations on discrete types.

Since the semantics of Ada remain defined even in error conditions, the necessary checks can be explicitly specified.

3.3.7 Stack Usage Analysis

The stack is a part of the memory shared by different subprograms and used for storing data local to the subprogram, temporary data and return addresses generated by the compiler. Stack Usage Analysis is a particular form of shared resource analysis that establishes the maximum possible size of the stack required by the system and whether there is sufficient physical memory to support this maximum stack depth. Also, some compilers use multiple stacks and this analysis needs to be conducted for each stack.

Another aspect of Stack Usage Analysis is to ensure that there is no stack-heap collision at run-time. This analysis is avoided when dynamic heap allocation is prohibited.

Stack Usage Analysis is made simpler for a programming language such as Ada where subprogram calls and return semantics are unambiguous, and where there is a clean distinction between static and dynamic types.

3.3.8 Timing Analysis

The overall objective of this analysis is to establish temporal properties of the input/output dependencies. A common and an important aspect of this analysis is the worst-case execution time for the correct behaviour of the overall system.

Certain programming language features or design approaches make timing analysis difficult, e.g. loops without static upper bounds and the manipulation of dynamic data structures.

The static typing of Ada and the unambiguous semantics of its control structures facilitates these analyses. Also, the pragmas Reviewable and Inspection_Point ensures that there is traceability from the source code to the object code to facilitate timing analysis.

3.3.9 Other Memory Usage Analysis

This analysis is required for any resource that is shared between different 'partitions' of software. These forms of analysis include, but are not limited to, memory (heap), I/O ports, and special purpose hardware, which perform specific computations or watchdog timer functions.

Analysis will show the absence of interference between Ada and other programs such as low-level and hardware device drivers and resource managers. In particular heap memory should usually be avoided and IO devices rigorously partitioned. The pragma Restrictions (No_Allocators) can be used to ensure no explicit use of the heap and pragma Restrictions (No_Implicit_ Heap_Allocation) to ensure no implicit usage of the heap.

3.3.10 Object Code Analysis

The purpose of Object Code Analysis is to demonstrate that object code is a correct translation of source code and that errors have not been introduced as a consequence of a compiler failures.

This analysis is sometimes undertaken by manual inspection of the machine code generated by the compiler. Alternatively, the compiler vendor may provide details of the mapping from the source code to object code so that manual checks are simpler to undertake. Unfortunately, it is not currently within the state of the art to formally verify the equivalence of source code and the generated object code.

The Ada pragma Inspection_Point provides a means of tracing from source code to object code. This facility means that mechanical checks can be undertaken in a similar way to that for a simpler language as illustrated in [16].

3.4 Testing

3.4.1 Principles

Testing (sometimes known as dynamic analysis) is the execution of software on a digital computer, which is often the target machine on which the final application runs. Testing has the advantage of providing tangible, auditable, evidence of software execution.

There are many testing techniques and new ones are being invented continually. This section is limited to those procedures that are required by various software standards. It is not intended to be an exhaustive encyclopaedia of the various testing techniques known at the present time.

Testing can be performed at various levels of software (and system):

- Software module level (individual procedures or functions),
- Software integration testing (i.e. module integration testing),
- Hardware/Software integration testing, and
- System testing.

The testing procedures described below focus on the first two aspects i.e. module and module integration testing, since the choice of programming language has a direct impact on the ease or difficulty of testing. Within this framework, there are two basic forms of testing:

- Requirements-based (or black-box) testing, and
- Structure-based (or white-box) testing.

Since exhaustive testing is infeasible for any realistic program, one approach to limit the number of test cases to partition the data domain into equivalence classes and their boundary values.

3.4.2 Requirements-Based Testing

The requirements-based testing methods aim to show that the actual behaviour of the program is in accordance with its requirements. For this reason, these methods are sometimes also called ‘functional testing’ or ‘black-box testing’. This is to highlight the fact that the program structure is not taken into account. There are two common methods for conducting requirements-based testing:

Equivalence Class Testing
This approach partitions the variables’ range (both inputs and outputs) into subclasses such that values in the ‘normal’ range as well as on the boundaries of the range represent the ‘equivalent’ of the entire range of data values. For variables that have enumerated types, all values need to be included in some, but not necessarily the same, equivalence class. If an integer type (or fixed or floating type) is used which includes positive, negative, and zero values, then the equivalence classes should jointly include at least one value from each of these ranges. Finally, in the case of real types (fixed or floating types), it requires testing taking into account the resolution (besides the range) of the variables.

Boundary Value Testing
This approach enhances the equivalence class testing by requiring values outside the ‘normal’ range. This form of ‘stress’ testing or ‘robustness’ testing generates additional confidence in terms of handling invalid values (both inputs and outputs).

3.4.3 Structure-based Testing

The objective of these testing methods is to increase the confidence in software by exercising the program beyond the requirements-based testing mentioned above.

Examples of the methods used, and their definitions, are:

Statement Coverage
Every statement in the program has been invoked at least once.

Decision (Branch) Coverage
Every point of entry and exit in the program has been invoked at least once, and every decision in the program has taken all possible outcomes at least once.

Modified Condition/Decision Coverage
Every point of entry and exit in the program has been invoked at least once, every condition in a decision in the program has taken all possible outcomes at least once, every decision in the program has taken all possible outcomes at least once, and each condition in a decision has been shown to independently affect that decision’s outcome. A condition is shown to independently affect a decision’s outcome by varying just that condition while holding fixed all other possible outcomes.

These structure-based testing approaches apply only to source code of high-level languages such as Ada. Ada includes features especially useful for constructing test-specific code. For example, a test point subprogram – intended to examine the values of ‘hidden’ data items – can be made a subunit and replaced with a null body in the delivered code. Similarly, a private child package can be used to allow test access to a package’s state without the need to modify the package itself.

3.5 Use of Verification Techniques in these Guidelines

When considering the previous verification techniques, certain groupings can be formed. The verification techniques within a group have similar properties with respect to the complexity of that technique when applied to a particular language feature. The following table shows the groups, and abbreviations, used in the section Assessment of Language Features (Section 6).

Type of Method	Group Name	Method Name
Analysis	Flow Analysis (**FA**)	Control Flow
		Data Flow
		Information Flow
	Symbolic Analysis (**SA**)	Symbolic Execution
		Formal Code Verification
	Range Checking (**RC**)	Range Checking
	Stack Usage (**SU**)	Stack Usage
	Timing Analysis (**TA**)	Timing
	Other Memory Usage (**OMU**)	Other Memory Usage
	Object Code Analysis (**OCA**)	Object Code Analysis
Testing	Requirements-based Testing (**RT**)	Equivalence Class
		Boundary Value
	Structure-based Testing (**ST**)	Statement Coverage
		Branch Coverage
		Modified Condition/Decision Coverage

Table 2 : Types of Verification

For convenience, some methods are grouped together as follows:

Flow Analysis
The three methods under this group name are Control Flow, Data Flow and Information Flow (Analysis). The basic methods of control flow and data flow are combined with information flow analysis.

Symbolic Analysis
Two methods are grouped together here: Formal Code Verification and Symbolic Execution. Both these methods share a mathematical foundation and typically use tools which analyse the source text of a program algebraically.

Requirements-based Testing
The Equivalence Class and Boundary Value are closely related in that the Boundary Value approach extends the choices by considering values outside the normal range of variables.

Structure-based Testing
Three testing methods are combined here, which in order are: Modified Condition/Decision Coverage, Branch Coverage and Statement Coverage. These methods are closely related and are often tested with the help of the same tool.

4. General Language Issues

4.1 Writing Verifiable Programs

Standards and guidelines for high-integrity software demand assurances of its fitness for purpose, based on its static analysis (a number of different forms of reasoning about the specification, design, and code, checking their properties and relating them, see section 3.3), and also dynamic analysis (observing and checking behaviour in execution, of components of the software or of the entire system, see section 3.4).

The choice of the verification techniques to be used for a software development is one part of the definition of the overall development process for that software. Coding standards and language subsets can only be defined after the analysis methods have been selected, to ensure that none of the analysis methods are compromised by the use of inappropriate language features.

The main part of this document (Section 6) gives details on the interaction of verification techniques and specific language features. This section looks more generally at the requirements for analysable programs.

In the context of high-integrity systems, fitness for purpose depends on the behaviour of a system being predictable, and demonstrably conforming to a precise specification of its requirement. Historically, demonstration of conformance to requirement specifications by static and dynamic analysis has usually been attempted retrospectively, after a system has been developed. This approach has rarely proved a cost-effective way of producing the desired results, for several reasons. Firstly, because the design and implementation processes have not been chosen with verification in mind. Secondly, the static or dynamic analysis technologies have been difficult and costly to apply, and were not very revealing. And finally, where such analyses have revealed errors, these were detected so late that their correction was extremely costly.

The high cost of repeated testing as mistakes are uncovered and rectified, and improvements in the technology of static analysis, have led to a realisation that there are significant cost benefits in performing static code analysis as the code is produced. This accords well with the movement, as specification methods have improved, towards 'Correctness by Construction' - the 'lean engineering' view of software development. Here static analysis methods, where they are relevant, feature as an integral part of the development process. Their purpose is to establish the validity (in particular respects) of intermediate products before they are used to develop further ones. This constructive approach also contrasts with retrospective verification in matters of style: where certain language features are positively required in order to support a particular model. The retrospective approach leads to a list of language prohibitions. Although the constructive approach is more demanding, conceptually, it uses language and tools to better effect, improving technical quality as well as reducing costs.

With motivations such as these for performing verification, it can be seen that there are four different reasons for needing or rejecting particular language features within this context:

1. Language rules to achieve predictability,
2. Language rules to allow modelling,
3. Language rules to facilitate testing,
4. Pragmatic considerations.

In addition, there are also reasons to enhance the language by adding forms of annotations.

In the following sections these matters are considered in detail.

4.1.1 Language Rules to Achieve Predictability

Language rules to achieve predictability are independent of the analysis methods to be used. It is a requirement, in all high-integrity system development, that the program source code is unambiguous. All possible forms of language ambiguity must be prevented in some way.

The ways in which programs written in a high-level language can be ambiguous are well-known. A few cases are discussed below.

Side-effects
It is generally accepted that side effects in subprograms are undesirable. Side effects can make order of evaluation of an expression a significant issue, and can make repeated calls to a function return different results. Many programming languages permit functions to have side-effects.

Effects of elaboration order
Languages that support default initialisation or elaboration will be dependent on the order of evaluation within a unit and the order in which units are evaluated.

Effects of parameter-passing mechanisms
Parameters to a subprogram in a high level language can be passed as a copy or passed as a reference. The results of an execution may depend upon which method is chosen by an implementation (where a choice is permitted).

Such ambiguities make the formal analysis of a program difficult. The uncertainties can be reduced by writing programs that ensure they do not depend on these types of issues.

There is often an attempt in high integrity systems to eliminate offending language features to remove the uncertainties or implementation dependencies. In practice, this is often not feasible, since they occur because of the interactions of features or are fundamental to the programming system.

4.1.2 Language Rules to allow Modelling

All static analysis methods involve the construction of some kind of model of program source text (or object code), and the application of algorithms or reasoning processes to this model, to check whether it has some particular properties. It is possible that if a particular feature of a language is used at all, or in a particular way, the model would be unable to capture those aspects of the program text that are important to the analysis to be performed. As a consequence, to ensure validity of a static analysis method some language features must be disallowed or their use rigorously controlled.

Assuming that language rules, or other devices, exist to ensure a program is meaningful, this section considers what is required for the various forms of modelling and analysis methods to be well founded and, at least in principle, applicable. The next section will consider what more has to be done to make them useful and economically viable.

The simpler kinds of static analysis, in particular control, data, and information flow analysis, essentially involve the study of paths (corresponding to execution histories) on a directed graph model of a program (effectively a flowchart). The arcs of the program graph have labels which capture, to different extents, syntactic and semantic properties of the program statements. For control flow analysis only the graph structure is required, for data flow analysis each arc label specifies which variables the corresponding statement reads and updates, for information flow analysis the dependencies between read and updated variables must be defined a little more precisely. Conceptually, formal code verification (and proof of absence of run-time errors) is also based on paths (or execution histories) on such a graph, but here the arc labels are partial functions of the set of program variables, defining the conditions for traversing the arcs and the transformations of variables performed in doing so. Thus, it is possible to envisage all these forms of static analysis as being applied to the same structural model, but with progressively 'richer' descriptions of the program objects and statements.

The inclusion of exception flows significantly complicates the graph structure and tends, in the general case, to the analysis becoming intractable; the goto statement also introduces difficulties. Otherwise, there are no obstacles to control-flow analysis, which only requires and provides relatively limited information.

Data-flow analysis begins to take an interest in the data objects of a program. When using this method difficulties of distinguishing between dynamically-selected components of composite objects are first encountered. Two issues are: arrays must be treated as entire objects (although their components, selected by dynamically-computed indices, can reappear in formal code verification), and dynamic creation of objects in the course of execution (using access types) cannot be represented. Information flow analysis essentially requires the same constraints.

It is clear that such static models of programs only apply to sequential code. If the program contains concurrent threads (e.g. tasks) then each thread must be analysed independently in terms of its flow, with the concurrency aspects being addressed by different models and reasoning processes. With a concurrent system the key objective of modelling is to prove that the system cannot enter undesirable states such as deadlock. The main static techniques for achieving this are finite state automata [3], petri-nets [4],[5] and process algebras [6],[7],[8] using model checking or mathematical proof. All try to construct the set of all feasible histories for the system to show that unsafe conditions cannot be reached. The degree to which thread synchronisation impacts on the flow analysis of individual threads determines the complexity of the overall verification problem. The more asynchronous the model, the more straightforward the flow analysis of each thread. If there is only asynchronous data passing between threads then each thread can be verified in isolation. Moreover, the complete timing/scheduling analysis of the entire concurrent system becomes tractable.

Surprisingly perhaps, the domain to which formal code verification is applicable in principle is no more restricted; indeed, it is somewhat larger, as composite objects can be considered more generally. There is a large gap, however, between what is analysable in principle and what can be analysed in practice and a number of strategic choices arise. If the dynamic semantics of the language are made as simple as reasonably possible, the burden of verification is placed as far as possible on static analysis tools. So, for example, if a program is deemed exception free by formal proof, coverage testing is significantly simplified.

In conclusion, the adequacy of modelling for different analysis methods gives rise to some language constraints. But just as important in determining how a programming language should be used, are the precise reasons for using the analysis methods and the ways in which they are to be employed - in other words, the choice of software development process.

4.1.3 Language Rules to Facilitate Testing

Demonstration of a program's correct dynamic behaviour, by testing, is an important part of its verification. In practice the development of critical systems usually involves two different forms of testing: investigative, informal or debugging testing; and formal coverage testing of a kind or kinds required by various standards. As noted above the detection and elimination of errors by static analysis methods early in the project lifecycle can greatly reduce testing costs by shifting the emphasis of testing from debugging to demonstrating correctness.

The constraints placed on programming languages to facilitate testing are generally less significant than those demanded by static analysis techniques; indeed, for requirements-based ("black box") testing there are no language feature implications at all. Although language features do have some impact on structure-based ("white box") testing, the use of static analysis techniques as part of the development process will ensure that language features which might complicate such testing will be avoided.

Language features that obstruct coverage testing are those such as dispatching which introduces very dynamic control flow; goto which complicates control flow; and features which complicate the view taken of data, for example: variant records, unchecked conversions and dynamic pointer usage. Predefined exceptions present a particular difficulty for coverage testing because it is often not possible to reach all the paths these introduce by external stimulation of unmodified code.

Language features that assist testing are those which constrain data values by strong and static typing and those which assist in locating errors by indicating their presence close to the point where they arise; here predefined exceptions are beneficial.

There is a general tension between the dictates of good software engineering, which encourages information (or more accurately "detail") hiding, and testing which often seeks to monitor the values of internal system state. Language features which enable these conflicting requirements to be reconciled are of particular value and avoid distorting the design just to achieve testability. Ada has particular strengths in this area which often allow test harnesses to have access to "hidden" data in non-intrusive ways. Examples include:

- A child package may be used to monitor state in the private part of its parent without requiring any change to that parent package.
- Subunits may be used to place the source for an embedded subprogram in a separate file thus allowing the construction of a suitable test driver for it. Test point subprograms in packages can also usefully be placed in a subunit and replaced with a null subprogram in delivered code.
- Parameterless functions may be used to return the value of "trimming variables" which are often found in control systems. These variables behave as constants at the program level but can be dynamically adjusted by direct memory access during rig testing. The use of parameterless functions to return their value reconciles the software's view that they are constants with the testing need to adjust their values.
- The use of a package to contain "test point" variables. These are "write-only" variables used solely for test monitoring purposes. At the point where the (hidden) value to be monitored is generated a copy can be passed to this package and used to update its state; this is preferable to distorting the design by making the data directly visible and emphasises the distinction between state needed by the software for its correct behaviour and that introduced for testing purposes only.

4.1.4 Pragmatic Considerations

The considerations of side effects and elaboration order described in section 4.1.1 impose conditions essential to the validity and technical relevance of static analysis methods. Further linguistic issues are important to the tractability and eventually the economic viability of their application.

The question of what conditions make the static analysis methods easy to apply, and what makes their results meaningful and useful, have essentially the same answers: a program should be well-designed.

All the static and dynamic analysis methods and construction techniques are most effective when software is well-structured, with every module having a single entry point and a single exit point. Although data flow analysis can still be applied to ‘spaghetti code’, it will be less effective in finding data flow errors and anomalies. Similarly, information flow analysis can be extended to ‘spaghetti code’, for which the unsurprising outcome is that almost every variable may depend on every other. In constructing proofs, it is preferable to have building blocks with pre-conditions on their single entry points and post-conditions defining state at their single exits. For all these reasons language subsets for safety-critical programming invariably exclude goto statements, return statements from procedure subprograms and apart from at the top most level, exception handlers are avoided.

Proving correctness of programs, like reasoning about them informally, is about relating fragments of code to fragments of specification. Decomposition of functionality is very important. Verifiability depends on simplicity of contextual information i.e. scope and visibility should be as limited as reasonably possible. So it is important to pay attention to architecture, the location of state, access to state, and embedding.

For such reasoning to be practical, the use of two-valued logic is preferred, where predicates are either true or false but not undefined. To achieve this it is useful for all variables to have valid values. This implies that there should not be undefined variables, a condition that can be met in three different ways:

- by language constraints, e.g. imposing initialisation of all variables in their declarations;
- by using a data flow analyser, to ensure that no variable is ever read before being initialised;
- by a tool-assisted software development process that ensures proper initialisations.

Of these options, the latter two are preferred, since it is usually better for all initialisations to be meaningful. The conclusion is that for high-integrity software development, the language employed is more than a pick-and-mix selection of language features. It must be built as a coherent whole as a means of expressing a chosen development philosophy, that is compatible with tools used to implement it.

4.1.5 Language Enhancements

A programming language should be used in a style that enhances its fitness for purpose. The rules discussed above for predictability and modelling are restrictions on the language: the elimination of syntactic forms or other, more subtle features. Fitness can also be enhanced by adding elements to the standard language, to facilitate derivation or verification of a program given its specification.

Tools and techniques that relate the source code to a formal specification, such as Formal Code Verification, often need information extra to the program source in order to work efficiently. Such information describes program properties that follow from the laws of the language and that the programmer intuitively believes but are not expressed directly in the language syntax. Examples include loop invariants, relationships between formal parameters, and state hidden inside other service-providing packages.[1] Capturing this information enables proofs that are otherwise computationally hard.

One approach to verification is to capture these properties by embedding annotations (formal comments), with their own syntax and semantics, in the source code, together with static design rules that relate the annotations to program objects. An annotation can express an invariant as an equivalence between program objects, and can reveal hidden state in packages without compromising their integrity.

The literate programming paradigm [18] provides another way of relating source code to specification. In literate programming, program and program fragments are embedded within a larger document, that also contains natural language description and other formal notations. One can relate and check objects in different fragments, and assemble a complete program for compilation.

Other development paradigms involve automated generation of package templates from a specification. As well as the package code, the toolkit may supply additional information to the programmer, to exclude unwanted interactions across module boundaries.

All of the above approaches facilitate correctness-by-construction, by narrowing the gap between a package or component and its specification, and by enabling its verification in a stand-alone manner at any time during development.

[1] It is a design goal of Ada that a client routine should not access state variables in another package (to prevent the client abusing them, among other good reasons). However knowledge of the state is needed to reason about the behaviour of the whole program.

4.2 The Choice of Language

It was noted in the Introduction to this report that any language that is to be used for the implementation of high integrity systems must:

- be strongly typed,
- support a range of static types,
- have consistent semantics that are defined in an international standard,
- support abstractions and information hiding,
- have available validated compilers.

Ada is unique in its compliance to all these attributes. Nevertheless it may be necessary to restrict or prevent the use of certain features to achieve full predictability, and to allow all the forms of static and dynamic analysis considered important. However verification poses few problems with the core of the language.

Rules of Ada usage are determined by considerations of the elimination of ambiguity, the feasibility of modelling and analysis, and the constructive use of static analysis. In some cases it is found that a language feature is undesirable (e.g. goto statements) or that it renders some kind of static analysis intractable (e.g. general exceptions) or infeasible (e.g. full tasking). More frequently the difficulty stems from the use of several features in combination in a particular manner. The most satisfactory way of applying appropriate restrictions in this case is through annotations, and rules between these and the Ada code. The annotation system requires very careful design if it is to be secure. The most important concern is with the use of good programming style; with ways of using Ada that provide good implementations of specifications and designs, and that can to a large extent be rigorously verified as they are constructed. Experience indicates that when these conditions are met, dynamic analysis is also greatly simplified.

The production of a complete and coherent annotated subset of Ada, with adequate tool support, deemed technically sound and economically viable by its users, is a major undertaking. It requires considerable financial investment, and years of patient labour by its developers and those who wish to apply it. These guidelines do not provide a single subset but rather describe the properties that such a subset should have.

Consideration of the language requirements for correctness by construction is a matter of positive choices. It may lead to some restrictions on the use of Ada, but these should be based on sound engineering judgement of the most appropriate combination of language features. Here the aim is to use Ada as the vehicle to support the chosen design and development paradigms, in the course of code production - which may for example suggest a particular way of using packages to implement object templates.

To perform all necessary constraint checks as a program components are constructed, it is necessary to utilise specification and design information, and relate it to the program architecture and code. To do this tools such as [17], and [19] use a system of annotations (‘formal Ada comments’), as discussed in 4.1.5. Introduction of an annotation system may also bring with it some stylistic restrictions on the use of Ada, for instance to reduce overloading and limit visibility. Annotations may also be necessary to fully support modular development. For example, to reason about the process performed by a module, knowledge of precisely what information it is accessing, and what are its effects, direct and indirect is required; its use or action on a global variable cannot be ignored.

The correctness by construction approach, with correctness checking module-by-module, may also require rules to avoid unwanted properties arising from the incremental development. For instance, it must be possible to prevent recursion (usually frowned upon in safety-critical applications) by rules applicable at the module level, rather than by checking for recursion when a system is complete.

It is reiterated that Ada is currently the only viable language with sufficient industrial heritage that can provide the framework for static analysis and correctness by construction. As indicated above some restrictions on the use of the language are necessary. Section 6 of these guidelines provides a detailed assessment of these features.

5. Criteria for Assessment of Language Features

Control of the language features used in the implementation of high integrity systems is one essential ingredient to the use of a high order language. Ada was expressly designed to facilitate this effort. Yet, like all modern general purposes languages, Ada offers a broader variety of features than strictly required for any specific application domain. Hence, important design decisions need to be made to determine which language features best suit the implementation and verification requirements that emanate from the specific application domain.

The reasoning behind this can be shown by an example. High integrity control systems often handle physical data that continuously varies. The processing involves the input/output of these quantities from/to digital to analogue converters. Within the program, the natural way to handle such data is by means of an Ada real type (fixed or floating point). The validation process for fixed point is slightly different than for floating point. Hence, a design decision should be made as to which form of real type should be used. In doing so, one should also consider that real types are only suitable for modelling continuously varying quantities and, therefore, not needed at all in some types of applications.

The choice of language features is determined by the nature and criticality of the application and the verification techniques to be employed. In practice, however, the availability of software tools may further constrain the choice of language features. For instance, if formal verification with proof is required, then the use of any real types might not be viable so that the application might have to be coded using integers only.

These guidelines rate language features using three categories as follows:

Included

A feature is ‘included’ if it is directly amenable to the designated verification technique. Not surprisingly, most Ada features are rated ‘included’ for most verification techniques. Included features enable the analysis to be undertaken and directly support the production of high integrity code.

Allowed

A feature is ‘allowed’ if the designated verification step is not straightforward, but is still achievable; or if the use of the feature is necessary and the use of the problematic verification technique can be effectively circumvented.

Excluded

A feature is ‘excluded’ if there is no current cost effective way of undertaking the designated verification technique. Assurance of exclusion requires some form of verification.

6. Assessment of Language Features

The Ada features are split in fourteen groups closely related to chapters of the LRM. These are:

- types with static attributes;
- declarations;
- names, including scope and visibility;
- expressions;
- statements;
- subprograms;
- packages (child and libraries);
- arithmetic types;
- low level and interfacing;
- generics;
- types with dynamic attributes;
- exceptions;
- tasking;
- distribution.

Each group of features is assessed separately. For each group of features assessed, these guidelines include one evaluation table and one textual section providing constructive guidance to use of the designated features.

The table indicates how a given language feature performs with respect to each of the nine groups of verification techniques identified in Table 2. The rating 'included' is denoted by the abbreviation 'Inc' (plain faced) in the relevant entry; the rating 'allowed' by the abbreviation '**Alld**' (bold faced); and the rating 'excluded' by the abbreviation '**Exc**' (bold faced). Some of the ratings, including all marked as allowed or excluded, are accompanied by explanatory notes providing the rationale for the assessment.

How to Use these Guidelines
The user of these guidelines should proceed in three steps, as follows:

1. Determine the verification techniques required from the relevant application specific standards or guidelines.
2. Identify and understand the objectives to be satisfied by each of the verification techniques.
3. Using the tables in section 6, determine the actual rating of the language features.
4. Confirm that the resulting choice of subset and the additional verification steps for any allowed features can satisfy the programming and verification requirements. This step should take into account available tools.

In some situations a verification technique is hard to apply because of the interaction of two or more language features. When this occurs all that is addressed is the difficulty under the feature that is primarily responsible for the problem. For example, the use of representation clauses has impact on a number of features; but these issues are all considered in the low level programming section (section 6.9).

Predefined Language Environment
The Guidelines give no explicit consideration to the predefined language environment. If components of the environment are used in an high-integrity application, then three situations arise:

1. The components are written in Ada. In this case, the Guidelines in section 6 apply.
2. The components are part of the run-time system. In this case, the Guidelines in section 7 apply.
3. If neither of the above applies, then no guidance is provided.

6.1 Types with Static Attributes

The strong typing mechanism in Ada is a significant contribution to software engineering. All values are associated with the type that is appropriate to their domain, and the type definition covers precisely the set of applicable values. This section is only concerned with types and subtypes that have static bounds and attributes, and that are statically allocated. Moreover, this section ignores interactions with Representation Clauses, which are dealt with in Section 6.11.

Ada95 has added the notion of statically matching subtypes to Ada83. When a compiler determines that one subtype does not statically match another type, it will prevent any usage of the first type in inappropriate places. In addition, conversions can only occur between types that have a common ancestry, e.g. all numeric types, or types of the same derivation class.

The rules for the derivation of types, explicit and implicit conversion between types, and mechanisms for the derivation, extension and overriding of primitive subprograms are complete, consistent and beneficial to programming of high integrity systems. Values associated with one type must be explicitly converted to other types. This forces designers to be explicit about conversions and makes the conversions visible in the source code. In addition, conversions can occur only between types that have a common parentage, i.e. all numeric types, or types that have been derived from the same parent type. The name-based typing and derivation support software engineering principles and prevent many classes of errors.

Abstract tagged types and subprograms provide significant capability to Ada at virtually no run-time cost. A type that is declared abstract prevents objects of that type from being declared, although type derivation (or extension) is permitted, and these new types may then have objects and actual primitive operations. Abstract subprograms allow the definer of the type to make unavailable certain operations that would ordinarily be available for the type. When a tagged type is extended, abstract primitive operations ensure that the extender of the type provides 'real' code for each primitive operation, and does not accidentally use default or incomplete operations. Guidance is provided on the prevention of run-time dispatching, since the subprogram called is not statically determinable.

6.1.1 Evaluation

Feature	FA	SA	RC	SU	TA	OMU	OCA	RT	ST
Anonymous Types	Inc	Inc	Inc	Inc	Inc	Inc	Inc	Inc	Inc
Subtypes	Inc	Inc	Inc	Inc	Inc	Inc[1]	Inc	Inc	Inc
Enumerated Types	Inc	Inc	Inc	Inc	Inc	Inc	Inc	Inc	Inc
Character	Inc	Inc	Inc	Inc	Inc	Inc	Inc	Inc	Inc
Boolean	Inc	Inc	Inc	Inc	Inc	Inc	Inc	Inc	Inc
Integer	Inc	Inc	Inc	Inc	Inc	Inc	Inc	Inc	Inc
Derived	Inc	**Alld**[2]	Inc	Inc	Inc	Inc	**Alld**[2]	Inc	Inc
Arrays	Inc	Inc	Inc	Inc	Inc	Inc	Inc	Inc	Inc
Records	Inc	Inc	Inc	Inc	Inc	Inc	Inc	Inc	Inc
Discriminated Records	Inc	**Exc**[3]	Inc	Inc	Inc	**Exc**[3]	**Alld**[3]	Inc	**Exc**[3]
Tagged Types without 'Class	Inc	**Alld**[2]	Inc	Inc	Inc	Inc	**Alld**[2]	Inc	Inc
Class Wide Operations	**Exc**[4]	**Exc**[4]	Inc	Inc	**Alld**[4]	Inc	**Alld**[4]	Inc	**Exc**[4]
Abstract Types & Subprograms	Inc	Inc	Inc	Inc	Inc	Inc	Inc	Inc	Inc

Table 3 : Types with Static Attributes

6.1.2 Notes

1. The use of 'Image for any subtype should be avoided. 'Image returns an unconstrained string which requires unbounded memory techniques for implementation.
2. Derivation of a type causes an automatic overloading of all primitive subprograms of the type. This makes functional coverage harder, but still tractable since the effects are contained to the unit that does the derivation, plus units that depend on the original unit.
3. Discriminants on records can be used to create unconstrained objects, to make some components inaccessible in some variants, and to define indefinite generic formal parameters and private types. This leads to significant analysis complexity, potential execution-time exceptions, and can cause the use of dynamic memory techniques.
4. The analysis of class-wide calls involves dispatching and hence must consider every candidate object type and analyse the associated subprogram. This can be a difficult task because types can be extended anywhere in the partition.

6.1.3 Guidance

Integer types should always be declared with explicit ranges, instead of deriving from one of the predefined integer types.

Avoiding Run-Time Dispatching
User invoked dispatching only occurs if 'Class is used. Prevention of dispatching can be enforced by the use of pragma Restrictions(No_Dispatch).

The use of derived types or tagged types with 'Class requires that all operations on the types are checked to ensure that the operations called according to the language rules are the ones the application design requires.

6.2 Declarations

A declaration associates a name with an entity and describes some characteristics of that entity. The Ada language uses these characteristics powerfully both *statically* (at compile time) and *dynamically* (at run time) to ensure that entities are not used in inappropriate ways.

6.2.1 Evaluation

Feature	FA	SA	RC	SU	TA	OMU	OCA	RT	ST
Named numbers	Inc	Inc	Inc	Inc	Inc	Inc	Inc	Inc	Inc
Constants	Inc	Inc	Inc	Inc	Inc	Inc	Inc	Inc[4]	Inc
Variables	Inc	Inc	Inc	Inc	Inc	Inc	Inc	Inc	Inc
Initialisation [2]	Inc	Inc	Inc	Inc	Inc	Inc	Inc	Inc	Inc
Aliased Object	**Alld**[3]	**Alld**[3]	Inc	Inc	Inc	Inc	**Alld**[3]	Inc	Inc
Unconstrained Object [1]	Inc	Inc	Inc	Inc	Inc	Inc	Inc	Inc	Inc
Declarative Part in Block Statement	**Alld**[5]	**Alld**[5]	Inc	Inc	Inc	Inc	Inc	Inc	**Alld**[5]

Table 4 : Declarations

6.2.2 Notes

1. The use of discriminants in the creation of unconstrained objects buys high-integrity systems more problems than the advantages it may procure. The use of objects of a discriminated subtype may force the compiler to defer to run time important decisions on the management of the object (e.g. allocation, access). Those cases may prove difficult to analyse. The entries in Table 4 have been assigned assuming that discriminants are *not* used for the creation of unconstrained objects.
2. The initialisation of variable objects in high-integrity systems should always be static and explicit. The former property ensures that the initialising expression can be verified at compile time. The latter ensures that the initialisation is deliberate.
3. Aliased objects should be used with care as they can allow the creation of views of an object which may inadvertently modify the intended properties of the original object. The entries in Table 4 have been assigned assuming 'safe' utilisation of aliased objects and general access types. In particular, objects of a discriminated subtype or which have components associated with representation clauses are *fragile* against inappropriate updates through access values and, therefore, difficult to ensure by analysis.
4. Boundary Value Testing is more problematic if a constant is assigned a non-static value on scope entry.
5. Ada allows block statements to enclose a declarative part. Whereas the use of this feature is one of the language means for encapsulation and hierarchical program structuring, it also presents some drawbacks to flow and symbolic analysis as well as to structural coverage. For example, the limitation to visibility and control of scope-local variables that can be obtained for test purposes from outer parts of the program.

6.2.3 Guidance

Declarations should be used to encapsulate the program design in as exact and static a manner as possible. For instance, constraints should be as tight as possible and also static so that program properties can be statically verified.

Use of Named Numbers
Named numbers are a good thing in that they denote the value of a static expression which is evaluated, with full precision, at compile time. This makes them the natural target of all numeric expressions in the program for this would eliminate the possibility that `Constraint_Error` may be raised at run time during the evaluation of the expression. For the same reason, named numbers should be used at all times in the executable code in the place of numeric literals.

Initialisation of Variables
All variables should be given a meaningful value before use. Failure to do so may raise a predefined exception or cause erroneous behaviour at run-time.

Initial values may be given by:

1. Associating an explicit initialisation expression with the variable at the point of its declaration.
2. Making an assignment to the variable that will be executed prior to references to it.

Controlled assignments to uninitialised variables can conveniently be made in a subprogram which is called prior to the use of the variable.

For state variables in packages, assignments may also be made in the package elaboration part. A consistent approach to the initialisation of package state variables should be adopted.

In all cases, Data Flow Analysis should be used to confirm that every object has been assigned a value before it is used. The effectiveness of the analysis is undermined if variables are initialised unnecessarily (sometimes called 'junk initialisation').

Use of Aliased Objects
The strong typing of Ada facilitates the achievement of type-safe access to variables through access values. Furthermore, the accessibility rules of the language help in creating objects which are impervious to access by general access types. Static tools can also be defined using e.g. ASIS to complement the power of the compiler in the determination of 'unsafe' use of aliased objects. The use of Ada aliased objects is therefore preferable and safer than the use of pointer or reference objects in other languages.

6.3 Names, including Scope and Visibility

Entities are denoted by names controlled by the rules for scoping and visibility.

Name de-referencing in Ada is usually quite straightforward and determinable at compile time with a few notable exceptions. Renamings can be used to introduce short names for use in a restricted scope. Object renaming declarations can be used to provide a short name for a component of an object. Overloading of names can enhance the readability of a program if applied judiciously.

Nesting of packages inside other packages provides information hiding and containment, but does not increase the scope levels (i.e. cause the information to be nested at deeper levels on the program stack or task stack). There is a tension between nesting and non-nesting of subprograms inside other subprograms or tasks. Many opportunities arise to simplify algorithms by placing some of the logic inside a local function or procedure. On the other hand, high integrity software processes usually enforce the isolated testing of all executable units.

6.3.1 Evaluation

Feature	**FA**	**SA**	**RC**	**SU**	**TA**	**OMU**	**OCA**	**RT**	**ST**
Names	Inc	Inc	Inc	Inc	Inc	Inc	Inc	Inc	Inc
Static renaming	Inc	Inc	Inc	Inc	Inc	Inc	Inc	Inc	Inc
Renaming – complex or dynamic evaluation	**Exc**[1]	**Exc**[1]	**Exc**[1]	Inc	Inc	Inc	**Exc**[1]	Inc	Inc
Overloading	**Alld**[2]	**Alld**[2]	Inc	Inc	Inc	Inc	**Alld**[2]	Inc	Inc
Nesting package spec	Inc	Inc	Inc	Inc	Inc	Inc	Inc	Inc	Inc
Nesting package body, subprogram	Inc	Inc	**Alld**[3]	Inc	Inc	Inc	**Alld**[3]	Inc	**Alld**[3]

Table 5 : Names, including Scope and Visibility

6.3.2 Notes

1. The use of complex forms of renaming, i.e. those which require run-time evaluation of bounds or object components; or those which extend component lifetime, is excluded because it hinders symbolic analysis, flow analysis and range checking; and complicates object code analysis as it embeds run-time code that has no associated visible source-code.

2. If multiple subprograms of the same name are used (whether overloaded, overridden or homographic), it should be shown that the intended subprogram is in fact called.

3. The textual inclusion of packages or generics inside a package body, private part of a package, subprogram body, task or protected type creates challenges for coverage-based testing. External tests that stimulate the enclosing unit

can rarely exercise all of the branches, conditions, or statements in the enclosed unit. Similarly range coverage tests become problematic.

6.3.3 Guidance

Names are always resolvable at compile-time, but may involve range checks. Two aspects involving names are considered below.

Renaming
Renaming can improve readability but runs the risk of making aliasing hard to detect. Hence reviews, perhaps supported by tools, are needed to ensure correct usage. Such reviews are greatly simplified if continued use of the original name of a renamed entity is avoided.

The subtype indication in a renaming declaration should statically match the subtype of the renamed object. Similarly, the profile of a subprogram renaming declaration should be subtype-conformant with the profile of the renamed subprogram. A subprogram can be renamed in order to furnish new default parameter values. A renaming-as-body can be used in cases where a subprogram can be implemented directly by calling some other subprogram.

Nesting
The textual inclusion of units inside package specifications and private part of library units provides information containment without sacrificing accessibility of the module for testing. Nested units in the private part can be accessed by child packages of the parent unit.

Packages should be library-level units, visible sub-units or child library packages. Where standards demand unit testing of all subprograms it may be necessary to avoid the declaration of subprograms locally within other subprograms. Accessibility for unit testing can also be achieved, without the need to compromise program structure, by making local subprograms subunits and placing their source in a separate file.

6.4 Expressions

An expression is a formula defining how a value is to be calculated. The value of an expression is determined by evaluation of the formula using the current values of the operands that appear in the formula. Operands are any of names, literals, function calls, allocators, type conversion, qualified expressions, or aggregates.

Every expression has a fixed type, and this is the type of the value resulting from its evaluation. The type is determined either from the types of the operands within the expression or from the context of the expression. In an imperative language like Ada expressions are readily available to review and analysis. This facility is supported by the strong typing of expression results.

A static expression cannot change in value and can be determined at compile time. Static expressions are useful to capture abstract constants, eg pi, in a way which provides reviewability and eliminates much runtime complexity.

6.4.1 Evaluation

Feature	FA	SA	RC	SU	TA	OMU	OCA	RT	ST
Operators with composite operands[1]	Inc	Inc	Inc	Inc	Inc	Inc	Inc	Inc	Inc
Logical Operators	Inc	Inc	Inc	Inc	Inc	Inc	Inc	Inc	Inc[2]
Short-circuit control forms	Inc	Inc	Inc	Inc	Inc	Inc	Inc	Inc	Inc[2]
Relational operators	Inc	Inc	Inc	Inc	Inc	Inc	Inc	Inc	Inc
Membership Tests	Inc	Inc	Inc	Inc	Inc	Inc	Inc	Inc	Inc
Slices	Inc	**Alld**[3]	Inc	Inc	**Alld**[3]	Inc	**Alld**[3]	Inc	Inc
Qualified Expressions	Inc	Inc	Inc	Inc	Inc	Inc	Inc	Inc	Inc
Aggregates[4]	Inc[5]	Inc	Inc	Inc	Inc	Inc	Inc	Inc	Inc
Static Exp.	Inc	Inc	Inc	Inc	Inc	Inc	Inc	Inc	Inc
Type Conversion Simple	Inc	Inc	Inc	Inc	Inc	Inc	Inc	Inc	Inc
Type Conversion other	Inc	**Alld**[5]	Inc	Inc	Inc[5]	Inc	**Alld**[5]	Inc	Inc
Indexing	Inc	Inc	Inc	Inc	Inc	Inc	Inc	Inc	Inc

Table 6 : Expressions

6.4.2 Notes

1. If one or more of the operands is a composite object then the compiler may insert loops into the object code that are not in the source code. Examples where this may occur are relational operations on arrays and logic operations on boolean arrays. If the size of these objects is not static then the timing analysis must be based on the worst case (which is probably, but not necessarily, when the objects have the largest possible size).

2. When testing decisions with multiple conditions, the value of every condition will need to be considered. If the short circuit forms are used then the values of some conditions will be insignificant in some test cases and need not be considered in those test cases. Achieving full coverage of compound decisions is more difficult when the conditions within the decision are not independent. If short circuit forms are used then the effect of dependencies between conditions is reduced.

3. The bounds of the range that defines the slice must be of the type of the index of the array. Two or more slices of the same array may overlap (so that components of the array appear in more than one slice). This may make understanding of the code more difficult, although the language definition ensures that the effects of overlapping slices are well defined. Slices will introduce loops into the object code that are not visible in the source and this may make timing analysis more difficult. Slices of packed arrays will introduce further timing complexity. Slices defined by ranges that are static subtypes will be simpler to analyse.

4. If an aggregate is to be assigned to an object and the aggregate references that object then the object code will be more complex than it might otherwise be:

   - The enforced use of a temporary makes the code more obscure and compiler dependent.
   - The compiler may require heap space for the temporary memory but heap management algorithms may not be present.
   - Data values are much more difficult to trace, as they are transferred through the hidden temporary.

5. There is a wide variety of type conversions for which the underlying system must be shown to implement the proper behaviour of the program. In particular conversions of composite and non-static objects generate additional code that must be traced back to source code. Conversion of composite objects usually requires a temporary object, heap storage, and unpredictable timing. View conversions (e.g. between tagged types, or applied to actual in out and out

parameters) requires dynamic checks as does conversion between generalised access types. Conversion between access types with different storage pools is potentially erroneous.

6.4.3 Guidance

For multi-termed expressions it is advisable to constrain the compiler's actions by fully parenthesising the expression. The programmer should have a clear view of the required order of evaluation of subexpressions and should enforce this view. This reduction in non-determinacy will significantly simplify verification and improve readability. For example, proof of exception free evaluation is obviously easier if there is only one order of evaluation.

Aggregates should be used as actual parameters and function return values in preference to temporary variables, and as the right-hand side of assignments to composite objects in preference to separate assignments to individual components. This makes it easier to verify that all individual components have been set, makes explicit any dependencies on the composite object assigned to, and makes maintenance more robust should the type definition change. Aggregates used to initialise objects should be static wherever possible, to avoid the need to build a temporary object (this is a specific implementation permission). Named aggregates should be preferred over positional except for large, heterogeneous, static arrays, and explicitly named choices should be preferred over '`others`' (viz '`1..n => ...`'). The former is more likely to require compiler-generated code to fill holes and the latter to require dynamic storage.

When two types are not readily convertible, or fall into one of the cases noted below, the conversion should be coded explicitly, with look-up tables, function calls or case statements:

- composite objects;
- tagged and indefinite types;
- in out or out parameters in a procedure call;
- access and generalised access types.

Complete test coverage may be difficult to achieve for expressions that create implicit loops in the object code (e.g. aggregates, expressions with composite operands). For example it may not be possible to create operands for which a loop is executed zero times, and this omission will need justification.

Be aware of the expressions which may create implicit loops in the object code as complete test coverage may be difficult to achieve. For example it may not be possible to create operands for which a loop is executed zero times - which will then need justification.

6.5 Statements

Statements are the basic commands, both simple and compound, that make up the Ada programming language. This section is concerned only with those statements that are not specific to a particular area of the language. For example, the Accept statement is not considered as it relates only to Tasking (section 6.13).

6.5.1 Evaluation

Feature	**FA**	**SA**	**RC**	**SU**	**TA**	**OMU**	**OCA**	**RT**	**ST**
Null	Inc	Inc	Inc	Inc	Inc	Inc	Inc	Inc	Inc
Assignment	Inc	Inc	Inc	Inc	Inc[1]	Inc	Inc[1]	Inc	Inc
Block	Inc	Inc	Inc	Inc	Inc	Inc	Inc	Inc	Inc
Goto[2]	**Exc[2]**	**Exc[2]**	Inc	Inc	**Exc[2]**	Inc	Inc	Inc	**Exc[2]**
For loop	Inc	Inc	Inc	Inc	Inc[3]	Inc	Inc	Inc	Inc
While loop	Inc	Inc	Inc	Inc	**Alld[5]**	Inc	**Alld[4]**	Inc	**Alld[4]**
Simple loop with exit	Inc	Inc	Inc	Inc	**Alld[5]**	Inc	Inc	Inc	Inc
Case	Inc	Inc	Inc	Inc	Inc	Inc	Inc	Inc	Inc
If	Inc	Inc	Inc	Inc	Inc	Inc	Inc	Inc	Inc

Table 7 : Statements

6.5.2 Notes

1. There may be an impact on timing analysis when assignment statements are between non-scalar values, e.g. when assigning values to arrays. This impact should be predictable, but must be considered when the analysis takes place.

2. The Goto statement is exceptional because its use is contrary to all principles of structured programming. There are no circumstances in which Goto can be used where the use of some other construct is not preferable on grounds of good practice, readability, and aesthetics. Given this, the use of Goto within high integrity systems is almost not an issue, and the reasons for not using it based on the applicability of the set of analysis techniques (although compelling) are almost irrelevant.
3. The timing analysis of a For loop gives precise results if the range through which the iteration takes place is statically determinable. The analysis is more difficult, and the results less useful, if the range is dependent on one or more variables. Timing analysis is complicated significantly by the use of an Exit statement within a For loop because the exact behaviour of the loop is harder to predict.
4. When generating the object code for a While loop, a compiler may generate more than one test for exit from the loop i.e. a single statement in source code may map to multiple pieces of object code. This makes Object Code Analysis more difficult, and also makes it harder to ensure that all branches and statements have been fully exercised when testing Structural Coverage.
5. Timing analysis is difficult for While loops, and any other loops where a condition is required to be satisfied before exit. It is not usually possible to determine accurately how many times the loop will be traversed before the exit condition is satisfied. The use of annotations to capture maximum loop counts is recommended.

6.5.3 Guidance

The use of statements of the type described in this section is fundamental to any structured programming language; these are the basic tools that build the underlying program structure. They allow the use of loops and conditional branching, essential features of the vast majority of meaningful programs. Because of their fundamental nature, the behaviour of most of these constructs is well-defined. That does not, however, necessarily mean that it is always absolutely predictable. For this reason, the ease of applying a specific analysis technique to a type of statement may depend on how that statement is used.

Apart from Goto, and to a small extent loops, there is no need to restrict the use of these basic statements (unless there are difficult-to-resolve timing issues), and to do so would place a great burden on the programmer. There is a need for caution, however, and good programming style should be used at all times. Loops, Case, and If, in particular determine the main structure of a program (or subprogram), and using these in an effective, well-structured, manner can make the whole analysis and testing process much simpler to perform.

Functional Correctness is readily applicable to nested loops only if the following are true: an exit statement must only transfer control to the level above, and the exit condition must be tested on each iteration.

6.6 Subprograms

Procedures are a basic unit of abstraction for statements and are an essential element of any imperative programming language. Ada subprogram specifications allow the mode and subtype of each parameter to be specified, allowing both compile-time type checks and run-time constraint checks on parameters in a call. Ada's compilation environment (library) requirements, and strong type checking eliminate most forms of incorrect invocation of a subprogram.

6.6.1 Evaluation

Feature	FA	SA	RC	SU	TA	OMU	OCA	RT	ST
Procedures	Inc	Inc	Inc	Inc	Inc	Inc	Inc	Inc	Inc
Functions	Inc	Inc[1]	Inc	Inc	Inc	Inc	Inc	Inc	Inc
Default Expression	Inc	**Alld[2]**	Inc	Inc	Inc	Inc	**Alld[2]**	Inc	Inc
Indefinite Formal Parameters	Inc	Inc	Inc	**Alld[3]**	**Alld[3]**	**Alld[3]**	Inc	Inc	Inc
Indefinite Return types	Inc	Inc	Inc	**Exc[4]**	**Exc[4]**	**Exc[4]**	Inc	Inc	Inc
Inline Expansion	Inc	Inc	Inc	Inc	Inc	Inc	**Alld[5]**	Inc	**Alld[5]**
Return in Procedures	**Alld[6]**	**Alld[6]**	Inc	Inc	Inc	Inc	**Alld[6]**	Inc	Inc
Aliasing	**Exc[7]**	**Exc[7]**	Inc	Inc	Inc	Inc	Inc	Inc	**Alld[7]**
Access Parameters	**Alld[8]**	**Alld[8]**	Inc	**Alld[9]**	Inc	**Alld[9]**	Inc	Inc	Inc

Table 8 : Subprograms

6.6.2 Notes

1. If functions have side effects, order of evaluation can become a significant issue, making formal analysis difficult.
2. The use of default expressions for some parameters is a convenience, but comes at a cost. Implementations may differ in how they support default parameters, making time and space analysis more difficult. The traceability of source code to object code is more difficult and variation between implementations is more marked if the default expression is not static. If default expressions are allowed, then the above issues could be addressed by a tool or code reviews.
3. Procedures and functions can have parameters of indefinite types. This may lead to a requirement for dynamic storage.
4. Functions can have indefinite return types. The storage for this object will need to be allocated dynamically.
5. Inline expansion of a subprogram call can be used to eliminate parameter passing overhead and may reduce the execution time of a program. Code size, however, can be increased, and the tracing of object code to source code can be more difficult.
6. Return statements can make the natural flow of control more apparent but returns from deeply nested structures can be obscure and cause difficulties for flow analysis, object code analysis etc. Only allowing returns at the outermost scope is an effective restriction.
7. When parameters are aliased (to non-locals or other parameters) then program proof based upon substitution will be incorrect. Similarly, informal reasoning can easily be in error. Hence if Formal Code Verification or symbolic execution is being used as an analysis technique, the absence of aliasing is required. The absence of aliasing should be determined by the use of tools and code reviews.
8. **{Aliasing}**
9. **{Accessibility checks}**

6.6.3 Guidance

The following restrictions on parameters are advised to eliminate the problems caused by indefinite formal parameters: no catenation of one-dimensional arrays and no discriminated records. Subtype conversions in subprogram calls must also be used with caution.

Static analysis must be used to ensure parameters with ‘out’ mode are assigned a value in all execution paths, and return statements are always encountered in functions. Side effects from parameter evaluation should always be avoided. Similarly, it is usually advisable to prevent functions from having side effects (although there are some circumstances in which side effects are unavoidable, but these must be verified separately). The use of non-local variables should always be documented.

Recursion, and mutual recursion, is usually prohibited for flow analysis, stack analysis and timing analysis. Direct recursion is easy to detect. Mutual recursion is harder to detect in general; however, if pragma `Elaborate_Body` is applied to all library units, and tagged types and generic units are not used, then mutual recursion can occur only between subprograms in a single library unit. This is more readily analysed. Pragma `Restriction (No_Recursion)` does not prevent recursion from occurring, and if recursion does occur the execution is erroneous. Implementations might check for violations of the restriction, and may generate somewhat simpler code.

Overloading can make the program reader's job more difficult. The advice of the Ada Quality and Style guide seems reasonable: use overloading judiciously, for widely used subprograms that perform similar actions on arguments of different types, and preserve the conventional meaning of overloaded operators.

Unconstrained types should be excluded as return types. This includes discriminated records, class wide types, and the concatenation operator ‘&’.

6.7 Packages (child, and libraries)

Packages are Ada's basic unit of modularity. Therefore packages are fundamental to the creation of any Ada program. Packages allow the partitioning of a program into parts that interact using well-defined interfaces. This can facilitate the analysis of a program by limiting the interactions between its parts.

High integrity programs can sometimes be structured so that the code that deals directly with some critical aspect of the system can be encapsulated in a package body. This is ideal, as Ada's language rules then guarantee that this code is called only using the interface defined in the package specification. Furthermore, any local data used in this critical code is protected from tampering.

Packages define three different levels of isolation. Entities declared in the public part of a package specification are visible wherever the package itself is visible. Entities declared in the private part are visible to the package body, and also within any child packages (if the package is a library unit). Finally, entities declared in the body are visible only within the body. These levels of isolation permit designers and implementers to implement stand-alone service packages, subsystems of cooperating packages, or packages that export all significant items. Subsystems can therefore be built with exactly the proper amount of visibility and security for the systems being designed. Issues to do with nested packages are considered in Section 6.3.

Child library packages provide a powerful mechanism for building subsystems. Because it is closely related to subunits there is little compiler impact, but the most effective ways of using this capability are still under consideration.

6.7.1 Evaluation

Feature	**FA**	**SA**	**RC**	**ST**	**TM**	**OMU**	**OCA**	**RT**	**ST**
Specifications	Inc	Inc	Inc	Inc	Inc	Inc	Inc	Inc	Inc
Bodies	Inc	Inc	Inc	Inc	Inc	Inc	Inc	Inc	Inc
Initialisation	Inc[1]	Inc	Inc	Inc	Inc	Inc	Inc[1]	Inc	Inc
With Clause	Inc	Inc	Inc	Inc	Inc	Inc	Inc	Inc	Inc
Private[2]	Inc	Inc	Inc	Inc	Inc	Inc	Inc	Inc	Inc[2]
Use Clause [3]	Inc	Inc	Inc	Inc	Inc	Inc	**Alld[3]**	Inc	Inc
Use Type [3]	Inc	Inc	Inc	Inc	Inc	Inc	**Alld[3]**	Inc	Inc
Child Private	Inc	Inc[4]	Inc	Inc	Inc	Inc	Inc	Inc	Inc
Child Public	Inc	Inc[5]	Inc	Inc	Inc	Inc	Inc	Inc	Inc
Separate	Inc	Inc	Inc	Inc	Inc	Inc	Inc	Inc	Inc
Elaborate Body	Inc	Inc	Inc	Inc	Inc	Inc	Inc	Inc	Inc

Table 9 : Packages (child, and libraries)

6.7.2 Notes

1. Ratings apply when the initialisation code in the package body accesses and updates objects and subprograms declared within the same package. Access to non-local objects and subprograms cause difficulties, especially with elaboration order.

2. Operations and objects defined in private parts cannot be explicitly verified from units having visibility only to the specification.
3. The use clause brings excessive overloading and possibly hiding of names. Human understanding and code review are made difficult. Use type clauses only bring into view the operations and literals of the type named in the clause. This provides a constrained view of the operations on a type.
4. Child library units export new interfaces that permit access to the private part of the parent. This view is not exported, but may not have been considered when the parent was written.
5. The ratings assume that child library units do not 'with' units that 'with' any parent units. Child library units can create dependencies on units that are dependent on the parent. Multiple views of a type and its primitive operations may be created. Ada language rules are designed to make only one path legal, but there may be confusion on the part of human designers and coders.

6.7.3 Guidance

Any complex initialisations can be done by procedures or assignments instead of during elaboration. The use of such initialising procedures for library-level packages may be preferred, as long as they are called only after the main subprogram has commenced, since elaboration of the package body variables and subprograms have all occurred, but it must still be shown that the call to the initialisation subprogram occurs before any use of the package - a non-trivial exercise, especially if any concurrency exists in the program.

The type defined private in Ada has at least two views - the public partial view and the complete private view. Language rules permit some differences between these definitions, such as adding or removing discriminants, adding limited, tagged, or aliased in the full view, and placing primitive subprograms in the private part. It is recommended that derivations from a type that has a partial view only occur where the full view (and hence all primitive subprograms) is visible, i.e. in the private part, in the private part of child packages, in private child packages and in package bodies.

Use Clauses
Most large software development projects and high integrity software development projects place significant restrictions on use clauses. The 'use type' clause and the renames clause provide alternatives that make primitive operations available without the wholesale import of another package's name space.

Child packages
Child packages permit the aggregation of packages into hierarchies of packages, and allow a subsystem to be extended without forcing the basic definition of the system (in the parent unit) to be modified or recompiled. Tools that analyse coverage must take into consideration all of the ways that child packages can be included in a program.

Child packages should not 'with' units that may 'with' the parent, and should not derive or extend types that have been declared in the private type of a parent unit.

6.8 Arithmetic Types

The ability in Ada, in contrast to most other languages, to define types and subtypes with static bounds substantially aids the review, analysis and verification process. For instance, simple tools can often affirm that uses of integer variables cannot raise an exception, say in an array indexing operation.

6.8.1 Evaluation

Feature	FA	SA	RC	SU	TA	OMU	OCA	RT	ST
Integer types	Inc	Inc	Inc	Inc	Inc	Inc	Inc	Inc	Inc
Modular types	Inc	**Alld**[1]	**Alld**[1]	Inc	Inc	Inc	Inc	Inc	Inc
Float types	Inc	**Alld**[10]	**Alld**[2]	Inc	**Alld**[7]	Inc	Inc[4]	**Alld**[10]	Inc
Fixed types	Inc	**Alld**[9, 10]	**Alld**[2,5]	Inc	**Alld**[8]	Inc	**Alld**[3,8]	**Alld**[5,10]	Inc
Dec/fixed types	Inc	**Alld**[6]	Inc	Inc	**Alld**[6]	Inc	**Alld**[6]	**Alld**[6]	Inc

Table 10: Arithmetic Types

6.8.2 Notes

1. The predefined operations on modular integer types are not those of classical mathematics. In consequence, care is needed to ensure that the operations perform the intended function. The same reservations apply to the use of the Modulus attribute.
2. If the program is required to be exception-free, then an analysis is required of the range of values that variables can have to demonstrate that each numeric operation will indeed be exception-free. This work is typically labour-intensive and is often more difficult with fixed point since the range of values is more restricted.
3. Many fixed point operations require run-time support from the compiler (for instance re-scaling). This support is context dependent and can be quite complex. The effort in showing the compiler does handle every operation correctly is significant and has led some developments to use floating point instead. Errors have been observed in validated compilers in this area, but special tests are available to check implementations.
4. Floating point requires complex (hardware/software) support. Such complexity may require specific validation measures.
5. The rounding of some fixed point operations is not defined in the language and the implementation need not provide any information on the actual rounding performed. This implies that the rounding could be context-dependent or even change with a new release of the compiler.
6. Support for decimal fixed point is typically only provided by compilers supporting the Information Systems Annex (Annex F). The predefined operations on decimal fixed point types that do not give an exact result are defined to truncate. In consequence, it may be easier to validate program which uses decimal fixed point types than those which use ordinary fixed point types. In other respects, the validation issues are the same as for ordinary fixed point types.
7. The timing of floating point operations are typically data-dependent. Hence computing the worst-case execution time can be awkward if an analysis of the data is required.
8. Fixed point operations require compiler support for which timing and object code analysis is more complex.
9. Type conversion between fixed point types whose delta is not a power of 2 may introduce addition rounding errors. Hence such types should be avoided. Placing a representation attribute clause for Small to match the delta is recommended for all fixed point types.
10. Due to the potential machine-dependence of the rounding of real operations, the computation of boundary value test cases can be awkward (see Equivalence Class Testing). Similarly, the rounding presents problems with Symbolic Analysis.

6.8.3 Guidance

As noted in 6.2.3, named numbers should be used when the application permits compile-time evaluation.

Continuously varying quantities should be modelled by means of Ada real types. It is not usually appropriate to use integer types since there is no simple way of multiplying integer values and re-scaling the result. Hence an important design decision is to use either ordinary fixed point types or floating point types (or less likely, both).

The inherent problem with real types is that rounding is performed which introduces a degree of implementation defined behaviour into the program which does not occur with integer types.

Analysis of real expressions requires analysis of the ranges within which the result of an expression is guaranteed to lie. A floating point expression has a value in a range which has a size relative to the value of the expression and a fixed point expression has a value in a range which has an absolute size.

When using real arithmetic:

1. Be aware of whether the implementation supports the Numerics Annex (Annex G) as the accuracy of real arithmetic is guaranteed only for implementations that support this annex.
2. For maximum portability of code, only use implementations that support Annex G, and ensure that the strict mode is used.
3. For implementations that support Annex G, the accuracy of the predefined operators is defined in terms of model numbers of the relevant type. Although the set of model numbers is implementation defined, it is straightforward for a user to define a set of model numbers that will be provided by all implementations of interest (provided only they have a common radix for number representation). By analysing the accuracy of real arithmetic in terms of this set of model numbers all implementations are guaranteed to provide the analysed accuracy or better.

4. If future implementations are unknown then the characteristics assumed by the accuracy analysis should be recorded and it is then easy to determine whether a new implementation conforms (and so whether the previous analysis still applies or needs to be repeated).
5. Currently, most of the major processor chips directly support floating point. The older chips do not, and neither do some specialised chips. Hence in some cases, the use of floating point is not viable due to the lack of hardware support and because software support is too slow. If the systems design would allow for either floating point or fixed point, then the choice is critical since it influences the coding, the testing and the qualification of the compiler. If it is decided not to use floating point, then the pragma Restrictions(No_Floating_Point) can be used to enforce no explicit use. However, the compiler may still make implicit use of floating point (say, for complex numeric conversions), which may require compiler options to remove.

Not specifically listed in the above table is the use of attributes. There are numerous numeric attributes which can conveniently be divided into two classes:

- Those whose use provides no special problems:

```
Adjacent,           Aft,                Ceiling,
Compose,            Copy_Sign,          Delta,
Denorm,             Digits,             Exponent,
Floor,              Fore,               Fraction,
Leading_Part,       Round,              Rounding,
Scale,              Small,              Truncation,
Unbiased_Rounding.
```

- Those which are low-level:

```
Machine,            Machine_Emax,       Machine_Emin,
Machine_Mantissa,   Machine_Overflows,  Machine_Radix,
Machine_Rounds,     Model,              Model_Emin,
Model_Epsilon,      Model_Mantissa,     Model_Small,
Remainder,          Safe_First,         Safe_Last,
Scaling,            Signed_Zeros.
```

These attributes must be used with care to ensure portability.

6.9 Low Level and Interfacing

Low level constructs and interfacing exist to allow an Ada program to interact with:

- elements of the machine, e.g. memory addresses;
- other hardware elements of the system, e.g. tape and screen devices;
- other software elements of the system, e.g. databases, GUI; and
- other languages, e.g. C, FORTRAN, Machine Code.

Ada is very useful in providing a mechanism for interchanging information between different languages. It is also useful in allowing these mechanisms to be performed in a very constrained way. In particular all of the low level features described in the following table must be encapsulated in program units that clearly isolate their behaviour from the rest of the program. The ability of Ada to support this encapsulation and isolation is a key advantage obtained from the use of Ada.

The use of low level constructs and interfacing can be seen to be a mechanism for dealing with elements outside of the Ada system under consideration.

6.9.1 Evaluation

Feature	**FA**	**SA**	**RC**	**SU**	**TA**	**OMU**	**OCA**	**RT**	**ST**
Package System	**Alld**[4]	**Alld**[4]	**Alld**[4]	**Alld**[4]	**Alld**[4]	**Alld**[4]	**Alld**[4]	**Alld**[4]	**Alld**[4]
Machine Insertion	**Alld**[5]	**Alld**[5]	**Alld**[5]	**Alld**[5]	**Alld**[5]	**Alld**[5]	Inc	**Alld**[5]	**Alld**[5]
Unchecked Access	**Exc**[8]	**Exc**[8]	**Exc**[8]	**Exc**[8]	**Alld**[8]	**Exc**[8]	**Alld**[8]	**Exc**[8]	**Exc**[8]
Unchecked Conversion.	**Alld**[7]	**Alld**[6]	**Alld**[6]	Inc	Inc	Inc	Inc	**Exc**[6]	Inc
Interfacing Pragmas	**Alld**[10]	**Alld**[10]	**Alld**[10]	Inc	**Alld**[10]	**Alld**[10]	**Alld**[10,11]	**Alld**[10]	**Alld**[10]
Representation Clause	Inc	**Alld**[1]	Inc	**Alld**[1,2]	**Alld**[2]	Inc	**Alld**[2]	**Alld**[2]	**Alld**[2]
Representation Attributes	**Alld**[3]	**Alld**[3]	**Alld**[3]	**Alld**[3]	**Alld**[3]	**Alld**[3]	**Alld**[3]	**Alld**[3]	**Alld**[3]
Streams	**Exc**[9]	**Exc**[9]	**Exc**[9]	**Exc**[9]	**Exc**[9]	**Exc**[9]	**Exc**[9]	**Exc**[9]	**Exc**[9]
Address Clause [12]	Inc	Inc	Inc	Inc	Inc	**Alld**[13]	**Alld**[14]	Inc	Inc

Table 11 : Low level and interfacing

6.9.2 Notes

1. Representation clauses can have interactions with other features that make their semantics, storage use, or timing properties difficult to predict.
2. Representation clauses may lead to object code for initialisation or unpacking. Also, such clauses can require that the compiler emits very different machine code than is usually the case, leading to difficulties in tracing from source to object code and a risk of incorrect object code.
3. Considered as expressions (i.e., not when specified in a representation clause). These attributes are then (typically) implementation-dependent compile-time constants.
4. Package system, including its children, is compiler-dependent. The facilities it provides should be evaluated by comparison with equivalent features described elsewhere in Section 6. For example, constants exported by package system are included (see Section 6.2.1).
5. Machine code is not constrained in its effect and must, therefore, be used with extreme care and encapsulated in the smallest unit possible.
6. An unchecked conversion gives an interpretation of a representation in a new (and possibly unrelated) type. The language rules do not specify what this interpretation will be (or whether it is even valid).
7. Some flow information can be lost (if flow information is retained for components of a composite.)
8. Unchecked access value creation, and subsequent use of the access value, can lead to dangling references or corruption of data.
9. Streams require class wide types and access parameters, and are therefore difficult to analyse.
10. The effect of an imported subprogram needs to be documented in such a way that its callers can be analysed.
11. There may be object code generated to account for differences in calling convention.
12. Address clauses should not be used to introduce aliasing by overlapping or superimposing variables in memory.
13. Analysis of memory use must consider the impact of the location of objects at specific addresses.
14. Object Code Analysis must ensure that the specified address of a variable is directly generated for all accesses of that variable and that the variable has not been optimised to a register, for example. Use of pragma volatile may be appropriate.

6.9.3 Guidance

If the system has a defined integrity level and the Ada code in the system is required to interact with some other element, then assurances should be obtained that the other element has the same integrity as the Ada code itself. These guidelines consider COTS software to be another element that ranges from complete software packages to code segments that have not explicitly been created for the current system. The latter end of this range could be considered as re-use.

If ‘allowed’ low-level features are used, it is recommended that their use be encapsulated within a small package body. This reduces the possibility of an interaction with some other feature used in a different part of a program, and can facilitate analysis.

Representation clauses on enumeration types should be avoided, except when confined to a unit body to interface to foreign systems. A type with a representation clause should not be a derived type, nor used as a source of a type derivation, nor should it be used as a loop parameter, case expression, array index, in a 'Succ or 'Pred clause, or in a type conversion.

If a record definition is subject to representation clauses, such as packing, alignment, layout or ordering clauses, then the declaration should be restricted to a compilation unit body. Care is required to ensure that there is not a conflict between representations applied to the record, components of the record, and objects of the record (example: ‘pragma pack’ and ‘‘aliased’ may be mutually incompatible). Assignment to or from such a record should be component-by-component to avoid timing issues and unanalysed object code.

Representation Clauses
Such clauses in general change the machine-code emitted by the compiler. They should be used solely to enable such changes and not to ‘confirm’ an existing representation. An enumeration representation clause with non-contiguous values will cause the code for many operations on the enumerated type to be obscure and hard to relate to the source text; hence such operations should be avoided. Similarly, record representation clauses can make operations such as assignment, equality and conversion non-trivial and hence every operation should be used sparingly and in a localised fashion.

6.10 Generics

Generics provide a powerful mechanism for constructing large-scale programs through the parameterisation of program units (packages and subprograms) with types, objects and subprograms. Abstract algorithms and data types can thereby be constructed in terms of parametric types and operations, specifying only as much as necessary. A generic instantiated with different sets of parameters yields different program units that share the same algorithmic structure.

Generic units can be used to:

- define abstract data types;
- parameterise a procedure by another procedure (e.g. a loop iterator);
- replicate a unit.

The first is by far the commonest and most useful.

Generics can appear at library level or nested within other units. Generic units can be children or parameters to other generics, enabling the construction of whole parametric software subsystems.

Use of generics enhances program reliability in several ways. It facilitates reuse, eases maintenance, reduces source code size and helps avoid human replication error. In principle it also assists with static analysis and testing: if a generic algorithm is verified once, all instances of the generic can be considered automatically verified. In practise, the complexity of the instantiation process mitigates against this ideal.

Analysability of a generic feature depends not only on the semantics and behaviour of the feature, but also on how it is compiled: whether by macro-expansion or code-sharing. In general shared code is highly parametric, because small changes to actual parameters make dramatic differences to efficiency (e.g. composite vs. elementary types). Code coverage is difficult to achieve, because there are rarely sufficient instances to test all options. The complexity of mapping from the generic unit directly to object code impedes validation as well as robustness and reliability. In practice a majority of compilers adopt macro-expansion, or a hybrid where only simple cases share code. It is recommended that generics compiled by code-sharing are excluded for high integrity systems. The following analysis assumes this recommendation.

Analysis of a generic feature might be undertaken on the generic unit, or on its instances. The former offers the possibility of ‘once-and-for-all’ verification: if the generic is shown to have some property, then that property is inherited by all of its instances. In principle it is possible to perform some analyses, notably Symbolic and Flow Analysis, on a generic unit alone, although tool support is presently weak. This is a corollary of Ada's contract model, which states that the formal specification contains sufficient information to determine the legality of the generic body, whatever actual parameters are supplied. Verification conditions at the point of instantiation must also be satisfied. This might involve work, or the imposition of minor restrictions, e.g. to enforce range constraints on formal parameters.

Alternatively, the analyses can be performed on the individual instances. Other techniques (Object-Code Analysis, testing) must be applied this way. Success criteria might be specified individually for each instance, on the generic itself and derived ('instantiated') for the instances, or both (provided consistency is maintained). Annotations and test points needed to express the criteria can be attached to the source code, a compiler intermediate form such as ASIS (where one exists), or onto the object code via a debugger. This last is highly effective for dynamic testing and has been used with success in high-integrity systems. It is presently unclear what static analysis annotations might apply to generics or instances, or how to attach them; this presents a challenge to tool builders. However there is no reason to believe that static analysis techniques cannot evolve to incorporate generics in some form.

Analysis of instances offers more immediate potential than analysis of whole generic units. Although it has to be repeated for each individual instance, the tool requirements are much lighter and the complex step of validating the instantiation is avoided.

6.10.1 Evaluation

The table below assumes a macro-expansion implementation, with analysis tools acting on an intermediate expanded form.

Feature	FA	SA	RC	SU	TA	OMU	OCA	RT	ST
generic units (packages & subprograms) [12]	**Alld**[1]	**Alld**[1]	Inc	Inc	Inc	Inc	**Alld**[2]	Inc	Inc
formal subprograms	**Alld**[3]	**Alld**[3]	**Alld**[4]	Inc	Inc	Inc	**Alld**[2]	Inc	Inc
in objects	Inc	Inc	Inc	**Alld**[5]	Inc	Inc	**Alld**[2]	Inc	Inc
in out objects	**Alld**[3]	**Alld**[3]	**Exc**[6]	Inc	Inc	Inc	**Alld**[6]	Inc	Inc
type parameters[7]	Inc	Inc	Inc	Inc	Inc	Inc	**Alld**[7]	Inc	Inc
default parameters	Inc	Inc	Inc	Inc	Inc	Inc	**Alld**[8]	Inc	Inc
formal packages	**Alld**[9]	**Alld**[9]	**Alld**[9]	**Alld**[9]	**Alld**[9]	**Alld**[9]	**Alld**[9]	**Alld**[9]	**Alld**[9]
generic children	**Alld**[9]	**Alld**[9]	**Alld**[9]	**Alld**[9]	**Alld**[9]	**Alld**[9]	**Alld**[9]	**Alld**[9]	**Alld**[9]
non-library and nested generic units	**Exc**[10]	Inc	Inc	Inc	Inc	Inc	**Alld**[2]	Inc	**Exc**[11]
non-library and nested instances	Inc	Inc	Inc	Inc	Inc	Inc	Inc	**Exc**[11]	**Exc**[11]

Table 12: Generics

6.10.2 Notes

1. Tool support is not available at the present time. Potential name capture problems when an actual parameter shares a name with a local of the generic unit.

2. The mapping between source and object code is complex for generics. In particular, implicit operations to perform assignment, equality, constraint checking, etc., are not obvious in the source code and obstruct understanding. This comment applies to object-code analysis of all aspects of generics.

3. Subtype constraints on generic formal `in out` objects and on parameters to generic formal subprograms are disregarded. This means that constraint exceptions cannot be predicted looking at the generic alone.

4. Subtype constraints on parameters to formal subprograms are disregarded: those pertaining to the actual are applied at run time. To simplify the semantics, formal subprograms should statically match the actual.

5. An instance of a generic in object declares a new object, initialised by the actual value, in the unit containing the instantiation, and the stack grows there. This may cause a problem, particularly at library level, where no such unit is readily identifiable.

6. An instance of a generic in out object is an alias of the actual object, with constraints those of the actual, not the formal. The alias complicates the code mapping.

7. `type parameters' covers all kinds of type: private, limited private, scalar, derived, tagged, array and access. None are

excluded, but in general the more closely-defined the type, the harder it is to relate source and object code, because more primitive operations occur. If the formal is a derived type, re-export of new derivations from the generic causes confusion between primitive operations.

8. When binding defaulted formal parameters, different visibility rules apply if the box <> version is used. The mechanism is prone to human error.

9. Formal packages and generic children provide a very powerful mechanism for encapsulating software components and subsystems for reuse, thereby increasing the reliability and maintainability of software. They are high level features: all semantic issues are resolved at instantiation and there is no impact on code generation. Therefore it seems unlikely that they will impede analyses at the instance level. However there is presently little field experience in their use and compilers have not been exercised in this area. Therefore they cannot be recommended for high-integrity Ada at the present time.

10. Flowcharts cannot be assembled for nested units.

11. Non-library generics or instances cannot be breakpointed on a per-instance basis.

12. {Note based on Tucker Tafts comments about restrictions to be written by BAW}

6.10.3 Guidance

Generics offer great potential for improving program reliability and maintainability. Against this, engineering experience and support in the high-integrity field is limited. Therefore for generics should be allowed where compilers are considered reliable, users have experience, and there exist support tools appropriate to the application.

It is recommended that only macro-expanding compilers be used, and that analysis is performed on each instance in preference to the generic unit.

There are a number of small issues with generics that obstruct analysis, comprehension, validation or maintenance. These should be excluded from high-integrity systems.

Exclusions to enable analysis:

- generics compiled by code-sharing,
- generics and instances not declared at library-level,
- nested generics,
- formal in out objects.

Exclusions for reasons of human comprehension and ease of maintenance:

- formal tagged and derived types,
- default parameters to generic units,
- subprogram parameters with constraints that do not statically match the actual. In addition features which involve other excluded areas of the language, such as dynamic types, should not be available to the user.

6.11 Types with Dynamic Attributes

Unboundedness in storage is incompatible with high integrity systems since the occurrence of `Storage_Error` is unacceptable. This implies that types with dynamic attributes are either excluded or should be used with extreme care.

The collection associated with an access type, when the maximum number of objects is static, is logically equivalent to an array of the objects. However, the handling of arrays by tools and programmers is generally more mature than for collections and hence the guidance here is to prefer arrays.

The requirement for staticness has essentially excluded variant records (which require discriminants, see Section 6.1) and run-time dispatching.

{DT to add paragraph describing what ‘restricted access types’ means in the table}

6.11.1 Evaluation

Feature	FA	SA	RC	SU	TA	OMU	OCA	RT	ST
Unconstrained array types – including strings[1]	Inc	Inc	Inc	Inc	Inc	Inc	Inc	Inc	Inc
Full access types	**Exc**[2]	**Exc**[2]	Inc	**Exc**[2]	**Exc**[2]	**Exc**[2]	Inc	Inc	Inc
Restricted access types	**Alld**[4]	**Exc**[3,4]	Inc	Inc	Inc	Inc	Inc	Inc	Inc
General access types	**Alld**[4]	**Alld**[4]	Inc	Inc	Inc	Inc	Inc	Inc	Inc
Access to subprogram	**Exc**[5]	**Exc**[5]	Inc	Inc	**Alld**[5]	Inc	**Alld**[5]	Inc	Inc
Controlled types	**Exc**[6]	**Exc**[6]	Inc	Inc	Inc	Inc	**Alld**[6]	Inc	**Exc**[6]

Table 13 : Types with Dynamic Attributes

6.11.2 Notes

1. Note that the concatenate function returns a type of an unconstrained array. Refer to Section 6.4 for more information.
2. Full access types employ the run-time system to allocate from the heap and other memory areas, making memory use unpredictable. It also introduces unbounded aliasing problems.
3. Pool-specific access types use memory similarly to array-based data types. However, they are novel to Ada95 and there is no maturity in their use or analysis.
4. Pools and general access types encourage aliasing of data. See restrictions in Section 6.2.
5. Fundamentally disrupts control flow, and makes it difficult to export analysis results of subprograms into calling subprograms. This exclusion can be enforced by the pragma Restrictions(No_Access_Subprograms).
6. **{'introduces hidden control flows' note by DT in his email of 25/3/98}**

6.11.3 Guidance

As noted in the introduction to this section dynamic behaviour is to be minimised in high integrity systems. Appropriate enforcements can be provided by the use of pragma Restrictions(No_Implicit_Heap_Allocation), pragma Restrictions(No_Allocators), pragma Restrictions(No_Access_Subprograms).

Although the evaluation table has 'Inc' against nearly all the testing based verification techniques, it should be noted that the effectiveness of these techniques may well be reduced. For example, a problem arising from the inappropriate use of aliasing may well be difficult to find during Requirements-based Testing and Structure-based Testing. It is also true that code inspection techniques will be made more complex (and hence error prone) by the use of these dynamic features.

6.12 Exceptions

Ada has well-defined semantics even under error conditions. The language allows the user to detect erroneous conditions and to specify the required behaviour under such conditions at run-time.

Predefined exceptions concern erroneous situations detected by the runtime environment. The implicit raising of predefined exceptions poses some problems where high integrity software is concerned because the location in the program and the moment it occurs at runtime cannot be easily predicted.

Whereas a predefined exception is raised automatically when an associated constraint is violated, user exceptions provide the means to specify erroneous situations whose occurrences must be explicitly detected.

The exception mechanism leads the designer to an intellectual dilemma:

- its use makes the verification more difficult. So the use of exceptions must be prohibited;
- its use allows residual errors to be detected and handled. So the exception features are a key part of a language for high integrity applications [15].

Solutions to this conflict will be proposed in the section dealing with guidance.

If the use of exceptions is prohibited this does not in itself prevent predefined exceptions being raised. The use of pragma Restrictions(No_Exceptions) is recommended, but again the program will become erroneous if a run-time exception does occur.

The evaluation table below has been produced with the assumption that exceptions are going to be used.

6.12.1 Evaluation

Feature	FA	SA	RC	SU	TA	OMU	OCA	RT	ST
Declaration (user)	Inc	Inc	Inc	Inc	Inc	Inc	Inc	Inc	Inc
Raising (predefined)	**Alld[2]**	**Alld[2]**	Inc	Inc	Inc	Inc	**Alld[3]**	Inc	**Alld[4]**
Raising (user)	**Alld[5]**	**Alld[5]**	Inc	Inc	Inc	Inc	Inc	Inc	**Alld[5]**
Handler (predefined)	**Alld[6]**	**Alld[6]**	Inc	Inc[7]	Inc[8]	Inc	Inc	Inc	**Alld[9]**
Handler (user)	**Alld[5]**	**Alld[5]**	Inc	Inc	Inc	Inc	**Alld[5]**	Inc	Inc
Propagation	**Exc[10]**	**Exc[10]**	Inc	Inc	**Alld[10]**	Inc	**Alld[10]**	Inc	**Alld[10]**

Table 14 : Exception

6.12.2 Notes

1. Predefined exceptions should not be raised explicitly.
2. Symbolic Analysis and Flow Analysis are problematic, as the locations at which predefined exceptions are raised are not generally known.
3. Object Code Analysis is difficult because detection is either explicit or implicit via a hardware mechanism.
4. It is difficult to find a test sequence that will bring about the error conditions necessary to raise the predefined exception.
5. In general, it is difficult to define the execution sequence that will cause the precondition of the exception to become true.
6. The actual state provoking the branching to the handler beginning cannot be established (Symbolic Analysis). Consequently, the actual state of the data values at the raising time is not known (Flow Analysis).
7. The stack is used or not to reach the handler depending on the technique used to implement the exception mechanism: conventionally, the Static Mapping which uses a table or the Dynamic Tracking which uses the stack.
8. Timing Analysis depends on the technique used to implement the exception mechanism.
9. It was previously mentioned that it is not possible to control the program from its inputs to raise a predefined exception and thus to execute the associated handler.
10. Once an exception has been propagated, then neither Flow Analysis nor Symbolic Analysis can be undertaken and other forms of verification become difficult.

6.12.3 Guidance

In the introduction, the conflict concerning the use of the exception mechanism was highlighted. On one hand, the previous section dealing with evaluation shows the difficulties induced by this mechanism when applying verification techniques. So, its use seems to be prohibited when High Integrity Systems are concerned. On the other hand, it is not possible to say that its non-use just reduces the language expression capabilities as it occurs for the other features. Indeed, when exceptions are not used, the errors cannot be handled but their existence is not avoided [13]. To solve this issue, the proposed guidelines concern three design strategies.

A first design choice (called ‘exclusion strategy’) consists of excluding the use of exceptions. This may be obtained by means of the pragma `Restrictions(No_Exceptions)` written in the Ada programs or by using compiler options provoking the same effect. However, the absence of erroneous states usually leading to an exception raising must be demonstrated. This is easily obtained concerning potential user exceptions as their raising is explicit. The proof that no predefined exceptions can be raised is more complex, but may still be tractable (see Section 3.3.5).

The second design choice (called ‘belt-and-braces strategy’) can be used if the absence of predefined exception raising cannot be guaranteed. It consists of introducing an ‘others’ exception handler at global level which does a very simple restart/reset operation (if such an operation is compatible with the application). As signalled in the notes, the actual state of the variables is not well defined when the exception handler is reached. So, this strategy requires the implementation of a mechanism ensuring the recovery of a well-known program state at program resumption time (for instance, a ‘recovery cache mechanism’). To conclude, it should be noted that the implementation of such a strategy may be very complex when the program includes tasks. In particular, resumption of the task in which an exception raising occurs is frequently not acceptable when synchronisations exist. In this case, a more complex resumption policy must be considered to handle the phenomenon called ‘domino effect’.

The third design choice (called ‘mastered exception mechanism using’) authorises the use of exceptions in a simple way. In particular, the following guidelines must be considered to make easier the verification techniques applications. Exception mechanisms can only be used (if required) to handle errors occurring at run-time and not rare events, such as ‘end of file is reached’. Predefined exceptions should not be raised explicitly. Predefined and user exceptions must be handled close to the raising location. In particular, propagation phenomenon should be avoided. Here again, the exception handling must guarantee that the program state is well-defined.

6.13 Tasking

High-integrity systems traditionally do not make use of high-level language features such as concurrency. With Ada, these language features can be prohibited by the use of pragma Restrictions(Max_Tasks=0), pragma Restrictions(No_Protected_Types) and pragma Restrictions(No_Delay). The view that tasking should not be used is despite the fact that such systems are inherently concurrent. Concurrency is viewed as a ‘systems’ issue. It is visible during design and in the construction of the cyclic executive that implements the separate code fragments, but it is not addressed within the software production phases. Notwithstanding this approach, the existence of an extensive range of concurrency features within Ada does allow concurrency to be expressed at the language level with the resulting benefits of having a standard analysable approach that can be checked by the compiler and supported by other tools.

The requirement to analyse both the functional and temporal behaviour of high integrity systems imposes a number of restrictions on the concurrency model that can be employed. These restrictions then impact on the language features that are needed to support the model. Typical features of the concurrency model are as follows.

1. A fixed number of tasks.
2. Each task has a single invocation event, but has a potentially unbounded number of invocations. The invocation event can either be temporal (for a time-triggered task) or a signal from either another task or the environment. A high-integrity application may restrict itself to only time-triggered tasks.
3. Tasks only interact via the use of shared data. Updates to any shared data must be atomic.

These constraints furnish a model that can be implemented using fixed priority scheduling (either preemptive or non-preemptive) and analysed in a number of ways:

1. The functional behaviour of each task can be verified using the techniques appropriate for sequential code. Shared data is viewed as just environmental input when analysing a task. Timing analysis can ensure that such data is appropriately initialised and temporally valid.
2. Following the assignment of temporal attributes to each task (period, deadline, priority etc), the system-wide timing behaviour can be verified using the standard techniques in fixed priority analysis [1],[2].

To implement this concurrency model in Ada requires only a small selection of the available tasking features. At the Eighth International Real-Time Ada Workshop (1997) the following profile (called the *Ravenscar Profile*) was defined for high-integrity, efficient, real-time systems [9].

The Ravenscar Profile allows the following.

1. Task type and object declarations at the library level (that is, no hierarchy of tasks).
2. No unchecked deallocation of protected and task objects.
3. No dynamic allocation of task or protected objects (this is not part of the profile but is included here to be consistent with our overall approach to dynamic allocation - see Section 4.1.2).

4. Tasks are assumed to be non-terminating.
5. Library level protected objects with no entries (to ensure atomic updates to shared data).
6. Library level protected objects with a single entry (for invocation signalling). This entry has a barrier consisting of a single boolean variable; moreover only a single task may queue on this entry (this is a static property that is easily verified).
7. 'Real-Time' package.
8. Atomic and volatile pragmas.
9. Delay Until statements
10. Count attribute (but not within entry barriers).
11. Task identifiers.
12. Task discriminants.
13. Protected procedures as interrupt handlers.

The inclusion of protected entries allows event based scheduling to be used. For many high integrity systems only time triggered actions are employed, hence such entries and their associated interrupt handlers are not required.

The profile defines dispatching to be *FIFO within priority* with protected objects having *Ceiling Locking*. However it also allows a non-preemptive policy to be defined. Cooperative scheduling (that is, non-preemption between well defined system calls such as 'delay-until' or the call of a protected object) can reduce the cost of testing as preemption can only occur at well-defined points in the code. It can also reduce the size of the run-time.

With either dispatching policy, the Ravenscar Profile can be supported by a relatively small run-time. It is reasonable to assume that a purpose-built run-time (supporting only the profile) would be efficient and 'certifiable' (i.e. built with the evidence necessary for its use in a certified system). An equivalent run-time for a constrained Ada83 tasking model has already been used in a certified application. Static checks are possible to ensure any program conforms to the profile.

6.13.1 Evaluation

Feature	FA	SA	RC	SU	TA	OMU	OCA	RT	ST
Ravenscar Profile	**Alld**[2]	Inc[1]	Inc	Inc[3]	Inc[3]	Inc[4]	Inc	Inc	**Alld**[5]
Other Tasking Features	**Exc**[6]	**Exc**[6]	Inc	**Exc**[7]	**Exc**[7]	**Exc**[7]	**Exc**[7]	Inc	**Exc**[8]

Table 15: Tasking

6.13.2 Notes

1. Due to the lack of synchronisation between tasks, symbolic analysis can be undertaken on a per task basis.
2. Each task is analysed as if it were a separate program. Data flow and information flow between tasks is an issue but can be addressed by viewing any data that flows between tasks to be output or input for the two tasks involved. Hence existing methods of dealing with inputs from the environment are adequate. With the Ravenscar profile there is no control flow between tasks.
3. System timing behaviour and stack usage are fully predictable with this profile.
4. The static task structure precludes the use of any memory structure other than the stack.
5. With the preemptive dispatching policy, structural coverage is deemed not to include the testing of all possible preemption points. Rather, structural coverage is focused on the behaviour of each task. If this view is not appropriate then non-preemptive (and cooperative) dispatching must be employed. This explicitly defines the points at which a task can be preempted. With non-preemption, interrupt handling (for example, to deal with a regular clock interrupt) is still allowed but the interrupt is not permitted to release a higher priority task for execution. A non-interruption dispatching policy is also possible. With this policy no interrupts are used and each task release condition must be checked whenever the run-time system is invoked.
6. Where appropriate the synchronous concurrency aspects of the application may be formalised using techniques such as finite state automata [3] Petri-Nets [4],[5] or a process algebra [6],[7],[8]. Note these aspects are rarely formalised in the systems view of concurrency. They are excluded due to lack of experience in using these techniques in high integrity applications and because there remain some research questions over the scaleability of the techniques to real problems. Other tasking features such as requeue and asynchronous transfer of control have not yet been formalised.
7. If dynamic task creation is allowed then memory usage cannot be predicted and static scheduling analysis is not feasible.

8. As the behaviour of one task is more synchronously linked to the executions of other tasks, the need to directly test multi-tasking programs increases. Notions of coverage for general tasking programs are not fully defined and hence the general model is excluded for this type of verification.

6.13.3 Guidance

With the profile, each task should be structured as an infinite loop within which is a single invocation event. This is either a call to 'delay until' (for a time triggered task) or a call to a protected entry (for an event triggered task).

The use of the Ravenscar profile allows timing analysis to be extended from just the prediction of the worst-case behaviour of an activity to an accurate estimate of the worst-case behaviour of the entire system. The computational model embodied by the Ravenscar profile is very simple and straightforward. It does not include, for example, the rendezvous or the abort, and hence does not allow control flow between tasks (other than by the release of a task for execution in the event triggered model). But it does enable interfaces between activities (tasks) to be checked by the compiler.

Preemption execution, in general, leads to increased scheduleability and hence is more efficient in the use of the system's resources (e.g. CPU time). As preemption can occur at any time, it is not feasible to test all possible preemption points. Rather, it is necessary for the run-time system (RTS) to guarantee that the functional behaviour of a task will not be affected by interrupts or preemption. For a high integrity application evidence to support this guarantee would need to be provided by the compiler vendor (or RTS supplier). For the Ravenscar profile the RTS will be simple and small. There is ample expertise in the industry to be confident that high integrity Ada RTS are feasible and will be available.

Not only does the use of Ada increase the effectiveness of verification of the concurrency aspects of the application, it also facilitates a more flexible approach to the system's timing requirements. The commonly used cyclic executive approach imposes strict constraints on the range and granularity of periodic activities. The Ravenscar profile will support any range and a fine level of granularity. So, for example, tasks with periods of 50ms and 64ms can be supported together. Moreover, changes to the timing attributes of activities only require a reevaluation of the timing analysis. Cyclic executives are hard to maintain and changes can lead to complete reconstruction. Finally, note that the inclusion of a small number of event triggered activities does not fundamentally change the structure of the concurrent program or the timing analysis, but it does impose significant problems for the cyclic executive. Polling for 'events' is a common approach in high integrity systems; but if the 'event' is rare and the deadline for dealing with the event is short then the time triggered approach is very resource intensive. The event triggered approach will work with much less resources. These guidelines are not intended to imply that event triggering is better than time triggering. The point here, is that the Ravenscar profile deals with both approaches and the migration from one to the other.

Many, but not all, of the constraints defined by the profile can be enforced by use of the Restrictions pragma. For those for which the pragma does not apply, it is still a static syntax check to determine if a program complies to the profile. The only exception to this rule is the assumption that a task is non-terminating. This can not, of course, be checked but any implementation of the profile will protect itself against task termination.

6.14 Distribution

Although many high-integrity systems are distributed, it is rare for the programming activity to directly address notions of distribution. Nevertheless, the features that Ada defines to support the programming of distributed systems are important and can have a role in even single processor high-integrity systems. This is particularly true when different criticality subsystems are to be hosted on to the same processing resource; an approach that is becoming more common in a number of application areas, for example, Integrated Modular Avionics.

Ada provides a number of categorisation pragmas that allow library units to be partitioned into distinct groups that do not share variables or physical addresses. The interactions between these separate partitions are well-defined and analysable (i.e. there can be no hidden interactions between the partitions). This separation of addresses does not necessarily imply separate memory spaces as this is an implementation issue. However, if memory protection is required (to isolate specific subsystems) then Ada's partition is the obvious way of representing the necessary enclosures at the program level.

On a single processor system the only two categorisation pragmas that are needed in order to facilitate the effective use of partitions are *pure* and *remote-call-interface*. A pure package is a restricted form of *pre-elaborate* which is itself a useful category as it designates a library unit that can be elaborated without the execution of any code at run-time. In a genuine distributed system, two further pragmas are usually required: *remote-types* and *shared-passive*.

6.14.1 Evaluation

Feature	FA	SA	RC	SU	TA	OMU	OCA	RT	ST
Pre-elaborate	Inc	Inc	Inc	Inc	Inc	Inc	Inc	Inc	Inc
Pure	Inc	Inc	Inc	Inc	Inc	Inc	Inc	Inc	Inc
Remote-call-interface	Inc	Inc	Inc	Inc	Inc	Inc	Inc	Inc	Inc
Remote types	Inc	**Exc**[1]	Inc	Inc	Inc	Inc	Inc	Inc	**Exc**[1]
Shared Passive	Inc	**Exc**[1]	Inc	Inc	Inc	Inc	Inc	Inc	**Exc**[1]

Table 16: Distribution

6.14.2 Notes

1. The programming of distributed systems is however somewhat immature; the construction of distributed systems and the commensurate issues of fault tolerance and survivability introduce many systems issues that go well beyond mere programming. The analysis and testing of distributed systems is not straightforward.

6.14.3 Guidance

Partitions are a useful form of structuring and hence the three standard pragmas should be employed, where appropriate, to provide additional information about the properties of library units.

7. Compilers and Run Time Systems

It is imperative that software written for high integrity systems in a high level language be interpreted the same way by developers, reviewers and auditors. Safety related and security guidelines and standards require that the language be endorsed by a national or international standards agency. In addition, some standards may require that compilers be validated to show conformance to the language definition and that this be performed by an independent certification organisation.

7.1 Language issues

Ada compilers must undergo such validation testing, and evidence of the results of such testing is maintained by the Ada Information Clearing house as a public record.

An Ada compilation system used for safety critical systems must conform to the Ada language definition as defined in the Ada standard [ARM]. A fully conformant compiler must pass all applicable ACVC tests which exercise the compiler, linker and the run-time system together. ACVC tests will also be applied to the annexes which have been implemented which must also conform to the Ada standard. The tool selection process for a safety critical application will need to ensure that the compilation system and the annexes to be used have a validation basis.

The compiler may have options which control the level of optimisation or other code transformations which affect traceability between source and object code. As these options have a direct impact on the correctness of the object code, it is important that the compiler passes the validation suite with the options to be used when building the operational software, and that the run-time libraries be verified with the same options.

The production of a language subset compiler specifically for safety critical applications is not recommended. A compiler used by many in a broad range of applications is more likely to uncover latent faults which may be present in the compilation system. Many safety standards require assessment of the quality of the compilation system through an analysis of reported and fixed faults by the compiler vendor. Quality reports, evidence of regression testing, review of the fault tracking system may be required. Such evidence may be provided under terms of non-disclosure by the compiler vendors.

For the reasons outlined earlier in this report, safety critical projects will use a subset of the Ada language. The application code must be analysed to ensure that the appropriate subset is used. This analysis may be done by code inspections, by the use of source analysis tools, or by the compiler itself, if it has this capability.

The Ada Semantic Interface Specification [ASIS] provides a standard mechanism for obtaining information about an Ada program or its components. As ASIS is an ISO standard, portable tools may be written which perform analysis of the language constructs used. Traversal algorithms, access to semantic information and structural constructs are provided for Ada source code under investigation. This representation exposes scope, visibility, overloading and other attributes to provide analysis tools the same information that a compiler code generator would use. Tools may thus be written which are independent of the compiler, but use the intermediate representation offered by the compiler.

An alternative subsetting method would be to use compilers directly. The compiler may be able to enforce a subset when directed to do so by the user through compiler options, or use of pragma restrictions.

A compiler processing in subset enforcement mode will be unable to pass many tests of the ACVC suite. Even the reporting packages of the ACVC may use features of the language which are outside of the safety critical subset.

The level of confidence in the subset compiler increases if the algorithms used in the subset and the full compiler are unchanged. It is important the implementation of pragma restrictions, or compiler subsetting options does not affect the general algorithms used by the compiler (control flow optimisations, memory management etc. should be the same whether pragma restrictions is used or not, or the differences well documented).

7.2 Compiler Qualification

The compiler and linker are development tools. They transform source to object and through this process may produce translation errors. Before a development tool can be used on a high integrity application with total trust, it must be qualified. At this point in time this is beyond the state of the art to qualify software as complex as a compiler. This level of distrust forces some verification to be performed at the object code level. This is usually performed through dynamic testing on the target computer with a specified coverage (say, Modified Condition/Decision Coverage to 100%). Alternatively, object code analysis may be performed, see section 3.3.10.

Verification through testing may possibly uncover compiler faults. The most serious are translation errors where the object code generated implements the semantics of the source program incorrectly. Faults discovered should be reported to the compiler manufacturer where they can be logged and tracked using an error reporting system. Subsequent compiler versions may improve the compiler correctness by fixing the reported errors.

A list of residual errors of errors not fixed in a given version should be available to the users. The use of a restricted subset compiler helps in this area as it is usually the combinations of complex constructs which causes the biggest problems for the compiler.

A compiler may generate correct but sub-optimal code, for example a range check on a variable which has already been checked. Although this may not affect the semantics of the program, it may introduce additional paths which are not covered through testing. This complicates coverage analysis at the highest levels of criticality. If traceability between source and object code cannot be demonstrated, then other verification means must be used to show that the compiler has not inserted incorrect code.

Additional code produced by the compiler must be found and traced and classified into 'dead' or 'deactivated' code. Dead code must be removed. Deactivated code must be documented and traced to a derived requirement.

7.3 Run-Time System

The run-time system (RTS) forms part of the operational software. It has a direct impact on the integrity of the application. Consequently the RTS needs evidence of verification to an integrity level which corresponds to the integrity level of the application or higher.

The RTS consists of several classes of routines used by the Ada program. Some routines are linked in automatically by the compiler to initialise and manage the target environment (e.g. set up memory bases for stacks, set up interrupt vectors and so on.) Some routines are linked in on demand by the code generator. The compiler may on occasion use run-time routines to implement operations which require many instructions (e.g. bit_block_move on an architecture which does not have an instruction to perform this). Some routines are made visible by the Ada language and supplied through packages supplied by the RTS. (e.g. Ada.Synchronous_Task_Control) The tasking system will include routines to declare, activate and perform synchronisation between tasks on behalf of the user. Calls to these routines are generated through the use of the underlying constructs.

The RTS may be supplied entirely by the developers of the Ada compiler, or a reduced Ada RTS may interface to a language independent RTS. With both approaches, a substantial effort must be undertaken to demonstrate that the RTS implements the semantics of the language. It must also be shown that the RTS executes with bounded execution times, and uses machine resources in a predictable way.

Under DO-178B guidelines, the functionality of a system and the verification activities undertaken provide evidence for certification of the specific system. Software is not certified, consequently, the RTS cannot be certified. Certification evidence can be produced for the RTS, but the materials produced will be scrutinised for each system in which the RTS is used. The requirements for certification does not diminish because it is Commercial Off The Shelf (COTS). Although broad use may improve the pedigree of a RTS, it does not diminish the responsibility for safe operation.

The requirements of the underlying safety or security standard must be satisfied for the RTS and evidence must be made available in accordance with the criticality level of the application.

8. References

8.1 Applicable Documents

These documents are formal or informal standards and guidance material which those developing critical systems may be expected to follow.

[ARM] ISO/IEC 8652:1995. Information technology - Programming languages - Ada.

[ARP 4754] ‘Certification Considerations for Highly-Integrated Or
Complex Aircraft Systems’. 1996

[ARP 4761] ‘Guidelines and Methods for Conducting the Safety
Assessment Process on Civil Airborne Systems and Equipment’. 1996

[ASIS] ISO/IEC 15291. Information technology - Programming languages - Ada Semantic Interface Specification.

[AQS] Ada 95 Quality and Style: Guidelines for Professional Programmers. SPC-94093-CMC. Ada Joint Program Office. October 1995.

[Common Criteria] ISO CD 15408 (Parts 1, 2 and 3). Evaluation Criteria for Information Technology Security.

[DO-178B] Software Considerations in Airborne Systems and Equipment Certification. Issued in the USA by the Requirements and Technical Concepts for Aviation (document RTCA SC167/DO-178B) and in Europe by the European Organization for Civil Aviation Electronics (EUROCAE document ED-12B), December 1992.

[DS 00-55] Defence Standard 00-55, ‘Requirements for Safety Related Software in Defence Equipment’, Ministry of Defence, (Part1: Requirements; Part2: Guidance). August 1997.

[EN-50128] CENELEC, Railway Applications: Software for Railway Control and Protection Systems. Draft of EN 50128:1997. November 1997.

[FDA] ODE Guidance for the Content of Premarket Submission for Medical Devices Containing Software. Draft, 3rd September 1996.

[GAMP] Supplier Guide for Validation of Automated Systems in Pharmaceutical Manufacture. ISPE 3816 W. Linebaugh Avenue, Suite 412, Tampa, Florida 33624, USA.

[IEC 601-4] IEC 601-1-4: 1996. Medical electrical equipment --- Part 1: General requirements for safety 4: Collateral Standard: Programmable electrical medical systems.

[IEC 880] IEC 880:1986. Software for computers in the safety systems of nuclear power stations.

[IEC 61508] IEC 61508: Draft. Functional safety: safety-related systems. Parts 1-7. Draft for public comment, 1995. **{Update to new reference}**

[ISO/IEC 15026] ISO/IEC 15026:1997. Information technology, Software Engineering, Software Integrity - System and Software Integrity Levels.

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

[MISRA] Development Guidelines For Vehicle Based Software. The Motor Industry Software Reliability Association. MIRA. November 1994. ISBN 0 9524156 0 7.

[NASA] NASA Guidebook for Safety Critical Software --- Analysis and Development. NASA Lewis Research Center. 1996.

[NRC] Review Guidelines on Software Languages for Use in Nuclear Power Plant Safety Systems. Nuclear Regulatory commission. NUREG/CR-6463. June 1996.

8.2 Referenced Documents

These documents contain material that uses of this Guide may find useful.

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

[2]A.Burns and A.J. Wellings, Real-Time Systems and Programming Languages (2nd Edition), Addison-Wesley, 1997

[3] J. Carroll and D. Long, R. Alur and D. Dill, Automata for Modelling Real-Time Systems, in Automata, Languages and Programming, editor M.S. Paterson, Lecture Notes in Computer Science, Vol 443, 1990.

[4] J. Peterson, Petri Net Theory and the Modelling of Systems, Prentice Hall, 1981.

[5] B. Berthomieu and M. Diaz. Modeling and verification of time dependent systems using time Petri nets. IEEE Transactions on Software Engineering, 17(3):259--273, 1991.

[6] C.A.R. Hoare, Communicating Sequential Processes, Prentice-Hall International, 1985

[7] R. Milner, Communication and Concurrency, Prentice-Hall International, 1989

[8] **{Get author from AB}** Theory of Finite Automata, Prentice-Hall, 1989.

[9] T. Baker and T. Vardanega, Session Summary: Tasking Profiles, Proceedings of the 8th International Real-Time Ada Workshop, ACM Ada Letters, pp5-7, 1997

[10] W J Cullyer, S J Goodenough and B A Wichmann, 'The Choice of Computer Languages in Safety-Critical Systems', Software Engineering Journal. Vol 6, No 2, pp51-58. March 1991.

[11]D Graham and T Gilb, 'Software Inspection', Addison-Wesley, 1993, ISBN 0-201-63181-4.

[12] IEEE Standard Glossary of Software Engineering Terminology, IEEE Std 610.12-1990.

[13] Motet G., Marpinard A., Geffroy, J. C., 'Design of Dependable Ada Software', Prentice Hall, 1996

[14] G J Myers, 'The Art of Software Testing', Wiley, 1979.

[16] D J Pavey and L A Winsborrow. Demonstrating Equivalence of Source Code and PROM Contents. Computer Journal. Vol 36, No 7. pp654-667. 1993.

[17] David Guaspari, Carla Marceau and Wolfgang Polak. Formal Verification of Ada Programs. IEEE Transactions on Software Engineering, vol 16, no 9, September 1990.

[18] W Sewell. “Weaving a Program --- Literate programming in WEB”, Van Nostrand Reinhold. 1989.

[19] J G P Barnes ‘SPARK’

9. Appendices

10. Index

{This needs to refer to paragraph numbers in the final version (difficult to implement in Word)}

You can submit comments in a variety of ways, which in order of preference are as follows:

- By E-mail to hrg@cise.npl.co.uk;
- By post to B A Wichmann, National Physical Laboratory, Teddington, Middlesex, TW11 0LW, UK;
- By FAX to +44 181 977 7091, marked for the attention of B A Wichmann.

11. Status

Working draft ISO/IEC/JTC1/SC22/WG9 (Ada) / HRG.

This a completely informal document and should not be quoted.

Please comment to HRG (e-mail to hrg@cise.npl.co.uk), but do not quote.

This is version 3.4 reflecting changes made following the ESTEC Meeting, March 1997.

Owners of the different sections (as of 11/03/98):

1. Introduction	Brian
2. Scope	Brian
3. Verification Techniques	Praful
3.1 Traceability	
3.2 Reviews	
3.3 Analysis	
3.4 Testing	
3.5 Use of verification techniques in these Guidelines	
4. General Language Issues	Peter
4.1 Writing Analysable Programs	
4.2 The Choice of Language	
5. Criteria for Assessment of Language Features	Tullio
6. Assessment of Language Features	Tullio
6.1 Types with Static Attributes	David
6.2 Declarations	Tullio
6.3 Names, including scope and visibility	Brian
6.4 Expressions	David
6.5 Statements	Alan
6.6 Subprograms	Alan
6.7 Packages (child and libraries)	Peter
6.8 Arithmetic types	Brian
6.9 Low level and interfacing	Alan
6.10 Generics	David
6.11 Types with Dynamic Attributes	Brian
6.12 Exceptions	Gilles
6.13 Tasking	Alan
6.14 Distribution	Alan
7. Compilers and Run Time Systems.	George