Engineering Software Systems for Customer Acceptance

Adrian Hilton

Engineering Software Systems for Customer Acceptance

Adrian Hilton
Praxis Critical Systems Ltd
20 Manvers Street
Bath BA1 1PX, England
adrian.hilton@praxis-cs.co.uk

Abstract

Building a software system is a well-understood problem with a wide range of solutions, each suitable for some classes of system but not for others. The commercial success of a software system, however, depends on its acceptance by the customer. Therefore, the developer must demonstrate that a system is fit for its purpose. A common view is that following a specified software or systems development process is adequate for this purpose. However, as software and safety standards move from a prescriptive to goal-oriented form, this demonstration of fitness will become better tailored to each system.

In this paper we examine how existing processes and products can be used to build an evidence-based case for high-assurance system acceptance. We draw on our own experience of developing and delivering such systems, and make practical recommendations for improving acceptance rates. We show how existing technologies and tools can support this process.

1. Introduction

There are as many ways of building software systems as there are software engineers. Each approach has strengths and weaknesses, and is appropriate for some classes of system but not for others. An essential step towards the successful delivery of a system is to choose a development approach that is suited to the needs of that system and its customers.

1.1 Building an acceptable system

We understand that building a high-assurance software system is different to writing an application to manage home finances, but there is a fundamental point in common. To be judged successful, the systems must be accepted by their customers. A home finance application must be easy to use, run reliably on the customer's home PC and get its arithmetic right. If it does not, people will not buy it.

Similarly, a flight control system must keep an aircraft stable throughout its flight, integrate with the rest of the aircraft's avionics, and be demonstrably safe to the level of integrity required by the relevant aviation safety body. If it does not, either the customer will refuse to accept it or the safety body will not issue the required flight certification.

1.2 An historical perspective

Brooks [4] wrote of the state of the software engineering art in 1975, and updated the 20th anniversary edition of his book with a review of the progress that software engineers had made. Brooks' conjectures were that:

1. system development time does not scale in an inverse-linear relation to team size, and indeed that adding more manpower to a late project makes it later (the "mythical man-month");
2. there is no single development, in either technology or management technique, which promises an order of magnitude improvement within a decade in productivity, reliability or simplicity ("no silver bullet");
3. after building one system successfully, the design and development of a follow-on system is prone to balloon out with pointless features and an elephantine design (the "second system effect"); and
4. a small number of documents, in a sea of project documentation, become the critical pivots around which every project's management revolves ("the documentary hypothesis").

History appears to have borne out these conjectures, which have passed into every-day software engineering practice. We do not attack any of them in this paper; indeed, we advocate reading Brooks as a classic introduction to the essentials of project management.

Instead, we assume that the system development is planned and carried out with an eye to these laws, and stay focused on the task of producing the system that the customer wants. We do not aim to reduce the time taken to develop a system. We aim to avoid all the extra development time resulting from having to rework the finished system after the customer has rejected it.

1.3 Focus

Our focus in this paper is on high-assurance systems, defined as systems which have an external mandate to meet some safety, security, liveness or performance criteria. Our requirement is to develop and deliver this system and accompanying evidence such that:

- the customer is likely to agree that the system meets the specified functional requirements;
- the safety body is likely to accept that the system is suitably safe;
- the commercial risks of system development are identified and minimized; and
- the cost of the development is minimized.

Demonstrating that a high-assurance system meets its functional and safety requirements is not a trivial task, and is system-dependent. In this paper we present tactics and strategies which we believe will assist in this process.

Our primary focus is on safety-related systems, since we have more experience in this field, but the issues raised relate well to other classes of non-functional requirements such as security, survivability and high-performance computing.

1.4 Structure

Section 2 describes current best practice in constructing good functional and safety requirements for a system. Section 3 examines the issues involved in working to mandated development standards. Section 4 looks at the techniques and tools used to build dependable and demonstrably correct systems from a set of requirements. Section 5 describes the benefits and limitations of testing.

At the end of each of these sections we collate the evidence-gathering steps discussed in the section. This provides a summary of the evidence that could be produced in the corresponding development phase.

Finally, section 6 draws together the main conclusions from the analysis work.

2. Meeting requirements

It is no good building a perfectly safe system if the system does not do what the customer requires. Nor is it worth meeting all the functional requirements if the software has a MTBF of a quarter of the required duration. In this section we address the task of identifying what system to build, and calculating how safe it needs to be.

2.1 Requirements engineering

Gathering and managing requirements is as integral to the system development as producing code. The Standish CHAOS reports of 1995 and 2003[31, 32] and the study by Taylor [33] estimate that between 30% and 48% of IT projects fail due to requirements-related problems, even though the stage at which the projects fail is usually late in the development cycle. The 2003 Standish report also notes that the fraction of originally defined features that projects deliver has decreased from 67% to 54% in the past year.

Most high-level functional requirements will come from the customer, and these will have a profound effect on the design of the system. Since late requirements changes are well-known to have serious impact on development timescales, the developer and customer should expend significant effort and time on a thorough and structured requirements elicitation. The customer is responsible for getting all their requirements written down clearly; the developer is then responsible for ensuring that the final implementation can clearly be shown to satisfy each of these requirements.

Requirements engineering methods such as KAOS [5] or the Praxis Critical Systems REVEAL® method [1] [11] provide a framework for requirements engineering that can be tailored to a particular system.

When producing requirements, we must bear in mind that we must eventually demonstrate that the system fulfils each requirement. For that reason, requirements should be directly testable wherever possible. The developer should also plan how the traceability of requirements to code is going to work. Having a complete map from each requirement to the sections of code which implement it may be excessive in many cases, but the developer should assume that late requirement changes will arrive, either from the customer or from the development itself.

If an existing requirement is changed, the developers must be able to go straight to the relevant parts of the design, code and tests to change them appropriately. It may even be possible to present a justified estimate of impact before the requirement change gets the green light; the cost may be enough to cause the change to be dropped. The developer must be careful to check the effect of the change

[1] REVEAL is a registered trademark of Praxis Critical Systems Limited

thoroughly; a small functional requirement change could conceivably affect non-functional requirements such as system safety or performance.

2.2 Is it safe enough?

Safety engineering is an engineering discipline in itself. Leveson's text "Safeware"[17] provides a good introduction to the subject. Her analysis of accidents such as the Therac-25 over-exposure of patients to ionising radiation[18] illustrates the consequences of inadequate system safety analysis.

Standards such as UK Defence Standard 00-56[22] define a safety management scheme for safety-critical systems development. The process typically incorporates features including:

hazard analysis: demonstration that all risks posed by the system are acceptable, identifying and setting safety requirements;

fault tree analysis: breaking down each hazard into component faults, proceeding recursively to identify the basic faults of the system and their effect on system safety; and

the safety case: an argument based on the above documents and the system development process, stating why the system meets the required safety level.

The level of required safety can be measured in several ways, but the approach taken by 00-56 is the use of *safety integrity levels* (SILs) to express the expected minimum time between failures. SIL-1, the lowest level, requires no more than one failure per 10^5 hours of high-demand operation. SIL-4, the highest level, requires no more than 1 failure per 10^9 hours. Since this is just over 114,150 years we can immediately see that system testing to demonstrate this level of reliability with any confidence will likely be impractical. The statistical limitations of testing have been discussed in detail by Littlewood[19].

Safety engineering can tell the developer how safe a system component must be, and the safety case is a key piece of safety evidence, but how to attain the required safety level is a separate question.

2.3 Is it too safe to build?

Systems can also be too safe. The higher the required safety level for a system, the more expensive it will be to build. Building a system which is much safer than required is likely to be a waste of money.

White box safety [29] is an approach that aims to reduce the amount of a system that is at a given level of criticality. It breaks down a single system into its components, identifies the hazards in the system, and tries to exclude system components from the causes of the hazards. When successfully applied, a SIL-4 system might be broken down into a core safety task which must still be SIL-4, but other components would have their SIL reduced or (ideally) removed.

One strategy to avoid is the movement of safety-critical functionality from conventional software into an area of the system not covered by a prescribed safety standard. This was superficially attractive with prescriptive standards such as Defence Standard 00-55 [21], and programmable logic was one such destination for the functionality. The obvious problem is that it does not make the system any more safe; instead, when the developed system is presented to the certification authority they are likely to reject it and demand that the safety case address the programmable logic component. Retroactively building safety into an existing system is notoriously expensive and difficult.

The interim UK Defence Standard 00-54[23] addresses safety-related electronic hardware for precisely this reason. Indeed, the requirements it places on systems with the higher SILs are so stringent that no implementations are known to have been certified to those levels of safety. The conclusion to draw is that if you wish to build a SIL-3 or SIL-4 system then you should keep safety functionality out of programmable logic. Hilton and Hall have described how existing high-assurance software engineering techniques could be applied to programmable logic [12, 13] but a practical demonstration of this has not yet been made.

2.4 Formal notations

Expressing requirements in English (or a similar written language) trades off ease of reading and writing against ambiguity. The use of formal notations for expressing some or all of a system's requirements removes ambiguity, at the cost of increased effort to write (and then comprehend) the requirements.

Common formal notations used in system specification include Z[30], VDM[15], CSP[14] and B[1]. The choice of notation will be dictated by what must be specified and what tools are available.

One additional benefit of formal notations includes the ability to prove correct sections of the system. Semi-automatic proof tools have reduced the amount of skilled work involved in producing such proofs. CSP, for instance, is generally good at specifying protocols of communication between objects, and is supported by the FDR tool[8] used to identify deadlocks and livelocks in a CSP system. Z can also be used effectively; in the development of the SHOLIS helicopter guidance system [16] which involved extensive formal specification and proof work, Z proof was found to be significantly the most efficient phase at finding faults.

From the point of view of generating evidence of correctness, formal notations provide a strong argument that the correct system was specified (*as long as* the customer was able to read, understand and agree with the specification). A formal proof of correctness is generally much easier to verify than to write, so any such proof will contribute towards the evidence of correctness as long as the property proven is relevant to the customer's requirements or system safety.

2.5 Evidence collation

Summarising the points of evidence that we can collect from the requirements phase:

- The use of a mature stakeholder-focused requirements engineering method is a valid piece of evidence that the customer's requirements have been captured and accurately expressed.
- A safety case is evidence that the required safety of the system has been measured, and that the system being built will meet that level of safety.
- A "white box" safety analysis is key safety evidence that the components developed to SILs below that required for the main system do not compromise overall system integrity.
- A readable specification in a formal notation is evidence that the system requirements have been unambiguously specified. Proof carried out on this specification demonstrates a level of self-consistency in the specification.
- It is especially important in this phase for the customer to be able to understand and approve the generated documents.

3. Working to standards

Current high-assurance standards in use in the domain of civil and military avionics include RTCA DO-178B[28] and UK Defence Standards 00-55 and 00-56 [21, 22]. These were included in a comparison of avionics standards by Pygott [26]. The key conclusion to the report was that the standards had roughly the same overall objectives, but differed in how they recommended that the developer attain the objectives.

Defence Standard 00-55, for instance, emphasises formal methods as key to achieving the higher levels of integrity whereas RTCA DO-178B regards formal methods as immature and requires justification of their use in a development. This may be partly a result of time – DO-178B was produced in 1992, whereas issue 2 of 00-55 was five years later – but also reflects differing opinions in the certification community about the value of such methods.

3.1 Standards evolution

UK Defence Standards undergo periodic rewriting: 00-55 and 00-56 are at issue 2 already, and issue 3 is due to appear soon. The rewritings reflect both feedback from practical application of the previous standards and advances in the state-of-the-practice of system development. The changes from issue 1 to issue 2 of 00-55 reflect industrial comments that the approach prescribed in issue 1 was too hard to apply in general, although at least one project was successfully developed under issue 1[16].

If experts dispute such issues, and standards documents show that conflict, how do we find a generic development process applicable to all standards? How can we anticipate the requirements of future versions of existing standards? We cannot, but we can focus on the areas of agreement noted above: the standards aim to support the process of producing a system which is demonstrably safe at a quantifiable level.

3.2 Goal-based safety

Within the United Kingdom there is a move away from prescriptive standards (of which issue 1 of 00-55 was a good example) towards documents that support development by describing a wide range of ways that the development process can produce evidence of safety and correctness.

Penny et al.[25] describe practical experience with this "goal-based" form of safety standard in the development of SW01, part of the regulations for ground-based air traffic services in the UK. They split evidence into two forms: *direct*, which directly relates to the safety of the system (such as evidence that static analysis has been carried out and no dangerous faults found), and *backing* which shows that the direct evidence is credible and sound (such as test reports and error history of the static analysis tool used).

Penny et al. conclude that there are sound benefits to the developer and to the certification authority in the use of goal-based safety. Although it represents a significant shift from common current practice, and requires support from a wide range of tools and techniques in the development process, it appears to be a promising approach; an approach which this paper aims to support.

3.3 Evidence collation

The decision to follow a standard should be presented as evidence in itself, along with a brief analysis of why this standard was appropriate.

- Following a prescriptive development standard will normally be taken as substantial evidence of adherence

to best practice. The hazard is that this is heavily dependent on the certification authority, and the approach prescribed may not stand the test of time.

- Adopting a goal-based standard, if available, is the option we advocate. However, the developer should ensure that their tools and techniques can produce adequate direct and backing evidence; the results of this check should be presented as evidence in itself.

4. Building the right system

Given good functional and safety requirements, how do we produce a system design and implementation that can be shown to satisfy them? This section describes how to collect evidence of correctness and safety during program development.

4.1 Designing to succeed

A system design expresses decisions made by developers about how they plan to address the functionality in general terms. It typically partitions the system into modules and describes data flow between the modules. Designs may evolve out of requirements, be dictated by the available toolsets and methodologies, or simply come out of the heads of the system developers.

UML is an increasingly common notation used for capturing designs. It has the great strength of a standardised notation which many engineers have studied and understand. Unfortunately it lacks any semantics, a feature which is important for the adoption of UML across a wide range of application domains but which leads to frequent disagreements about what a UML diagram actually means. It is up to engineers in each application domain to agree on a standard semantics, and in real-time and high-assurance systems this has not yet happened. The use of UML in a system design should take account of this.

The principles of system design have been discussed at length elsewhere[24] and will not be repeated here. We will restrict ourselves to observing that the design should admit traceability back to functional requirements, and onwards to the implementation.

The designer should also note Amey's comments[3] on the difference between encapsulation and hiding. To show that code is free of unintended side effects (and hence order-of-evaluation dependencies) it is vital for the designer to identify the location of system state, even if it appears irrelevant to the compiler. This illustrates a general principle of design, that one should not unnecessarily sacrifice structure and information early in the design process even if it will be lost in a later phase.

4.2 Implementation languages

For a high-assurance system, the choice of implementation language will be dictated by the availability of a trusted compiler for the host system. Typically this restricts the developer to the "mainstream" languages of C, C++ or Ada. Assembly language can clearly be used, but the problems of designing large systems in this are well known; typically, it is used in small amounts for time-critical processing or low-level interfacing. Higher-level languages incorporating garbage collection such as Java and Perl allow quick writing of programs but current interpreters for these languages are unsuited to real-time or high-assurance applications.

Auto-generation of code from a high-level design by a tool is attractive since the implementation step is apparently attained "for free". In practice, however, little additional programming expertise can be supplied by the generation tool except in very specific target applications, so what you get for free is what you pay for (in information-theoretic terms). For high-assurance system either the generator itself must be validated (which returns us to the problem of writing a safe and correct program) or the output code must be analysed; such code is not normally structured for readability or susceptible to automatic analysis.

If a program is then written in C or Ada, the developer can choose to use a subset of the full language. This is normally done to avoid known pitfalls with code constructs outside the subset, and is intended to demonstrate to the certification authority that the code will exclude a class of common errors. The MISRA rules for C[20] define such a C subset; similarly, SPARK[2] Ada defines annotated subsets of Ada 83 and 95[2, 7].

To produce evidence that the program code contains no code constructs outside the chosen subset, a suitable tool is required. Such a tool can be a simple parser if only the language syntax is restricted; however, it may be possible to take advantage of the reduced complexity of the programming language and do more substantial semantic analysis. This leads us to the technique of static analysis.

4.3 Static analysis

Developers sometimes equate static analysis with the `lint` tool found in many C compiler tool sets, used to identify unused variables and ignored function return values. This severely underestimates the technique's potential usefulness.

Static analysis is the compile-time identification of certain properties in a program. The developer must decide which properties are important to identify, and then ensure

[2]Note: The SPARK programming language is not sponsored by or affiliated with SPARC International Inc. and is not based on the SPARC^{TM} architecture.

that the program description being compiled admits such identification. Clearly, the richness of the program description will determine the level of analysis that can be carried out. If the program description is ambiguous, as is the case with any non-trivial subset of C, the potential of static analysis is limited.

Praxis Critical Systems has substantial practical experience in using the SPARK Examiner, a static analysis tool that enforces the SPARK Ada subset. The subset excludes Ada language features prone to cause error (e.g. aliasing, operator overloading, dynamic memory allocation), but also insists on the use of *annotations* – Ada comments marked with an extra symbol, ignored by the compiler but used to provide program design information to the Examiner. These annotations provide information that can be efficiently checked, then fed into the verification of annotations higher up the calling tree in a SPARK Ada program.

Our experience is that the SPARK Examiner is visibly very effective at preventing a wide range of errors at the subprogram level. Moreover, since we require code to pass analysis before compilation, such error detection avoids the overhead resulting from a unit or system test failure. In the SHOLIS development [16], for instance, we found that unit tests picked up very few errors; the static analysis had already found the errors unrelated to proof, so they were fixed before any testing occurred. A side effect is that compilations of SPARK-checked code seldom fail.

Conversely, static analysis of Ada code tested to RTCA DO-178B [28] Level A revealed potentially safety-critical errors that had avoided detection. Moreover, there was no significant difference in error rates between Level A and Level B tested software, despite the substantial extra testing effort in Level A. This analysis was described in detail by German and Moody[9] in a general assessment of static analysis tools and techniques. They conclude that static analysis is an effective technique, but its nature and depth should be selected with regard to the criticality of the code under examination.

From the point of view of generating evidence of correctness and safety, the static analysis of a program provides a single report which assures the certification authority that certain classes of error (e.g. use of uninitialised data) are not present in a program. Of course, the value of this analysis depends on the confidence in the correctness of the static analysis tool.

4.4 Program proof

Program proof may follow on from static analysis. A program proof is a rigorous demonstration that a program satisfies one or more specified properties. Unlike static analysis, proof cannot typically be done automatically in a relatively short time; advances in proof tool technology and processor speed have helped, but the construction of a non-trivial proof will often require human intervention.

A necessary condition for proof is the use of an unambiguous language with a well-defined semantics. Without this, it is not possible to construct meaningful proof rules. C, and even Ada, do not meet this test. A language such as SPARK can have such a semantics, but even then there are limitations. Ada's floating point rules, for instance, are such that we advocate great caution when using floating point arithmetic in SPARK. The greatest hazard in proof is an incorrect manual proof step.

Proof may be used to demonstrate a general property of a program (e.g. that an Ada program never raises a run-time error), or to demonstrate that a subprogram has a specific property. We now find it practical to run a nightly proof calculation on our SPARK software development work, where we aim to demonstrate total freedom from run-time errors.

The proofs provided as evidence of correctness will be accompanied by sequences of proof steps, each of which can be validated efficiently. The main work of validation will be in ensuring that the supplied proof rules are in fact correct, and that the proofs were carried out on the version of code that was shipped. The value of this evidence is a powerful argument that all the relevant system specifications (themselves unambiguously specified) were satisfied.

4.5 Evidence collation

Summarising the points of evidence collected during the implementation phase:

- A system design in a well-understood notation indicates to the certification authority how the safety responsibilities are divided in the system.
- The choice of a mainstream high-level language removes doubts about compiler correctness and allows focus on the program itself.
- Choosing an enforceable subset of a language is evidence that a class of errors due to unsafe language features have been excluded.
- Static analysis of a program can demonstrate that the program has a range of useful properties (e.g. never reading uninitialised data), but the properties which can be demonstrated depend heavily on the language subset chosen.
- Program proof is strong evidence of correctness, but is more labour-intensive than static analysis.

5. The limits of testing

Testing is a vital part of system development. The main kinds of testing are:

- informal testing by developers that the feature they are developing works at least approximately as designed;
- unit testing to exercise each component of a program (typically by subprogram or module, depending on the implementation language);
- functional testing to check that all requirements are covered; and
- system testing to verify that the entire system operates as designed without any errors.

"White box" testing is informed by the design and implementation details of the system, and aims to attain some level of coverage of the code. "Black box" testing concentrates on the required output, ignoring system structure. As a result, black box tests may normally be constructed earlier in the product lifecycle than white box tests, and are more amenable to re-use in future system variants.

However, we must remember what testing alone cannot achieve. Modern testing techniques are efficient and successful within a limited framework, but (as noted above) even the most stringent testing can miss an error that techniques such as static analysis can detect.

5.1 Aims and achievements of testing

Testing, as Dijkstra said[6], can only show the presence of errors, not their absence. Functional testing aims to show that functional requirements are met, but at best can show that no errors occur while the function is being exercised in a range of common ways.

Unit testing aims to exercise each individual component (unit) in a program. There are formal notions of how thoroughly a unit has been tested – statement coverage, branch coverage, MC/DC etc. – but the limiting factor in unit testing is often the person writing the test. They should know the required result of each test before writing it. The temptation to derive the test result from the code is substantial, so unit test results should ideally be written before the unit is written. But then, the tests are unlikely to cover all of the unit's statements or paths.

System testing can only realistically exercise a small section of the system's state space. Detecting and counting errors during continuous system test can give an indication of the number of *detectable* errors remaining in the system, but can never assure the developer, certification authority or customer that all the errors are gone.

If a particular testing aim is to reduce the number of errors encountered by users then usage models for the system can be useful to direct testing effort to frequently-used sections of the code. This may not be appropriate if the testing aim is to reduce the likelihood of *any* occurrence of a particular class of error (such as system reset).

5.2 Untestable conditions

As noted previously, for systems with a required SIL 4 rating it is not practical to achieve that level of confidence with testing alone. There are also more specific aspects of program correctness which are difficult to achieve by testing. Absence of run-time errors can only be shown by testing if the test exercises every path in the entire program for all values of input data. This is normally computationally infeasible.

5.3 Evidence collation

It is not easy to produce convincing evidence that effective testing has been carried out. Use of an independent team for testing can increase the chance of testing oversights or unjustified assumptions being caught. It may be useful for such a team to perform a random audit on tests and sections of code, checking them against required practice and against the existing requirements.

Summarising the points of evidence collected during testing:

- Test results are necessary evidence for any program, but often not sufficient.
- Independent testers reduce, but do not remove, the difficulty of producing convincing evidence of testing.
- Functional testing is useful evidence that the customer's requirements are covered, at least to some degree; it will provide increased assurance if the requirements were properly managed and tracked.
- Thorough unit testing is evidence of diligence, but may not be an efficient way of detecting errors, and does not provide much practical evidence of correctness or safety.
- System testing is primarily a statistical exercise, and for high-SIL systems cannot provide sufficient statistical confidence that the system is safe.

6. Conclusions

In this paper we have examined the problem of producing a correct and safe system, to the satisfaction of a customer and a certification authority. This is a practical problem which many software engineers face daily.

We have adopted a standards-neutral approach of collecting evidence of safety and correctness as we progress through the system development process. We have seen how existing and emerging tools and techniques can contribute to a strong evidence-based argument that the system does what it is supposed to, safely.

Perhaps one of the most definite pieces of evidence of system correctness and safety is a program warranty. For the SIL-2 CDIS air traffic control system[10], written in 200KLOC of C, Praxis delivered the system with a ten year warranty. One claim (and fix) was made during customer system testing; no further claims were made during the warranty period. If this could be done with 1990s technologies, how much more should we expect from software engineering in this new decade?

6.1 Acknowledgements

Thanks are due to Rod Chapman and Helen May from Praxis Critical Systems Limited, and to Jon Hall from the Open University, for their comments on an earlier version of this paper.

References

[1] J.-R. Abrial. *The B Book*. Cambridge University Press, 1996.

[2] P. Amey. SPARK – the SPADE Ada kernel. Technical Report 1.0, Praxis Critical Systems Ltd., 1999.

[3] P. Amey. Logic versus magic in critical systems. In D. Craeynest, editor, *6th Ada-Europe International Conference on Reliable Software Technologies, Proceedings*, volume 2043 of *Lecture Notes in Computer Science*. Springer-Verlag, May 2001.

[4] F. P. Brooks, Jr. *The mythical man month: essays on software engineering*. Addison Wesley Longman Inc, anniversary edition, 1995.

[5] A. Dardenne, A. van Lansweerde, and S. Fickas. Goal-directed requirements acquisition. *Science of Computer Programming*, 20, 1993.

[6] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

[7] G. Finnie and R. Wintle. SPARK 95 – the SPADE Ada 95 Kernel. Technical Report 1.0, Praxis Critical Systems Ltd., October 1999.

[8] Formal Systems (Europe) Ltd. *FDR User Manual*, May 1997.

[9] A. German and G. Mooney. Air vehicle software static code analysis lessons learnt. In Redmill and Anderson [27], pages 175–193.

[10] J. A. Hall. Using formal methods to develop an ATC information system. *IEEE Software*, 12(6), March 1996.

[11] J. Hammond, R. Rawlings, and A. Hall. Will it work? In *Proceedings of the 5th International Symposium on Requirements Engineering*, August 2001.

[12] A. Hilton and J. Hall. On applying software development best practice to FPGAs in safety-critical systems. In R. W. Hartenstein and H. Grünbacher, editors, *Proceedings of the 10th International Conference on Field Programmable Logic and Applications (FPL'00)*, volume 1896 of *Lecture Notes In Computer Science*, pages 793–796. Springer-Verlag, August 2000.

[13] A. J. Hilton and J. G. Hall. Mandated requirements for hardware/software combination in safety-critical systems. In *Proceedings of the workshop on Requirements for High-Assurance Systems 2002*. Software Engineering Institute, Carnegie-Mellon University, September 2002.

[14] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, 1985.

[15] C. B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall International, 1986.

[16] S. King, J. Hammond, R. Chapman, and A. Pryor. The value of verification: Positive experience of industrial proof. In J. M. Wing, J. Woodcock, and J. Davies, editors, *FM'99 — Formal Methods; Proceedings*, volume 1709 of *Lecture Notes in Computer Science*. Springer-Verlag, September 1999.

[17] N. Leveson. *Safeware: System Safety and Computers*. Addison-Wesley Publishing Company, 1995.

[18] N. Leveson and C. Turner. An investigation of the Therac-25 accidents. *Computer*, July 1993.

[19] B. Littlewood and L. Strigini. Validation of ultrahigh dependability for software-based systems. *Communications of the ACM*, 36(11):69–80, 1993.

[20] MIRA. *Guidelines for the Use of the C Language in Vehicle Based Software*, April 1998.

[21] Defence Standard 00-55 issue 2, August 1997. Requirements for Safety-Related Software In Defence Equipment.

[22] Defence Standard 00-56 issue 2, December 1996. Safety Management Requirements for Defence Systems.

[23] Interim Defence Standard 00-54 issue 1, March 1999. Requirements for Safety Related Electronic Hardware in Defence Equipment.

[24] D. L. Parnas. Designing software for ease of extension and contraction. *IEEE transactions on software engineering*, pages 128–138, March 1979.

[25] J. Penny, A. Eaton, P. G. Bishop, and R. E. Bloomfield. The practicalities of goal-based safety regulation. In Redmill and Anderson [27], pages 35–48.

[26] C. H. Pygott. A comparison of avionics standards. Technical Report DERA/CIS/CIS3/TR990319/1.0, UK Defence Evaluation and Research Agency, August 1999.

[27] F. Redmill and T. Anderson, editors. *Proceedings of the 9th Safety-Critical Systems Symposium*. Safety-Critical Systems Club, Springer-Verlag, 2001.

[28] Requirements and Technical Concepts for Aviation Inc. *DO-178B / EUROCAE ED-1D: Software Considerations in Airborne Systems and Equipment Certification*, December 1992.

[29] A. Simpson and M. Ainsworth. White box safety. In *Proceedings: Avionics Conference and Exhibition*. ERA Technology Ltd., 1999. ERA Report 99-0815.

[30] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall International, second edition, 1992.

[31] The Standish Group. *The CHAOS report*, 1995.

[32] The Standish Group. *What are your requirements? 2003*, 2002. Research note.

[33] A. Taylor. IT projects sink or swim. *BCS Review*, pages 61–64, January 2001.