Technical Report No: 2001/01

FPGAs in Critical Hardware/Software Systems

Adrian Hilton

Gemma Townson

Jon G. Hall

September 2002

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

http://computing.open.ac.uk

FPGAs in Mission Critical Hardware / Software Systems

Adrian J. Hilton Gemma Townson[*] Jon G. Hall[†]

September 26, 2002

Abstract

FPGAs are being used in increasingly more complex roles in critical systems, interacting with conventional critical software. Established safety standards require rigorous justification of safety and correctness of the conventional software in such systems. Newer standards now make similar requirements for safety-related electronic hardware, such as FPGAs, in these systems.

In this paper we examine the current state-of-the-art in programming FPGAs, and their use in conventional (low-criticality) hardware/software systems. We discuss the impact that the safety standards requirements have on the co-development of hardware/software combinations. and suggest adaptations of existing best practice in software development that could discharge them. We pay particular attention to the development and analysis of high-level language programs for FPGAs designed to interact with conventional software.

1 Introduction

FPGAs 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 use of FPGAs in this role impacts safety-critical system development in that hardware/software combinations become subject to relevant standards, such as UK Defence Standard 00-54 [?] and IEC 61508 [?], which, in particular, will require safety and correctness arguments for the programmable logic components in addition to, and in combination with, those for the conventional software.

removed: requirements for the

In addition, technological improvements mean that program development for FPGAs has become more like software development in terms of program

[*]Praxis Critical Systems Ltd., 20 Manvers Street,Bath BA1 1PX

[†]Department of Computing, The Open University, Walton Hall, Milton Keynes

size, syntax, complexity, and the need to clarify a program's purpose and structure. Taken together, safety-critical use and technological improvement have important implications for hardware/software co-development.

This paper describes the impact that requirements from mandated standards have on hardware/software co-development, in the context of existing software development best practice for safety-critical systems. We then show how best practice might be adapted to FPGAs 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. 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 (Def Stan 00-54) [**?**] – an interim standard for the use of safety-related electronic hardware (SREH) in UK defence equipment[1] – relates to systems developed under a safety systems document such as IEC 61508 [**?**]. Other standards include [**?**].

Def Stan 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 design[2];
>
> (§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.

Def Stan 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 [**?**] is suggested as an example of a suitable language.

Although Def Stan 00-54 is interim, the concerns which it expresses about existing practices and its suggestions for process improvements are worth careful

[1] All the authors are British, and so are most familiar with British standards. Other such standards include [**?**]

[2] 'Custom design' refers to the non-standard components of the electronic component under examination; in particular, this includes an FPGA's program data

scrutiny. In particular, the requirement for 'formal representation' which supports reasoning about FPGA 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 Def Stan 00-54.

Examine public AAvA info

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

Best practice in the development of software for safety-critical systems is to be found in software which has been developed to a software standard (such as Def Stan 00-54) and which is in current use. To illustrate what is involved in safety-critical software development, then, we consider the SHOLIS helicopter-landing system, developed to UK Defence Standard 00-55 [?] as currently in use on Royal Navy Duke-class frigates. (From this, we will extrapolate to what would be needed for hardware/software co-development.)

In SHOLIS, extensive use was made of formal methods development technologies. These included: specification and proof in Z; tool assisted static analysis of program code; and semi-automated proof of program properties, such as absence of run-time exceptions. Formal methods proved useful: 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 demonstrably improved developer and user confidence in the system. (For detail of the development of SHOLIS system, the interested reader is referred to [?].)

I'm doubtful as to whether this section/paragraph adds much

As we wish the same benefits to be available to hardware/software co-development, we see that best practice should, in particular, facilitate similar analytical techniques for FPGAs and their relationship with system software. Unfortunately, the locus of application of formal methods as applicable to sotware is not known to include FPGAs. In particular, there are possible complications brought on by the use of FPGAs are: the highly parallel nature of their computations; the difficulties of interfacing to other system components; and issues of timing.

Whereas timing issues and interfacing can be resolved (to some extent) by the use of simulation and the synchronous design for the FPGA [, Need ref here], and asynchronous interfacing to other components, respectively. However, correct analysis of the parallel structure of FPGAs remains key to extending software development best practice to cover them.

4 Current FPGA Usage

["..." means "to be researched and filled in by Gemma"]

4.1 Devices

The mainstream devices are X by Xilinx and Y by Altera ...

4.2 Programming Languages

The low-level hardware description languages (HDLs) for FPGAs are VHDL and Verilog. These, similar to the relationship between machine code in software, allow precise control over timings and performance but are hard to use correctly. Pebble[**?**] is a simpler, synchronous HDL that allows translation to VHDL or netlist format.

than what, or is it simpler to use?

The increased capacity and clock speed of modern FPGAs makes it possible to trade-off performance for ease-of-device-programming. Handel-C[**?**], for instance, uses the syntax of ANSI-C to form a language which is partly imperative and partly declarative: C language constructs describe control and data flow; extensions add parallel constructs such as channels and indicate parallel execution of statements. Declarations inform the Handel-C compiler about other system entities such as RAM and ROM blocks, and implementation details such as which resources are to be shared.

It is worth noting here that, although Handel-C removes some undesirable features of C programs such as type ambiguity and some of the less clear control flow features, it retains problems such as the if/if/else syntax ambiguity and adds undesirable features such as inferred variable widths. It is not a language designed for safety-critical systems.

Why is it worth noting here? Is it because someone might ask whether you couldn't use it in the development of safety-critical systems? If so, it might be better to make that explicit

4.3 Applications

The following systems use FPGAs (and Handel-C) in substantial roles ...

5 FPGAS in Safety-Critical Systems

In this section we first consider the form that safety and correctness arguments for FPGAs should take, and then look into their combination with those for software.

then(?)

Ref for "Safer C"

5.1 FPGA Safety and Correctness Arguments

added

changed

FPGAs may be built into safety-critical systems *ab initio* when the system is first designed or as part of a re-engineering of a software-based system in which they replace current software functionality. In the former case, design languages are likely to be higher-level, and require transformation to lower-level FPGA specific designs. In the latter case, it is possible that the existing software is that starting point for the FPGA design. In either case, their incorporation brings with it a need to be able to reason formally about the safety and correctness of programs executing on the FPGA (to satisfy Def Stan 00-54). Hence, we identify three requirements for a FPGA design methodology:

1. *verifiability and validity*: the ability to demonstrate that programs satisfy their requirements specification;
2. *refinability*: the ability to refine high-level designs into code while demonstrating equivalence between them; and
3. *interoperability*: the ability to reason about behaviour at the interface between software and FPGA.

perhaps not the best term, interfacability?

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.

This section is a bit confused from here on. The logical development should leave us looking at (what I have, perhaps, erroneously called verifiability and validity, refinability, and interoperability) but, anyway, three separate things. Below the first confusingly merges into the second...needs restructuring a little. We also need to be able to audit back to them when we discharge them later on. Next comments on section 6

5.2 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 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 [**?**], 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 [**?**, §5.3-5.4] 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.

5.3 Small-Scale Refinement

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

Pebble's structure is simple 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 demonstrated 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.

6 Design Refinement

Given a detailed Z specification, we wish to develop a system design into a hardware-software implementation, demonstrably maintaining correctness. We note that, typically, FPGAs will form only a relatively small adjunct to software. In this case, a useful stepping stone for FPGA design would be a software language that could act as the target of refinement from Z and as well as being able to be compiled directly into SRPT processes.

Our candidate language for this role is SPARK Ada [?, ?]. SPAKR Ada is a subset of the Ada language [?, need ref here]. The use of SPARK Ada is well-known in safety critical system development; indeed, it was the main implementation language for the SHOLIS project. Proof of concept is established, in [?, ref], the authors have produced a design for a SPARK interpeter which allows arbitrary sections of SPARK programs to be compiled directly into programmable logic. In combination with its traditional software use, this shows that SPARK Ada is a good candidate.

6.0.1 Benefits of SPARK Ada

SPARK Ada has many desirable characteristics as a language for form development. It has a formal semantics defined in Z[?, ref here]; tool support from the SPARK Examiner static analysis tool[?, ref here]; and uses the strong type system of Ada[?, ref here]. SPARK Ada is also strongly recommended for use in developing SIL 4 systems[?, ref here].

Methodologically, SPARK Ada is supported by the INFORMED method [?]. INFORMED provides for the top-down development of a system from its specification . Specifically, INFORMED helps identify the boundaries between the SPARK Ada program and any “real world” devices. INFORMED analysis is key to the design of our system.

In Z?

Does this discharge one of our three requirements?

Why is it key? Is it just an important component?

next two paras: Are they relevant? If so why?

SPARK Ada programs do not currently include the Ada tasking (parallel processing) statements. Audsley and Ward [?] describe how to model a parallel Ada program using the Ravenscar Profile safe tasking model [?] to schedule SPARK Ada programs on programmable logic devices, allowing worse-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.

SPARK Ada can, with very few extensions to the language, replace Handel-C in such systems. Many of the features which Handel-C adds to C (e.g. specific bit-width typing and casting restrictions) are present in Ada; others (e.g. forbidding side-effects in expressions) are implemented in SPARK.

But you said it was not used!

SPARK Ada includes features that assist in compiling imperative structures to hardware. The information and data flow known to a SPARK Examiner makes it feasible to detect potential race conditions in code intended for parallel execution. SPARK also allows the proof that no variable in the program goes outside of its declared numeric range; this is a substantial aid to both compilation and correctness. Ada defines compiler directives for representing values of enumerations and the specific addresses of variables.

We should say earlier that these form part of the annotations

automatically detect?

Parallel constructs present more of a challenge. Fine-grained parallelism would need an explicit program marker. One option would be use of a `pragma parallel` on a subprogram, marking the subprogram's statements to be executed in parallel. Channels between threads could be modelled using Ravenscar protected objects.

The greatest benefit from this work is that an entire system program could be represented with a SPARK program; parts to be implemented on an FPGA would be subject to static analysis both in isolation and as part of the larger system. The FPGA program would then have been implemented in a high-level language designed (and proven effective) for use in safety-critical systems.

Our current investigations involve producing a map from SPARK Ada onto Handel-C; given this, we can then work on going directly from SPARK to VHDL.

6.1 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* [?], 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.

6.2 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 [**?**] 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.

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; and
- requires specific testing techniques for the programmable logic component.

7 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.

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.

height 90mm width 82mm process.pdf

Figure 1: Development Process

Any subsection identified as suitable for implementation in an FPGA is modified to have the appropriate parallel constructs and interface declarations (e.g. to on-chip RAM and ROM). This modified code is statically analysed with the rest of the system. At compile time, the FPGA-specific code is compiled into VHDL; within the main Ada program, it is replaced by a veneer sending data to and from the FPGA via a method appropriate to the particular system (e.g. memory-mapped I/O or a bus interface package).

8 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 FPGAs to a high standard of integrity. This process combines established safety-critical software development tools with techniques for developing correct programmable logic programs.

At the level of implementation we have identified the deficiencies of existing FPGA programming languages and proposed the extension of the SPARK Ada high-integrity programming language to fill this gap. We have examined the problem of testing hardware/software systems and identified existing and proposed test methods appropriate for the task.

8.1 Acknowledgements

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