

Languages for Critical Systems

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

January 4, 1999

Abstract

The most critical computer systems have to be shown to be correct to regulatory authorities or perhaps that all reasonable care has been taken should a legal claim arise. The choice of programming language for such systems can aid analysis and management of the complexity within the software. The paper shows that Ada 95 has the necessary attributes to contribute in the construction of such systems.

1 Introduction

Six years ago, I wrote with two colleagues a paper about the choice of programming language in critical systems [2]. The technology has changed enough since then for the same question to be asked again.

1.1 Is the choice important?

The major problem with software is handling complexity. Compared with six years ago, we can build much bigger and more complex systems. The difficulty is demonstrating that such systems are safe. For the highest integrity safety systems, one needs to ensure *beyond reasonable doubt* that the system is safe. If such assurance is not forthcoming, people might be killed as with flight-critical software or a railway signaling system.

The choice *is* important because the source text of the software is its authoritative manifestation — the natural language specification is of little direct use since that is not executed and the binary program which is executed is too obscure to reason about. The choice is also important because of essential differences which can make validation feasible, difficult or practically impossible [18].

On the other hand, if safety software is very small indeed, small enough that writing it in assembler is not unreasonable, then the actual choice is probably not critical.

In this paper, we assume that a choice is being made for a critical system which is not so small that assembler would be feasible and hence we are looking for language features which would positively aid managing the complexity of the software.

2 Language features

We consider various features in order of increasing usefulness upon our goal of demonstrable safety.

Modularity. All modern languages claim to have this and hence it cannot be used to make an effective choice.

Information hiding. Subroutines provide a limited degree of information hiding which is universally available and hence can be assumed. The key issue is if the language can enforce a design which hides critical information from the majority of the code. The language will then assist in the task of demonstrating that critical properties are being implemented. For instance, the design might state

that the only updates to a specific variable must be in subroutines X, Y and Z. Does the language ensure that updates in the (potentially) hundreds of other subroutines cannot update the variable? To enforce such restrictions, the language must have a module concept of some type: packages in Ada [13], classes in C++ and Java, and modules in Modula-2.

Data integrity. This requires a number of features: Type-checking is required across subroutine and module boundaries. Pointers must be bound to specific types so a de-referencing gives a type-secure result. Index bounds of arrays must be checked, as must references for a nil or undefined value. C and C++ essentially fail these requirements due to the lack of reference and index checking. These failures can be overcome by using a subset of features, but that requires policing and checking that no loopholes remain. Data integrity also requires that no unassigned values are used which could cause the code to malfunction. In practice, this requires that each subroutine has defined set/use characteristics. The parameter modes in Ada can be used to ensure this. The call-by-reference in almost all other languages is inadequate and one has to rely upon design information not directly apparent from the source code.

Control-flow integrity. One needs to be assured that the program will not lose control of the processor. This requires data integrity to ensure that the program instructions are not overwritten by data. In the case of a switch/case statement, it must be impossible to execute a jump outside the program code. This might require data integrity or even knowledge of the machine code generated by the compiler. Interrupt processing is a source of problems with control flow, and at the highest integrity level, the advice is clear — don't allow interrupts.

Range constraints. Of course, one would like to know that all data values are 'correct', but this cannot always be guaranteed. It would require a mathematical proof of correctness of software which can be very difficult and too expensive to achieve. An intermediate step is to show that variables are in the intended range, and therefore a declarative means of expressing this is highly advantageous. Ada and Pascal have this feature, C, C++ and Java do not.

3 Predictable execution

Reasoning from the source text of a program is a powerful tool in demonstrating safety [16]. This implies that, given a program in the chosen language, it should be possible to show how it will execute without testing it! Indeed, we know that evidence from testing will never be sufficient for the most critical systems[6, 8].

In fact, virtually all programming languages fail to achieve the goal of predictable execution, but if a well-designed subset is used, many languages come very near.

To consider the issue in more depth, take a single statement within a subroutine which just undertakes integer addition:

```
A := B + C;
```

We need to ensure that B and C have been assigned a value. However, if B and C are parameters passed by reference, this will depend upon the rest of the program. This type of dependence can make the effective verification impossibly difficult since one even has to ensure that no cyclic dependencies of this form exist. A static analysis tool will help to ensure that B and C have been assigned only if this can be determined from the one subroutine alone.

Having determined that B and C have been assigned, we need to ensure that the result of the addition is well-defined. This requires that the mathematical sum lies within some range. This check is again only a practical proposition if B and C are known to be within a small static range. Hence the need for static ranges, as noted above.

Even the addition (of assigned values) itself is handled quite differently in various languages:

- If the mathematical result of the addition overflows the machine capacity, in Pascal, C or C++, the resulting program is undefined. (Pascal implementations traditionally make a check, while those for C and C++ do not.)

- If the result overflows in Ada, an exception is raised.
- If the result ‘overflows’ in Java, the program produces a defined value, but is not the usual mathematical one.

Hence only Ada provides a fail-safe capability by allowing the system design to undertake a controlled shut-down (say) if an exception is raised. Moreover, if an exception is *not* raised, then with Ada you know the correct mathematical result was produced. The second-best language (of those listed) is Pascal, since the static range constraints (used well) permit verification with reasonable effort.

Our single statement concludes with the assignment to A. This action is never problematic with C, C++ or Java, since even if the incorrect result is produced, the assignment is made! With Pascal and Ada there is at least a reasonable verification step of using static constraints on B and C to show that A is within the correct bounds. An assignment like $I := I + 1$, one needs to ensure that the code cannot be executed when I has the largest legal value.

4 Some comments on IEC 61508

Another draft has been issued of parts of the IEC generic standard on safety with programmable systems [4]. Part 7 is much improved on previous versions and gives good overall guidance without being specific to a programming language. The summary table does give cause for concern since several distinguishing differences between the languages mentioned elsewhere are lost. For example, it would appear that C when used in conjunction with static analysis is regarded as effective as Ada, although the lack of data integrity is admitted.

As noted above, the basic problem is managing complexity and therefore the key aspect is the ability to analyse large programs. Small, simple programs could be written in almost any language without problems. As the size of a program grows (for any fixed safety integrity level) so does the need for language attributes to aid analysis [16]. Hence I would conclude that C is satisfactory for small programs but quite inappropriate for larger ones (since it lacks data integrity and range constraints). Hence it would be an advantage if the size (or complexity) of the application was factored into the tables of the languages recommended.

Another issue with the current draft of 61508 is that it appears to reflect best practice of a few years ago rather than the current position. Of course, safety standards do have to be conservative in their approach to new technology. Embracing the latest fad would be quite wrong. Specific aspects which seem to be to reflect the past are:

1. Pascal is rarely used, and it is now difficult to purchase a compiler for a modern machine which is reliable enough for current systems. (A great shame, but a fact.)
2. BSI-QA and NCC have not provided language validation services for three years, and it seems very unlikely they will in the future. Hence qualifying a compiler must be done by other means.
3. The concern about the large size of Ada has effectively disappeared since several demanding safety systems have now been very successfully developed in the language.
4. PL/M is rarely a choice for newly developed systems and it does not seem appropriate in my view to give such prominence to a proprietary language. (The Intel manual has a disclaimer which would appear to rule out its use in critical systems.)

5 The Ada Guidelines

The new Ada 95 standard has specific capabilities to handle the concerns of safety system development. Moreover, experience with Ada 83 has shown that compilers are very reliable, and that the use of an appropriate subset is effective. Ada is also the language of choice in the defence and aerospace sector in the UK [5].

Although Ada 95 makes explicit provision for safety and security applications [14], it became apparent that this is insufficient. Providing a technical capability is one thing, ensuring that it can be effectively used

is another. The agreed approach was to prepare an ISO Technical Report *Guidance for the use of the Ada programming language in high integrity systems* [7, 19]. The objective was to ensure that adherence to safety or security standards using Ada was as effective as possible.

This undertaking has been performed under the ISO Ada standardization body. This has the virtue of ensuring that the Ada content is technically correct, but has the risk that the guidance is not adequate for the developers of safety or security systems (who are not, in general, Ada experts).

The conventional approach to high integrity systems is to use the level of criticality to determine the techniques to be applied. We found that this did not work for two reasons: firstly, the main problem area is the highest integrity systems (which implies that the division into integrity levels is ineffective since all the material is for the highest level) and secondly, different sectors apply different approaches, regardless of the integrity level.

The method adopted has been to assume that systems are developed to safety or security standards which will require specific verification techniques. For instance, the civil avionics standard places substantial emphasis on dynamic testing [10, 17], whereas the UK defence standard places emphasis on formal methods [3]. Although these differences are unfortunate, for instance it is unclear how UK military avionics systems should be developed, the Ada guidelines should be made to be effective in every industrial sector.

Given specific verification techniques, then the Guidelines identify the language features which provide barriers to the use of these techniques. From this, the developer can choose an appropriate subset. This approach does not give a single subset of the language which would have the danger of inhibiting the use of language features even when their use is well-understood. In fact, one of the interesting developments in the Guidelines is the ability to exploit the light-weight concurrency model in Ada 95. I believe that this is a very important advance since many systems are essentially concurrent. The typical current practice is to use round-robin processing which is hard to understand and validate.

5.1 Another Guideline

Some interesting comparisons can be made with the UK automotive industry guidance on the use of C [9]:

1. The C Guidelines start with stating that the ‘spirit of C’ (i.e. trust the programmer) is not appropriate for safety systems. The Ada guidelines are very much within the spirit of the language which is to provide as many compilation checks as possible [12].
2. Since the C guidelines are targeted on only one industry, the process of drafting the guidelines has been much quicker (and simpler?).
3. The C guidelines explicitly exclude the highest integrity level, while the Ada guidelines are mainly concerned with the highest level.
4. Much of the C guidance is concerned with style and intelligibility, while for Ada we assumed that this is addressed by use of other material, such as [11].
5. The higher level of the Ada language meant that many issues had to be addressed which have no equivalent in C (exceptions, generics, tasking, overloading, etc).

Very wisely, the C Guidelines explicitly excludes consideration of C++.

6 Other issues

The technical features of a language are not the only issue to be considered in selecting a language for the development of a critical system. However, if the technical concerns are ignored there is a danger that validation costs will be excessive.

Programmer experience is clearly vital. However, this should only be judged (in my view) in relation to critical systems of comparable size to the new one. Good experience developing a small system in C (say) is no guide to the use of the same programming team for producing a large critical system.

Compiler availability and maturity for Ada compilers is good — stress testing Pascal and Ada compiler at NPL has shown that Ada compilers are *much* more reliable in spite of the increased size of the language [20]. Tool support for verification and validation is also good for Ada, especially the SPARK system [1, 15].

7 Conclusions

Having been involved with the Ada language since 1975, it comes as no surprise that I would recommend it to those developing critical systems. Those that are *not* using the language should ensure that alternative technical measures are being used to achieve the same ends.

References

- [1] J G P Barnes. High Integrity Ada — The SPARK approach. Addison-Wesley 1997.
- [2] 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.
- [3] Interim Defence Standard 00-55, “The Procurement of Safety Critical Software in Defence Equipment”, Ministry of Defence, (Part1: Requirements; Part2: Guidance). 1st August 1997.
- [4] IEC 61508: Functional safety: safety-related systems. Parts 1-7. Draft for public comment, 1998. (Parts 3 and 7 are concerned with software.)
- [5] 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>
- [6] B Littlewood and L Strigini. Validation of Ultra-High Dependability for Software-based Systems. Comm ACM. Vol 36, No 11, pp69-80, 1993.
- [7] Guidance for the use of the Ada programming language in high integrity systems. ISO IEC/JTC1/SC22/WG9. Version 3.4. May 1998. (Enquiries to baw@cise.npl.co.uk)
- [8] B Littlewood and L Strigini. The Risks of Software. Scientific American. November 1992, pp38-43.
- [9] Guidelines for the use of the C language in vehicle based software. Motor Industry Research Association. May 1998. (Enquiries to misra@mira.co.uk)
- [10] 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.
- [11] Ada 95 Quality and Style: Guidelines for Professional Programmers. SPC-94093-CMC. Ada Joint Program Office. October 1995.
- [12] Taft, T. High-Integrity Object-Oriented programming in Ada 95. Keynote Presentation Ada-Europe 97 reported in Dirk Craeynest, notes on Ada-Europe’97 International Conference on Reliable Software technologies. Ada-Europe News, Number 25, December 1997.
- [13] B A Wichmann. Why Ada is for you. Programming Language Choice – Practice and Experience. Edited by Mark Woodman. Thomson Computer Press. London. ISBN 1-85032-186-8. 1996. pp125-134.
- [14] B A Wichmann. Agreeing the Safety and Security Annex. Ada Yearbook 1994. Edited by C Loftus. IOS Press. pp141-146. 1994

- [15] B A Wichmann. Strategy on the Use of SPARK. NPL Report 227/94. June 1994.
- [16] 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.
- [17] B A Wichmann. A Review of a Safety-Critical Software Standard. NPL. June 1994.
- [18] B A Wichmann. Why it is difficult producing safety critical software? Ingenuity. (ICL's Technical Journal). May 1995 pp96-104.
- [19] B A Wichmann. High Integrity Ada. SafeComp '97.
- [20] B A Wichmann. Some Remarks about Random Testing. To be published (available from the author).

A Document Details (not for publication)

A.1 Status

Draft.

A.2 Project

1INTTV00.

A.3 File

Stored on the Sun in file baw/scs/choice2.

A.4 History

- First draft written 5th May 1998.
- Revised to reflect comments by Tony Mansfield, 7th May 1998.
- Revised to reflect comments/approval by Dave Rayner, 14th May 1998.

A.5 Actions

- E-mail to Anca Vermesan.