High Integrity Ada

B A Wichmann,
National Physical Laboratory,
Teddington, Middlesex, TW11 0LW, UK
E-mail: baw@cise.npl.co.uk

March 27, 1997

Abstract

This paper describes the approach being taken by an ISO group to produce Guidelines for the use of Ada when developing high integrity applications.

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 grow, so do the demands for improved techniques for their production.

Hence there is a need to ensure critical systems have the properties required, and this can only be achieved by analysis of the software in addition to conventional dynamic testing. Unfortunately, analysis of software written using low-level languages is prohibitively expensive, since it is necessary to analyse every instruction in the program merely to ensure the integrity of the data or control flow. On the other hand, the strong typing in the Ada language facilitates such analysis by reducing the potential means by which data can be overwritten or the control flow changed. Hence not only are high integrity systems important, but Ada has appropriate attributes to provide the assurance needed in their design.

Given that Ada is being used for a high integrity application, then further confidence can be gained by providing guidance on the use of the language. This guidance material identifies those features of Ada for which additional verification steps should be performed to ensure that their use is appropriately controlled. Following such guidance should provide all the assurance that the high integrity application requires.

The use of Ada to produce high integrity applications is recommended on the grounds that:

1. The semantics of Ada programs are well-defined, even in error situations.
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.

Annex H of the Ada 95 standard has been developed to ensure, as far as a language standard can, that the special requirements for high integrity applications can be met. The primary areas in which such additional assurance is needed are for safety and security.

In the UK, Ada is the language of choice in the Defence and Aerospace sector for high integrity applications. For an analysis of this sector, see the Foresight report on High Integrity Real Time Software [12].

These proposed Guidelines are being considered by the HRG group which operates under the auspices of the ISO Ada group (ISO/JTC1/SC22/WG9).

2 Ada Guidelines

The proposed Guidelines we are developing are specific to Ada 95 [17], the current ISO Ada standard. The desire is to effectively complete the technical work on the Guidelines in 1997 so that newer high integrity Ada systems can use the language with confidence. Indeed, since Annex H of the Ada 95 standard has specific facilities to aid the development of such applications, we conclude that very high confidence can be gained by suitable use of the language. The Guidelines would be a separate ISO document in addition to the Ada standard.

The context within which the Guidelines would be used must be carefully considered. The consensus of the approach that the ISO group wishes to adopt is summarised in the following points:

- The Guidelines must be appropriate for the development of software which is required to meet specific safety and security standards, such as: DO-178B [32], IEC 880 [18], IEC 1508 (Part 3) [15], ITSEC [31], IEC 601-1-4 [16] and UK Defence Standard 00-55 [30]. In consequence, we attempt not to repeat any material in these standards (and to avoid any inconsistencies with them).

- The Guidelines assume the appropriate use of language constructs as available in the Ada Quality and Style Guide [37].

- Where possible, we reference other Software Engineering terminology and standards [25, 28], to minimise the size and complexity of the Guidelines (which also makes the main content Ada-specific).

- There is an implicit assumption that appropriate risk assessment methods have been employed (which arise from the safety and security standards listed above). General advice on risk management is readily available, see [11, 14].

- It is assumed that appropriate quality management procedures are employed consistent with ISO 9001 [19, 20] in the context of a suitable life-cycle [21].

- The Guidelines are not aimed at specific, quantified reliability targets. It is widely understood that for the highest integrity levels, it is not possible to measure the reliability, at least by testing [24].

Given this approach, we believe that there are few viable options to achieving high integrity other than the use of the Ada language. Six years ago the author recommended (with others) Pascal as well as Ada [27], but the maturity and functionality of Ada now make it significantly superior to Pascal.

A major need for such Guidelines is to present a common view for suppliers, users and certifiers. For instance, major defence projects are frequently multi-national and therefore require a common resolution of methods to produce high integrity software. Also, if a certification body is unhappy about the use of a specific Ada feature, then very expensive additional verification steps could be required, or even worse, a major re-working could be required. These Guidelines should reduce the risk of such unplanned requirements.

In some cases existing (non-Ada specific) Guidelines, which have been developed for one specific industrial sector, are being used as input for our Guidelines which are Ada-specific but for any sector. Examples of such Guidelines are: US-nuclear [35], US-space [36], US-medical [13], European-railway [26] and UK-motor [29].

One specific issue has caused the group to revise the approach taken. Almost all of the safety and security standards have a number of levels to reflect to risk or integrity of the system. The initial approach was to have Guidelines with four levels to mirror those in the majority of the standards. A number of problems were noted with this approach:

1. Given that a system is being developed to a standard with four levels (like IEC 1508, say), then the assumption would be that there is a simple mapping between the two. This is unlikely to be the case.

2. In those cases in which the standard has a different number of levels (like ITSEC, with 7 levels), then the correspondence would not be clear.

	Group Name	Method Name
Analysis	Functional Correctness	Formal Code Verification
		Symbolic Execution
	Flow Analysis	Control Flow
		Data Flow
		Information Flow
	Stack	Stack usage
	Timing	Worst Case Execution Time
	Range Checking	Range Checking
	Other Memory Usage	Other Memory Usage
	Object Code Analysis	Object Code Analysis
Testing	Structural Coverage	MCDC
		Branch Coverage
		Structure Coverage
	Equivalence Class	Equivalence Class
	Boundary Value	Boundary Value

Table 1: Methods

3. Some requirements do not map well into the concept of levels, but are derived from the system requirements. This is in contrast to an ISO standard which provides means of determining the software integrity required from the system requirements [23]. An example of a specific requirement is for worst-case execution time which could be needed for any integrity level system.

The Guidelines nevertheless need to reflect the requirements of very different systems which we handle by means of specific **methods**. These methods are tabulated in Table 1 to reflect their heirarchical relationship.

Language issues will then be driven by the requirement (or not) to use these methods. For instance, if Formal Code Verification is a requirement, then those language features for which no appropriate formal model exist must be excluded from use within the application. In general, language features are classified in three ways:

Included. These features can be included since their analysis and use provides no essential difficulties, ie, have tractable and well-understood verification techniques. All the straightforward features of Ada 95 are in this category.

Allowed. These features have known, but well-understood, problems in their use. Additional verifications steps may be needed. The Guidelines will specify the problems and known approaches to their resolution. Management may specify that a specific approval process must be invoked, or that authorization be obtained.

An example of a feature in this category would be Unchecked_Conversion, since it is likely to be needed in many applications, but has known difficulties for validation. An additional validation step might be to check the object code produced by the compiler.

Excluded. These features are essentially incompatible with the method being employed. The only effective resolution is not to use the feature.

One other possible approach to producing high integrity Ada is the standardization of a subset of the language, an example of which is the SPARK subset of Ada 83 [9]. This approach was considered by ISO and rejected. The subsequent meetings for the HRG considered the development of Guidelines.

The Guidelines approach has the following advantages over a specific subset:

- A single subset could only be optimal for at most one integrity level.

- Specific control over the use of language features is already provided in Annex H of the Ada 95 standard. (Hence the user can 'design his own' subset by permitting or not around 20 specific features.)
- As validation technology improves, the optimal subset grows. In practice, subsets are enforced by tools which would then imply changes to the tools as well. (We give an example below in which at some future date one can reasonably expect the concurrency features of Ada to be approved for the high integrity systems.)
- Some low-level features of Ada are required by many applications, even if their use is very tightly controlled and is only a tiny fraction of the code. (Hence, if a subset is enforced without exception, these low-level features would have to be included, but that would give a subset having verification problems. If the subset is not strictly enforced, then can it provide the assurance needed?)

2.1 Content of the Guidelines

The main sections of the proposed Guidelines are as follows:

Verification Techniques. This section enumerates the methods in Table 1 and their relationship to the Guidelines.

Language Usage. This section catalogues the language features in terms of the **included**, **allowed** and **excluded** categories together with specific guidance which is related to language features (and their interaction, of course).

Compilers and Run Time Systems. These two key elements have several properties which implies that specific guidance must be given.

Extracting information about an Ada program merely from the source text is potentially very fruitful due to the strong type checking and well-controlled interfaces that the language provides. Compilers already extract such information for code generation, but in the past, most of the information has been unavailable for validation and verification. This implies that validation tools for Ada 83 have had to repeat the source text analysis undertaken by a compiler. This is now changing due to the Ada Semantic Information System (ASIS) standard [1]. This standard provides an interface for tools to extract information from the Ada library provided the compiler supports the standard.

For high integrity systems, it would appear that ASIS provides a basis for tools undertaking quite sophisticated analyses. At this stage, the Guidelines would not necessarily *require* the use of ASIS. An ASIS-based tool which lists the **excluded** and **allowed** features in an Ada program is clearly desirable. ASIS is obviously limited to the Ada source text and therefore cannot directly handle requirements involving, say, the external environment.

The ISO Ada group is fortunate in having access to a very detailed study undertaken in Canada on the application of Ada 95 for high integrity applications [2, 3, 4, 5, 6]. Although some members of the Ada group contributed to this study and commented upon it, the resulting reports are not the consensus Guidelines that we are proposing here. In consequence, the material needs to be reformulated into an agreed form for the ISO publication.

At the lowest level, the Canadian study reports on the suitability of the majority of the Ada 95 constructs for their use in high integrity applications. Their report only considered the highest level of integrity, while ISO will take into account the analysis required and classify this by a three-way system. However, the reworking of much of this information into the **included**, **allowed** and **excluded** categories should not be too difficult.

2.2 Difficult issues

Obtaining high integrity requires careful judgement which is more difficult as the complexity of systems increases. The Ada language provides the means of controlling complexity, but there are pitfalls for the unwary which the Guidelines will enumerate. For some issues, the best approach is hardly obvious and we consider some of these below.

Research versus standardization

The research community would like to address the intellectual challenge of demonstrating the validation of the more complex features of Ada. However, it is only established methods which have already been shown to be satisfactory which can be used with confidence. Tool support is vital here, since complete reliance upon manual validation methods is not practical.

An example of a research area is concurrency, which is directly supported in Ada 95 by two mechanisms: the rendezvous and protected types. There are clear advantages in using concurrency when the external world with which the system reacts is naturally concurrent. However, race conditions, deadlock and unsynchronized update can easily occur unless extreme care is taken. A key issue here is to ensure that test cases have deterministic execution, a view which has strong support from the certification bodies.

The ISO group has not resolved this issue, but it does appear that for the highest levels of integrity, Ada protected types could be classed as **allowed**, following ideas presented in [7]. In general, it would seem that many Ada features which would currently be classified as **allowed** could at some future date when the verification technology improves as **included**.

New features of Ada 95

It would appear that the new features of Ada 95 would present a significant challenge for the developers of high integrity applications. However, Ada 95 actually simplifies many aspects due to the following:

- Ada 95 has defined the language more closely, leaving fewer aspects which can vary between implementations.
- Annex H allows the user to restrict the usage of language features in a way that is enforced by the compiler.
- Hooks are provided so that the actual object code produced by a compiler can be audited by the developer of high integrity systems. (Of course, the auditing may be aided by tools.)
- Many of the extensions in Ada 95 are not to the basic language but in the form of additional required packages. (Such packages need only be considered if actually used in an application, whereas some basic language features have an impact even if they are not used. The additional packages have defined properties, such as the numerical accuracy of the mathematical functions, which can clearly simplify program verification.)

Of course, there are some challenges for high integrity usage in new features of Ada 95, such as: protected types (mentioned above as a means of providing concurrency in a more easily validated form than the Ada 83 rendezvous), and object-oriented extensions.

The object-oriented extensions provide the basic ability to add functionality with extensions to records (tagged types) and new operations on such records. However, this is done is a manner which allows the user to control dynamic dispatching of subprogram calls, which could otherwise require significant additional validation effort. Similarly, the programmer can restrict the use of class-wide operations to ease the verification burden. This contrasts with fully object-oriented languages like Smalltalk (or Java) which is essentially dynamic and hence would be regarded as unsuitable for the highest integrity systems.

Dynamic testing versus static analysis

Static analysis and dynamic testing are complementary ways of comparing an implementation with its specification. Testing allows the most *direct* comparison between implementation and operational requirements. However, because exhaustive testing is impossible — even modified condition decision coverage (MCDC), which is usually very onerous, does not cover more than a fraction of possible execution histories — testing in general cannot show the absence of errors.

In contrast, because 'size' of a static analysis task is determined by the size of a program source text, rather than the number of its possible execution sequences, *in principle* static analysis can be complete, in showing absence of errors *of some classes*. However, static analysis is applied only to models of a program, derived from its source text using a precise definition of the programming language. Since the validity of the models must be checked, and since they may not capture all aspects of program execution (for instance of timing, or resource utilisation), a substantial amount of testing remains essential.

Currently, the cost of validation of high-integrity software, principally incurred in the dynamic testing phases, may exceed half the total project cost. Such figures are commonly reached for avionics software, for which the MCDC testing needed to meet DO-178B requirements [32] is enormously expensive. The high costs can largely be attributed to the fact that the dynamic testing reveals flaws — often of a fundamental nature, such as mis-statement or misinterpretation of requirements specifications — very late in the development life-cycle. Their correction then is very costly, as is the subsequent repetition, usually several times over, of large parts of the MCDC process.

Solutions are being found to this problem by using a 'lean engineering' or 'correctness by construction' approach [38]. This exploits the static semantic rigour of Ada, and the possibility of performing strong static analysis checks on Ada source texts (because the language is relatively well defined), to detect errors in specification, design and coding early in the life-cycle, as the software is being constructed.

Experience indicates that this approach substantially changes the character of software development [38], involving much more interaction between programmers and designers at relatively early stages of software development. However, even where MCDC test coverage is still required, validation costs have been dramatically reduced, because testing takes on a more confirmatory role, rather than repeated late revelation of errors, with costly repair of these.

It is proposed that the Guidelines should provide some information on how Ada can be exploited in achieving correctness by construction, with particular emphasis on the following aspects:

- How Ada 95 can be exploited, in constructing verifiable designs.
- Guidance on Ada usage to assure predictable behaviour and validity of program models employed in static analysis.
- Discussion of the range of static analysis techniques applicable to Ada 95 programs in particular (from the static-semantic checks required of compilers, through data and information-flow analysis to proof of absence of run-time errors and formal code verification).
- Guidance on ways of using Ada 95 that facilitate dynamic test.
- Discussion of co-ordination/management of a correctness by construction regime for Ada 95 software development.

Exception handling

At the highest integrity level, it is accepted that code should be exception-free. This is the view taken in the Canadian study and the SPARK subset, for instance. If one can be assured that code is exception-free, then one can use the pragma Suppress, which can simplify the code generated by the compiler, thus reducing verification costs.

At lower levels of integrity, the optimal situation is not so evident. Showing the code is exception-free is often expensive so that if this step could be avoided, there would be savings. As Ada provides the ability to detect and recover from the raising of an exception, the natural approach at the lower levels would be to allow for such an eventuality. Indeed, handling an exception can protect the system against some faults. The analysis of programs in which exceptions can be raised can be complex, but facilities like ASIS should make this practical. In consequence, it appears that this language feature needs to be handled quite differently at the various integrity levels.

Timing

Resource usage is a key requirement for the validation of systems. With predominantly static storage usage, and requirements in Ada 95 for compilers to produce memory usage information, storage is not a major problem. Unfortunately, timing is getting very complex with the pipe-lining and caches in modern processing units.

This area is more hardware than language dependent. However, it is necessary to provide worse case execution times from the object code. It remains to be seen if useful bounds can be obtained for the next generation of processors.

Fixed point versus floating point

Numerical computation on physical quantities can be logically undertaken in either floating point or fixed point. The older processor chips without built-in floating point have often forced designers to use fixed point. The compiler support for fixed point is relatively complex and hence provides an additional verification burden. In contrast, the recent fault in the division operation in the Pentium processor has illustrated the dangers inherent in the complexity of the floating point hardware (even if the compiler support is then easier to validate).

For the Ada Guidelines, it would appear that both fixed point and floating point must be considered with an analysis of the validation issues in both cases.

3 Conclusions

The belief is that with approved ISO Guidelines on the application of Ada for high integrity applications, developers can produce systems with greater confidence and wider acceptability, which will meet the requirements of both the certification bodies and the users.

4 Acknowledgements

This work would not have been possible without the support of the membership of the HRG, the ISO High Integrity Ada group. Those members who have contributed to the HRG work so far are (in alphabetical order): John Barnes (UK), Praful V Bhansali (Boeing), Alan Burns (University of York), Bernard Carré (Praxis), Dan Craigen (ORA, Canada), Mike P. DeWalt (FAA), Robert Dewar (Ada Core Technologies), David Guaspari (ORA), C. Michael Holloway (NASA), Mike Kamrad (Computing Devices International), Stephen Michell (Maurya Software), Alexander Miethe (Competence Center Informatik), George Romanski (Thomson Software Products), Mark Saaltink (ORA, Canada), Michael K. Smith (Computational Logic Inc), James C. Stewart (U.S. Nuclear Regulatory Commission), Adam Tacy (UK MoD), Phil Thornley (British Aerospace Defence), David Tombs (Defence Research Agency), and Tullio Vardanega (ESTEC).

References

[1] Ada Semantic Information System. Working draft. 1st November 1996. Available on the Internet:

`public/AdaIC/work-grp/asiswg/asis/v2.0`

[2] Dan Craigen, Mark Saaltink, Steve Michell. "Ada95 and Critical Systems: An Analytical Approach." In Proceedings of "Reliable Software Technologies: Ada Europe'96, Alfred Strohmeier, Editor. Lecture Notes in Computer Science, Volume 1088, Spring-Verlag, 1996.

[3] Steve Michell, Dan Craigen, Mark Saaltink. "Using Analytical Approaches for High Integrity Ada95 Systems." International Real-time Ada Workshop, Ravenscar, U.K. April 1997. To appear in Ada Letters.

[4] Dan Craigen, Mark Saaltink, Steve Michell. "Ada95 Trustworthiness Study: A Framework for Analysis." ORA Canada Technical Report TR-95-5499-02, November 1995.

[5] Mark Saaltink, Steve Michell. "Ada95 Trustworthiness Study: Analysis of Ada95 for Critical Systems." ORA Canada Technical Report TR-96-5499-03a, January 1997.

[6] Mark Saaltink, Steve Michell. “Ada95 Trustworthiness Study: Guidance on the use of Ada95 in the Development of High Integrity Systems,” Version 1.0. ORA Canada Technical Report TR-96-5499-04, November 1995.

[7] A Burns and A J Wellings. Restricted Tasking Models. Ada real-time workshop. 1997.

[8] British Computer Society Specialist Group in Software Testing. Standard for Software Component Testing (Working Draft 3.0). Glossary of terms used in software testing (Working Draft 6.0). October 1995. Available free on the Internet (until copyright is assigned to BSI):

`http://www.rmcs.cranfield.ac.uk/~cised/sreid/BCS_SIG/index.htm`

[9] B A Carré and T J Jennings. SPARK — The SPADE Ada Kernel. University of Southampton. March 1988.

[10] J Dawes. “The VDM-SL Reference Guide”. Pitman Publishing. 1991. ISBN 0-273-03151-1

[11] Guidelines on Risk Issues. The Engineering Council. February 1993. ISBN 0-9516611-7-5.

[12] Defence and Aerospace Panel: Technology Working Party report on High Integrity Real time software. Available free on the Internet:

`http://www.npl.co.uk/npl/collaboration/partners/foresight/index.html`

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

[14] Safety-related systems — Guidance for engineers. Hazards Forum. March 1995. ISBN 0 9525103 0 8.

[15] IEC 1508: Draft. Functional safety: safety-related systems. Parts 1-7. Draft for public comment, 1995. (Part 3 is concerned with software which is the relevant part for the ISO Ada Guide.)

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

[17] ISO/IEC 8652:1995. *Information technology — Programming Languages — Ada*.

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

[19] EN ISO 9001:1994, Quality systems — Model for quality assurance in production and installation.

[20] ISO/IEC 9000-3: 1991. Quality management and quality assurance standards — Part 3: Guidelines for the application of ISO 9001 to the development, supply and maintenance of software.

[21] ISO/IEC 12207: 1995. Information technology — Software life cycle processes.

[22] ISO/IEC 13817-1:1996 Information technology — Programming languages, their environments and system software interfaces — Vienna Development Method — Specification Language — Part 1: Base language.

[23] DIS ISO/IEC 15026: 1996 Information technology — System and software integrity levels.

[24] B Littlewood and L Strigini. The Risks of Software. Scientific American. November 1992.

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

[26] CENELEC, Railway Applications: Software for Railway Control and Protection Systems. Draft of EN 50128:1995. November 1995.

[27] 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.

[28] J A McDermid (Editor). Software Engineer’s Reference Book. Butterworth-Heinemann. Oxford. ISBN 0 750 961040 9. 1991.

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

[30] Defence Standard 00-55, “The Procurement of Safety Critical Software in Defence Equipment”, Ministry of Defence. Available free on the Internet:

`http://www.mod1ndrl.demon.co.uk/0055/0055.html`

[31] “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.)

[32] 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.

[33] J M Spivey. The Z Notation, A Reference Manual, SECOND EDITION. Prentice Hall International Series in Computer Science. 1992.

[34] B A Wichmann, A A Canning, D L Clutterbuck, L A Winsborrow, N J Ward and D W R Marsh. An Industrial Perspective on Static Analysis. Software Engineering Journal. March 1995, pp69-75.

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

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

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

[38] J Sutton and B Carre: Tri-Ada Conference 1995.

A Document Details

A.1 Status

Draft.

A.2 Project

2INTMM10.

A.3 File

Stored on the Sun in file baw/ada/hrg/safecomp.tex.

A.4 History

- First draft written 8th January 1997.
- Revised after a review by Graeme Parkin, 10th January 1997.
- Revised to take into account the comments of Alan Burns, David Guaspari and Phil Thornley. Also revised to take into account the existence of ISO 15026, 23rd January 1997. (Points made but no change made are: no names given to the levels; and a detailed discussion of tools versus subsets not provided).
- Revised to take into account comments by Dave Rayner (NPL) and Bernard Carre. Also noted the DS00-55 to all on the Internet, 30th January 1997.
- Revised to take into account changes agreed at February HRG meeting and full references given by Dan Craigen, 25th March 1997. Note that integrity levels still appear in the section on exceptions, but this seems OK to me.

A.5 Actions

- E-mail to HRG.