Technical Report N° 2000/06

On Applying Software Development Best Practice to FPGAs in Safety-Critical Systems

Adrian Hilton,
Jon G. Hall

2000

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

http://computing.open.ac.uk

On Applying Software Development Best Practice to FPGAs in Safety-Critical Systems

Adrian Hilton, Jon Hall

The Open University

Abstract. New standards for developing safety-critical systems require the developer to demonstrate the safety and correctness of the programmable logic in such systems. In this paper we adapt software development best practice to developing high-integrity FPGA programs.

1 Introduction

Programmable logic devices are increasingly important components of complex and safety-critical systems. Standards such as the emerging UK Defence Standard 00-54 [6] and IEC 61508 [3] now require developers to reason about the safety and correctness of programmable logic devices in such systems. In addition, programming such devices is becoming more like programming conventional microprocessors in terms of program size, complexity, and the need to clarify a program's purpose and structure.

This paper looks at existing best practice in software development and shows how it might be adapted to programmable logic devices without incurring undue overhead in system development time. It does not consider the issue of testing programmable logic programs.

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. The *integrity* of such a system is measured in terms of the probability of total or partial failure. [3] defines four integrity levels, SIL 1, SIL 2, SIL 3 and SIL 4, with the highest level (SIL 4) specifying a frequency of less than 1 failure per 10^8 hours of operation.

Since many safety-critical systems affect public safety, governmental and associated oversight agencies have drawn up standards documents for the development of safety-critical systems. Newer standards are starting to require the rigorous demonstration of safety for programmable logic components that has been required for software for many years.

UK Defence Standard 00-54 [6] is a new interim standard for the use of safety-related electronic hardware (SREH) in UK defence equipment. It relates to systems developed under a safety systems document such as IEC 61508 [3].

Def Stan 00-54 is appropriate if an electronic component of the system is identified as having a safety integrity level of SIL 1 or greater. The techniques described in the document are to be used to analyse complex electronic designs for systematic failures. The standard contains the following recommendations which are of particular interest to us.

> (§12.2.1) A formally defined language which supports mathematically based reasoning and the proof of safety properties shall be used to specify a custom design;
> (§13.4.1) Safety requirements shall be incorporated explicitly into the Hardware Specification using a formal representation; and
> (§13.4.4) Correspondence between the Hardware Specification and the design implementation shall be demonstrated by analytical means, subject to assumptions about physical properties of the implementation.

where 'custom design' refers to the non-standard components of the electronic component under examination, and in particular to an FPGA's program data.

Def Stan 00-54 also notes that widely used standard HDLs without formal semantics, such as VHDL and Verilog, present compliance problems if used as a design capture language: Z [7] is suggested as an example of a suitable language.

Def Stan 00-54 is interim, and may well change at formal issue as did Def Stan 00-55. Nevertheless, the concerns which it expresses about existing practices and its suggestions for process improvements are worth careful scrutiny. A formal language which supports reasoning about programmable logic behaviour will assist developers to comply with this standard; without the ability to reason formally, it is not possible to meet the requirements of §12.2.1, §13.4.1 and §13.4.4.

3 Applying Software Best Practice to Programmable Logic

Programmable logic devices, 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; as noted above, Def Stan 00-54 requires this analytic reasoning. Here we have three distinct needs for a semantics of FPGA programs, to be able to:

- demonstrate that programs satisfy their specifications;
- refine high-level designs into code while demonstrating semantic equivalence between them; and
- 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.

3.1 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 specifications against a given implementation. The key weakness of model checking is that it is time-consuming, and usually will only be able to tell you *whether* your system is correct, not where it is weak.

In this paper we adopt the second strategy which is often initially harder: 'develop the requirements into an implementation'. This development is the process of *refinement*; step-by-step application of a set of laws which transform an abstract specification into a concrete implementation. This approach requires more 'up-front' investment of time and effort. However, the correctness of the implementation with respect to the specification is guaranteed, excepting the possibility of human error in the refinement steps.

Both of these approaches require the ability to reason analytically about FPGA programs. We address this in the following section.

3.2 Analytical Reasoning

Synchronous Receptive Process Theory (SRPT), described in [1], was developed from Josephs' Receptive Process Theory [4] 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_X each of which has a set of inputs $I \subseteq \Sigma$ and output events $O \subseteq \Sigma$ where $I \cap O = \emptyset$. 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. Each trace $t : \mathbb{N} \rightarrow \mathbb{P}(I \cup O)$ specifies a possible sequence of sets of events for the process at each tick of the global clock.

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.

3.3 Design Refinement

We wish to refine a FPGA program design from the Z specification language to an implementation, maintaining demonstrable correctness. Refining the specification directly to SRPT is possible but hard work. A useful stepping stone would be a software language that could act as the target of refinement from Z and then be compiled into SRPT processes. One candidate is SPARK Ada [2], a subset of the Ada language. SPARK Ada has a formal semantics defined in Z, tool support from the SPARK Examiner static analysis tool, and the strong type

system of Ada. SPARK Ada is also strongly recommended for use in developing SIL 4 systems.

Given an SRPT description of the program, we could attempt to compile it into VHDL, but maintaining correctness would be hard. VHDL lacks a semantics, with vendor implementations differing significantly. FPGA netlists will vary in semantics depending on the target device. One intermediate option is to use a language such as Pebble [5]. 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 could be mapped directly onto Pebble with minimal effort.

This development process is illustrated in Figure 1.

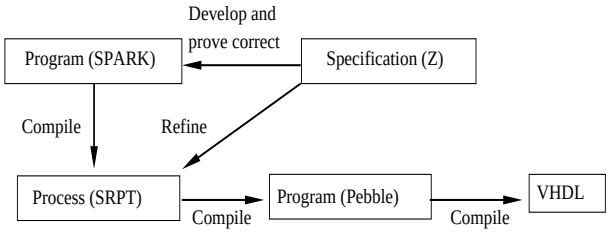

Fig. 1. Development Process

3.4 Conclusion

We have seen how a forthcoming safety standard places 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.

References

1. Janet E. Barnes. A mathematical theory of synchronous communication. Technical report, Oxford University Computing Laboratory, 1993.
2. Jonathan Garnsworthy and Bernard Carré. SPARK - an annotated Ada subset for safety-critical systems. *Proceedings of Baltimore Tri-Ada Conference*, 1990.
3. International Electrotechnical Commission. *Functional Safety of Electrical / Electronic / Programmable Electronic Safety-Related Systems, IEC Standard 61508*, March 2000.
4. Mark Josephs. Receptive process theory. *Acta Informatica*, 29:17–31, 1992.
5. 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)*, number 1482 in Lecture Notes In Computer Science, pages 9–18. Springer-Verlag, September 1998.
6. Requirements for safety related electronic hardware in defence equipment, March 1999. Interim Defence Standard 00-54 Issue 1.
7. J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, 2nd edition, 1992.