The choice of computer languages for use in safety-critical systems

by W.J. Cullyer, S.J. Goodenough and B.A. Wichmann

The paper reviews the choice of computer language for use in safety-critical systems. The advice given reflects both civil and military requirements. A comparison is made between assembly-level languages, the language C, CORAL 66, Pascal, Modula-2 and Ada. It is concluded that a well defined sub-language is essential for use in safety-critical projects, and a guide is provided for project managers and designers on the characteristics which such a subset should possess.

1 Introduction

Computers are progressively invading every area of civil and military procurement. In hospitals, patients are monitored using equipment that contain microprocessors. Aircraft rely even more heavily on computers for navigation and safe operation. Military designers are working on advanced fighter aircraft which are inherently so unstable that human pilots cannot fly them without computing systems calculating the required deflections of the control surfaces every few milliseconds. Nuclear reactors and plants handling toxic chemicals use microprocessors and medium-sized mainframes to provide monitoring and alarm systems.

The applications listed above have one thing in common; design errors in the electronic hardware of the computers or mistakes in computer programs may result in disastrous accidents. During recent years, the name High Integrity Computing has been applied to such critical computing applications. In the worst case, accidents caused by design flaws may be of such severity that lives are lost or massive damage is done to the environment.

Typically in these applications, the role of the computer is to accept inputs from various sensor systems, perform some calculations or logic and then provide output to actuators. All designers of critical systems should know the consequences of component failure in service. Random failures of microchips, sensors and actuators are coped with by using redundancy; for example, triplicated 'autoland' systems in aircraft and quadruplicated energency shutdown systems in nuclear reactors.

In the scientific disciplines that relate to safety-critical computing, efforts within UK government research establishments are being concentrated in three areas. First, research is being carried out into methods of writing technical specifications for high-integrity hardware and software, using formal mathematical methods. Secondly, RSRE and the University of Southampton, amongst others, have been working on the problems of developing and verifying the correctness of high-integrity software, and NPL has been concerned with the validation of compilers [1] and floating point arithmetic [4]. Thirdly, a team from the University of Cambridge has devised new methods of designing provably correct microprocessor hardware. Interestingly, these methods themselves use highly complex, unproven computer programs. Related work in the US is mainly from computational Logic Inc. [5–7].

During the last seven or eight years, NPL and RSRE have provided consultancy in many areas of computer applications in which human life is at stake. Based on these interactions with industrial design teams, it is clear that very great care must be exercised in the choice of methods when designing highly critical 'black boxes'. Often the advice sought has related to the choice of computer language to be employed in the development of a specific product. This choice is only one element in the overall management judgements when starting a project. However, there are many other decisions about the techniques to be adopted which are equally crucial (see Section 2).

This paper attempts to provide advice to designers and managers on the choice of computer language to use when developing systems on which human life may depend. A comparison is made between the use of structured assembly languages, C, CORAL 66, Pascal, Modula-2 and Ada. This list should cover many of the civil and military projects which will enter development in the next few years. This paper makes it clear that 'unsafe' constructions exist in all known assembly and high-order languages. The flaws that exist may, in the worst case, lead to an unsafe system. Hence, the advice given favours the use of well defined subsets of the commonly available languages.

2 Methods of design for safety-critical systems

This topic must be addressed before debating the language issues. In summary, we believe that the integrity of

computer-based systems will be improved if the following principles are heeded:

- the safety-critical elements of the system must be identified by hazard analysis and clearly delineated in the top-level documents;
- there should be an overall rigorous, mathematical specification of the functionality of the safety-critical components, e.g. in Z [8–10] or VDM [11];
- all safety-critical elements must have a redundant design, i.e. be multichannel implementations, although not all the channels need be programmable;
- the software in these channels should be subjected to independent verification.

More details on these topics are given in a draft UK Defence Standard 00-55 [12, 13]. It follows from the list above that the computer language used for the implementation should be capable of being used in a formal mathematical environment. In addition, the syntax and semantics of this language must be known precisely enough to enable techniques such as static code analysis [14, 15] to be used, via a suitable translator.

Given the requirements for safety and the required coupling to formal methods of specification and verification, it is possible to draw up a list of questions which seek to establish whether a language, or sub-language, is a likely candidate. Attempts have been made to find a short list of technical questions which can be used as a filter during the selection of a language. The following Section gives the conclusions from these studies.

3 Questions to be asked when choosing a language

The questions below should be the minimum set considered by a project manager and design team when agreeing on the choice of computer language.

☐ **Wild jumps:** can it be shown that the program cannot jump to an arbitrary store location (i.e. can the control flow be totally determined)?
☐ **Overwrites:** are there language features which prevent an arbitrary store location being overwritten?
☐ **Semantics:** are the semantics of the language defined sufficiently for the translation process needed for static code analysis to be feasible?
☐ **Model of maths:** is there a rigorous model of both integer and floating point arithmetic within the language standard?
☐ **Operational arithmetic:** are there procedures for checking that the operational program obeys the model of the arithmetic when running on the target processor?
☐ **Data typing:** are the means of data typing strong enough to prevent misuse of variables?
☐ **Exception handling:** if the software detects a malfunction at runtime, do mechanisms exist to facilitate recovery? (e.g. global exception handlers, which may in themselves introduce hazards if used unwisely.)
☐ **Safe subsets:** does a subset of the language exist which is defined to have properties that satisfy these requirements more adequately than the full language?
☐ **Exhaustion of memory:** are there facilities in the language to guard against running out of memory at runtime? (e.g. to prevent stack or heap overflow.)
☐ **Separate compilation:** does the language provide facilities for separate compilation of modules, with type checking across the module boundaries?
☐ **Well understood:** will the designers and programmers understand the programming language sufficiently to write safety-critical software?

The question dealing with exception handling needs some extra comment. Programs can go wrong at runtime. Possible failures (exceptions) must be dealt with by the applications programmer. The usual solution is to classify the types of failure that are foreseen and design recovery mechanisms for each class. Languages such as Ada provide direct support for such designs, by offering built-in exception handling routines. A more radical approach is to try and design languages which are 'exception free', for example, Currie's NewSpeak [16].

The list of questions given here may be modified, based on experience over the next two to three years, but for the moment it will be used as a yardstick to measure, very approximately, the merits and pitfalls of candidate languages. The authors request details of any relevant experience from readers of this paper.

4 Languages available

4.1 Method of assessment

The assessments given in this Section attempt to differentiate between assembly-level languages, C, CORAL 66, Pascal, Modula-2 and Ada. Project managers may encounter proposals for the use of other languages. If so, our strong advice is that professional help should be sought. In order to give non-specialists some order of merit when comparing languages, an elementary notation has been adopted for appraising the answers to the questions listed in Section 3. The symbols used are

X: the facility is not provided, and this may result in equipment that is unsafe.
?: the language provides some protection, but there remains a risk of malfunction.
***:** sound protection is provided, and good design and verification should minimise the risk of serious incident.

The resulting Tables should not be regarded as fixed. New research and development of sub-languages policed by formal methods may tend to enhance particular assessment as we move into the 1990s. This is particularly true in relation to Ada, which at the moment is immature for this application area. All the assessments given in the Tables should therefore be treated as lower bounds, arising from the state of current scientific knowledge.

4.2 Assembly-level languages

Until recently, virtually all safety-critical computer-based devices were programmed at assembly level [17]. Even if compilers for high-order languages were available for the target processor, the design team and the certification authorities involved felt that the 'visibility' provided by

Table 1 Assessment of assembly-level language

Short title	Structured assembler	RSRE VISTA
Wild jumps	*	*
Overwrites	?	?
Semantics	?	?
Model of maths	?	*
Operational arithmetic	?	*
Data typing	?	?
Exception handling	X	?
Safe subset	?	*
Exhaustion of memory	*	*
Separate compilation	X	X
Well understood	*	*

assembly-level programming was a vital element in producing a safe system. This confidence has not been borne out in practice, for reasons that are discussed below. It is fair to comment that many designers use some form of a 'structured assembly language' [18, 19] when designing products, i.e. an elementary language which has some high-order language constructs but which nevertheless translates one-for-one into machine instructions.

Two possibilities are worth assessment. First, a number of UK defence contractors have devised or purchased the capability to write good programs in structured assembly languages, such as PL/M/-80 and SPADE 8080. These techniques have been considered in detail by Clutterbuck and Carré [20]. The second assessment considered below is the structured assembly language VISTA, designed by RSRE specifically for use with the new VIPER microprocessor. This has a more rigorous basis than other structured assembly languages and runs on the VIPER chip. Take the list of questions one by one.

Table 2 Assessment of the C language

Short title	C
Wild jumps	?
Overwrites	X
Semantics	X
Model of maths	X
Operational arithmetic	X
Data typing	X
Exception handling	?
Safe subset	X
Exhaustion of memory	?
Separate compilation	X
Well understood	?

- **Wild jumps:** structured assemblers offer good protection against wild jumps, as does VISTA.
- **Overwrites:** there is no bound checking in most assemblers, and so it is possible to write to an address outside of the intended range. This is also true for VISTA.
- **Semantics:** the semantics of structured assembly languages are reasonably well defined.
- **Model of maths:** it is very difficult to assess the arithmetic invoked in assembly-level languages. Side-effects, such as the toggling of overflow and carry flags, make interpretation difficult. VISTA is much stronger in this respect, in that it is based on a rigorous model of addition and subtraction.
- **Operational arithmetic:** on most target microprocessors the behaviour of elementary operational programs can be understood informally. Architectures such as VIPER are needed to produce high confidence.
- **Data typing:** there is some data typing in structured assembly languages but protection is incomplete.
- **Exception handling:** there is no recovery mechanisms in structured assembly-level languages. In the VIPER processor, exceptions such as integer overflow cause the chip to stop, recovery being via the external system. The VISTA language itself does not support the handling of exceptions.
- **Safe subsets:** structured assembly-level languages available commercially have been derived informally from parent languages that possess a number of dangerous constructs. Great care is needed when programming in the simplified structured language.
- **Exhaustion of memory:** providing that recursion is barred, structured assembly languages should avoid the risks inherent in running out of memory.
- **Separate compilation:** the linkers provided with the assembly-level languages may produce unexpected side-effects. It may well be better to work with monolithic programs.
- **Well understood:** designers' understanding of structured assemblers can be very good. However, there may be a mistaken belief that you are 'closer to the machine' and hence 'safer'.

Overall, it is acknowledged that the very careful use of structured assembly languages may be acceptable in projects where the risk to human life is comparately low.

4.3 The C language

This Section is based on a draft of the ISO Standard for C. This Standard has been produced to reflect the language implemented by existing C compilers. This approach is very different from the ISO Standards for Pascal and Modula-2, in which an attempt was made to provide a more stringent definition. A consequence of this approach with Pascal is

that, even eight years after the Standard was agreed, several significant implementations do not conform to the Standard. With C, this position should not arise, although the price paid for this is that the Standard is weak and most legal programs can behave differently on conforming systems.

☐ **Wild jumps:** the C equivalent of the case statement is achieved by means of a switch statement. If no matching statement is found to the switch expression (and there is no default), then no action is performed. This is more secure than the unchecked facility in Pascal and Modula-2. An insecurity in flow control exists in the calling of a procedure/function by means of a variable.
☐ **Overwrites:** since store access is implicitly via pointers and pointers can be formed without restriction, very few forms of access and assignment can be shown to be safe.
☐ **Semantics:** the language definition for ISO documents current practice, rather than defining a uniform behaviour.
☐ **Model of maths:** the language supports both signed and unsigned integer arithmetic. Unsigned integer arithmetic is modulo the word length without overflow detection and is therefore inherently insecure. Since C does not support subranges, it is difficult to show that any integer calculation cannot overflow. The sign of the result of integer division is defined by the implementation rather than the language (as in Pascal, Modula-2 and Ada). Division by zero is undefined, although recovery by a signal is possible.
☐ **Operational arithmetic:** the action taken on integer overflow is undefined, although recovery by a signal is possible. The variability permitted by the Standard will make effective validation very difficult.
☐ **Data typing:** the data typing is very weak in C. The type checking that does exist can be overridden by means of the cast operators.
☐ **Exception handling:** the C language defines six signals that can be used by a programmer to recover from a detected event (such as divide by zero). Unfortunately, there is no requirement that the corresponding events are detected, and hence the signal facility will depend on the properties of a particular implementation.
☐ **Safe subsets:** no safe subset seems feasible for the language.
☐ **Exhaustion of memory:** there is no specific signal for storage overflow, and hence any recovery seems unlikely.
☐ **Separate compilation:** the separate compilation system is insecure. However, good tools are conventionally used with C (e.g. 'make' and 'link'), which ensures a textual consistency of program texts. Conformance to the new ANSI/ISO Standard for C should help, since the facility for parameter specification eases the checking problem for compilers.
☐ **Well understood:** the C language is widely understood by the large (and growing) UNIX community. Hence, there is little problem in this aspect of the language; good courses and textbooks are available.

Overall, the use of C for safety-critical software must be very strongly deprecated. In particular, the language is significantly worse than an assembly language because many aspects of the definition are undefined, unspecified or vary between implementations. It might be thought that C++ would be a potential language for this area being a derivative of C with object-oriented design features. Since all C programs are supposed to be acceptable to C++ systems, no additional security seems possible. Of course, a subset of C++ might well be a contender, but the relationship of a subset of a superset to C is too tenuous to draw any conclusions at this stage.

4.4 CORAL 66

Industry now has access to validated CORAL 66 compilers, by virtue of the tests applied by the UK MoD. This gives some confidence that a particular compiler obeys the broad requirements of the language definition [21], which in itself is an informal document in the mathematical sense. However, little theoretical analysis of CORAL 66 has been attempted, largely because until recently it had not been used in the development of highly safety-critical equipment. Over the last few years, a handful of UK defence contractors have developed subsets of CORAL 66 for use in critical areas of procurement. Take the list of questions.

- **Wild jumps:** except for the SWITCH statement, CORAL 66 provides reasonable protection against jumping to an arbitrary location.
- **Overwrites:** the use of anonymous references allows any location to be overwritten. Such constructs should be excluded from safety-critical subsets.
- **Semantics:** the semantics of CORAL 66 are uncertain in a number of areas. Translators for static code analysis handle only subsets of the language.

Table 3 CORAL 66 and subset

Short title	CORAL 66	CORAL subset
Wild jumps	?	*
Overwrites	X	*
Semantics	?	*
Model of maths	?	?
Ooperational arithmetic	X	?
Data typing	?	?
Exception handling	X	X
Safe subset	X	?
Exhaustion of memory	?	*
Separate compilation	?	?
Well understood	?	*

- **Model of maths:** some work has been done on the analysis of fixed point arithmetic in CORAL 66, but there is no definitive model of the arithmetic.
- **Operational arithmetic:** there is little confidence in the precise mathematical behaviour of operational programs derived from unrestricted CORAL 66 source text.
- **Data typing:** user-defined data typing does not exist in CORAL 66. There is a small fixed set of types.
- **Exception handling:** there are no recovery mechanisms in CORAL 66.
- **Safe subsets:** subsets of CORAL 66 exist and have been found to be safer than unrestricted use of the language.
- **Exhaustion of memory:** Providing recursion is avoided, there should not be any problem at runtime.

Separate compilation: CORAL 66 does not have any standard method of separate compilation. Manufacturers have produced their own schemes for linking modules, which include limited type checking.

- **Well understood:** designers understand CORAL 66 text reasonably well (at least in the UK), and a certain degree of maturity is apparent in software delivered to the UK MOD.

Table 3 shows the markings, which are believed to be fair for the full language and for a typical sub-language, of the type used in recent UK defence contracts.

It is not surprising that the unrestricted use of CORAL 66 is not a good option. The language was designed 20 years ago, well before research had begun to show the characteristics needed for 'safe' languages. As the appreciably higher assessment for CORAL subsets indicates, it is possible to use CORAL more safely, if a rigorous code of practice is used to eliminate the bad features of the language. Provided static code analysis or a similar technique is employed to check the source text down to and including full semantic analysis, good CORAL implementations can be produced.

4.5 Pascal

As designed originally by Wirth and codified subsequently in the ISO Standard, Pascal is a comparatively well defined language. Nevertheless, as described in the answers to the questions below, it is perfectly possible to write dangerous programs in unrestricted Pascal. This has been demonstrated conclusively in a number of examples given by Wichmann [22].

However, the last seven years' work by Dr. Carré at the University of Southampton and at NPL have produced a detailed scientific knowledge of the semantics of Pascal and the execution of programs on a given target processor. From the work at Southampton has come SPADE Pascal, a subset of ISO Pascal which is suitable for safety-critical applications. Based on this theoretical knowledge, there is every reason to treat subsets of ISO as serious candidates when designing high-integrity systems.

Apply the now familiar questions.

☐ **Wild jumps:** there is a potential trap in the Pascal CASE statement, which may lead to a jump to an arbitrary location, but otherwise the control flow constructs are sound.

☐ **Overwrites:** Pascal is comparatively weak in the area of overwriting, since this can be done by indexing using an unchecked value or via an erroneous pointer.

TABLE 4 ISO Pascal and SPADE Pascal

Short title	ISO Pascal	SPADE Pascal
Wild jumps	?	*
Overwrites	?	*
Semantics	?	*
Model of maths	*	*
Operational arithmetic	?	*
Data typing	?	*
Exception handling	X	X
Safe subset	X	*
Exhaustion of memory	?	*
Separate compilation	?	?
Well understood	*	*

☐ **Semantics:** the semantics of ISO Pascal are well understood, and both ISO Pascal and the SPADE subset have a formal definition.

☐ **Model of maths:** there is a firm model for integer arithmetic but no such model for REAL arithmetic (conventionally, floating point).

☐ **Operational arithmetic:** the behaviour of integer arithmetic can be determined precisely and is checked by the compiler validation process. The behaviour of REAL arithmetic is not defined and, in consequence, the validation suite only performs a partial check.

☐ **Data typing:** Pascal types are well defined and compilers detect many erroneous assignments. However, variant records create holes in the protection.

☐ **Exception handling:** No exception handling is provided in Pascal. SPADE Pascal, analysed by static code analysis, will not encounter integer overflows due to the inclusion of range checks in the intermediate language.

☐ **Safe subsets:** as noted above, the SPADE Pascal subset is very well defined. This bars the use of variant records and severely limits constructs that employ pointers.

☐ **Exhaustion of memory:** with programs written in ISO Pascal there is no certain way of ensuring that the heap space is not exhausted at runtime. SPADE Pascal is deterministic in this respect.

☐ **Separate compilation:** Pascal does not provide a separate compilation system. Module systems provided by industry vary in security.

☐ **Well understood:** Pascal is readily understood by programmers and those projects that have used SPADE Pascal have found interpretation very easy.

It is clear that Pascal has similar defects to the other languages considered in this paper. What gives the language the edge, for the moment, is the intensive theoretical studies into its imperfections in universities and research laboratories. Table 4 shows the details of the assessment.

Therefore, it is concluded that SPADE Pascal has just about the same integrity as hoped for in the future from an Ada sub-language (see Section 4.7). However, currently the SPADE Pascal tools are more highly developed than their Ada equivalents.

4.6 Modula-2

The Modula-2 language has a substantial fraction of the power of Ada but is only of the same degree of complexity as ISO Pascal. In some respects, it can be regarded as a highly suitable language for safety-critical software, being strongly typed and with modules for information hiding. ISO has agreed to standardise the language, and this work is being undertaken by BSI. The key questions are answered for the language defined in this new Standard. It is not appropriate to answer the questions for the original language [23], since this definition is not precise enough.

- **Wild jumps:** the case statement is insecure unless all the possible options have been covered. It is easy to cover all cases by means of an 'otherwise' clause. In addition, a simple static check can ensure that all the cases have been covered (a useful warning in a compiler). Attempting to execute a case that has not been covered raises an exception, which is easy for an implementation to trap. Another form of insecurity in flow control is the calling of a procedure/function by means of a procedure/function variable. If the value of the variable has not been set, then an exception is raised. Unfortunately, in this situation, detection is difficult at runtime and virtually impossible (in general) statically.
- **Overwrites:** the potential insecurity of access via/to an uninitialised pointer is the same as for ISO Pascal. Access to array elements is also the same as for Pascal.
- **Semantics:** the language definition for ISO is being produced in VDM [11], and hence it is thought that this will provide a better basis for analysis than any other Standard language. VDM is underpinned by mathematics.
- **Model of maths:** the model of arithmetic used is that of an implementation-defined sub-range of the integers. This is rather more precise than Pascal. For floating point, it is hoped that the ISO Standard will be written to reference a newly proposed Standard for floating point [4].
- **Operational arithmetic:** if the limitations of a finite word length for integers is exceeded, then Modula-2 specifies that an exception is raised. The action is then dependent on the implementation. The type of action is checked by the validation procedures, as for Pascal. (A prototype Modula-2 validation suite is available, which will be used for formal validation once the Standard is agreed and the suite revised accordingly.)
- **Data typing:** although Modula-2 is a strongly typed language, there are three loopholes to the type rules:

Table 5 Modula-2 and a subset

Short title	Modula-2	Modula-2 subset
Wild jumps	?	*
Overwrites	?	*
Semantics	*	*
Model of maths	*	*
Operational arithmetic	?	?
Data typing	?	*
Exception handling	?	?
Safe subset	?	*
Exhaustion of memory	?	?
Separate compilation	*	*
Well understood	*	*

  - ☐ unsafe use of variant records, as in Pascal;
  - ☐ use of an explicit unsafe conversion function;
  - ☐ use of parameters of type WORD, which matches any parameter type.
- **Exception handling:** no recovery strategy is required in the proposed Standard. However, it is expected that the exceptional situations, which are simple for implementations to detect, will be mandated with at least one option in a compiler.
- **Safe subsets:** a safe subset would exclude case statements with uncovered cases and the three forms of type loopholes noted above.
- **Exhaustion of memory:** no provision is made for trapping stack overflow. The allocation of space on the heap is, in effect, in control of the user. Hence, the user can write allocation routines that ensure that the heap space is suitably bounded.
- **Separate compilation:** the separate compilation system is type-secure. This is checked by the validation procedures.
- **Well understood:** the Modula-2 language has been available for some years. The ISO Standard will differ mainly in being more rigorously defined. It is a relatively simple language and therefore should not present any difficulty to the designers and programmers of safety-critical software.

Table 5 needs to be interpreted with care. First, it applies only to the Modula-2 Standard, when agreed. Some assessment has been performed of existing implementations, which has shown that they are not appropriate for critical software. Secondly, although a Modula-2 subset looks good, it may lack adequate functionality for a specific application. For instance, the handling of low-level input-output may

require assembly language (this applies to other languages as well). In addition, a user-written space allocator for the heap, written to trap heap overflow, needs to use an unsafe feature of the language. Nevertheless, it is felt that of the Standard languages, Modula-2 is inherently more secure than the others considered here.

4.7 Ada

With the availability of validated compilers and simple Programming Support Environments (PSE), Ada is now a practical language for small to medium-sized projects. However, the semantics of parts of the Ada language have not been specified fully in the definition [24], and compiler writers are having to make pragmatic decisions in implementing certain constructs. Even in projects that are not safety-critical, discipline is required in the use of the language if reliable systems are to be produced. This is typical of any new high-order language in its first few years of use. After a while, designers learn which design techniques and language constructs should be avoided.

As regards subsets, or sub-languages, several attempts are under way in the UK to produce codes of practice that will constrain programmers to the use of those Ada constructs for which the semantics are certain. A number of the problems are listed by McGettrick [25]. Much work remains to be done for safety-critical systems, although 'Safe Ada' [26] and SPARK [27] show some promise of yielding a solution. Therefore, the assessment given below of the characteristics of a sub-language that could appear is of necessity, based on theoretical considerations rather than experience.

As before, list the answers to the key questions.

☐ **Wild jumps:** there are no constructs in Ada that allow a jump to an arbitrary location.

Table 6 Ada and a hypothetical sub-language

Short title	Ada	Ada subset
Wild jumps	*	*
Overwrittes	?	*
Semantics	?	?
Model of maths	?	*
Operational arithmetic	?	*
Data typing	*	*
Exception handling	*	*
Safe subset	X	?
Exhaustion of memory	X	?
Separate compilation	*	*
Well understood	?	*

☐ **Overwrites:** provided that all low-level facilities, such as UNCHECKED_CONVERSION, are avoided, there is no way an Ada program can overwrite a random location.

☐ **Semantics:** about 800 queries have been raised with the US Department of Defense over the definition of Ada. Probably enough is known to form a small sub-language, which can be translated accurately and allow static code analysis to be done, but it must be noted that uncertainties still exist about the definition.

☐ **Model of maths:** there is a good model of Ada arithmetic. The Ada compiler validation suite does some of the necessary conformity checks.

☐ **Operational arithmetic:** because of the theoretical work on Ada, the arithmetical behaviour of simple operational programs can be predicted.

☐ **Data typing:** the Ada strong typing rules are powerful and provide appreciably more security than in CORAL 66 or Pascal.

☐ **Exception handling:** all runtime errors in Ada result in exceptions being raised, with more than adequate means of writing handlers being provided to the programmer. This can be a two-edged sword, since dangerous recovery mechanisms may be designed.

☐ **Safe subsets:** as noted above, work is in hand to derive sub-languages from Ada.

☐ **Exhaustion of memory:** it is not easy to analyse an Ada program to establish that it will not run out of resources at runtime.

☐ **Separate compilation:** Ada has a firm separate compilation system, with type checking across module boundaries.

☐ **Well understood:** as Ada is so new, there must be doubts about the extent to which the language constructs are understood.

Overall, Ada passes through the filter of questions in a comparatively smooth manner. Table 6 shows the authors' assessments of its merits and problems, together with a theoretical marking for a high-quality sub-language. A more detailed assessment of Ada appears in Reference 3.

Given the above assessment, it might be thought that the use of Ada in an unrestricted manner would be a reasonable option. For the reasons given in this paper, the authors believe that it is too early to use the unrestricted Ada language when developing safety-critical systems. Preconditions for the use of the language should be greater industrial experience of using Ada, on less critical systems, and the development of a sound sub-language. A tool, which could provide a high degree of confidence in Ada programs (suitably written), is proposed in Reference 2.

5 Conclusions and recommendations

The clearest conclusion to be drawn from this paper is that a code of practice, designed to enforce a subset of the chosen language, is an essential element when implementing safety-critical software. Considering the assessment in Tables 1 to 6, the position, regarding the choice of computer language, is judged to be as follows.

• The languages that design teams should consider as candidates for use in high-integrity systems are, according

to the assessments in this paper, and in descending order of merit

- ☐ ISO Pascal subsets supported by validation tools, (e.g. SPADE Pascal);
- ☐ an Ada sub-language, when available;
- ☐ a Modula-2 sub-language, when available;
- ☐ a CORAL 66 subset.

If used carefully and backed up by the use of at least static code analysis (and preferably formal proofs), these languages should enable designers to satisfy the requirements of national and international Standards for safety-critical systems.

- If analysis of the hazards suggests that the risks are comparatively low, the second group of languages that may be considered includes, in no particular order

- ☐ structured assembly languages;
- ☐ DoD Ada, with minimal restrictions;
- ☐ ISO Pascal, with minimal restrictions;
- ☐ Modula-2, with minimal restrictions.

In the authors' judgement, systems developed using any one of these second choice languages will *not* satisfy the strictest requirements of national and international Standards for safety-critical equipment.

- Based on the assessments in this paper, the use of the following languages is to be deprecated when safety is an issue:

- ☐ unrestricted use of assembly languages;
- ☐ C (despite its many adherents);
- ☐ unrestricted use of CORAL 66.

As acknowledged at several points in this paper, the above lists will vary as time progresses and revised forms of this paper, or something equivalent, will need to be issued about every three years, to give an updated view to managers and design teams.

It is interesting to speculate on future directions in this area. More secure languages would help, such as NewSpeak, but development here is very slow as gaining acceptance for a new language is very difficult. More realistically, tools can be used to ensure compliance with a safe subset; this paper shows that such subsets are useful and much more secure than full Ada or Pascal, for example. One of the authors has his own solution to the insecurities in Ada [2] which is tool-based, rather than introducing a new language.

If program proof is to be attempted, then good design is essential. Developments such as object-oriented design can help but should not be thought of as a panacea. A functional programming style aids program proof significantly, but such a strict style leads to code with unacceptable performance in many cases.

This paper does not consider the problem of the reliability of high-level language compilers. It should be noted that this is a serious issue, as revealed by a special tool [28]. Proven software components may provide an answer, as in Micro-Gypsy [7].

6 Acknowledgment

7 References

[1] WICHMANN, B.A., and CIECHANOWICZ, Z.J. (Eds.): 'Pascal compiler validation' (Wiley, 1982)

[2] WICHMANN, B.A.: 'Low-Ada: an Ada validation tool'. NPL Report DITC 144/89, 1989

[3] WICHMANN, B.A.: 'Insecurities in the Ada programming language'. NPL Report DITC 137/89, January 1989

[4] WICHMANN, B.A.: 'Towards a formal specification of floating point', *Comput. J.*, 1989, pp. 432–434

[5] MOORE, J.S.: 'PITON: a verified assembly level language'. Computational Logic Inc., September 1988

[6] SMITH, M.K., CRAIGEN, D., and SAALTINK, M.: 'The nanoAVA definition.' Computational Logic Inc., June 1988

[7] YOUNG, W.D.: 'Verified compilation in Micro-Gypsy.' Computational Logic Inc., October 1989

[8] SUFFRIN, B.: 'Z Handbook, draft 1.1'. Oxford University Programming Research Group, October 1985

[9] HAYES, I.: 'Specification case studies' (Prentice-Hall, 1987)

[10] SPIVEY, J.M.: 'Understanding Z' (Cambridge University Press, 1988)

[11] JONES, C.B.: 'Systematic software development using VDM' (Prentice Hall, 1989)

[12] MOD: Policy Statement on Defence Standard 00-55. Director of Standardization, MoD, March 1988

[13] MOD: Interim Defence Standard for the Procurement of Safety-Critical Software. Director of Standardization, May 1989

[14] CARRÉ, B.A.: 'SPADE static code analysis manual'. Programm Validation Ltd. April 1985

[15] RTP. MALPAS Users' manuals. Rex, Thompson and Partners, April 1986

[16] CURRIE, I.F.: 'NewSpeak — an unexceptional language', *Softw. Eng. J.*, 1976, **1**, (4), pp. 170–176

[17] CULLYER, W.J.: 'Hardware integrity', *Aeronaut. J.*, August/September 1985

[18] WIRTH, N.: 'PL360: a programming language for 360 computers', *J. ACM*, 1968, (1), **15**

[19] WICHMANN, B.A.: 'PL515: an Algol-like assembly language for the DPP 516'. NPL Report, April 1970

[20] CLUTTERBUCK, D.L., CARRÉ, B.A.: 'The verification of low-level code', *Softw. Eng. J.*, 1988, **3**, (3), pp. 97–111

[21] IECCA: IECCA Official definition of CORAL 66. HMSO, 1970

[22] WICHMANN, B.A.: 'Notes on the security of programming languages' in LIBBERTON, G.P. (Ed.): '10th Advances in Reliability Technology Symp: (Elsevier, 1988)

[23] WIRTH, N.: 'Programming in Modula-2' (Springer-Verlag, 1988)

[24] ICHBIAH, J.D. *et al.*: Reference manual for the Ada programming language. ANSI MIL-STD 1815A (also ISO 8652)

[25] McGETTRICK, A.: 'Program verification using Ada' (Cambridge University Press, 1982)

[26] HOLZAPFEL, R., and WINTERSTEIN, G.: 'Ada in safety-critical applications'. Ada Europe Conference, 1988

[27] CARRÉ, B.A., and JENNINGS, T.J.: 'SPARK — The Spade Ada Kernel.' University of Southampton, March 1988

[28] WICHMANN, B.A., and DAVIES, M.: 'Experience with a compiler testing tool'. NPL Report DITC 138/89, 1989

The paper was received on 16th October 1989.

W.J. Cullyer is Lucas Professor of Electronics, with the Department of Engineering, University of Warwick, Coventry CV4 7AL, UK; S.J. Goodenough is with the Air Defence and Air Traffic Control Group, RSRE, Malvern, Ministry of Defence, WR14 3PS, UK; and B.A. Wichmann is with the Division of Information Technology and Computing, National Physical Laboratory, Teddington, Middlesex, TW11 0LW, UK.