

Practical Experiences of Safety- and Security-Critical Technologies

Peter Amey and Adrian J Hilton

Praxis Critical Systems Ltd., 20 Manvers Street, Bath BA1 1PX; Tel: +44 1225 823761; email: {peter.amey/adrian.hilton}@praxis-cs.co.uk

Abstract

In this article we identify the special properties of systems intended for use in ultra-reliable domains and the qualitative shift in development methods needed to achieve those properties. The advantages (and disadvantages) of Ada are introduced in the contexts of the ISO HRG report on High-Integrity Ada and of the SPARK sub-language. The demands of common, important development standards are described together with appropriate and cost-effective techniques for meeting them. Finally project experience illustrating successes in meeting the main standards is discussed.

Keywords: high integrity, safety, security, case study, SPARK, Ada.

1 Introduction

Safety- and security-critical systems have a fundamental common property, that they are not just correct and reliable but that they can be *shown* to be correct and reliable. Most modern critical systems contains software, and often (though not always) the system relies on the operation of the software to achieve safety or security. In this case the software is designated *safety-critical* or *security critical*, and is often termed generically *high-integrity* software.

High-integrity software is code where reliability is more important than efficiency, cost, time-to-market or functionality. Examples are security systems, financial systems where down-time incurs a large cost, and cases where a costly one-off mission capability may be lost such as a Mars Lander.

The unique characteristic of high-integrity software is the need to be able to show, before there is any service experience, that a system will meet its requirements. It is a problem qualitatively different from normal software, since the standards involved may be very high (e.g. requiring the equivalent of no more than 1 failure every 114,000 years of operation). It is not enough to be just “more careful” in development!

In this article we consider how such assurance may be obtained for the software component of a system, in particular for software components written in Ada. We draw on our experiences with illustrative case studies in developing and verifying safety-critical and security-critical systems.

2 Methods of Verification

We define the following terms:

- *verification* as “the process of determining that a system (or its component) meets its specification”;
- *validation* as “the process of determining that a system is appropriate for its purpose”; and
- *certification* as “persuading an external regulatory body that a set of specific requirements have been met and/or processes followed”.

Techniques of verification include:

- inspections / reviews, e.g. requirements or code;
- testing, e.g. requirements tests or unit tests; and
- analysis e.g. flow analysis, model-checking or partial program proof.

Inspections are uniquely flexible, since the primary tool is the engineer, and can be done early on. However they are inherently informal and fallible, and what inspection is feasible is limited by the *semantic precision* of the object being inspected and the *semantic gap* between the objects being compared.

(Dynamic) testing spans the entire development testing and can potentially identify errors in requirements, specifications, code, the compiler and the hardware. However, exhaustive testing is rarely feasible or even possible, and for high-integrity systems the high levels of assurance required the testing time becomes impractical. Littlewood [1] and Butler [2] note the limitations of Bayesian testing and the implications for reliability of any failures during a very long testing process.

There are also the practical problems arising from the need for a complete or mostly complete system before testing can begin. The testing phase of a system development is frequently a bottleneck for the entire development, and over-running testing is an expensive risk for most projects.

Analysis, by contrast, can be conducted early on in the development process and can establish properties that cannot be shown statistically, such as the proof of absence of run-time errors or freedom from timing deadlocks. However it can only *compare* objects (e.g. the code against the specification), and what can be achieved is limited by the precision of the descriptions and notations used.

Showing “correctness” is therefore harder than building correct systems; the key is to use a range of verification techniques. Since bug detection and correction is expensive, the focus should be on:

- preventing bugs from occurring;
- using techniques to find bugs early on; and
- using final testing as a *demonstration of correct behaviour* rather than as a method of finding bugs.

3 Reliable programming

The language in which we choose to program will affect the way we think about the program. A developer is less likely to think about abstractions if programming in machine code or assembler; she will think more in terms of the target machine if programming in C. For larger systems, language support for abstraction and encapsulation is vital if the program is to remain manageable and verifiable.

3.1 Formality and uncertainty

Machine code is *formal*, in that the target machine provides an operational semantics for it. However we cannot normally reason about the code, we can only observe its behaviour. Early reasoning about program correctness saves money and reduces risk, but is hampered by the lack of formality of most programming languages.

Typical causes of uncertainty in programming languages include deficiencies in the language definition (e.g., the semantics of integer division in C) and deliberate implementation freedoms (e.g. order of evaluation of expression components and parameter passing mechanisms). This leads to either:

- *ambiguity*, where program behaviour cannot be predicted from the source code; or
- *insecurity*, where violations of a language rule cannot be detected.

Ambiguities are resolved by compiler authors. Insecurities are left for the user to discover. Neither of these situations are attractive, but there are possible solutions:

- invent new languages without these problems;
- work with *dialects* associated with compilers; or
- use logically coherent language *subsets* to overcome ambiguities and insecurities.

When inventing new languages the very small user base leads to poor or non-existent tools, staff shortage and a lack of training support; this approach is impractical for most projects.

3.2 Dialects

When using a dialect, the first step is to find out what your compiler does. Its behaviour can then be documented and included in coding standards and review checklists. This can be an effective solution, but there are problems.

First, the compiler’s behaviour must be known; it must also be consistent. If it changes with new releases, or between the host and the target, then the dialect may be invalid. Developers must also know when it matters; we must recognise that implementation-dependent behaviour is present and know what the compiler’s behaviour is in this case.

You cannot expect to avoid all anomalous behaviour by this method, but it may be valuable in special cases e.g. small, specialised processors where only one company provides support, or for an established compiler where a small number of problems are known but service experience provides confidence in the rest of its behaviour.

3.3 Subsets

The need for subsetting was described by Wichmann [3]: “No currently standardised language could be recommended without reservation for the most critical applications without subsetting”. To be useful, however, language subsets need to be:

- simple;
- application oriented;
- predictable;
- formally verifiable; and
- be supported by sound tools.

A subset comprises a base language, a set of troublesome language features which are removed, a set of limitations on the way that remaining features may be used, and optionally the introduction of *annotations* to provide extra information. We can construct subsets that vary on four axes: precision (security and lack of ambiguity), expressive power, depth of analysis possible and the efficiency of the analysis process. There are complex trade-offs in this model.

The fundamental trade-off is between the discipline which programmers accept in order to reduce bug *insertion*, and the effort which they are prepared to make in bug *detection*. For example, unrestricted C provides little or no protection from bug insertion, whereas Ada requires extra discipline (e.g. strong typing) which reduces the bug insertion rate.

A qualitative shift in what is possible will only occur when the precision of the subset becomes exact.

3.4 MISRA C

MISRA-C is a C subset defined by the UK motor industry research association (MIRA) and its associated software reliability association (MISRA). It comprises 127 “rules” presented in the form of a coding standard, and its designers regard it as suitable for “SIL 3” systems; they recommend Ada or Modula-2 if available.

The base language of MISRA C is ISO/IEC 9899:1990 + the 1995 technical corrigendum. It removes troublesome C features such as pointer arithmetic, and imposes limitations such as all *switch* statements having to have a final *default*. It does not use extra annotations.

There was no attempt to make the MISRA C subset logically coherent and free from ambiguity and insecurity, and its machine checkability is well short of 100%. Some rules cannot be machine-checked at all, e.g. Rule 4: “Provision should be made for appropriate run-time checking”, and the exact meanings of some other rules are unclear and much debated. Nevertheless, following the subset rules should prevent many common C errors.

3.5 The Ada HRG

The Ada HRG is an ISO committee under WG9 to investigate high-integrity Ada issues, responsible for interpreting and developing Annex H of the LRM and focused on developing Ada in the high-integrity field. Its members include tool vendors, users, Government (e.g. the UK Ministry of Defence) and academics.

HRG report ISO/IEC JTC1/SC22/WG9 “Programming Languages – Guide for the Use of the Ada Programming Language in High Integrity Systems” [4] does not define a subset but identifies the basis on which a subset might be selected. The general approach is to identify verification techniques, then compare each technique with Ada language features and assess the level of compatibility.

Particular points of interest are exceptions and tasking. Exceptions are *essential* for high-integrity applications but must be *avoided* because of the difficulties they cause for flow and symbolic analysis. HRG had to reconcile these views; propagation of exceptions is clearly a key issue. For tasking they chose to define the “Ravenscar Profile”, of which more later.

3.6 SPARK and other Ada subsets

SPARK [5] is a sub-language of Ada with particular properties that suit it to the most critical applications:

- completely ambiguous;
- free from implementation dependencies;
- all rule violations are detectable;
- formally defined and tool supported.

The base languages are ANSI/MIL-STD-1815A-1983 (SPARK 83) and ISO-8652;1995 (SPARK 95). SPARK removes the troublesome language features (e.g. tasking, access types), limits the way remaining features may be used (e.g. limitations on the placement of exit and return) and introduces annotations to provide extra information.

SPARK annotations describe program design information. They are related to executable code by static-semantic rules, which are checked mechanically by the SPARK Examiner tool. This tool also checks language subset compliance, does system-wide data flow and information flow analysis, formal verification including proof of safety and security properties, and can demonstrate prior to execution that a program is “exception free”. SPARK is compatible with both HRG guidance and compiler-defined high-integrity subsets.

Other Ada subsets tend to fall into two groups: informal coding standards involving fairly arbitrary exclusion of language features, such as System “Safe Ada”, and subsets generated by compiler back-end and run-time library considerations. The latter are exemplified by GNAT Pro High Integrity Edition, C-Smart and Raven.

4 An overview of standards

There are many standards relating to software development for safety-related or security-related systems. Each domain (e.g. rail, aerospace, automotive) has its own standards. They vary in their power to mandate or recommend particular practices. In this section we discuss some representative standards and their implications for software development practice.

4.1 Software development techniques

Common aspects of standards’ recommended or mandated techniques for software development include:

- hazard or risk analysis;
- defining safety integrity levels;
- defining the system lifecycle;
- formal methods for requirements and design;
- modelling various system properties;
- choice of language, RTOS etc.;
- validation, verification and testing (VVT); and
- accumulation of evidence in a safety case.

We now examine selected techniques in more detail.

4.2 Hazard analysis

Hazard analysis is the process of identifying hazards and their causes. A *hazard* is defined by Leveson [6] as “a certain state of a system that, combined with other conditions in the environment, will lead inevitably to an accident or loss”. An example of a hazard for a car is “the wheels fall off while the car is travelling at speed”. Each hazard in a system has an estimated probability of occurrence and an associated severity.

Associated terms are *risk* and *safety*. *Risk* is a combination of probability and severity: the above hazard is severe, but (hopefully) very unlikely in most modern cars, so does not carry great risk. There are three main “risk regions”: broadly acceptable, As Low As Reasonably Practical (known as ALARP, of which more later) and intolerable. *Safety* is freedom from unacceptable risk.

Various formalised hazard analysis techniques exist, such as Hazard and Operability (HAZOP), Failure Modes and Effects Analysis (FMEA) and Fault Tree Analysis. Their common theme is the creation and maintenance of a hazard log, detailing the identified hazards and stating what steps have been taken to mitigate them.

4.3 Risk and ALARP

Risk itself is regulated (in the UK) on the basis of being reduced As Low As Reasonably Practical (ALARP). This stems from a UK Court of Appeal decision on the 1949 case *Edwards vs. The National Coal Board* where Judge Asquith noted :

“...a computation must be made by the owner in which the quantum of risk is placed on one scale and the sacrifice involved in the measures necessary for averting the risk (whether in money, time or trouble) is placed in the other, and that, if it be shown that there is a gross disproportion between them - the risk being insignificant in relation to the sacrifice - the defendants discharge the onus on them.” [7]

4.4 Safety integrity levels

It is common when developing against a safety standard for the system to be assigned a *safety integrity level* (SIL), typically in the range 1 (low) to 4 (high). The techniques (and costs) for system validation will be defined by the standard for each SIL.

A SIL is a general indicator of the quality required for design / build processes. It is applicable to both software and hardware, and indicates the probability that the system meets its requirements and avoids systematic failure. However, it is often misunderstood and misused.

It is important to note that if software is developed to e.g. SIL 4 for a particular system then it is *not* automatically SIL 4 for a different system. The hazards against which the software was originally may not include all the hazards of the new system, so although the software might be high quality it has not been shown to be safe or secure in the new environment.

4.5 The safety case

A safety case is a collection of evidence for safety of a system in the form of an argument for overall safety. It will relate various verification and analysis activities to avoidance or mitigation of the system hazards. Typically a safety case may contain or reference:

- the configuration management plan
- the software verification plan
- the software quality assurance plan
- standards for design and coding
- specifications for requirements, design and tests
- the results of software verification
- a software configuration index; and
- a traceability matrix.

5 DO-178B

RTCA DO-178B “Software Considerations in Airborne Systems and Equipment Certification” [8] is a set of guidelines produced by the civil avionics industry for good practice in producing avionics software. It is neither a

process document nor a standard, but in practice it has been endorsed by the FAA and JAA and is regarded as the “Bible” for civil avionics software.

DO-178B defines five levels of criticality for software based on consequence of failure, from E (no effect) through to A (catastrophic) in a similar approach to SILs. Only levels A and B are regarded as safety-critical.

5.1 Qualification and verification

For the output of each development step, the standard requires that either the tool that produced that output (e.g. the compiler) be *qualified*, or the output itself (e.g. object code) be *verified*. Since most tools in the object generation path are not qualified, the object code itself must be verified. This is achieved by specified test coverage criteria according to the criticality level. The coverage can be of the source code rather than the object code if source-to-object traceability can be achieved.

5.2 Coverage

In modified decision/condition branch coverage (MC/DC) testing, each value independently affecting a decision must be executed. In all cases the guidelines require coverage analysis to show that the coverage required has been attained. At Level A, the coverage must take place on the object code if the object code is not *directly traceable* from the source code – if Ada run-time checks are turned on in the compiler, for instance, then it is unlikely that such traceability exists.

Note that it is often possible to “cheat” on coverage. One way of reducing the number of tests required in Ada is to translate an if-else if – else block into a single lookup into a constant array, reducing many tests (one for each branch, for instance) to a single test – there is no concept of “covering” the data in such an array. For this reason, and others, coverage should not be regarded as a gold standard.

5.3 DO-178B Critique

DO-178B is a software correctness standard, not a system safety standard; there is no relation of the software to the system hazards, the developer can only state “the whole box has been tested to level A”.

There is undue emphasis on testing activities in the document. This may reflect its age (released in 1992) but means that there is little or no credit for analysis and review which are both worthwhile activities. It can create an environment where getting to the testing phase quickly is seen as the prime aim, and where MC/DC is widely misinterpreted as “exhaustive” testing when we have seen how it can be circumvented to some extent.

The role of the Designated Engineering Representative (DER) is a specialist designated by the FAA is to establish compliance with the DO-178B process, not system safety. In this respect DO-178B is very different from more modern safety-related standards.

5.4 Economic arguments

The cost of testing to DO-178B rises as the criticality level rises. Most testing work needs a test rig, the development of which is a back-end high-risk process. MC/DC requires many more tests than statement coverage; one estimate is that it multiplies the cost of testing by 5.

As noted, the required testing for level A is likely to be harder in Ada than in C. Is the best solution therefore to “hack in C”? No – this is because it is much cheaper to adopt a process that reduces errors *before* going into formal test. This was the approach taken when developing software for the C-130J Hercules II aircraft.

5.5 C-130J development

The C-130J was a project by Lockheed Martin which aimed to produce a much-improved version of the venerable C-130 Hercules transport aircraft. It included a new propulsion system with 29% more thrust and 15% better fuel efficiency, advanced avionics and control systems, two mission computers and two backup bus interface units to give dual redundancy to the aircraft systems.

The development process emphasised “Correctness by Construction” (CbC), focused on “doing it right first time”. The requirements of DO-178B were kept in mind but not allowed to dominate the development. Development was driven by verification, aiming to verify objects as soon as they were created.

Formality was introduced early, in the specification phase (using the CoRE requirements engineering process and Parnas table to specify in/out mappings) and in the programming language (Ada 83 and SPARK). There was a close map from the requirements to the code: “one procedure per table”. Static analysis (with the SPARK Examiner) was part of the development process

5.6 C-130J results

There was an unexpected benefit of using SPARK; the language requires all data interactions to be described in annotations, but some interactions were not clearly specified in the CoRE tables (e.g. validity flags). The coders could not ignore the problem, and so the requirements/specification document became a dynamic document as ambiguities were resolved during coding.

Early static analysis picked up a number of errors which would have been unlikely to surface during testing. The testing required by DO-178B was greatly simplified as a result. Very few errors were found during testing, which was done for less than 20% of the normal industry cost, and the overall cost of the level A system was half that of typical non-critical system development cost.

5.7 C-130J UK verification

The UK Ministry of Defence commissioned Aerosystems International to perform retrospective static analysis of *all* the C-130J critical software, using a variety of tools e.g. the SPARK Examiner and SPADE toolset for proving SPARK

code against CoRE, and MALPAS for Ada. All anomalies were investigated and “sentenced” by system safety experts. The results were instructive:

- significant safety-critical errors were found by static analysis in code developed to DO-178B Level A;
- proof of SPARK code was shown to be cheaper than other forms of static analysis performed;
- SPARK code had only 10% of the residual errors of full Ada, and Ada only 10% of the residual errors of C; and
- there was no statistically significant difference in residual error rate between DO-178B Level A, Level B and Level C code.

The results of the C-130J development are detailed by Sutton and Croxford [9] and Amey [10] and compared by German [11] to the results of other development methods.

6 UK Defence Standards

The UK Ministry of Defence uses Defence Standard 00-56 Issue 2 [12] as its primary safety standard. This references Defence Standard 00-55 for specific requirements and guidance on developing safety-related software. 00-56 is a system-wide standard and defines the safety analysis process to be used to determine safety requirements, including SILs. It promotes an active, managed approach to safety and risk management.

6.1 Defence Standard 00-55

00-55 Issue 2 is applied to software components of various SILs. It defines the process and techniques to be followed to achieve a level of rigour appropriate to the SIL. It requires the production of a software safety case to:

- justify the suitability of the development process, tools and methods; and
- present a justification based on objective evidence that the software satisfies the safety aspects of the software requirement.

Compared to DO-178B, 00-55 puts a heavy emphasis on the use of formal methods and proof. It still requires stringent testing. It uses integrity levels S1 to S4 rather than criticality levels E to A, but the main difference is that 00-55 focuses on software safety whereas DO-178B focuses on software correctness.

6.2 SHOLIS – a 00-55 project

The Ship Helicopter Operating Limits Instrumentation System (SHOLIS) is a system for assessing and advising on the safety of helicopter flying operations on board Royal Navy and Royal Fleet Auxiliary vessels. Power Magnetics and Electronic Systems Ltd. was the prime contractor. Praxis Critical Systems developed all the operational software to *Interim* Defence Standard 00-55 (1991).

SHOLIS is a dual redundant system, implementing fault tolerance by hot-standby. The two data processing units

(DPUs) are located at opposite ends of the ship to increase tolerance of battle damage, and are not synchronised in any way. It was the first system with software developed to Interim 00-55 at S4.

The final requirements numbered 4000 statements, with 300 pages of Z, English and mathematics. The code was 27,000 lines of non-blank non-comment Ada plus 54,000 lines of SPARK flow annotations and 20,000 lines of SPARK proof annotations.

6.3 SHOLIS verification

The major verification activities were:

- review and Z proof of the System Requirements and System Design;
- SPARK static analysis and proof of the code;
- worst-case timing, stack-use and I/O estimates;
- module, integration and system validation tests based on Z specifications and requirements;
- testing to cover 100% statement and MC/DC;
- acceptance, endurance and performance testing; and
- review of everything with a documentation trail.

As of January 2001 the system had passed all system validation, endurance and performance tests. The acceptance test for the customer was completed with no problems, and following sea trials some cosmetic changes were requested but no faults, crashes or unexpected behaviour were reported.

6.4 SHOLIS conclusions

Z proof and system validation tests were the most cost-effective phases at finding faults. Traditional module testing was arduous and found few faults, except in fixed-point numerical code. The strong static analysis removed many common faults before they could be tested. Software integration work was trivial. More detailed conclusions and fault metrics are given by King et al. [13].

The proof work demonstrated proof of exception freedom on the whole system, which was a significant win. In the proof work, the major lesson at the time (1998) was that a big computer is far cheaper than the time of the engineers using it. Moving on to today, the proof simplification work would be better done distributed on personal workstations, reducing the original simplification time by 1-2 orders of magnitude.

The conclusion of the SHOLIS development work is that the use of formal techniques such as proof and static analysis not only satisfies the stringent requirements of UK defence standards, but provides a real payback in terms of reduced testing time and increased system assurance.

6.5 00-56 revision

Defence Standard 00-56 is being revised, and the second public draft for comments was released in January 2004.

The new version incorporates Defence Standards 00-55 and 00-54 (safety-related electronic hardware) merged under the title “Programmable Systems”. We will briefly examine the new approach in this draft standard, mindful that it may change before release as Issue 3.

The approach in the new 00-56 is to require the developer to provide evidence that the system is safe for its intended purpose throughout its life. This will include a safety case, an auditable safety management system, risk assessment, hazard analysis, and a demonstration that risk is tolerable or ALARP. The developer must define the safety standards that they will meet, and then demonstrate compliance.

The Code of Practice in 00-56 refers to the software as a Safety Related Programmable System (SRPS). It requires *validated* safety arguments or claims for the SRPS, supported by:

- direct evidence from deductive analysis, demonstration and review;
- process evidence from following good practice; and
- qualitative evidence from good design.

Tools and processes which might undermine the SRPS integrity (e.g. compilers or analysis tools) must be qualified in some way. The standard also explicitly requires consideration of counter-evidence i.e. testing and in-service failures.

7. The Common Criteria

Security-critical applications are as important as safety-critical applications. We now consider the international Common Criteria security evaluation guidelines, and see how a security application (the Mondex Certification Authority) was developed and verified.

7.1 Common Criteria concepts

The Common Criteria for IT Security Evaluation [14] aims to set a “level playing field” for all developers in the participating states, and aims for international mutual recognition of evaluated products. It combines and replaces the US “Orange Book” and the UK ITSEC scheme.

The Common Criteria (CC) defines two types of IT security requirement:

- functional (defining the behaviour of a system or product); and
- assurance (for establishing confidence in the implemented security functions – is the product built well and does it meet its requirements?)

The CC defines seven Evaluation Assurance Levels – EAL 1 (lowest) through EAL 7 (highest). An EAL defines a set of functional and assurance components from the CC documents which must be met. EAL 7 roughly corresponds to ITSEC E6 and Orange Book A1.

7.2 The MULTOS CA

MULTOS is a multi-application operating system for smart cards. It allows applications to be loaded and deleted dynamically when the card is “in the field”. To prevent forging or the use of invalid applications, applications and card-enabling data are signed by the MULTOS Certification Authority (CA). At the heart of the CA is a high-security computer system that issues these certificates.

The CA has some unusual requirements:

- a target of 6 months between reboots and warm stand-by fault tolerance;
- a lifetime of decades, and must be supported for that long;
- to be tamper-proof and subject to the most stringent physical and procedural security; and
- meet the requirements of UK ITSEC E6.

All the requirements, design, implementation and on-going support for the CA were done by Praxis Critical Systems.

7.3 MULTOS CA development process

The overall development process conformed to ITSEC E6, minimising the reliance on COTS by assuming arbitrary but non-Byzantine behaviour. The COTS components (e.g. Windows NT, Backup tool and SQL Server) were not certified.

The security policy and top-level specification was formalised in Z, although no formal proof was carried out. The system was implemented in a mix of SPARK Ada, full Ada 95, Visual C++ and SQL, and tested top-down with coverage measurement. The mixed-language approach was due to selecting “the right tools for the right job” – implementing a GUI in SPARK Ada was clearly as wrong as implementing security-critical functions in C++.

The use of SPARK in particular was because it is almost certainly the only industrial-strength language that meets the requirements of ITSEC E6. Full Ada 95 was used where the necessary constructs (e.g. tasking) were not in the SPARK language at the time; even then, the use of Ada was “Ravenscar-like” in using simple, static allocation of memory and tasks.

7.4 MULTOS CA results

The use of Z for the formal security policy and system specification helped to produce an indisputable specification of functionality, and using Z, CSP and SPARK “extended” the formality into the design and implementation. The top-down incremental approach to integration and test was effective and economic. High-security systems incorporating COTS are possible, and indeed probably necessary.

Note that the CA was not formally evaluated against ITSEC E6; nevertheless, there was a clear benefit in security and reliability from developing to this standard.

8 Compilers and runtime

Having established and demonstrated approaches to developing high-integrity systems for the safety and security domains we now examine how developers should select a compiler and run-time for such a high-integrity system.

8.1 High-integrity compilers

A high-integrity Ada compiler should have:

- Annex H support;
- evidence of qualification against the Ada standard;
- optimisation and other switches;
- available and competent support;
- runtime support for high-integrity systems; and
- support for object code verification.

The HRG report [4] recommends validation of appropriate annexes – almost certainly A, B, C, D and H. Annex G (Numerics) may also be applicable for some systems. It does *not* recommend use of a subset compiler, although it recognises that a compiler may have a mode in which a particular subset is enforced. The main compiler algorithms should be unchanged in such a mode.

8.2 Compiler features

Annex H support and available pragmas will vary among compilers. One of the most important considerations is whether the compiler vendor has documented implementation decisions – demand this information from the vendor, and if they cannot or will not supply such information then find out why not!

“Qualification” of a full compiler as per the DO-178B (or similar) definition is beyond our reach at the moment. The pragmatic alternative is to combine:

- avoidance of “difficult to compile” language features in the high-integrity subset used;
- in-service history of the compiler;
- choosing the “most commonly used” compiler options;
- a study of the compiler faults history and database;
- verification and validation of the system; and
- object code verification as a last resort.

8.3 Runtime systems

In delivering a high-integrity system, a run-time library (RTL) must be verified to the same level of assurance as the rest of the application. Most Ada compiler vendors have responded to this need with products for Ada 83 and Ada 95. There are three main approaches to certifiable runtime systems: “small and certifiable”, Ravenscar or no runtime.

The “small and certifiable” approach is largely aimed at meeting the requirements of DO-178B up to and including

level A systems. The technical goal is to eliminate language features with an unacceptably large runtime impact, e.g. tasking, heap allocation, exceptions, predefined I/O. Examples are Aonix C-SMART and Rational VADSSc for Ada 83.

Ravenscar is a “profile” of tasking and other features in Ada 95 which is appropriate for high-integrity and hard real-time systems. It is designed to be a particularly efficient, simple runtime system implementation on a single processor target, amenable to static timing and schedulability analysis. An example implementation is Aonix Object Ada Real-Time/Raven. The SPARK subset now includes Ravenscar.

The idea of “no runtime” is to eliminate all language features which require any runtime library support. The advantage is that there is no COTS component to certify.

8.4 Runtime options and development

For all runtime options, the Board Support Package is necessary. This provides support for download and start up, cold boot (i.e. from ROM), hardware-specific initialisation, debugging, coverage analysis and some predefined simple I/O.

The BSP must be tested, but traditional testing techniques such as unit test and coverage analysis may not work on the BSP. In some projects we have fielded two BSPs: a “debug” version supporting all the above functionality for any application, and a “strip” version supporting only the functionality required for delivery of a specific application. This makes (manual) coverage analysis and testing easier, with no “dead” code.

8.5 Object code verification

If a compiler cannot be trusted sufficiently, manual verification of the object code may be required (OCV). This should be avoided if at all possible as it requires detailed knowledge of the language, compiler and target processor, and is hard and lengthy work.

The 1-step approach to OCV is to compare source code with disassembled object code directly. However the “semantic gap” between the two is very wide, especially for a rich language like Ada, and this therefore requires detailed knowledge of Ada compilation, code generation, language issues etc.

The 2-step approach is to review the Ada source against the compiler-generated intermediate language (IL), then review the IL against the disassembled object code. This means that the semantic gaps are narrower and splits the reviewing skills required, but not many compilers allow users access to IL.

From experience on the SHOLIS project, where a compiler feature caused large aggregates to be allocated on the heap although no heap was used in the runtime, we recommend always having someone on a project team who is capable of reading and reviewing the object code.

9 Conclusions

In this article we have examined the challenge of implementing high-integrity systems in the context of our experience with them. We have shown what the current safety and security standards demand, described three real-world commercial projects developed under these standards, and shown how the selection of tools and development processes can pay off in reduced development time and increased system reliability.

References

- [1] B. Littlewood and L. Strigini (1993), *Validation of Ultrahigh Dependability for Software-based Systems*, Communications of the ACM, November 1993.
- [2] R. W. Butler and G. B. Finelli (1993), *The Infeasibility of Qualifying the Reliability of Life-critical Real-time Software*, IEEE Transactions on Software Engineering, vol. 19 no. 1, January 1993, pp. 3—12.
- [3] B. A. Wichmann (1992), *Requirements for Programming Languages, Computer Standards and Interfaces*, National Physical Laboratory.
- [4] ISO/IEC JTC1/SC22/WG9, *Programming Languages – Guide for the Use of the Ada Programming Language in High Integrity Systems*, <http://www.dkuug.dk/jtc1/sc22/wg9/n359.pdf>
- [5] J. Barnes (2003) *High Integrity Ada – the SPARK Approach to Safety and Security*, Addison Wesley, ISBN 0-321-13616-0
- [6] N. G. Leveson (1995) *Safeware: System Safety and Computers*, Addison-Wesley, ISBN 0-201-11972-2
- [7] Judge Asquith (1949) *Edwards v. National Coal Board*, All England Law Report Vol. 1, p. 747
- [8] RTCA (1992) *Software Considerations in Airborne Systems and Equipment Certification*, DO-178B
- [9] J. Sutton and M. Croxford (1996) *Breaking Through the V&V Bottleneck*, Lecture Notes in Computer Science vol. 1031, Springer-Verlag
- [10] P. Amey (2002) *Correctness by Construction: Better Can Also Be Cheaper*, CrossTalk journal, March 2002
- [11] A. German (2003) *Software Static Code Analysis Lessons Learned*, CrossTalk journal, November 2003
- [12] UK Ministry of Defence (1996) *Safety Management Requirements for Defence Systems*, Defence Standard 00-56, Issue 2
- [13] S. King, R. Chapman, J. Hammond and A. Pryor (2000) *Is Proof More Cost-Effective Than Testing?* IEEE Transactions on Software Engineering, vol. 26 no. 8, August 2000
- [14] National Institute of Standards and Technology (1999) *Common Criteria for Information Technology Security Evaluation*, CCIMB-99-031, version 2.1, August 1999