Technical Report N° 2005/06

Developing Critical Systems with PLD Components

Adrian J. Hilton
Jon G. Hall

12th April 2005

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[1] and Jon G. Hall[2]

[1] Praxis High Integrity Systems, 20 Manvers Street, Bath BA1 1PX, England
Adrian.Hilton@praxis-his.com
[2] The Open University, Walton Hall, Milton Keynes MK7 6AA, England
J.G.Hall@open.ac.uk

Abstract. Understanding the roles that rigour and formality can have in the design of critical systems is critical to anyone wishing to contribute to their development. Whereas knowledge of these issues is good in software development, in the use of hardware – specifically programmable logic devices (PLDs) and the combination of PLDs and software – the issues are less well known. Indeed, even in industry there are many differences between current and recommended practice and engineering opinion differs on how to apply existing standards.

This has led to gaps in the formal and rigorous treatment of PLDs in critical systems. In this paper we examine the range of and potential for formal specification and analysis techniques that address the requirements of verifiable PLD programs. We identify existing formalisms that may be used, and lay out the areas of contributions that academia can make to allow high-integrity PLD programming to be as feasible as high-integrity software development.

Key words: FPGA, PLD, survey, programmable logic, parallel, process algebra, programming languages

1 Introduction

Programmable Logic Devices are increasingly important components of high integrity systems. By offloading tasks from the main CPU onto a PLD, system performance limits can be attained more easily. 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 Field Programmable Gate Arrays (FPGAs) such as the Xilinx Virtex family and Complex Programmable Logic Devices (CPLDs) such as the Altera FLEX 10K family. The practices which they recommend vary in rigour and in practicality. Adherence to these standards is

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. This paper will focus on the existing standards and techniques primarily used in European countries, although several of the standards examined have American origin or usage too.

Section 2 of this paper summarises the characteristics of PLDs and describes how and where they are used. In Section 3 we describe and analyse the main safety and security standards relevant to PLD program development. Section 4 describes current industrial practice including case studies of PLD use. Section 5 summarises recent research relevant to safe or provably correct PLD program development. Finally, Section 6 summarises the paper and indicates routes for future work.

2 Programmable Logic Devices

PLDs were a development of the simple Programmable Logic Array (PLA) which has been available in electronics design since the early 1980s. The early history of field-programmable logic is reviewed by Moore in [1]. The most common (and interesting) form of PLD in use is a Field Programmable Gate Array.

2.1 Device design characteristics

The key characteristics of an FPGA are that they can have their program contents changed upon power-up ("field-programmable") and have an internal structure of a regular array of logic cells ("gate array"). They provide a logic device of relatively low complexity, computing some function of a set of digital inputs to produce a set of digital outputs. This is done in a highly-parallel manner. FPGAs have semi-permanent state, held in programmed lookup tables, typically implemented as static random access memory (SRAM). These tables are programmed by the download of lookup table data from an external source.

FPGAs differ from other programmable logic devices (PLAs, PROMs or CPLDs) by allowing a more complex flow of data through themselves. They also differ from Application Specific Integrated Circuits (ASICs) by trading speciality of design for speed of development and economy of small-scale production.

2.2 Use of PLDs

PLDs are typically used in building a prototype system in place of a custom ASIC. It is significantly cheaper and quicker to use such devices when the alternative is a minimum production run of 5000 ASICs in a different company's fabrication plant ("fab"). A small-scale single run of ASIC production can easily

cost \$750,000 and take months from submission of VHDL design information to the fab to the arrival of the silicon.

There can be significant commercial gain in using PLDs rather than ASICs. Time-to-market is reduced, since there is not the delay in setting up and making the ASIC production run, and there is little overhead if an error is subsequently found in the device. There is also the potential for increased time-in-market, providing mid-life upgrades to the PLD code without having to replace the hardware.

The critical systems industry would like to implement systems based on PLDs for the reasons stated above, but cannot if the resulting systems exhibit the common failure modes of PLDs. One role of formality is to argue that PLDs are suitable for such systems.

2.3   Programming PLDs

The implementation of a PLD-based system can be done in many ways. The equivalent of microprocessor object code will be a device-specific "netlist" which specifies the data to be loaded into each cell and router of the device. To reach netlist form, several intermediate compilation steps are normally required; the place-and-route work involved in this compilation is NP-hard.

The majority of PLDs are programmed in VHDL [2] or Verilog[3], either directly or with a higher-level design language being compiled through them. These Hardware Description Languages (HDLs) have substantial standard libraries, allowing a certain amount of code reuse. They model the PLD as interconnected blocks rather than providing higher-level functions such as iteration or procedure call. Even if a higher-level language or design tool is used, it will normally compile its input into VHDL or Verilog.

There is a subset relation between *behavioural* and *synthesizable* VHDL. The former is an expressive imperative language incorporating explicit iteration, alternation and a constructive type system. The latter is a small and simple subset (Register Transfer Level) which can be compiled directly into combinations of logic gates and latches. Going from the former to the latter is non-trivial, and in general it is too difficult to automate the translation process.

Design languages at an abstraction level above HDLs tend to split into three variants:

1. explicit parallel general-purpose languages;
2. domain-specific languages designed to solve a certain class of problems in an inherently parallel way; or
3. modifications of existing imperative languages.

The classic general-purpose parallel programming language is occam[4]. The programming model underpinning occam has been developed initially into the Handel language, embedded in a functional programming syntax[5], and more recently into the commercially-supported Handel-C language[6]. Handel-C has a C-like syntax but incorporates explicit parallelism; the semantic model is much

closer to occam than to C. Another example, this time a compositional hardware language is Ruby [7]. Ruby was based on the idea that circuits are built from parts by a process of composition, which has mathematical properties similar to the composition of functions and relations. A modern development of Ruby is Lava[8], a prototype HDL developed by and in use at Chalmers University in Sweden. It trades off the expressiveness of behavioural VHDL or Verilog for compactness and simplicity of descriptions of common circuit layouts.

An example of a domain-specific language is the synchronous programming language Esterel[9], used to specify and implement action systems. This has been applied by Hammarberg et al.[10] in a demonstration hydraulic fluid detection system. A more specific language is CoreFire, described in [11]. Developers write CoreFire programs in a "sticks and bubbles" graphical notation of data flow, and compile them to high-performance applications which run on Annapolis Wild FPGA boards.

Commonly used imperative languages which have been compiled into PLDs include C[12, 13], Java[14] and Ada[13, 15]. The specific difficulty in using these languages is in expressing PLD-specific concepts such as fine-grain parallelism which is not normally part of the original language. System-C[16] and Handel-C are examples of how C's syntax can be extended to express parallel concepts.

2.4   PLD formalisms

Substantial effort was made in the 1980s and 1990s to develop a hardware design language that supported formal reasoning and abstraction, two features absent from HDLs such as VHDL and Verilog. The main exponent of this approach was ELLA[17], a non-proprietary language with a formal basis.

ELLA was not a strict competitor to VHDL and Verilog, but in practice it was treated as such. The relatively small size of hardware designs made design in existing HDLs feasible, if difficult. It may be that, as hardware designs and PLD dies continue to grow in size, high-integrity requirements will make ELLA or similar design languages more necessary. This change was seen in software with the emergence of structured design methods as program sizes grew beyond what one developer could manage; it is reasonable that a similar effect will eventually be seen in programmable logic program design.

The (parallel) formalisms that apply best to the massively parallel PLD structure are parallel process algebras such as CSP[18] and CCS[19]. The main problem in representing small-small digital logic constructs such as AND and OR gates with CSP is that CSP is not *receptive*; a CSP process representing a logic gate may refuse events representing voltage changes on its input wires, whereas the logic gate may not. A secondary problem is that CSP is asynchronous by design; processes only synchronise through shared events (or communication on channels). Most PLD designs are synchronous, with design blocks sharing a single clock. Therefore the receptive and synchronous aspects of the PLD architecture would have to be represented artificially in a CSP model. A better approach is to use an algebra incorporating these features, and the authors have successfully

applied the synchronous receptive process algebra SRPT[20] in a refinement system for PLD programming[21].

Recent work by Boulanger et al. [22] has attempted to use the B method to produce BHDL, a VHDL reformulation in B. This work is early and tool support is limited, but it represents a promising avenue for certain applications.

2.5 Summary

We have seen that PLDs present a different programming architecture to conventional microprocessors, and have examined different programming methods for this synchronous highly parallel model. We now discuss the demands that safety and security certification make for rigorous development and verification of PLD programs.

3 Current safety and security standards

The main safety standards relevant to PLD programming in Europe are:

- RTCA DO-254[23] which is an international civil aviation standard;
- UK Interim Defence Standard 00-54[24] and UK Interim Defence Standard 00-56[25] which are UK standards for defence-related systems;
- IEC 61508[26] which is a European standard intended to apply to a wide range of systems; and
- the Common Criteria[27] which is an international standard for developing secure systems.

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 relevant with 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.

In this section we analyse the content of each of these standards in detail.

3.1 RTCA DO-254 / EUROCAE ED-80

The airborne electronic hardware development guidance document RTCA DO-254 / EUROCAE ED-80[23] is the counterpart to the well-established civil avionics software standard RTCA DO-178B / EUROCAE ED-12B. 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 recommendations on suitable general practice. It is not a prescriptive standard; 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 between 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.

3.2 UK Defence Standards

Within the UK, Interim Defence Standard 00-54[24] 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 Issue 2[28].

The UK Defence Standards are currently being rewritten so that 00-54, and its software counterpart 00-55, are being rolled together into the 00-56 standard. Issue 3 of 00-56 has just (January 2005) been released as an interim standard. This version[25] explicitly equates regular software and PLD programs as safety-related complex electronic elements (SRCEE) in Part 2, §15.1.

Overview 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 new 00-56 is less prescriptive, instead requiring that "compelling evidence that safety requirements have been met. Where possible, objective, analytical evidence shall be provided." (Part 1,§11.3.1).

Risk is regulated (in the UK) on the basis of being reduced ALARP (As Low As is Reasonably Practical). This stems from a UK Court of Appeal decision on the 1949 case Edwards vs. The National Coal Board[29] 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."

This is significant because it means that if it is feasible and not disproportionately expensive to do formal analysis, and there is a demonstrable gain in reliability from this, then a UK court is likely to expect it to be done for the system risk to be regarded as ALARP.

High-integrity requirements Formal specification and analysis of PLD programs are *mandated* at all safety integrity levels for 00-54. This posed a practical problem for developers since in 1999 (its year of issue) there were no known tool-supported specification or proof notations which were generally applicable to PLD programming. Each project required a from-scratch selection of, and capability development in, notations and analysis techniques. This is risky and potentially expensive.

The new 00-56, as noted above, makes no prescription for methods to be used. However, the risk involved in using the SRCEE is required to be ALARP and specifically requires evidence to validate the safety argument including (Part 1, §19.2):

1. direct evidence from analysis;
2. direct evidence from demonstration (testing and/or operation), including quantitative evidence;
3. direct evidence extracted from the review process;
4. process evidence showing good practice in development, maintenance and operation; and
5. qualitative evidence for good design, including expert testimony etc.

The quantitative aspect of item 2 is significant because work by Littlewood[30] has shown that conventional testing *cannot* show that a system is highly reliable in a statistically significant way. This applies to systems at the SIL-3 or SIL-4 integrity levels, or Levels A and B in DO-254 terms.

00-56 also requires each tool in the compilation chain to have suitable arguments or analysis in place to show that it does not introduce significant errors into the system.

3.3   Other standards

IEC 61508 "Functional Safety of Electrical / Electronic / Programmable Electronic Safety-Related Systems"[26] 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.

PLDs have been shown to be particularly useful in implementing cryptographic functions, for instance the Advanced Encryption Standard (AES). The Common Criteria guidance for IT security evaluation[27] 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 rigorous 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.

4   Current practice

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

Gibbons and Ames[31] 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 3.1:

- 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 synthesised 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.

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

5 Recent research

Recent 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.

5.1 Specification and proof techniques

Established parallel specification notations such as CSP and LOTOS[32] 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 [33] 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, initially described in [21] and extended in [34], 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.

5.2   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 as described by Schubert[35]. It 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 beneficial for checking a complete design but cannot usually be applied until near the end of a development.

Model checking tools such as Solidify from Saros Technologies 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 3.1, 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[36] 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.

5.3 High-level programming

Imperative 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 is a modern high-level PLD programming language that owes much to the occam parallel programming language[4]. 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 5.2 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. [37] 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. 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[38] and by Audsley and Ward[39] 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. Ada has the advantage that its syntax is very close to the syntax of *behavioural* VHDL; however, *synthesizable* VHDL is more restrictive.

The authors have chosen a complementary approach, taking new programs written in the SPARK high-integrity annotated Ada 95 subset[40] and transforming identified coherent subsections directly into PLD software while maintaining overall (and justifiable) program correctness[38]. 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.

Declarative 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 [10] 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.

Pebble[41] 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[21] and so can be meaningfully analysed for high integrity systems. This means that any compiler which targets Pebble can conceivably avoid qualification by comparing the source language with the Pebble output. Pebble normally compiles directly to VHDL.

Ruby, as noted in Section 2.3, develops designs by composing sub-designs sequentially, in parallel and via functions. A recent marrying of Ruby and Pebble is Quartz[42] which uses a Ruby-like compositional design process and compiles the result into Pebble code. Quartz has a prototype compiler, as does Pebble, but they are not yet at the stage where they could be used in a high-integrity development.

6   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[43] 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 for PLD program development measure up against existing safety standards, providing specific advice to project teams considering the above recommendations. We now consider how formal techniques may increase the demonstrable integrity level of PLD programs.

6.1   Future work

Industry has 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 (formal?) compilers for these languages, building on related work in conventional software[44];
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.

The focus in this work should be on making this advances usable by a typical industrial development or verification team, with the minimum of change in the existing development process.

6.2 Safety and security standards

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-facetted 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. The equating of programmable hardware with conventional microprocessor-based software in these standards means that the rigorous analysis and specification techniques required for high-integrity software will require analogues in high-integrity programmable hardware.

6.3 Summary

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.

References

1. Moore, W., Luk, W., eds.: FPGAs: Edited Proceedings of the International Workshop on Field Programmable Logic and Applications. In Moore, W., Luk, W., eds.: FPGAs: Edited Proceedings of the International Workshop on Field Programmable Logic and Applications. (1991)
2. IEEE: IEEE Std. 1076-1987: IEEE Standard VHDL Language Reference Manual. (1991)

3. IEEE: IEEE Std. 1364-1995: IEEE Standard Description Language. (1995) Based on the Verilog(TM) Hardware Description Language.
4. Hoare, C.A.R.: occam Programming Manual. Prentice-Hall International (1984)
5. Page, I., Spivey, M.: How to program in Handel. Technical report, Oxford University Computing Laboratory (1993)
6. Celoxica Ltd.: Handel-C Language Reference Manual. 3.1 edn. (2002)
7. Jones, G., Sheeran, M.: Circuit design in Ruby. In Staunstrup, J., ed.: Formal Methods for VLSI Design, North-Holland (1990) 13–70
8. Claessen, K., Sheeran, M.: A Tutorial on Lava: A Hardware Description and Verification System. (2000)
9. Berry, G.: The foundations of Esterel. In Plotkin, G., Stirling, C., Tofte, M., eds.: Proof, Language and Interaction: Essays in Honour of Robin Milner. Foundations of Computing. MIT Press (2000)
10. Hammarberg, J., Nadjm-Tehrani, S.: Development of safety-critical reconfigurable hardware with Esterel. In: Eighth International Workshop on Formal Methods for Industrial Critical Systems, Linköping University, Elsevier (2003)
11. McHale, J.: The new frontier: Reconfigurable computing. Military and Aerospace Electronics (2002)
12. Sweeney, C.: Optimal features of hardware platforms. Technical Report 006, Embedded Solutions Ltd. (1998)
13. Sheraga, R.J.: ANSI C to behavioural VHDL translator, Ada to behavioural VHDL translator. The RASSP Digest **3** (1996)
14. Macketanz, R., Karl, W.: JVX — a rapid prototyping system based on Java and FPGAs. [45] 99–108
15. Ward, M., Audsley, N.C.: Hardware implementation of programming languages for real-time. In: Proceedings of the Eighth IEEE Real-Time Embedded Technology and Applications Symposium (RTAS'02), IEEE (2002) 276–284
16. Connell, J., Johnson, B.: Early HW/SW integration using SystemC v2.0. In: Proceedings of the Embedded Systems Conference, ARM and Synopsys Inc. (2002)
17. Morison, J.D., Clarke, A.S.: ELLA 2000; a Language for Electronic System Design. McGraw-Hill Book Company (1993)
18. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall International (1985)
19. Milner, R.: Calculi for synchrony and asynchrony. Theoretical Computer Science **25** (1983) 267–310
20. Barnes, J.E.: A mathematical theory of synchronous communication. Technical report, Oxford University Computing Laboratory (1993)
21. Hilton, A.J., Hall, J.G.: Refining specifications to programmable logic. In Derrick, J., Boiten, E., Woodcock, J., von Wright, J., eds.: Proceedings of REFINE 2002. Volume 30 of Electronic Notes in Theoretical Computer Science., Elsevier (2002)
22. Aljer, A., Devienne, P., Tison, S., Boulanger, J.L., Mariano, G.: BHDL: Circuit design in B. In Lilius, J., Balarin, F., eds.: Third International Conference on Application of Concurrency to System Design, Laboratoire d'Informatique Fondamentale de Lille, Université de Compiègne, Institut National de Recherche sur les Transports et leur Sécurité (2003) 241
23. RTCA / EUROCAE: RTCA DO-254 / EUROCAE ED-80: Design Assurance Guidance for Airborne Electronic Hardware. (2000)
24. MoD: Interim Defence Standard 00-54 issue 1 (1999) Requirements for Safety Related Electronic Hardware in Defence Equipment.

25. MoD: Interim Defence Standard 00-56 issue 3: Safety management requirements for defence systems. Technical report, UK Ministry of Defence (2005) http://www.dstan.mod.uk/.

26. International Electrotechnical Commission: IEC Standard 61508, Functional Safety of Electrical / Electronic / Programmable Electronic Safety-Related Systems. (2000)

27. Common Criteria: Common Criteria for Information Technology Security Evaluation. (1999)

28. MoD: Defence Standard 00-56 issue 2. Technical report, Ministry of Defence (1996) Safety Management Requirements for Defence Systems.

29. Asquith, J.: Edwards v. National Coal Board. All England Law Report **1** (1949) 747

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

31. Gibbons, W., Ames, H.: 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 (1999)

32. International Organisation for Standardisation: ISO/IEC 8809:1989; LOTOS: A formal description technique based on the temporal ordering of observational behaviour. (1993)

33. Breuer, P.T., Kloos, C.D., López, A.M., Madrid, A.M., Fernández, L.S.: A refinement calculus for the synthesis of verified hardware descriptions in VHDL. ACM Transactions on Programming Languages and Systems **19** (1997) 586–616

34. Hilton, A.J.: High Integrity Hardware-Software Co-design. PhD thesis, The Open University (2004)

35. Schubert, T.: High level formal verification of next-generation microprocessors. In: Proceedings of the 40th Design Automation Post-Conference, Intel Corporation, ACM Press (2003)

36. Stepney, S.: CSP/FDR2 to Handel-C translation. Technical Report YCS-2002-357, Department of Computer Science, University of York (2003)

37. Gupta, S., Dutt, N., Gupta, R., Nicolau, A.: SPARK: a high-level synthesis framework for applying parallelizing compiler transformations. In Ranganathan, N., ed.: Proceedings of the Sixteenth International Conference on VLSI Design, Center for Embedded Computer Systems, University of California at Irvine (2003)

38. Hilton, A.J., Hall, J.G.: High-integrity interfacing to programmable logic with Ada. In Llamosí, A., Strohmeier, A., eds.: Proceedings of the 9th International Conference on Reliable Software Technologies (Ada-Europe 2004). (2004)

39. Ward, M., Audsley, N.C.: 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)

40. Barnes, J.: High Integrity Software: The SPARK Approach to Safety And Security. Addison Wesley (2003)

41. Luk, W., McKeever, S.: Pebble — a language for parametrised and reconfigurable hardware. [45] 9–18

42. Pell, O.: Quartz: A new language for hardware description. final year project report, Department of Computing, Imperial College of Science, Technology and Medicine (2004)

43. Hilton, A.: Practical guide to certification and re-certification of AAvA software elements: Software for programmable logic devices. Technical report, QinetiQ (2003)

44. Stepney, S.: Incremental development of a high-integrity compiler: Experience from an industrial development. In: Proceedings of the Third IEEE High-Assurance Systems Engineering Symposium (HASE'98), Washington D.C. (1998)
45. Hartenstein, R.W., Keevallik, A., eds.: Field-Programmable Logic and Applications: From FPGAs to Computing Paradigm, 8th International Workshop (FPL'98), Proceedings. In Hartenstein, R.W., Keevallik, A., eds.: Proceedings of the 8th International Workshop on Field Programmable Logic (FPL'98). Volume 1482 of Lecture Notes In Computer Science., Springer-Verlag (1998)