TECHNICAL REPORT

ISO/IEC TR 15942

First edition
2000-03-01

Information technology — Programming languages — Guide for the use of the Ada programming language in high integrity systems

Technologies de l'information — Langages de programmation — Guide pour l'emploi du langage de programmation Ada dans les systèmes de haute intégrité

Reference number
ISO/IEC TR 15942:2000(E)

ISO copyright office
Case postale 56 • CH-1211 Geneva 20
Tel. + 41 22 749 01 11
Fax + 41 22 734 10 79
E-mail copyright@iso.ch
Web www.iso.ch

Printed in Switzerland

Contents

Foreword

Introduction

Section 4 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 5 provides the main technical material of this Technical Report by classifying Ada language features. Users of this Technical Report 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 Technical Report concludes, in Section 6, by providing information to aid the choice of a suitable Ada compiler together with its associated run-time system.

References to relevant standards and guides are provided. A detailed analysis of Ada95 for high integrity systems is available in References [CAT1, CAT2] and [CAT3] .

A comprehensive index is provided to ease the use of the Technical Report.

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.

Table 1: Levels of criticality in some 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

This Technical Report emphasizes the higher levels of criticality, for which the more demanding verification techniques are employed and for which Ada provides major benefits.

This Technical Report, 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], [DS 00-56], [ARP 4754] and [ARP 4761] may be useful in determining the criticality of a system if this is not covered by application-specific standards.

This Technical Report has been written for:

1. Those responsible for coding standards applicable to high integrity Ada software.
2. Those developing high integrity systems in Ada.
3. Vendors marketing Ada compilers, source code generators, and verification tools for use in the development of high integrity systems.
4. Regulators who need to approve high integrity systems containing software written in Ada.
5. Those concerned with high integrity systems who wish to consider the advantages of using the Ada language.

This Technical Report is not a tutorial on the use of Ada or on the development of high integrity software. Developers using this report are assumed to have a working knowledge of the language and an understanding of good Ada style, as in [AQS].

When proposals were made that a subset of Ada should be specified for high integrity applications, it was realized that the provision of the Safety and Security Annex in the Ada standard did not satisfy all the requirements of the developers of high integrity systems. In consequence, a group was formed under WG9 to consider what action was needed. This group, called the HRG, proposed and drafted this Technical Report over a three year period.

Conventions

In line with the Ada standard, the main text is in a Roman font. Ada identifiers are set in a sans-serif font, and the Ada keywords in a bold sans-serif font.

Postscript

The guidance provided here reflects the understanding of the issues based mainly on using the previous Ada standard in developing high integrity applications. Over the next few years, the current Ada standard will be used for further high integrity applications which will no doubt need to be reflected in a revision of this guidance. Specifically, further detail can be produced based upon the experience gained.

Instructions for comment submission

Informal comments on this Technical Report may be sent by e-mail to hrg@cise.npl.co.uk. If appropriate, the project editor will document the issue for corrective action.

Comments should use the following format:

```
!topic: Title which is a summary in one sentence
!reference TR 15942- ss.ss.ss
!from Author, Name, yy-mm-dd
!keywords keywords related to topic
!discussion
text of discussion
```


where *ss.ss.ss* is the section number, *yy-mm-dd* is the date. If the comment requests a change, a rationale for this and the substance of the actual change proposed would facilitate the processing of the comment.

Information technology - Programming languages - Guide for the use of the Ada programming language in high integrity systems

1 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 [ISO CD 15408].

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 this Technical Report is synthesized from an analysis of their requirements and recommendations.

1.1 Within the scope

This Technical Report assumes that a system is being developed in Ada to meet a standard listed above or one of a similar nature. The primary goal of this Technical Report 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 this Technical Report to mean all instantiations of the generic should be executed.

1.2 Out of scope

The following topics are considered to be out of scope with respect to this Technical Report:

- Domain-specific standards,
- Application-specific issues,
- Hardware and system-specific issues,
- Human factor issues in the application (as opposed to human factors in the use of the Ada language which is in scope).

2 Verification Techniques

Verification is the confirmation by examination and provision of objective evidence that specified requirements have been fulfilled [ISO 8402: 2.18] .

There are currently four approaches required by applicable standards and guidelines (see Section 7.1) to support the verification of software:

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

Each one of these is discussed below. Where appropriate, language-specific techniques that support each approach are discussed. Finally, these techniques are grouped into categories that can form a basis for the analysis of Ada language features. This analytical approach forms the basis for the assessment presented in Section 5.

2.1 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:

2.2 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 [11] 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 high integrity 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.

There is a distinct tradeoff between the use of language features to capture design intent and support abstraction versus the need for simplicity and predictability. This document leaves to coding guidelines, such as the Ada Quality and Style Guide [AQS], software engineering issues such as those concerned with readability and reusability. Those issues that are associated with quantifiable analysis are considered in the following sections.

2.3 Analysis

This Technical Report distinguishes 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 this Technical Report 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

for loop control variables, make it difficult to produce poorly structured Ada code. If the **goto** statement is not used and relatively minor restrictions are made on placement of **exit** and **return** statements, Ada code becomes inherently well structured.

2.3.2 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.

In most general-purpose languages, Data Flow analysis is a complex activity, mainly because global variables can be accessed from anywhere, and because subprogram parameters do not support **out**-only modes. The job can be made significantly easier in Ada which has packages to contain potentially shared data, and **out** mode parameters on subprograms.

2.3.3 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.

2.3.4 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 Flow, Data Flow and Information Flow analysis.

Symbolic Execution is a technique where the program is ‘executed’ by performing back-substitution; in essence, the right hand side of each assignment is substituted for the left hand side variable in its subsequent uses. This converts the sequential logic into a set of parallel assignments in which output values are expressed in terms of input values. Conditional branches are represented as conditions under which the relevant expression gives the values of the outputs from the inputs. To undertake this

```
      ≤ 0:
   X  =      A+B
   Y  =      D-C
   Z  =      not defined on this path (retains initial value)
A+B   > 0:
   X  =      A+B
   Y  =      D-C
   Z  =      D-C+1
```

These algebraic expressions give the output in terms of the input and can be compared (manually) with the specification of a subprogram to verify the code.

Symbolic Execution can also be used to assist with reasoning that run-time errors will not occur (e.g., Range Checking). The Symbolic Execution model is extended to include expressions indicating the conditions under which a run-time error may occur. If these expressions are mutually contradictory for a particular execution path then that path is free from potential run-time errors. In the above example, if Z has an upper declared bound of 10, then we have the condition that if A+B>0, then D-C+1<11 to avoid raising the exception on the upper bound.

2.3.5 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/security property can also be constructed.

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 VCs. 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 and termination of any recursion. Recursion is not normally permitted in high integrity systems. 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, similarly 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 high integrity 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

pragma Reviewable and **pragma** Inspection_Point ensure that there is traceability from the source code to the object code to facilitate timing analysis.

2.3.9 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, that perform specific computations or watchdog timer functions.

Other Memory Usage analysis will show the absence of interference between Ada and other components such as low-level and hardware device drivers and resource managers. In particular heap memory should usually be avoided and IO devices rigorously partitioned. Ada is particularly useful when doing such analysis since the **pragma** Restrictions (No_Allocators) can

pragma Restrictions (No_Implicit_Heap_Allocation) to ensure no implicit usage of the heap.

2.3.10 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 failure.

This analysis is sometimes undertaken by manual inspection of the machine code generated by the compiler. 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** Reviewable provides basic information to assist in tracing from source code to object code. **Pragma** Inspection_Point can be used to determine the exact status of variables at specific points. For the requirements of such an analysis, see [15].

2.4 Testing

2.4.1 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 behaviour [14].

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 is to partition the data domain into equivalence classes and their boundary values in order to limit the number of test cases.

2.4.2 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
The inputs and outputs of the component [BS 7925-1: 3.42] under test are divided into equivalence classes in which the values within one class can reasonably be expected to be treated by the component in the same way. The equivalence class for numeric data is a range having the same sign or zero. For data of an enumerated type, each value usually forms a class, since each value could be expected to be treated differently. For composite types, the equivalence classes are obtained by combining