

Mandated Requirements for Hardware/Software Combination in Safety-Critical Systems

Adrian J. Hilton
Praxis Critical Systems Ltd.
20 Manvers Street
Bath BA1 1PX
England
+44 1225 466991
adi@suslik.org

Jon G. Hall
The Open University
Walton Hall
Milton Keynes
England
+44 1908 652679
j.g.hall@open.ac.uk

ABSTRACT

Safety-critical systems are an important subset of high-assurance systems. Higher performance requirements have led to the increased use of combined hardware/software systems therein, with hardware devices taking processing load off software.

As might be expected, safety-critical systems have many requirements made of them by established standards. By implication, and now by emerging safety standards, such requirements must be discharged over hardware/software combinations, with important ramifications for best practice.

In this paper we discuss the impact that such requirements have on the co-development of hardware/software combinations, and suggest adaptations of existing best practice that could discharge them.

Keywords

Process, safety-critical, SPARK Ada, programmable logic

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 performance. The impact for safety-critical system development is that requirements for the hardware/software combination become subject to the relevant standards, such as UK Defence Standard 00-54 [16] and IEC 61508 [12], requiring reasoning about the safety and correctness of PLDs.

Moreover, technological improvements mean that PLD development becomes more like software development in terms of program size, complexity, and the need to clarify a program's purpose and structure. Taken

together, these have important implications for hardware/software co-development.

This paper assesses the impact that requirements from mandated standards have on hardware/software co-development, in the context of existing software development best practice for high assurance systems. In addition, we show how best practice might be adapted to programmable logic devices without incurring undue overhead in system development time.

2 SAFETY STANDARDS

A *safety-critical system* is a collection of components acting together where interruption of the normal function of one or more components may cause injury or loss of life.

Although the visibility of safety-critical components within larger systems is often low to the end-user, they are increasingly part of the lives of ordinary people and so form an important subset of high-assurance systems. Examples of such systems in general public life are air traffic control centres and railway signalling systems.

As such, their development is the subject of many standards. One example is UK Defence Standard 00-54 (DefStan 00-54) [16] – an interim standard for the use of safety-related electronic hardware (SREH) in UK defence equipment¹ – relates to systems developed under a safety systems document such as IEC 61508 [12]. Other standards include [18].

DefStan 00-54 contains the following requirements of the development process which are of particular interest to us:

(§12.2.1) [That] a formally defined language which supports mathematically based reasoning and the proof of safety properties shall be used to specify a custom design²;

¹Both the authors are British, and so are most familiar with British standards. Other such standards include [18]

²'custom design' refers to the non-standard components of the electronic component under examination; in particular, this includes a PLD's program data

(§13.4.1) [That] safety requirements shall be incorporated explicitly into the hardware specification using a formal representation; and

(§13.4.4) [That] correspondence between the hardware specification and the design implementation shall be demonstrated by analytical means, subject to assumptions about physical properties of the implementation.

DefStan 00-54 notes that widely used standard hardware description languages (HDLs) without formal semantics, such as VHDL and Verilog, present compliance problems if used as a design capture language: Z [19] is suggested as an example of a suitable language.

Although DefStan 00-54 is interim, the concerns which it expresses about existing practices and its suggestions for process improvements are worth careful scrutiny. In particular, the requirement for ‘formal representation’ which supports reasoning about programmable logic behaviour is expected to appear in the final version; with this in mind we describe a development process for hardware/software systems which aims to satisfy the requirements of DefStan 00-54.

We now examine the best practice in safety-critical software development to identify practices appropriate for systems incorporating software and programmable hardware in their architecture.

3 CURRENT BEST PRACTICE

The best practice in the development of software for safety-critical systems can be exhibited by software which has been developed to a software standard analogous to DefStan 00-54, and which is in current use. We take as an example the SHOLIS helicopter-landing system, in use on Royal Navy Duke-class frigates, which was the first software developed under UK Defence Standard 00-55 [15].

The development of SHOLIS is described in detail in [13]. The developers made extensive use of formal methods techniques including specification and proof in Z, use of static analysis tools to analyse the program code, and semi-automated proof of program properties such as absence of run-time exceptions. The Z proof phase was found to be *significantly* the most efficient phase at finding faults, and the ability to prove the absence of run-time errors improved developer and user confidence in the system. From this we see that best practice for the use of PLDs should facilitate similar analytical techniques.

The extra complications brought on by the use of programmable logic are the highly parallel nature of PLD computations, interfacing to other system components, and timing issues. Timing issues can be resolved to some

extent by simulation, and by use of a synchronous design for the PLD. Interfacing asynchronously to other components is a relatively well-understood problem. Correct analysis of the parallel structure of PLDs is therefore key to extending software development best practice to cover PLDs.

4 PLDS IN SAFETY-CRITICAL SYSTEMS

The Use Of Programmable Logic

Programmable logic devices have developed from the basic Programmable Logic Array (PLA) into the modern day Field Programmable Gate Array (FPGA). PLDs are capable of substantial computation rates in a restricted set of problems; with better development processes, this could lead to safety critical systems being better able to meet their real-time requirements. In other arenas, their use already improves efficiency; for example, Chodowicz *et al*, in [8], described programmable logic device implementations of the Advanced Encryption Standard algorithm final-round contenders which provided encryption throughput of between 7.5 and 16.8 Gbit/sec.

Moreover, and importantly for safety-critical systems, as well as improving efficiency, locating processing in PLDs may remove the need to design multiple CPUs into a system.

Safety Considerations

PLDs, and FPGAs in particular, may be built into safety-critical systems when the system is first designed or as part of a re-engineering of an older system. Such incorporation brings with it a need to be able to reason formally about safety and correctness of programs executing on the FPGA to satisfy DefStan 00-54. Here we have three distinct needs for a semantics of FPGA programs: to be able to

1. demonstrate that programs satisfy their requirements specification;
2. refine high-level designs into code while demonstrating semantic equivalence between them; and
3. reason about behaviour at the interface between software and programmable logic.

We develop these points in the rest of this section, with the objective of outlining a method to produce a correct FPGA program from a high-level specification.

Demonstrating FPGA Program Correctness

There are two choices for showing that a FPGA’s program satisfies its specification. The more common, *verification*, is ‘show that the implementation does what the requirements say’. One possibility is to use ‘model-checking’, automatic checking of finite state specifica-

tions against a given implementation. The key weaknesses of model checking are:

1. it is very CPU-intensive, due to the number of possible state transitions;
2. usually it will only be able to tell you *whether* your system is correct, not where it is weak; and
3. it does not prove properties in the *general* case, but only for the *specific* case of the actual states in the model being checked.

In this paper we adopt the second strategy which is often initially harder: a proof-theoretic approach to ‘develop the requirements into an implementation’.

We use Synchronous Receptive Process Theory (SRPT) to model formally the structure of FPGAs. SRPT, described in [5], was developed with the motivation of being able to reason about synchronous (clocked) events. It specifies a system as a set of events Σ , and a set of processes P_i each of which has a set of input and output events. Processes are defined in terms of output events in reaction to input events. SRPT has a denotational semantics expressed in terms of the *traces* of each process. Interested readers are referred to Barnes [5, §5.3-5.4] for the details of the semantics.

The structure of a FPGA can be considered as a collection of small SRPT processes reacting to input signals to produce output signals, when cells are viewed as processes and their routing is viewed as describing which signals pass to which process. In our work to date we have demonstrated a method of proof that a FPGA cell (modelled by an SRPT process) satisfies a specification in terms of event sequences in its traces.

Small-Scale Refinement

In [11] the authors demonstrate a refinement calculus suitable for developing abstract specifications into an implementation in the Pebble programmable logic programming language [14]. The calculus semantics are based on Back and Wright [4]. The calculus provides refinement rules for transforming specifications into SRPT processes, which can then be compiled automatically into a Pebble implementation.

Pebble is synchronous, low-level enough to compile to VHDL or netlist format without too high a probability of serious compiler error, and high-level enough to abstract away from device dependencies. SRPT processes at a suitably concrete level can be mapped directly into Pebble with minimal effort.

The authors demonstrate the use of this calculus by developing a provably correct carry look-ahead adder. It is, however, painstaking work and would be hard to

apply to developing all of a realistically sized real-world system. In the next section we examine the wider task of designing and implementing the rest of the system.

5 DESIGN REFINEMENT

Given a detailed Z specification, we wish to develop a complete system design into hardware-software implementation, maintaining demonstrable correctness. Since most of the implementation is likely to be in software, a useful stepping stone would be a software language that could act as the target of refinement from Z and then have parts compiled directly into SRPT processes.

Design Language

One candidate is SPARK Ada [10, 9], a subset of the Ada language. This was the principal implementation language for the SHOLIS project. SPARK Ada has a formal semantics defined in Z, tool support from the SPARK Examiner static analysis tool, and uses the strong type system of Ada. SPARK Ada is also strongly recommended for use in developing SIL 4 systems. The authors have produced a design for a SPARK interpreter which allows arbitrary sections of SPARK programs to be compiled directly into programmable logic.

INFORMED [2], a design methodology for SPARK Ada, provides a top-down development of a system from its specification. It aims to identify the boundaries between the SPARK Ada program and “real world” devices. INFORMED analysis is key to the design of our system.

SPARK Ada programs do not currently include the Ada tasking (parallel processing) statements. Audsley and Ward [3] describe how to model a parallel Ada program using the Ravenscar Profile safe tasking model [6] to schedule SPARK Ada programs on programmable logic devices, allowing worst-case execution time analysis. We can use the results of the INFORMED analysis to separate out the individual SPARK Ada programs which are executed in parallel.

Current development of SPARK Ada is incorporating the Ravenscar profile into the language; the ‘protected objects’ with which processes communicate are modelled as data streams. Other process interactions are mapped using a combination of suspension objects and atomic variables. Analysing the detail of the inter-process interaction is done with existing Ravenscar-specific tools.

Correct Refinement

The formal refinement of a state-based specification into a parallel process model is, in the general case, hard to manage correctly. One promising unified theory is *Circus* [7], an integration of the CSP process algebra and the Z specification language. This uses a Z schema to describe the state of each process and CSP-like action

to describe the control behaviour of each process. *Circus* has well-defined refinement rules for transforming specifications from abstract to concrete form.

Circus is appropriate to our development process at a higher level than SRPT. It gives us a way to refine down from an initial abstract specification to a collection of relatively independent processes, omitting specific timing descriptions as long as they are irrelevant. The developer would then translate these specifications to a SPARK Ravenscar system design, or (for certain identified processes) into an SRPT process specification.

Circus is as yet untested in an industrial-scale development; nevertheless, its framework and the rigour of its specification and refinement laws show promise for practical system specification.

Testing

A well-recognised method of increasing confidence in a system is the use of testing. Testing methods for conventional software are understood, and existing standards make various recommendations about rigorous approaches to testing e.g. statement and decision coverage, unit testing versus system testing and the use of dynamic test tools.

Unit testing of programmable logic devices presents new problems, such as detecting the effects of corruption to the programming bitstream. An approach such as that in [17] aims to identify stuck-at conditions for cells, open and shorted interconnections and coupling faults. The development testing plan must include such testing of the programmable logic components.

We have noted that high integrity systems require careful specification. Such specifications form a natural basis for testing, and it is therefore possible to write test cases from the specifications early in the development process; indeed, writing test cases before the implementation starts is not only possible but desirable since it gives the developer an immediate check as to whether his code satisfies the specifications.

A novel approach to testing is described by Aichernig [1]. Starting from a formal specification coupled with a set of test cases, mutation of the specification is used to check whether the test cases detect the mutation and hence whether the test cases adequately cover the specification. This is only possible with a well defined specification and refinement framework, but does make a contribution to the developer's level of confidence in the system.

In summary, then, the testing of a high integrity hardware-software system:

- is key to establishing confidence in the system;

- can exploit existing software techniques effectively as a side effect of possessing a rigorous specification;
- requires specific testing techniques for the programmable logic component; and
- may benefit from mutation testing of the system specifications.

6 PROCESS

The overall development process is illustrated in Figure 1. We start with an abstract specification in Z. Through the INFORMED method we identify the boundaries and components of our system. Any computation obviously suitable for programmable logic is split off into a separate specification and refined manually into SRPT processes then compiled into Pebble.

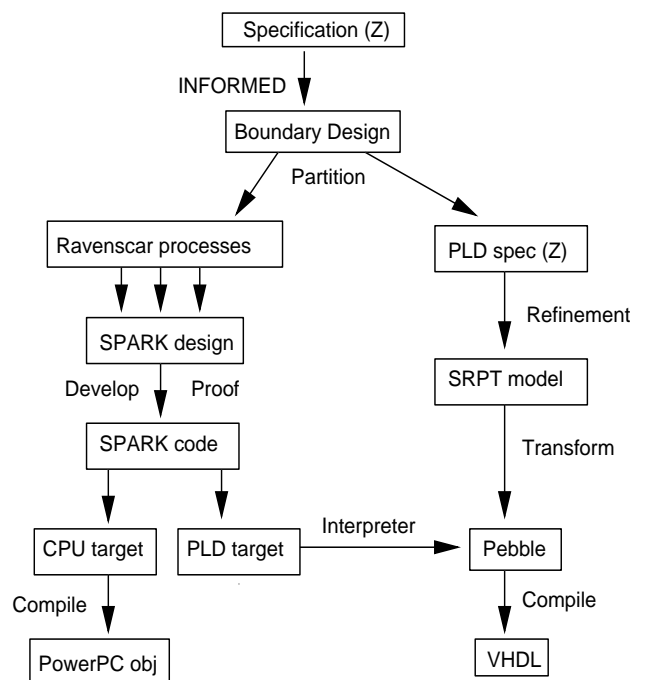


Figure 1: Development Process

The main Ada program is split into processes which interact using the Ravenscar tasking subset. Each process is developed into a SPARK Ada program, which is verified with the SPARK Examiner and proof checking tools.

At this point, any subsection of a program may be selected for mapping into programmable logic. The subsection is compiled into a PLD SPARK interpreter in Pebble, and replaced in the program with an interface to that interpreter. The remaining Ada code is compiled with a (verified) Ada compiler into the target hardware. The Pebble code is compiled into VHDL or netlist form suitable for programming the system PLDs.

7 CONCLUSION

We have seen how safety standards place requirements for analytical demonstration of the safety of systems incorporating programmable logic. We have identified key technologies and methods for such analysis, and proposed a process for developing programs for PLDs to a high standard of integrity. This process combines established safety-critical software development tools with techniques for developing correct programmable logic programs. We have examined the problem of testing such systems and identified existing and proposed test methods appropriate for the task.

Acknowledgements

Thanks are due to Peter Amey and Brian Dobbing for information on SPARK Ada and the Ravenscar Profile.

REFERENCES

- [1] Bernhard K. Aichernig. Contract-based mutation testing in the refinement calculus. In *Proceedings of REFINE 2002* [20].
- [2] Peter Amey. INFORMED design method for SPARK. Technical report, Praxis Critical Systems, 1998.
- [3] N. C. Audsley and M. Ward. Language issues of compiling Ada to hardware. Technical report, Real Time Systems Group, University of York, 2002.
- [4] Ralph-Johan Back and Joakim von Wright. Trace refinement of action systems. In *International Conference on Concurrency Theory*, pages 367–384, 1994.
- [5] Janet E. Barnes. A mathematical theory of synchronous communication. Technical report, Oxford University Computing Laboratory, 1993.
- [6] Alan Burns, Brian Dobbing, and George Roman-ski. The Ravenscar tasking profile for high integrity real-time programs. In *Reliable Software Technologies, Proceedings of the Ada Europe Conference*, volume 1411 of *Lecture Notes In Computer Science*, pages 263 – 275. Springer-Verlag, 1998.
- [7] Ana Cavalcanti, Augusto Sampaio, and Jim Woodcock. Refinement of actions in Circus. In *Proceedings of REFINE 2002* [20].
- [8] Pawel Chodowicz, Po Khuon, and Kris Gaj. Fast implementations of secret-key block ciphers using mixed inner- and outer-round pipelining. In *ACM/SIGDA Ninth International Symposium on Field Programmable Gate Arrays (FPGA '01)*, pages 94–102. ACM, ACM Press, February 2001.
- [9] Gavin Finnie and Ross Wintle. SPARK 95 – the SPADE Ada 95 kernel. Technical Report 1.0, Praxis Critical Systems Ltd., October 1999.
- [10] Jonathan Garnsworthy and Bernard Carré. SPARK - an annotated Ada subset for safety-critical systems. *Proceedings of Baltimore Tri-Ada Conference*, 1990.
- [11] Adrian Hilton and Jon G. Hall. Refining specifications to programmable logic. In *Proceedings of REFINE 2002* [20].
- [12] IEC Standard 61508, March 2000. Functional Safety of Electrical / Electronic / Programmable Electronic Safety-Related Systems.
- [13] Steve King, Jonathan Hammond, Rod Chapman, and Andy Pryor. The value of verification: Positive experience of industrial proof. In *FM'99 — Formal Methods; Proceedings*, volume 1709 of *Lecture Notes in Computer Science*. Springer-Verlag, September 1999.
- [14] Wayne Luk and Steve McKeever. Pebble — a language for parametrised and reconfigurable hardware. In R. 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.
- [15] Defence Standard 00-55 issue 2, 1997. Requirements for Safety-Related Software In Defence Equipment.
- [16] Interim Defence Standard 00-54 issue 1, March 1999. Requirements for Safety Related Electronic Hardware in Defence Equipment.
- [17] M. Renovell. A specific test methodology for symmetric SRAM-based FPGAs. In Reiner W. Hartenstein and Herbert 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 300–311. Springer-Verlag, August 2000.
- [18] Requirements and Technical Concepts for Aviation. Software considerations in airborne systems and equipment certification, December 1992.
- [19] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, 2nd edition, 1992.
- [20] University of Kent at Canterbury. *Proceedings of REFINE 2002*, July 2002.