

***Technical Report No: 2004/09***

***Developing Critical Systems with PLD  
Components***

***Adrian J. Hilton  
Jon G. Hall***

***15<sup>th</sup> March 2004***

---

***Department of Computing  
Faculty of Mathematics and Computing  
The Open University  
Walton Hall,  
Milton Keynes  
MK7 6AA  
United Kingdom***

***<http://computing.open.ac.uk>***



# Developing Critical Systems with PLD Components

Adrian J. Hilton<sup>1</sup> and Jon G. Hall<sup>2</sup>

- <sup>1</sup> Praxis Critical Systems, 20 Manvers Street, Bath BA1 1PX, England  
Adrian.Hilton@praxis-cs.co.uk
- <sup>2</sup> The Open University, Walton Hall, Milton Keynes MK7 6AA, England  
J.G.Hall@open.ac.uk

**Abstract.** Programmable logic devices (PLDs) are now common components of critical systems, and are increasingly used for safety-related or safety-critical functionality. Since 1999 avionics- and defence-related safety standards have advised and prescribed various approaches for PLD programming in safety-related systems. There are many differences between current and recommended practice, and safety engineers differ on how to apply the existing standards.

This paper describes past and current practice in programming PLDs in critical systems. It summarises the relevant safety and security standards and anticipates forthcoming changes to UK standards. It describes the work that the authors and others have done in the field of specifying, designing and proving correct PLD programs and maps out avenues of work that the authors believe necessary for PLD programming technology to keep pace with PLD functionality.

## 1 Introduction

Programmable Logic Devices (PLDs) are increasingly important components of safety-critical systems. By placing simple processing tasks within auxiliary hardware, the software load on a conventional CPU can be reduced, leading to improved system performance. They are also used to implement safety-specific functions that must be outside the direct address space of the main CPU. Technological improvements mean that PLD development has become more like software development in terms of program size, complexity, and the need to clarify a program's purpose and structure.

Standards for safety-related electronic hardware design and development have, since 1999, explicitly targeted FPGAs and CPLDs. The practices which they recommend vary in rigour and in practicality. These standards are hindered by the immature state of PLD program design, development and analysis techniques and tools relative to those available to safety-related software developers. There are now signs that the move towards high-level programming of PLDs, coupled with the adoption of existing specification notations and proof techniques, may enable more formal and rigorous PLD program development.

Recent work by safety engineers and safety authorities based in the United Kingdom has started to produce guidance for avionics systems project teams on how to use PLDs in safety-critical functions and make justified evidence-based arguments about their fitness for purpose.

In this paper we examine past and current practice in programming PLDs in critical systems, comparing and contrasting with current safety-critical software practice. We describe and analyse the main safety and security standards relevant to PLD program development. Based on this information we describe the results of our research to date along with other relevant research. Finally we identify the deficiencies in current theory and practice and suggest ways in which they could be overcome.

Section 2 describes the content of the main standards relevant to PLD program development. Section 3 describes current industrial practice including case studies of PLD use. Section 4 summarises recent research relevant to safe or provably correct PLD program development. Finally, Section 5 summarises the paper and indicates routes for future work.

## 2 Current safety standards

The main safety standards relevant to PLD programming are:

- RTCA DO-254[31] which is an international civil aviation standard;
- UK Interim Defence Standard 00-54[28] which is a UK standard for defence-related systems; and
- IEC 61508[18] which is a European standard intended to apply to a wide range of systems.

There are also international security-related standards such as the Common Criteria[7].

In this section we analyse the content of each of these standards, then compare and contrast them to identify the safety requirements that future PLD software development will have to satisfy.

### 2.1 RTCA DO-254 / EUROCAE ED-80

The airborne electronic hardware development guidance document RTCA DO-254 / EUROCAE ED-80[31] is the counterpart to the well-established civil avionics software standard RTCA DO-178B / EUROCAE ED-12B[30]. It provides a guide to the development of programs and hardware designs for electronic hardware in avionics. It covers PLDs as well as Application-Specific Integrated Circuits (ASICs), Line Replaceable Units (LRUs) and other electronic hardware. As well as being applied to systems aimed for Federal Aviation Authority acceptance, it may be used as a quality-related standard in non-FAA projects.

**Overview** DO-254 specifies the life cycle for PLD program development and provides guidance on how each step of the development should be done. It is not a prescriptive standard, providing instead recommendations on suitable general practice and allowing methods other than those described to be used. The emphasis is on choosing a pragmatic development process which nevertheless admits a clear argument to the certification authority (CA) that the developed system is of the required integrity.

DO-254 recommends a simple documentation structure with a set of planning documents that establish the design requirements, safety considerations, planned design and the verification that is to occur. This would typically be presented to the CA early in the project in order to agree that the process is suitable. This plan will depend heavily on the assessed *integrity level* of the component which may range from Level D (low criticality) to Level A (most critical). Note that the DO-254 recommendations differ very little for Levels A and B.

**High-integrity requirements** Appendix B of DO-254 specifies the verification recommended for Level A and Level B components in addition to that done for Levels C and D. This is based on a Functional Failure Path Analysis (FFPA) which decomposes the identified hazards related to the component into safety-related requirements for the design elements of the hardware program. The additional verification which DO-254 suggests may include some or all of:

**architectural mitigation:** changing the design to prevent, detect or correct hazardous conditions;

**product service experience:** arguing reliability based on the operational history of the component;

**elemental analysis:** applying detailed testing and / or manual analysis of safety-related design elements and their interconnections;

**safety-specific analysis:** relating the results of the FFPA to safety conditions on individual design elements and verifying that these conditions are not violated; and

**formal methods:** the application of rigorous notations and techniques to specify or analyse some or all of the design.

If tools are used for compilation or verification of the PLD software then DO-254 requires a certain amount of *tool qualification*. This may incorporate separate analysis of the tool software, appeals to in-service history of the tool, or direct inspection of the tool output. At higher integrity levels, in-service history alone is likely to be insufficient.

## 2.2 UK Interim Defence Standard 00-54

Within the United Kingdom, Interim Defence Standard 00-54[28] specifies safety-related hardware development in a similar way to DO-254. The main difference is that 00-54 is far more prescriptive than DO-254, and assumes that the development takes place within a safety management process as described in Defence Standard 00-56[27].

**Overview** We have analysed the requirements of Defence Standard 00-54 and its implications for PLD program design in [14]. For the purposes of this paper it suffices to observe that 00-54 makes strict demands on the rigour and demonstrable correctness of PLD programs, and that these are significantly stricter than those in DO-254. The requirements of 00-54 map closely to those of UK Defence Standard 00-55[26] which specifies safety techniques for use in conventional software development for safety-critical systems.

**High-integrity requirements** Formal specification and analysis of PLD programs are *mandated* at all safety integrity levels. This poses a practical problem for developers since there are no known tool-supported specification or proof notations which are generally applicable to PLD programming. Each project is likely to require a from-scratch selection of and capability development in a notation and analysis techniques, which is risky and potentially expensive.

Note that 00-56 is being updated at the moment. The draft release of the new version for public comment[25] indicates that the requirements of 00-54 will be folded into 00-56 as an extra volume of the standard, and that 00-56 will move from its current prescriptive requirements to a design where it prescribes only the forms of evidence that may be appropriate for designs of a given integrity; developers will have more freedom to chose their safety engineering approach, but will still need to formulate a rigorous safety argument based on accumulated evidence.

### 2.3 IEC 61508

IEC 61508 “Functional Safety of Electrical / Electronic / Programmable Electronic Safety-Related Systems”[18] is a standard which covers a wide range of systems and their components. Part 2 in particular gives requirements for the development and testing of electrical, electronic and programmable devices. Here the *programmable* part of the systems is not addressed in detail; there are requirements for aspects of the design to be analysed, but no real requirements for implementation language or related aspects. It is the experience of the authors that DO-254 is more directly usable for developers than IEC 61508 Part 2.

### 2.4 Other standards

PLDs have been shown to be particularly useful in implementing cryptographic functions, for instance the Advanced Encryption Standard[20]. The Common Criteria guidance for IT security evaluation[7] does not distinguish between software executing on a microprocessor, ASICs or programs executing on PLDs; they may all form part of the Target of Evaluation (ToE) and require equally rigid reasoning with respect to the security requirements identified in the Protection Profile or Security Target for the ToE. The formal and semi-formal assurance required for ASIC and software designs at Evaluation Assurance Levels 5 to 7 is therefore required for PLD programs too.

The German TÜV has produced a draft document [34] on the use of PLDs in safety-related systems, giving specific guidance on the general modes of failure of these devices and possible design and test mitigations. This document is interesting to read as a supplement to DO-254 / Defence Standard 00-56 Issue 3, providing techniques which can generate evidence towards a safety argument. Its recommendations include:

- the use of high-level description languages for complex designs (although it classes VHDL and Verilog as high-level, whereas modern languages such as Handel C and Esterel are much more abstract);
- verification of synthesis and layout steps by comparing simulation results;
- design of the layout with consideration of EMC effects (e.g. crosstalk); and
- use of techniques to detect both systematic and random failure.

## 2.5 Conclusions on standards' requirements

The available standards vary significantly in what they *prescribe* for PLDs and what techniques they *suggest* are applicable. Defence Standard 00-54 is the most prescriptive, but as noted above is likely to become less so with its incorporation in the new release of Defence Standard 00-56.

The common requirements of the standards are:

1. to operate under an appropriate quality / safety management system;
2. to plan the development process and the safety argument in advance;
3. to consider both random and systematic failures;
4. to qualify tools involved directly in the compilation chain;
5. to use analytic techniques (“formal methods”) to verify high-integrity programs; and
6. to conduct the verification based on identified system hazards.

## 3 Current practice

We now examine three cases where PLDs were used in a critical function for a system.

Gibbons and Ames[8] reported on the use of an FPGA in a space-based tethering experiment where an unanticipated power-up characteristic of the chosen FPGA caused the effective loss of the satellite incorporating it. This occurred despite extensive testing, and one reason was that it was not possible to reproduce the transient spike twice within several hours – a classic transient fault. It is clear from this experience that extensive testing is not sufficient for mission- or safety-critical FPGAs; it is equally true that even formal analysis and proof would be unlikely to detect such a problem.

The first author has experience of constructing a Level A safety argument against RTCA DO-254 for a subsystem based on multiple PLDs. The approach used incorporated most of the recommended Appendix B methods as listed in Section 2.1:

- a functional failure path analysis, relating each design component to identified system hazards;
- use of a system architecture to mitigate the consequences of isolated failure (provision of a backup operational mode); and
- elemental analysis of each design component against the functional failures identified, using a combination of full-coverage testing and manual inspection of VHDL.

There was insufficient product service experience of the VHDL compiler to qualify the tool according to DO-254 requirements; instead, the synthesized output was inspected manually to ensure that the critical design components were still present and correctly connected. Formal methods were not necessary because of the size of the design and the simple nature of its specification.

One approach to the use of FPGAs in a hostile environment has been described by Lima et al. [22] The main environmental hazard in their target domain (space) is corruption of volatile memory via bombardment by high-energy particles. The architecture adopted is a triple-redundant design with fault detection and periodic “scrubbing” to reset look-up tables to known values. This is a classic example of mitigating an unavoidable hazard; however, the design and function greatly complicates the task of arguing system correctness. The user must balance increased general reliability against demonstrable correctness.

We now examine what recent research and development has contributed towards the problem of producing high-integrity PLD programs.

## 4 Recent research

Research relevant to safety-critical PLD program design includes:

1. specification and proof of parallel systems, enabling a correct-by-construction approach to program design;
2. model checking techniques to verify safety properties of an existing PLD design at a HDL or netlist level; and
3. the design and use of high-level programming languages to enable PLD programming at a more abstract level, possibly in a domain-specific language or tool.

We describe each of these aspects, including their relation to the earlier mentioned safety and security standards where appropriate.

### 4.1 Specification and proof techniques

Established parallel specification notations such as CSP[17] and LOTOS[19] are capable of describing the highly parallel structure of a PLD program, but have not yet been applied generally as specification notations for actual PLD programs. A contributory factor is likely to be the over-complexity of the notations compared to the simple synchronous structure of most PLD programs.

Earlier work by Breuer et al [4] on production of a refinement calculus directly targeting VHDL has a solid theoretical base, and (in theory) allows the production of VHDL designs which are demonstrably correct. This work also fell foul of over-complexity, and without tool support was impractical to apply efficiently to PLD program designs.

The authors have used the SRPT synchronous receptive process algebra to implement a formal specification and refinement systems for synchronous PLD programs. This work, described in [15], establishes refinement as a practical technique for at least small PLD designs, and indicates that it may scale well for certain classes of design. It is targeted directly at the specification and proof of PLD programs, but currently lacks tool support.

Refinement in parallel systems is an area of active research; the authors anticipate significant developments in techniques and tool support in this area in the next few years.

## 4.2 Model checking

Model checking is the application of graph theory and finite state machines to decide whether a temporal logic formula is maintained across all possible system states. It has become practical to apply it to verifying key properties of complex modern processors, for example the non-floating point operations of the Intel Pentium IV microprocessor which was verified as described by Schubert[32].

Model-checking is effective at deciding whether a design conforms to certain safety properties, but is vulnerable to the *state explosion* problem where designs of increasing size quickly become impractical to model-check. It is a retrospective activity, which may be beneficial for checking existing designs but does not easily allow the design of programs which are demonstrably correct throughout their development.

Model checking tools such as Solidify[2] are now starting to be used in PLD program verification, and can provide assurance that the design has suitable safety properties across all possible states. This is a more powerful argument for safety than simulation, since it is practically impossible to cover all possible system states for any designs other than the very simple, but there remains the question of tool qualification. As noted in Section 2.1, DO-254 requires either direct verification of the tool or in-service history – inspection of the tool output does not help qualification in this case. Neither of these are currently available.

Stepney[33] has shown how a subset of CSP compatible with the FDR2 model-checking tool can be transformed into a program in a Handel-C language subset, thereby allowing a design to be model-checked for correctness before a compilable version of the design is produced. FDR2 has a long in-service history and would be easier to qualify for medium levels of integrity.

## 4.3 High-level imperative programming

Since 1996 there has been a steadily growing interest in compiling imperative languages into HDLs (and hence into PLDs). The most popular approaches have

been based around C language syntax, presumably for its immediate appeal to most developers, although this syntax often hides complex parallel programming issues not present in sequential C.

Handel-C[5] is a modern high-level PLD programming language that owes much to the occam parallel programming language[23]. It has been used in a range of industrial applications including military and aerospace, although the authors do not know of any use of a Handel-C program in a safety-critical function. As noted in Section 4.2 above, a Handel-C subset can be the target of a compilation from model-checked CSP, and there is a toolset which can perform the usual verification activities at each development stage. However, the Handel-C compiler is complex and as yet is not known to be amenable to qualification.

Gupta et al. [10] have described a synthesis process which transforms pointer-free non-recursive ANSI C to VHDL. Unusually, it places much of the parallel programming activity within the toolset; the programming language cannot express parallel concepts. Because of this, the approach suffers from the well-documented deficiencies of the C language with respect to safety and correctness; see Romanski[29] for elaboration on this. The fundamental question is how the developer can be sure that his programming intent has been captured and preserved by the compilation chain.

The conventional software programming language Ada 95 has been examined by the authors[16] and by Audsley and Ward[35] as a design and implementation language for PLDs. Audsley and Ward have addressed the compilation of legacy Ada code into a one-hot state machine, aiming to maintain the existing safety argument for the code by qualifying only the PLD-targeting compiler. This work is in progress but has demonstrated coverage of many Ada constructs including Ada's parallel programming features.

The authors have chosen a complementary approach, taking new programs written in the SPARK high-integrity annotated Ada 95 subset[3] and transforming identified coherent subsections directly into PLD software while maintaining overall (and justifiable) program correctness. Ada's strong numerical typing and SPARK's ability to prove programs free from run-time errors combine to simplify the transformation process. The SPARK toolset has strong in-service qualification evidence, although there is no formally released SPARK-to-HDL compiler as yet so any qualification argument would have to relate the compiled HDL to the original SPARK.

#### 4.4 High-level declarative programming

High-level declarative languages are less used in PLD programming, perhaps because the declarative nature of VHDL and Verilog means that the programming models of high-level declarative languages are less obviously different to what is already used. Nevertheless, they require consideration.

Esterel is a language designed for programming action systems, and so is not conventionally imperative but not fully declarative. It has a formal synchronous semantics so Esterel programs can be meaningfully analysed for correctness and safety. It was used by Hammarberg and Nadjm-Tehrani [11] to demonstrate

the use of an aircraft hydraulic system component. The Esterel program was compiled through VHDL, which is a common interim language for compilation high-level designs. This approach has the problem that verification must justify the semantic gap between Esterel and VHDL.

Cryptol [21] is an example of a domain-specific language (targeting cryptographic applications) which may be suitable for programming PLDs. It is more appropriate to security-critical rather than safety-critical systems, but as noted in Section 2.4 the fundamental requirements on the language and compiler are similar. As a functional programming language it is inherently easier to analyse than many imperative languages, but there remains the problem of demonstrating that the compilation chain may not introduce insecurities (e.g. leak information) or errors.

Ruby[9] was one of the earliest attempts to abstract away from common HDLs for FPGA programming. It develops designs by composing sub-designs sequentially, in parallel and via functions. It has been developed more recently into the Lava language[6], embedded in the Haskell functional programming language, which allows formal verification of hardware designs. It is not known to have been used in industrial PLD programming, but the combination of formal verifiability and small semantic gap between design and PLD implementation may make it appropriate for some implementations.

Pebble[24] can be viewed as a simplified synchronous (single clock) subset of VHDL. Its great strength is its simplicity; Pebble has been given a synchronous semantics by Hilton[13] and so can be meaningfully analysed for high integrity systems.

## 5 Discussion and conclusions

As part of an ongoing study into reducing avionics lifecycle costs, QinetiQ Ltd. have produced reports advising UK military Integrated Project Teams (IPTs) on the use of PLDs in Advanced Avionics Architectures. The report on PLD programming[12] recommends that IPTs:

- plan to develop and verify PLD programs in the same way as software programs;
- plan the safety argument from the start, and build up evidence throughout development;
- use mature tools, amenable to qualification and supported throughout the project life;
- use programming languages with clearly defined syntax, and ideally a full semantics;
- investigate the use of formal notations and analysis techniques to increase verifiability; and
- do not use PLDs just to avoid developing safety-critical software.

In this paper we have identified how existing tools, languages and technologies measure up against existing safety standards, providing specific advice to project

teams considering the above recommendations. We now consider how they might be combined to increase the demonstrable integrity level of PLD programs.

### 5.1 Future work

There is a clear need for some level of formal verification of PLD designs. Simulation alone is inadequate. Existing tools are not yet amenable to qualification. Formal notations and proof systems do not have appropriate tool support. There needs to be a combination of developments including:

1. industrial use of design languages with formal definitions;
2. development of qualifiable compilers for these languages;
3. more widespread use of model checking in industrial designs;
4. qualification of verification tools via a combination of in-service evidence and analysis of the tool design; and
5. investigation of refinement and proof techniques with a view to supporting complex PLD designs at the highest levels of integrity.

### 5.2 Summary

RTCA DO-254 is a recent standard, and developers are now just starting to apply it in practice. The authors' experience with it to date is that it admits justification of PLD program safety at Level A with the exact approach defined by the developer. The range of verification methods proposed for Level A and Level B PLD programs allows a comprehensive multi-faceted argument for program safety. The emerging issue 3 of UK Defence Standard 00-56 and the Common Criteria documents are consistent with this requirement of formal reasoning at the higher integrity levels.

In this paper we have described the current state of the art in the practice and theory of producing high-integrity PLD applications conforming to these standards. We have identified the key deficiencies in tools and languages and proposed ways in which they may be fixed. We have examined appropriate areas of research and described how they could be developed to be usable in real system developments. All of the components needed for high-integrity PLD programming exist; future work must focus on combining them effectively.

### 5.3 Acknowledgements

The authors are grateful to Brian Dobbing and David Cooper from Praxis Critical Systems Ltd. and Tim Murray from QinetiQ Ltd. for advice and information given during the writing of this paper.

## References

1. ACM SIGDA. *Eleventh ACM International Symposium on Field-Programmable Gate Arrays*. ACM Press, February 2003.

2. Static functional verification with Solidify. White Paper, 2001.
3. John Barnes. *High Integrity Software: The SPARK Approach to Safety And Security*. Addison Wesley, April 2003.
4. Peter T. Breuer, Carlos Delgado Kloos, Andrés Marín López, Atividad Martínez Madrid, and Luis Sánchez Fernández. A refinement calculus for the synthesis of verified hardware descriptions in VHDL. *ACM Transactions on Programming Languages and Systems*, 19(4):586–616, July 1997.
5. Celoxica Ltd. *Handel-C Language Reference Manual*, 3.1 edition, 2002.
6. Koen Claessen and Mary Sheeran. *A Tutorial on Lava: A Hardware Description and Verification System*, August 2000.
7. Common Criteria. *Common Criteria for Information Technology Security Evaluation*, August 1999.
8. Wally Gibbons and Harry Ames. Use of FPGAs in critical space flight applications – a hard lesson. In *1999 Military and Aerospace Applications of Programmable Devices and Technologies Conference*. Space Dynamics Laboratory, Utah State University, September 1999.
9. Shaori Guo and Wayne Luk. Compiling Ruby into FPGAs. In W. Moore and W. Luk, editors, *Field-Programmable Logic and Applications (FPL'95)*, volume 975 of *Lecture Notes In Computer Science*, pages 188–197. Springer-Verlag, August 1995.
10. Sumit Gupta, Nikil Dutt, Rajesh Gupta, and Alex Nicolau. SPARK: a high-level synthesis framework for applying parallelizing compiler transformations. In N. Ranganathan, editor, *Proceedings of the Sixteenth International Conference on VLSI Design*. Center for Embedded Computer Systems, University of California at Irvine, January 2003.
11. Jerker Hammarberg and Simin Nadjm-Tehrani. Development of safety-critical reconfigurable hardware with Esterel. In *Eighth International Workshop on Formal Methods for Industrial Critical Systems*. Linköping University, Elsevier, June 2003.
12. Adrian Hilton. Practical guide to certification and re-certification of AAvA software elements: Software for programmable logic devices. Technical report, QinetiQ, July 2003.
13. Adrian J. Hilton. *High Integrity Hardware-Software Co-design*. PhD thesis, The Open University, April 2004. To appear.
14. Adrian J. Hilton and Jon 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.
15. Adrian J. Hilton and Jon G. Hall. Refining specifications to programmable logic. In John Derrick, Eerke Boiten, Jim Woodcock, and Joakim von Wright, editors, *Proceedings of REFINE 2002*, volume 30 of *Electronic Notes in Theoretical Computer Science*. Elsevier, November 2002.
16. Adrian J. Hilton and Jon G. Hall. High-integrity interfacing to programmable logic with Ada. In Albert Llamosí and Alfred Strohmeier, editors, *Proceedings of the 9th International Conference on Reliable Software Technologies (Ada-Europe 2004)*, June 2004.
17. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, 1985.
18. International Electrotechnical Commission. *IEC Standard 61508, Functional Safety of Electrical / Electronic / Programmable Electronic Safety-Related Systems*, March 2000.

19. International Organisation for Standardisation. *ISO/IEC 8809:1989; LOTOS: A formal description technique based on the temporal ordering of observational behaviour*, 1993.
20. Kimmo U. Järvinen, Matti T. Tommiska, and Jorma O. Skyttä. A fully pipelined memoryless 17.8 Gps AES-128 encryptor. In *Eleventh ACM International Symposium on Field-Programmable Gate Arrays (FPGA'03)* [1], pages 207–215.
21. John Launchbury and Satnam Singh. An approach to compiling Cryptol to FPGAs. In *3rd Annual High Confidence Software and Systems Conference, Proceedings*, pages 137–146. Galois Connections and Xilinx, April 2003.
22. Fernanda Lima, Luigi Carro, and Ricardo Reis. Reducing pin and area overhead in fault-tolerant FPGA-based designs. In *Eleventh ACM International Symposium on Field-Programmable Gate Arrays (FPGA'03)* [1], pages 108–117.
23. INMOS Ltd. *occam Programming Manual*. Prentice-Hall International, 1984.
24. Wayne Luk and Steve McKeever. Pebble — a language for parametrised and reconfigurable hardware. In R. W. Hartenstein and A. Keevallik, editors, *Proceedings of the 8th International Workshop on Field Programmable Logic (FPL'98)*, volume 1482 of *Lecture Notes In Computer Science*, pages 9–18. Springer-Verlag, September 1998.
25. Defence Standard 00-56 issue 3 (public comment draft), July 2003. Safety Management Requirements for Defence Systems.
26. Defence Standard 00-55 issue 2, August 1997. Requirements for Safety-Related Software In Defence Equipment.
27. Defence Standard 00-56 issue 2, December 1996. Safety Management Requirements for Defence Systems.
28. Interim Defence Standard 00-54 issue 1, March 1999. Requirements for Safety Related Electronic Hardware in Defence Equipment.
29. George Romanski. Review of 'Safer C' (by Les Hatton). Technical report, Thomson Software Products, January 1996.
30. RTCA / EUROCAE. *RTCA DO-178B / EUROCAE ED-12: Software Considerations in Airborne Systems and Equipment Certification*, December 1992.
31. RTCA / EUROCAE. *RTCA DO-254 / EUROCAE ED-80: Design Assurance Guidance for Airborne Electronic Hardware*, April 2000.
32. Tom Schubert. High level formal verification of next-generation microprocessors. In *Proceedings of the 40th Design Automation Post-Conference*. Intel Corporation, ACM Press, June 2003.
33. Susan Stepney. CSP/FDR2 to Handel-C translation. Technical Report YCS-2002-357, Department of Computer Science, University of York, June 2003.
34. Requirements for the design and the use of ASIC's, FPGA's, EPLD's and comparable components in safety-related systems, April 1999. [http://tuvasi.com/asic\\_03.htm](http://tuvasi.com/asic_03.htm).
35. M. Ward and N. C. Audsley. Hardware implementation of the Ravenscar Ada tasking profile. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*. ACM Press, 2002.