Safety Arguments for use of an Ada to FPGA Compiler

Iain Bate; Department of Computer Science; University of York; UK

Simon Bates; Department of Computer Science; University of York; UK

John McDermid; Department of Computer Science; University of York; UK

Keywords: FPGAs, safety, software

Abstract

In this paper the use of Field Programmable Gate Arrays (FPGAs) has been investigated as an alternative target device for safety-critical software to the use of traditional microprocessor (e.g. Von Neumann-based architectures). FPGAs are a desirable alternative in comparison to microprocessors for reasons including: ease of analysis in comparison to modern microprocessors such as the PowerPC, and the obsolescence problems with using short-lifetime microprocessors in long lifetime systems. The work in this paper uses the Goal Structuring Notation to decompose the safety claims for FPGA-based systems. It then considers how the necessary evidence can be gathered. The results of this work are the following three contributions. Firstly, we identify the necessary evidence required to justify that the resulting FPGA-based system is of sufficient integrity. Secondly, we identify techniques that can be used to gather the required evidence. Thirdly, we show the relation to the evidence currently gathered for microprocessor-based systems and the guidance given by the Safety and Security Annex for the Ada programming language.

Introduction

The use of Field Programmable Gate Arrays (FPGAs) is being investigated as an alternative target device for safety-critical software to the use of Von Neumann style processors. The reasons for choosing an FPGA include; the behaviour of modern processors, such as the PowerPC is difficult to analyse [1], the production lifecycle of processors is becoming shorter leading to systems with longer lifecycles (e.g. Avionics) facing difficult and expensive production decisions (i.e. buy and store components, make the design technology transparent increasing cost and inefficiency, or regularly update the product), and FPGAs lend themselves to hardware software co-design. The work is being performed in the context of the following: critical systems, FPGAs that are re-programmable technology rather than based on write-once technology, and software written in safe subsets of Ada. However much of the argument and evidence requirements are generally applicable to any compiler and language. The work does not deal with architectural issues such as fault tolerance.

The starting point for this paper is the work performed on compiling suitable subsets of Ada such that they can be hosted on a FPGA [2]. (Suitable subsets are defined as those for which the resulting code can be predicted with reasonable cost effectiveness, e.g. SPARK Ada [3].) The reasons writing software and compiling it down to a digital design is preferred to producing a digital design in the first place are that software produced through an appropriate software engineering process is considered to be easier to change, more effective at representing complex designs, and provide better support for reusing existing designs and/or development approaches. The reasons Ada is preferred to other languages (e.g. Handel-C [4]) that already have support for compilation down to FPGA are that its use in critical systems is more extensive particularly for higher integrity applications and it is considered a better language for software engineering.

The aim of this paper is to identify the necessary evidence needed to justify that the resulting system is of sufficient integrity (i.e. the risk of hazards is acceptably low), and then identifying the techniques by which this evidence can be provided. Whilst achieving these objectives, an important goal is, where possible, to use the same approaches to development, verification and certification as would be used for software being hosted on a traditional Von Neumann architecture. This allows existing approaches and techniques to be reused and it also eases any migration that might take place between the two approaches.

Background

The following subsections provide background to the context for the applications to be executed on the FPGA, a survey of relevant certification standards, and summarises the compilation approach that has been developed.

Application Context:

As previously stated, this work is being performed in the context of critical systems up to and including the highest integrity levels prescribed by current safety standards with the functionality being produced in software, notably Ada, and then compiled for use on a FPGA. The nature of the application, whether continuous or discrete in nature, is not being assumed. However, it is being assumed that the software's source code is being produced using rigorous methods that keep systematic errors to a minimum. That is, the methods should ensure the source code meets its specification. However since the primary goal of the work is to ensure that the risk of hazards being introduced is acceptable, then the number of systematic errors in the software written in Ada is not relevant to the arguments presented.

Certification Standards:

There are number of safety standards and guidance documents that govern the development of hardware for use in safety-critical applications. Therefore, rather than presenting arguments targeted to a specific safety standard, we present the arguments that can be regarded as 'core' to all standards, e.g. the identification, avoidance and control of systematic errors in the transformation between the source code of the software and the eventual executable realisation.

A benefit of abstracting away from specific standards is that there is significant difficulty interpreting the current standards because the standards are largely separated out into hardware and software but this work effectively spans the two areas. Since the majority of the work is concerned with software and related tools for mapping the software to hardware and not the actual hardware itself, greater relevance is given to what the software standards raise on these issues. However issues from the hardware standards are still relevant because the internals of the compiler are based on mapping individual Ada language statements to circuit schematics, the latter of which should be verified using hardware techniques. The situation is eased because the hardware and corresponding software standards are very similar in their requirements meaning that we can mainly highlight common areas and identify any notable differences.

Compilation Approach:

The objective for the work is the compilation of Ada such that it can be predictably and efficiently represented on a FPGA. The current approach involves the following steps:

- Software is produced in a sequential form of Ada (e.g. SPARK) – note concurrent subsets of Ada (e.g. Ravenscar) may be supported at a later date.
- The compiler produced at York converts the Ada into EDIF (Electronic Design Interchange Format) format. The compiler is template-driven and features no optimisation. The templates are produced and verified through simulation using standard schematic design facilities provided by most electronic design suites. At this stage the information is still technology transparent and human readable.
- Standard tools are then used to place and route the EDIF design; this results in a file in NGD (Native Generic Design) format that is still human readable and is mostly (approx 90%) technology transparent. The non-technology transparent parts are related to device-specific components, which are infrequently used across FPGA families.
- The final stage is conversion to a bit stream that can be loaded onto a specific device.

Clearly as a result of this work developing certification strategies for the use of FPGA, some changes in the way the compiler operates might be needed.

Safety Argument

The purpose of this section is to present the safety argument for the execution of compiled software on a given platform using the Goal Structuring Notation (GSN). A short introduction to GSN is given in the following subsection. The safety argument that has been produced demonstrates how it can be shown the behaviour of the software as defined in the source code is as expected on the platform. This concentrates on whether the compiler and other tools (e.g. place and routing – the process by which a logical design is mapped to physical hardware) perform an appropriate transformation rather than whether the hardware

itself contains systematic errors. The argument assumes the software is written in a language (or subset) whose behaviour is amenable to verification, for example SPARK Ada [3].

Introduction to Goal Structuring Notation:

Within this and the following sections, we use the GSN [5] to outline the safety arguments that need to be made to support the use of software compiled and executed on a platform within a safety-critical system. Any safety case can be considered as consisting of requirements, argument, evidence and definition of bounding context. GSN - a graphical notation - explicitly represents these elements and (perhaps more significantly) the relationships that exist between these elements (i.e. how individual requirements are supported by specific arguments, how argument claims are supported by evidence and the assumed context that is defined for the argument).
The principal symbols in the notation are shown in Figure 1 (with example instances of each concept).

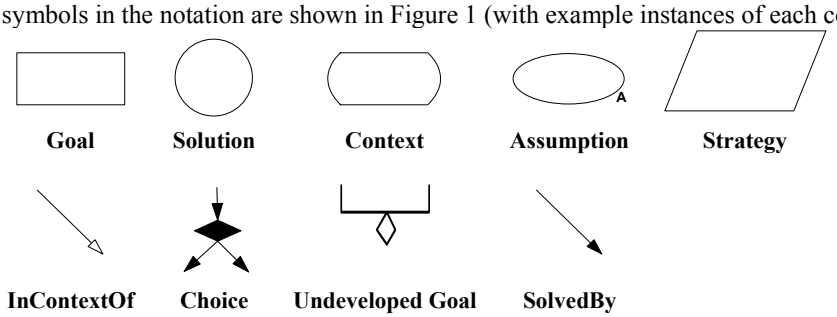

Figure 1 - Principal Elements of the Goal Structuring Notation

The principal purpose of a goal structure is to show how **goals** (claims about the system) are successively broken down into sub-goals until a point is reached where claims can be supported by direct reference to available evidence (**solutions**). As part of this decomposition, using GSN it is also possible to make clear the argument strategies adopted (e.g. adopting a quantitative or qualitative approach), the rationale for the approach (**assumptions**, **justifications**) and the **context** in which goals are stated (e.g. the system scope or the assumed operational role). For further details on GSN see [5].

Top Level Argument:

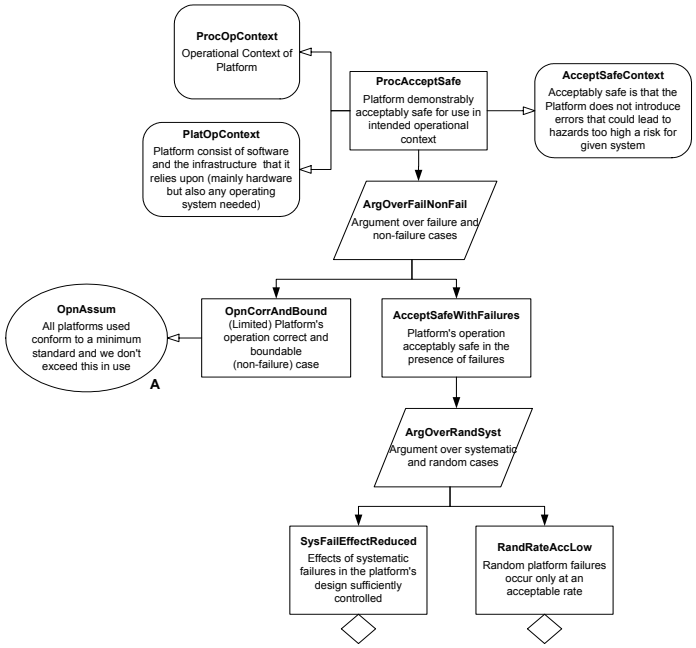

Figure 2 - Platform is Acceptably Safe

Figure 2 gives the top-level safety argument whose main goal is to show that the platform is safe to use in the intended operational context. (Part of the context for this argument is that the platform consists of an

executable form of the software and an infrastructure which is predominantly hardware but could include an operating system.) To achieve this aim it is necessary to show that the risk of errors in the system associated with the use of the platform is acceptable for the given system (goal **ProcAcceptSafe**). The acceptable risk is related to the system's integrity requirement. The argument is split into two parts; one part deals with the cases where the system does not fail while operating (goal **OpnCorrAndBound**), and the other with the cases where the system does fail (goal **AcceptSafeWithFailures**). The goal **OpnCorrAndBound** has an assumption associated that the platform conforms to a minimum standard, this refers to issues such as a maximum gate delay being specified and met.

The failure cases are further split into systematic and random failures. As previously stated, it is outside of the scope of this work to deal with how the system deals with failures during operation. Therefore, the systematic and random failure goals are left undeveloped (as indicated by the diamond under the goal). Instead, this work is to concentrate on expanding the argument for the non-failure case (i.e. trying to ensure errors do not occur in the design) – refer to the following subsection for further details. It should be noted that the previous work [1] concentrated on the actual execution on a specific platform rather than how the executable is realised.

Argument for the Prevention of Errors:

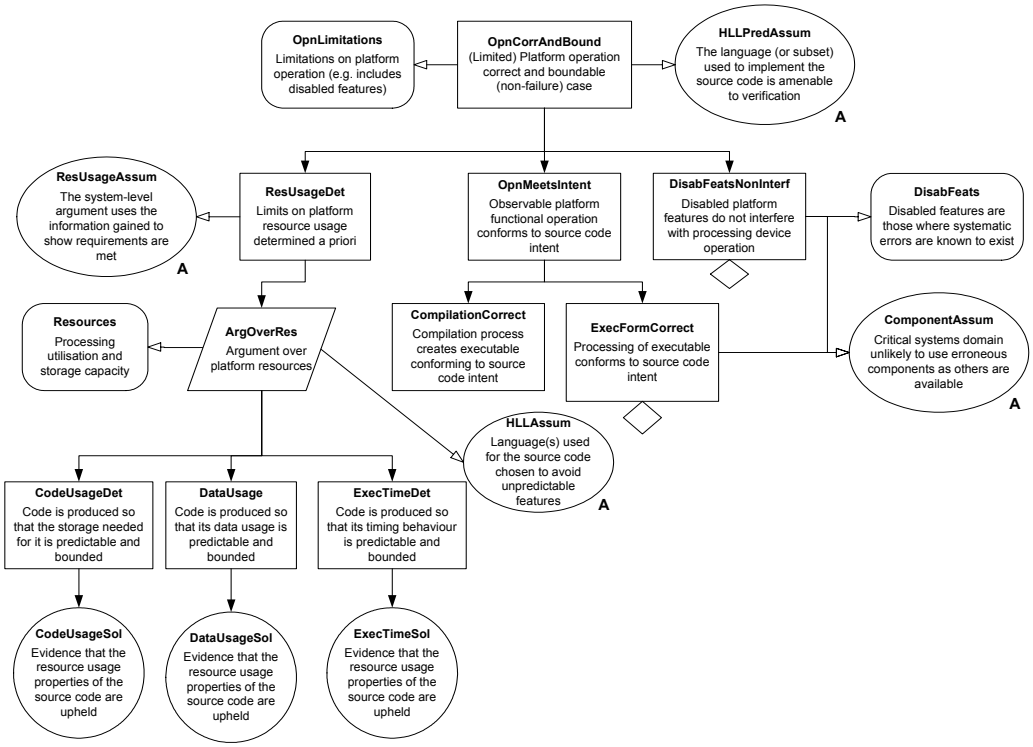

Figure 3 – Prevention of Errors

The argument to support goal **OpnCorrAndBound** is given in Figure 3. A key assumption (assumption **HLLPredAssum**) is that the software's source code has been implemented in a language (or subset) that is amenable to verification, i.e. its operation is predictable and boundable.

The argument initially splits into three parts:

- Goal **ResUsageDet** - that the maximum resource usage can be determined by virtue of how the software's source code is compiled and then executed – it is assumed that the system-level argument then uses the information gained about resource usage to show the system-level requirements are met.
- Goal **OpnMeetsIntent** - that the observable behaviour is as intended. This is split further into; goal **CompilationCorrect** which argues the compilation process results in an executable that conforms with the original intent of the source code, i.e. errors are not introduced, and goal **ExecFormCorrect** argues the processing of the executable conforms with source code intent. The correctness of the compiler is discussed further in the next subsection. Goal **ExecFormCorrect** refers to situations where errors in

the actual device lead to errors in operation and consequently source code intent not being met. This part of the argument is not developed further because it is unlikely components with known systematic errors or fabrication would be used in the critical systems domain when ones without these known problems exist – assumption **ComponentAssum**. Where errors are known to exist, then the goal **DisabFeatsNonInterf** would apply.

- Goal **DisabFeatsNonInterf** - that disabled features do not interfere with the platform's operation. This refers to where features of the platform are not used either because they are spare capacity or because there are errors in the devices as discussed in the previous point with respect to goal **ExecFormCorrect**. Again, this part of the argument is not developed further for the same reasons as the previous point.

Compilation Correct:

Figure 4 presents the argument for the goal **CompilationCorrect** that originated in Figure 3. Satisfaction of the goal is split into two parts.

The first part shows by a combination of manual inspection and evidence from testing on the target that the operational behaviour is as expected. The current practice for doing this involves each time the compiler is used re-performing a certain amount of testing on the target that was previously performed on a host. However as confidence grows in the compilers correctness, the amount of testing is reduced.

The second parts assesses whether the compilation process is correct. There are two parts to this; loading the resulting executable correctly onto the platform (evidence generated that an appropriate loading mechanism is applied) and consequently checking it (evidence that an appropriate verification mechanism is used, e.g. CRC checks), and showing that the executable is produced according to the Language Reference Manual (LRM). The latter of these is further explored in Figure 5.

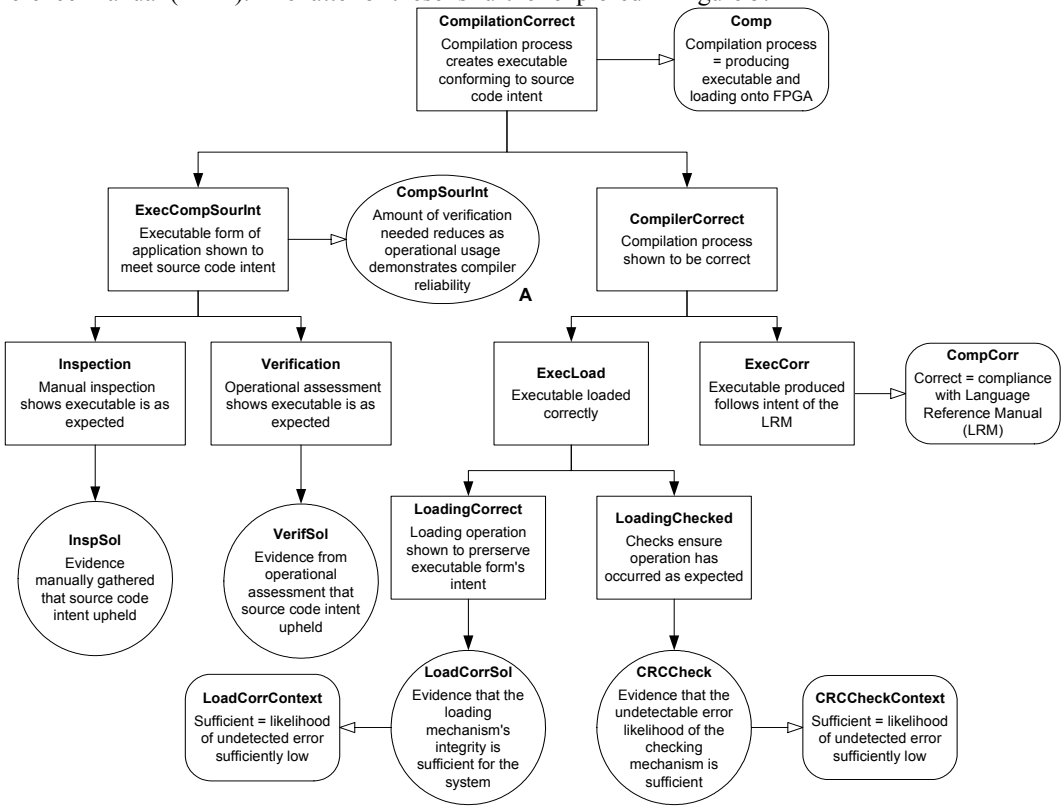

Figure 4 – Compilation Correct Argument

Figure 5 presents the safety argument for the executable being produced following the intent of the LRM. (The ability to gather evidence to support this argument depends on a sufficiently rigorous LRM being available.) This argument is split into two parts. The first part deals with whether the language syntax and synthesisability are checked before compilation begins. The second part deals with the language semantics

for each instruction, and combinations thereof, being checked and preserved during compilation. (The term *instruction* is used in this document to refer to the representation of a single Ada language statement through the compilation levels until it is ultimately realised in the executable form.) The syntax and semantics checks should be performed where possible through the various levels of compilation.
The need for evidence that semantics are preserved can be satisfied by a number of approaches leading to the necessary evidence being generated - the approaches can be used in combination. The approaches include; constructing the compiler formally so that the evidence is generated as it is designed, using formal verification via model checkers to generate evidence of correctness, and using non-formal techniques to generate the evidence. The formal techniques include; checking for semantic correctness and traceability through the compilation levels, and simulation and requirements-based testing performed sufficiently that coverage metrics indicate enough confidence has been gained in the approach.

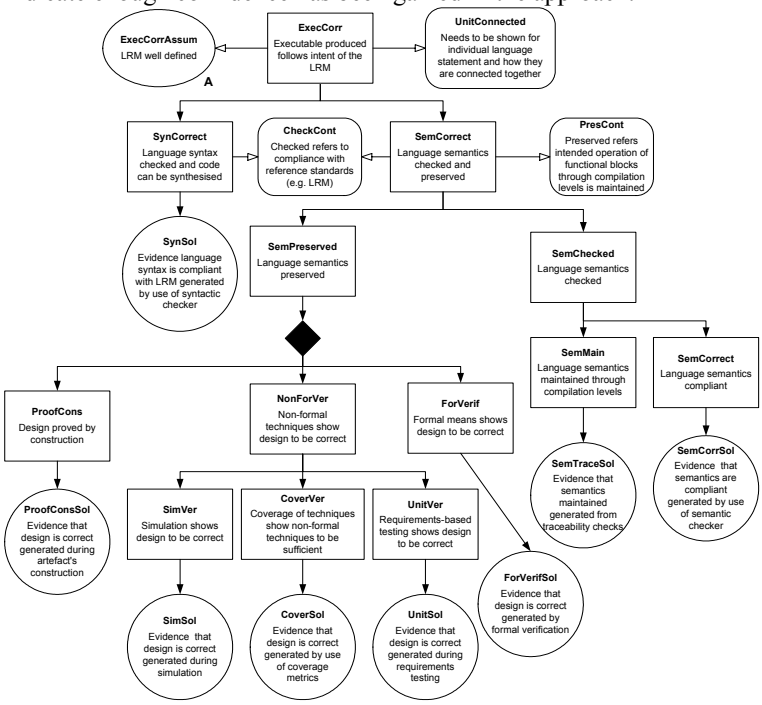

Figure 5 – Argument for Executable Conforms with LRM

Relationship with Existing Work:

In Whalen and Heimdahl [6] five requirements for high integrity code generation were established that are considered relevant when assessing our approach. Their consideration is given below:

1. Source and target languages must have formally well-defined syntax and semantics.
   This argument has been defined in Figure 5 *that there is a need to be able to check syntax and semantics against a rigorously defined enough language reference manual.*
2. The translation between a specification expressed in a source language and a program expressed in a target language must be formal and proven to uphold the meaning of the specification.
   This is satisfied by the arguments stated in Figure 4 *that evidence is needed that source code intent is upheld in subsequently generated code and in* Figure 5 *that both representations have a traceable path between them and conform to the LRM.*
3. Rigorous arguments must be provided to validate the translator (a compiler is an example of a translator) and/or the generated code.
   A rigorous argument and the evidence needed to satisfy it have been presented.
4. The implementation of the translator must be rigorously tested and treated as high assurance software.
   A rigorous argument and the evidence needed to satisfy it have been presented. Included in this is the argument stated in Figure 4 *that evidence is needed that source code intent is upheld in subsequently generated code.*
5. Generated code must be well-structured, documented and traceable to the specification.

Being able to show that source code intent is upheld in subsequently generated code as defined in Figure 4 *and that both representations have a traceable path between them and conform to the LRM given in* Figure 5 *satisfies this need.*

To show that the argument presented is complete, the evidence required of the software is compared in Table 1 to the evidence requirements discussed in the Ada HRG standard [7]. The reason the Ada HRG standard is used is that it collectively represents the requirements of certification standards for software. In addition, Table 1 contains evidence requirements for review and traceability which are demanded by most standards.

Category	**Verification Technique**	**Description**	**Coverage in Argument**
Quality Control	Traceability	Showing how and where requirement(s) are satisfied in the code.	Figure 5, solution **SemTraceSol**.
	Reviews	Checking the output of individual process stages, e.g. verification.	Figure 4, solution **InspSol**.
Analysis	Control flow analysis	Ensure code is executed in correct order and that it is structurally well formed.	Mainly source code issue but also Figure 4, solution **InspSol**.
	Data flow analysis	Ensure no variable can be accessed before it has been set.	Same as for control flow analysis.
	Information flow analysis	Identifies relationships between inputs and outputs in-order to ensure the correct dependencies are implemented and no other dependencies exist.	Same as for control flow analysis.
	Symbolic execution	Verify properties of the software, without resorting to formal proofs and proof tools.	Figure 5, solutions **SimSol**, **CoverSol**, and **UnitSol**.
	Formal code verification	Proving that the code is correct with respect to a formal specification.	Figure 5, solution **ForVerifSol**.
	Range checking	Verify that data values are within a specified range.	Same as for control flow analysis.
	Main memory usage analysis	Determine the main memory usage used and show that the processor has sufficient memory.	Figure 3, solutions **InterMemSol**, **IntraMemSol**, **CombMemSol,** and **TestConfSol2**.
	Stack usage analysis	Determine the maximum stack usage used and show that the processor has sufficient stack memory.	Figure 3, solutions **CallDepthSol**, **DynMemSol**, and **CombDynSol**.
	Timing analysis	Determine the maximum execution time for tasks and that execution times are bounded.	Figure 3, solutions **CodeUsageSol**, **DataUsageSol**, **ExecTimeSol,** and **TestConfSol1**.
	Other memory usage	Determine worst-case other memory usage (e.g. communications buffers) and whether the system has sufficient capacity.	Figure 3, solutions **CodeUsageSol**, **DataUsageSol**, **ExecTimeSol,** and **TestConfSol1.**
	Object code analysis	Determine the object code upholds the source code intent.	All solutions following on from goal **CompilationCorrect** in Figure 4.
Require-ments Based Testing	Equivalence class	Given that exhaustive testing is impractical, equivalence class testing subdivides the input and output data spaces into classes such that a test with any of the values within one class should give equivalent results.	Figure 5, solutions **CoverSol** and **UnitSim**.
	Boundary value	Builds on the equivalence class of testing to test at the boundaries of value classes, rather than just at some point in the class.	Same as for equivalence class.
Structure Based Testing	Statement coverage	Apply test cases so that each program statement has been invoked at least once.	Same as for equivalence class.
	Branch coverage	Apply test cases so that each decision in the program has taken each of its possible outcomes.	Same as for equivalence class.
	Modified condition / decision coverage	Show that each basic condition independently affect each basic decision.	Same as for equivalence class.

Table 1 – Coverage of the Ada HRG Verification Requirements in the Arguments

Roadmap for Future Work on FPGAs

In this section the intention is to go through the forms of evidence needed to support the arguments presented, discuss how it might be provided with the current tools and techniques and establish any future verification techniques that need to be developed.

Table 2 presents the result of the assessment. In the table; the first two columns define where the need for evidence is identified in the arguments that have been produced, the third column a description of the evidence needed, the fourth column a discussion of how the evidence requirement could be satisfied, and the final column whether the techniques are currently available.

Table 2 shows that most forms of analysis for gathering the evidence needed are currently available. However a number of specific caveats have been identified (in *italics*). There is also the general issue that whilst tools and techniques may exist they may not have been justified and qualified for use in the context of critical system applications.

Figure Ref	Solution Ref	Verification Needs	Proposed Technique	Available Now?
Figure 3	*CodeUsageSol*	Evidence that assumed predictability in the high-level language is maintained through to the platform.	Static analysis and code review to show that the maximum code size can be determined apriori. This needs to be performed in the context of a given device.	✓ *(Properties emerge from circuit design and place and route. Results normally verified using separate tool.)*
	DataUsageSol	Evidence that assumed predictability in the high-level language is maintained through to the platform.	Static analysis and code review to show that the maximum data size can be determined apriori. This needs to be performed in the context of a given device.	✓ *(Properties emerge from circuit design and place and route. Results normally verified using separate tool.)*
	ExecTimeSol	Evidence that assumed predictability in the high-level language is maintained through to the platform.	Static analysis and code review to show that the maximum and minimum timings of the executable can be determined apriori. This needs to be performed in the context of a given device.	✓ *(Properties emerge from circuit design and place and route. Results normally verified using separate tool.)*
Figure 4	*InspSol*	Evidence from manual inspection that executable meets source code intent.	Code review and traceability analysis on both individual source-level statements and program examples. This is similar to how object code analysis is currently performed on compilers.	✓
	VerifSol	Evidence from functional verification activities that executable meets source code intent.	Unit testing and scenario-based testing repeated on target and results compared. The unit testing can be performed on both individual source-level statements and program examples. This is similar to how object code analysis is currently performed on compilers.	✓
	LoadCorrSol	Evidence that the loading mechanism of executable onto the FPGA is of sufficient integrity.	Loading mechanism developed according to a suitable process.	✓ *(Loading mechanisms currently exist but their integrity needs checking.)*
	CRCCheck	Evidence from post-loading checks that the loading has indeed been performed as required. The checks should be performed after loading and to a limited extent during normal operation.	Appropriate checking mechanism based on CRC techniques developed according to a suitable process.	✓ *(Post-loading checks currently exist but their integrity needs checking.)*
Figure 5	*SynSol*	Evidence that language statements are syntactically correct at the various levels of compilation.	The code can be passed through appropriate checkers and results obtained. The checkers may be an integral part of a compile or a separate tool such as *Lint*. More than one compiler or an independent checker may be deployed to reduce the qualification needs of individual checkers	✓ *(There are a variety of compilers and tools such as lint that can perform these checks.)*
	ProofConsSol	Evidence gathered during the formal design of the FPGA compiler that language semantics are preserved at the various levels of compilation.	It is generally considered and accepted that it is infeasible and unnecessarily expensive to formally develop applications or their supporting tools, e.g. autocode generators. However, formal methods for logic circuits are more practicable than for software so some assessment may be possible. Otherwise, other forms of similar evidence such as from *SimSol* should compensate for the lack of formal assessment.	✓ *(Projects normally justify that formal design is unnecessary.)*

Figure Ref	Solution Ref	Verification Needs	Proposed Technique	Available Now?
Figure 5	*SimSol*	Evidence gathered during simulation that language semantics are preserved.	Perform simulation of individual language statements and their interfaces to show that their operation matches that expected. Simulation should be used more for FPGAs than for conventional Von Neumann architecture due to the improved tools available. The simulation can be performed on both individual source-level statements and program examples.	✓ *(Simulation can currently be performed at most, if not all, compilation levels.)*
	CoverSol	Evidence from the various forms of non-formal assessment (see *SimSol*, *UnitSol*) that the language semantics are preserved.	Assessment that sufficient confidence has been gained. The assessment would be similar to that normally performed as part of determining when sufficient functional verification has been performed. The coverage analysis can be performed on both individual source-level statements and program examples.	✓
	UnitSol	Evidence gathered during unit testing that language semantics are preserved.	Perform unit testing of individual language statements and their interfaces to show that their operation matches that expected. The unit testing can be performed on both individual source-level statements and program examples.	✓ *(Unit testing can currently be performed at most, if not all, compilation levels.)*
	SemTraceSol	Evidence gathered through the compilation steps that semantics preserved.	Techniques exist for tracing the meeting of requirements through a number of steps.	✓
	SemCorrSol	Evidence gathered through the compilation steps that semantics are complaint with reference manual.	Techniques exist for checking semantics' compliance. An often used tool is *lint* which is available for many languages.	✓ *(There are a variety of compilers and tools such as lint that can perform these checks.)*
	ForVerifSol	Evidence gathered through post-design assessment that semantics are complaint with reference manual.	Powerful model checking techniques exist for this purpose.	✓ *(Projects normally justify that formal approaches are unnecessary.)*

Table 2 – Verification Techniques to Support Safety Argument

In general, gathering evidence using the Ada targeted at an FPGA is easier than Ada targeted at a conventional Von Neumann architecture for the following reasons:

- The analysis techniques and tools available are more powerful and operationally mature. This means a compiler targeting an FPGA could be demonstrated as having a higher integrity than for a conventional microprocessor. Having a higher integrity compiler could reduce the amount of object-code analysis and target-based testing needed.
- The analysis techniques and tools available are able to operate at different level of abstractions (e.g. at the source code level, intermediate levels and executable form), and there are significantly more powerful and accurate simulators available. This would help in meeting some of the IMA goals – i.e. technology transparency and delaying the choice of hardware platform.
- The target is much simpler in the case of an FPGA and could be statically analysed whereas many of the modern microprocessors available can not easily be analysed [1, 2].
- The timing behaviour of a circuit on a FPGA is normally optimised and analysed for the worst-case which is more suitable for embedded systems than for a modern microprocessor. In contrast, it is normally the average case that optimisation is performed for and worst-case analysis is not generally available.

Summary and Future Work

This work has complemented the work on developing an Ada to FPGA compiler by producing a safety argument that defines the necessary evidence needed to justify the application software's source code is transformed and executed in a manner that introduces acceptable risk, and then identifying the techniques by which this evidence can be provided. When defining the argument the same approaches to development, verification and certification have been proposed, where possible, as for software being hosted on a traditional Von Neumann architecture. This allows existing approaches and techniques to be reused and it also eases any migration that might take place between the two approaches.

One of the main conclusions of this work is that the analysis techniques and tools available for FPGAs are more powerful and operationally mature. This means a compiler targeting an FPGA could be demonstrated as having a higher integrity than for a conventional microprocessor. Having a higher integrity compiler could reduce the amount of object-code analysis and target-based testing needed. Additionally, the analysis techniques and tools available are able to operate at different level of abstractions (e.g. at the source code level, intermediate levels and executable form), and there are significantly more powerful and accurate simulators available. This would help in meeting some of the IMA goals – i.e. technology transparency and delaying the choice of hardware platform.

References

[1] I Bate, P Conmy, T Kelly, J McDermid, *Use of Modern Processors in Safety-critical Applications*, The Computer Journal, 44 (6), 531-543, 2001.

[2] M. Ward, N. Audsley, *Hardware Compilation of Sequential Ada*, Proceedings of CASES'01, pp. 99-107, 2001.

[3] J. Barnes, *High Integrity Ada: The SPARK Approach*, Addison-Wesley, 1997.

[4] M. Bowen, *Handel-C Language Reference Manual*, Embedded Solutions Limited, edition 2.1, 1998.

[5] T. P. Kelly, *Arguing Safety – A Systematic Approach to Safety Case Management*, DPhil Thesis YCST99-05, Department of Computer Science, University of York, UK, 1998.

[6] M. Whalen, M. Heimdhal, *On the Requirements of High-Integrity Code Generation*, Proceedings of the 4th High Assurance in Systems Engineering Workshop, 1999.

[7] ISO, *Guide for the Use of Ada Programming Language in High-Integrity Systems*, ISO/IEC PDTR 15952:1998, 1998.

Biography

I. Bate, Research Fellow, Department of Computer Science, University of York, Heslington, York, YO10 5DD, U.K, telephone - +44 1904-432786, facsimile - +44 1904-432708, e-mail – iain.bate@cs.york.ac.uk.

Dr Iain Bate has been a Researcher within the Real-Time Systems Research Group within the Department of Computer Science at the University of York since 1994. His doctoral research, completed in 1998, focused upon establishing and demonstrating an approach to scheduling and timing analysis for safety-critical systems. His research has mainly been in the real-time systems, aspects of safety-critical systems.

S.A. Bates, MEng, Research Associate, Department of Computer Science, University of York, Heslington, York, YO10 5DD, UK, Tel: 0044 (0) 1904 433385, Fax: 0044 (0) 1904 432708, Email: simon.bates@cs.york.ac.uk

Simon Bates has been a Research Associate in the BAE SYSTEMS funded Dependable Computing Systems Centre (DCSC) at the University of York since October 2002. He graduated from the University of Manchester in 2002, where he attained a MEng (Hons) in Electronic Systems Engineering. Since taking up his role with the DCSC he has developed the following research interests: Safety Cases, Safety Case Architectures, Modular and Incremental Certification of Safety Related Systems, and Software Architectures.

Prof J.A. McDermid, Professor of Software Engineering, Department of Computer Science, University of York, Heslington, York, YO10 5DD, UK, Tel: 0044 (0) 1904 432786, Fax: 0044 (0) 1904 432708, Email: john.mcdermid@cs.york.ac.uk

John McDermid has been Professor of Software Engineering at the University of York since 1987 where he runs the high integrity systems engineering (HISE) research group. HISE studies a broad range of issues in systems, software and safety engineering, and works closely with the UK aerospace industry. Professor McDermid is the Director of the Rolls-Royce funded University Technology Centre (UTC) in Systems and Software Engineering and the BAE SYSTEMS-funded Dependable Computing System Centre (DCSC). He is author or editor of 6 books, and has published about 250 papers.