

# High-Integrity Interfacing to Programmable Logic with Ada

Adrian J. Hilton<sup>1</sup> and Jon G. Hall<sup>2</sup>

<sup>1</sup> Praxis Critical Systems, 20 Manvers Street, Bath BA1 1PX, England  
Adrian.Hilton@praxis-cs.co.uk

<sup>2</sup> The Open University, Walton Hall, Milton Keynes MK7 6AA, England  
J.G.Hall@open.ac.uk

**Abstract.** Programmable logic devices (PLDs) are now common components of safety-critical systems, and are increasingly used for safety-related or safety-critical functionality. Recent safety standards demand similar rigour in PLD specification, design and verification to that in critical software design. Existing PLD development tools and techniques are inadequate for the higher integrity levels.

In this paper we examine the use of Ada as a design language for PLDs. We analyse earlier work on Ada-to-HDL compilation and identify where it could be improved. We show how program fragments written in the SPARK Ada subset can be efficiently and rigorously translated into PLD programs, and how a SPARK Ada program can be effectively interfaced to a PLD program. The techniques discussed are then applied to a substantial case study and some preliminary conclusions are drawn from the results.

## 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 system performance. 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.

The airborne electronic hardware development guidance document RTCA DO-254 / EUROCAE ED-80[13] is the counterpart to the well-established civil avionics software standard RTCA DO-178B / EUROCAE ED-12B[12]. 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.

This paper examines how the requirements of DO-254 for high-integrity PLD program development may be satisfied by the use of Ada, and in particular the SPARK subset. We do not address ASICs or other electronic hardware. We also consider how to analyse PLD program correctness in the context of its interface to an Ada program running on a conventional microprocessor.

The Ada compilation work to date has taken insufficient advantage of the properties of SPARK Ada programs; we address this. In Section 4.2 we outline an algorithm for compiling SPARK Ada code into hardware.

## 1.1 DO-254 structure

DO-254 specifies the life cycle for PLD program development and provides guidance on how each step of the development should be done. It is not a prescriptive standard, providing instead recommendations on suitable general practice and allowing methods other than those described to be used. 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 for Levels A and B. We will focus on PLD programs of Level A and Level B integrity in this paper.

## 1.2 DO-254 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.

We show in this paper that using SPARK Ada supports the high-integrity verification methods of DO-254; specifically, elemental and safety-specific analysis, and is one way of using a formal notation and formal analysis techniques to increase confidence about PLD program design integrity.

## 2 Related work

### 2.1 Other standards

Within the United Kingdom, Interim Defence Standard 00-54[16] 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[15].

We have analysed the requirements of Defence Standard 00-54 and its implications for PLD program design in [8]. For the purposes of this paper it suffices to observe that 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. Formal specification and analysis are *mandated* at the higher integrity levels.

IEC 61508 “Functional Safety of Electrical / Electronic / Programmable Electronic Safety-Related Systems” [10] 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.

### 2.2 High-level imperative PLD programming

The practical domain-agnostic choices for embedded system development are currently Ada, C, C++ and (to some extent) Java. Java in its existing form is unsuitable for real-time systems as it requires periodic garbage collection. The Xilinx Forge toolset[6] compiles Java byte-code to Verilog, but is at an early stage of development and it is not yet known what limits it places on the source Java program.

C has been the basis for PLD programming in a number of language/compiler sets including Sheraga’s ANSI C to Behavioural VHDL Translator[14], System-C[5] and Handel-C[3]. Handel-C is the most promising approach and appears practical for efficient programming of low-criticality PLD applications. However the syntax and semantics of System-C and Handel-C are based on C and so they share C’s fundamental problems with high integrity.

C's failings are described by Romanski in [11]. He makes the key comment "The [C] language attempts to hide the underlying machine so that programs become portable between different machines. Unfortunately, the target characteristics show through." The lack of strong typing, substantial unspecified or implementation-dependent behaviour, and language constructs such as unbracketed single clauses and admissibility of assignment into conditions in C are viewed by Romanski as some of the chief deficiencies that make it unsuitable for inclusion in safety-critical systems, even if a "safe" subset is used.

C++ is almost a strict superset of C, and its additional object-oriented language features are generally irrelevant to hardware-specific concerns.

Additionally, if we wish to abstract away as many target hardware details as possible then the use of a low-level language such as C appears to be going in the wrong direction. This leaves Ada as the sole remaining practical choice. The close relationship between the syntax of Ada and VHDL further suggests that this is an appropriate choice.

### 2.3 Ada as a HDL design language

Ada was compiled to Behavioural VHDL by Sheraga[14], but a more modern approach to compilation has been detailed by Audsley and Ward[17, 19, 20]. They have developed the York Ada Hardware Compiler which compiles sequential SPARK Ada 95 through a circuit graph form to EDIF netlist format. This is a practical demonstration that such compilation can be done. Their emphasis is on worst-case execution time analysis rather than support of a DO-254 process.

Audsley and Ward have outlined in [18] how the Ravenscar tasking profile may be implemented on PLDs. Their emphasis is on worst-case execution time and scheduling analysis, leveraging the known timings of a PLD program. They combine the Ravenscar profile with sequential SPARK; now that SPARK Ada incorporates Ravenscar[4] this combination is supported by the SPARK Examiner. Although they outline how the Ravenscar constructs may be represented on a PLD, they have not yet demonstrated compilation of a multi-task Ravenscar program onto a PLD.

The York Ada Hardware Compiler appears to be the most advanced hardware compiler of (SPARK) Ada available at the moment. It has not been assessed as a tool for critical systems use. It remains to be seen whether communicating Ravenscar tasks can be efficiently compiled onto and run on a PLD, and whether this would be useful in PLD program design. The communication protocol between a PLD-based task and a software-based task must also be defined and analysed for correctness and efficiency.

## 3 Using Ada for high-integrity interfaces

Current PLD program development mainly uses the VHDL and Verilog hardware description languages (HDLs). Depending on the target domain there are a number of design tools available which generate an HDL description from a more

abstract design specification. However, it is common for designs to be written directly in VHDL or Verilog.

As is the case in software, the larger a hardware program is, the harder it is to write. In software we deal with this initially by programming in a higher-level language, reverting to low-level only when required for performance reasons. When no appropriate higher-level language is available, we apply well-established programming techniques such as abstraction, encapsulation and refinement to divide the program into manageable units.

When developing a complex embedded system the exact hardware-software partitioning is often not known until late in the development cycle. Additionally, target hardware may not be available for early testing. For this reason it is necessary to be able to represent most of the program in software for the purpose of host-based testing. Developing the program design in a single high-level programming language is a practical interim solution. The language must however make it simple to extract a program fragment into PLD form.

[7] argues that the SPARK 95 annotated Ada subset[2] is useful in the role of a PLD design language. SPARK supports design and implementation of programs at the highest levels of integrity, including DO-178B Level A and UK Defence Standard 00-55 S4. Its annotations allow design specification of desired program properties (data and information flow, partial correctness and design hierarchy) which are then checked against the actual code by the SPARK Examiner and SPADE proof tools.

SPARK aids PLD program design and compilation in ways which include:

- the enforcing tool (the Examiner) has an established product service experience on high-integrity projects;
- data flow annotations require developers to specify all global variables read or modified by a subprogram, allowing a direct view of all input and output data flow in the subprogram;
- the information flow analyser knows the exact inputs and outputs of each program statement, allowing automatic detection of independent sequential statements which may execute in parallel;
- SPARK permits proofs of absence of run-time errors and expression overflows, which allow PLD arithmetic blocks to be designed without having to consider how to detect and handle overflow; and
- pre- and post- condition annotations on subprograms allow the developer to specify additional restrictions on parameter and global input values, and verify that those restrictions are always met.

In particular, the ability to specify and prove properties of SPARK programs is a powerful tool when we are applying elemental analysis and safety-specific analysis as per Appendix B of DO-254. Static analysis and partial program proof are well-established formal methods with a history of use in industrial applications[1].

In Section 5 we give an example of the compilation of subprograms from a SPARK package body into VHDL.

## 4 The design options

When extracting a fragment of SPARK program code for execution on a PLD, the first task is to define the software-PLD interface. This is typically done with fully represented variables mapped onto memory locations corresponding to the PLD input and output pins, with additional variables to control asynchronous data sending and receiving. The developer must define and show correctness of this protocol before making any argument about the correctness of the software or the PLD.

To implement the extracted program fragment on the PLD there are three possible options: an interpreter, refinement and direct compilation. We reject an interpreter as impractical, given current PLD sizes. We now describe the other options.

### 4.1 Refinement

The pre- and post- condition annotations of SPARK allow the developer to specify the result of a subprogram computation. An example may be an integer square root routine:

```
procedure Sqrt(X : in Integer; Y : out Natural);
--# derives Y from X;
--# pre  X >= 0;
--# post (Y * Y <= X) and ((Y+1)*(Y+1) > X);
```

This defines the result  $Y$  as long as  $X$  is non-negative. The developer would then use the Examiner to generate proof conditions, and the SPADE tools to attempt to show that they are met. *As long as* any calling program only relies on the program meeting its specification, its implementation can be done in any way the developer likes.

This then allows us to replace the subprogram with an arbitrary PLD program, permitting the program to be optimised for minimum execution time outside the constraints imposed by the (essentially sequential) Ada language. However, to maintain integrity the developer is required to show somehow that the custom-designed PLD program meets the pre- and post-condition specification.

The authors have presented a formally defined specification and refinement system[9] suitable for designing parallel synchronous programs to run on programmable logic devices. The example developed in that article is a carry look-ahead adder optimised for execution time which it would make little sense to express in Ada. This then allows us to produce provably correct PLD program fragments as part of a larger Ada program design.

This can be combined with direct compilation, replacing one or more compiled subprogram logic blocks with the equivalent refined logic blocks.

## 4.2 Direct compilation

This is the approach normally taken for Ada program compilation for PLDs, and appears to be the most practical for programs of substantial size.

We suggest that, in order to maximise traceability from program design to PLD program, we adopt a recursive composition of subprograms with a stateless flow of information through the subprogram. Each Ada subprogram is represented by a HDL block with input wires corresponding to the subprogram input parameters and SPARK global input variables, plus a “start computation” wire which is asserted when all the subprogram input data is valid. The output wires of the block correspond to the subprogram output parameters and SPARK global output variables, plus a “computation complete” wire which is asserted when the subprogram computation is complete. The internal subprogram HDL design uses the control flow constructs described by Audsley and Ward in [17].

## 5 A case study

The first author has developed a substantial Ada real-time control program as the base for PLD compilation work. This program controls a hypothetical safety-critical embedded system, and has been developed using extant safety-critical program development techniques and tools. The code is publically available at <http://www.suslik.org/Simulator/> under the GNU General Public License.

### 5.1 The system

The system is the main control unit (MCU) for an endo-atmospheric interceptor missile, armed with a low-yield fission warhead. This system is clearly safety-critical; a detonation of the warhead at the launch site is a definite hazard to life. Of course, there are mission-critical requirements as well; if the warhead were never to go off, the missile targeted for interception would probably get through to its destination and be likewise a hazard.

The main hazard of the system will be detonation of the on-board warhead at an unsafe location (i.e., close to the launch point, or below a certain altitude). This dictates safety considerations such as having confidence in the estimated distance from launch point.

The control system itself consists of an interface to a standard 1553 bus, a number of packages monitoring system component data from the bus, and top-level packages reading data from these monitors and sending appropriate controls back onto the bus. The system components include a warhead, airspeed monitor and barometric sensor, supported by features such as a real-time clock and watchdog timer. An abbreviated system structure is shown in Figure 1.

### 5.2 The implementation

The program was developed using SPARK Ada 95 with full information flow analysis and run-time checking. Initially it was developed to run entirely in

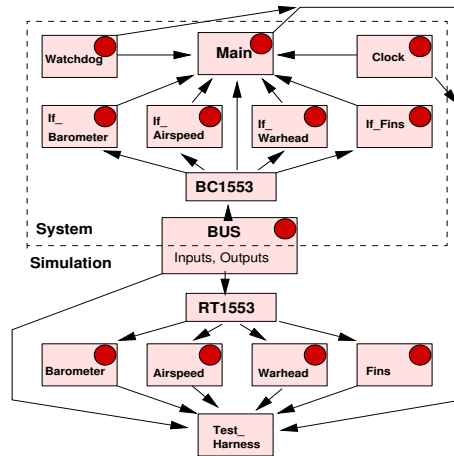


Fig. 1. Missile system design

software. Accompanying the main program is a set of simulation packages for the system components and an associated script-driven test harness for unit and system testing. The SPARK design boundary includes all packages but those directly related to testing.

The existing system code count is 504K of Ada files, with 16 800 lines. Of these, 2 500 lines are SPARK annotations, 2 900 are comments, 1 300 are blank, and the remaining 10 000 (forming 330K) are Ada code. 20K of this Ada was test-related code. There are 75 packages and public child packages, with 9 of those packages related to testing. This shows that the system is not trivial in size.

### 5.3 Identifying and extracting a fragment

We selected the `Nav` package as suitable for translation to hardware. This package provides subprograms for estimating the current missile heading, altitude and speed. It works by reading the barometer, compass, airspeed and INS sensor measurement and validity data, returning the data from the appropriate primary sensor if valid, and if invalid then estimating a result by combining data from the other (valid) sensors. There is a `Maintain` procedure which is called periodically from the main system polling loop and calculates updated sensor readings.

The original package has two SPARK abstract variables: `location_state` and `sensor_state` representing, respectively, histories of the recent sensor measurements and the sensors' validity states. The changes for PLD interfacing involved an extra abstract variable `fpga_inputs` representing the current sensor measurement and validity data output to the PLD. The `location_state` concrete variables were changed to be read-only variables memory-mapped to the PLD output pins and the new `fpga_inputs` concrete variables were write-only

variables memory-mapped to the PLD output pins. The bodies of the estimation subprograms were essentially unchanged from the original, reading the same named variables.

The main change was in the polled procedure `Maintain` which originally called `Handle_XX` procedures to update each calculated measurement. This was replaced with direct writing of sensor data to the PLD inputs. It resulted in a simpler naive information flow for the subprogram, although there was some information loss in that the dependency of the PLD outputs on the PLD inputs was not expressed. To implement this, the two moded abstract state variables would be replaced by an single unmoded abstract state variable `State` which would be changed at each write or read of PLD information.

#### 5.4 Transformation

The high-level structural steps of transformation of the selected `Handle_XX` subprograms into VHDL were:

1. replace global variables in the subprogram declaration and body with the appropriate PLD input and output vector names;
2. identify each subprogram's in and out argument and global data and create a VHDL architecture declaration for it;
3. add appropriate `Clock` and `Reset` inputs to the declaration;
4. connect the appropriate PLD input and output pins to the subprogram's inputs and outputs;
5. create the VHDL implementation for the subprogram by declaring architectures for the major Ada control flow elements;
6. add declarations for appropriate vectors to connect these architectures; and then
7. add the required connections between blocks and architecture inputs and outputs.

At the level of translating subprogram body code from SPARK Ada to VHDL, no initial effort was made to enable fine-grain parallelism. Instead, SPARK Ada program constructs (principally alternation and assignment) were mapped into the most directly corresponding VHDL representation (respectively, multiplexing from expression evaluation and data routing).

No compilation or simulation of the VHDL was done since it was a capability demonstration. A process for producing timing-robust VHDL from a SPARK design is clearly required for this transformation process to be practically useful. We must also compare execution time and PLD space usage for a benchmark set of Ada programs in order to form metrics-based arguments on the benefits of different compilation approaches.

#### 5.5 DO-254 considerations

Taking the main hazard of the system to be a detonation of the missile warhead in an unsafe altitude or distance, the `Nav` package is clearly safety-critical as it

provides best-effort estimates of the missile position and associated confidence levels. As such it is plausibly of Level A criticality.

A functional failure path analysis would quickly identify the safety-related proof conditions that would need to be discharged for each major design element (in this case, the SPARK subprograms). The stages of the safety argument would be:

1. a demonstration that the Ada subprogram satisfied the safety conditions, by an appropriate combination of static analysis, proof and manual inspection;
2. an argument that the software-PLD communication protocol used did not contribute to any hazards;
3. a mapping of the compiled VHDL to the original Ada, combined with the safety argument for the Ada; and
4. an argument that the safety-related elements of the VHDL are preserved in form in the compiled netlist.

We do not, for brevity, complete this argument.

## 6 Discussion and conclusions

We have described the requirements of RTCA DO-254 for high-integrity PLD programming. We have shown how Ada, and in particular SPARK Ada, is a suitable design language for high-integrity PLD programs. This has been done with the PLD programs forming part of a large hardware-software system. We have compared different approaches to translating Ada program fragments to an HDL, and concluded that a combination of refinement from formal specifications and direct compilation of SPARK Ada is effective and appropriate within the scope of RTCA DO-254.

We now examine what gaps in existing practice this work has revealed, and how these gaps may be filled.

### 6.1 Lessons from extraction

The extraction process yielded the following data points:

- Relatively little of the package specification changed. The abstract state variables gained SPARK modes, and one extra output abstract variable was required, but the global and derives annotations did not change greatly.
- Most of the work in the package body involved mapping concrete state variables onto the correct area of memory. External global data (from the sensors) was passed directly onto the PLD inputs.
- The transformation was not quite automatic, but was effected quickly and was amenable to manual inspection for correctness.
- The SPARK annotations were very helpful in characterising the inputs and outputs quickly, making VHDL architecture declarations simple to write.

- Bit widths could be easily calculated manually, and minimised by use of `pragma Pack()` and Ada representation clauses. There seems no reason why these widths could not be estimated by a relatively simple tool, given a SPARK syntax tree.
- The guarantee of no expression overflow given by the Examiner `-exp` flag (and subsequent proof) would greatly simplify the process of writing VHDL to compute arithmetic expressions.

The recursive block-based compilation method proposed in Section 4.2 should be measured against the existing York compilation algorithm with respect to execution time, PLD space usage and ease of comparison of the compiled netlist with the original Ada. This will give a measure of what program forms are best suited to each compilation method.

## 6.2 Language

The existing Ada language appears adequate for non-specialised design of a class of PLD program. The SPARK Ada language improves on this by allowing static analysis of program structure and admitting formal proof of relevant program properties.

The lack of simple generics in SPARK Ada is likely to be cause developer pain on larger projects or when reuse is required. To implement a subprogram with an array argument, SPARK requires the type of the array be known statically at each point of subprogram call. Allowing parameterisation by an enumeration or numeric type would violate the existing SPARK subset but facilitate once-only declaration of the subprogram.

## 6.3 Summary

RTCA DO-254 is a new 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-faceted argument for program safety.

SPARK Ada has been demonstrated to be a practical source language for PLD program compilation. We have shown how key features of SPARK Ada increase confidence that both the original and compiled program are correct. We have linked an existing rigorous parallel specification and refinement system with the specification and proof model of SPARK Ada, allowing provably correct PLD program fragments.

We and other developers have provided theory and tools to allow compilation of SPARK Ada programs onto PLDs at the highest level of integrity for DO-254. This work now requires a real-world case study to measure its practicality, susceptibility to verification, and computational efficiency.

## References

1. Peter Amey and Rod Chapman. Industrial strength exception freedom. In *Proceedings of ACM SIGAda Annual International Conference*. ACM Press, December 2002.
2. John Barnes. *High Integrity Software: The SPARK Approach to Safety And Security*. Addison Wesley, April 2003.
3. Matthew Bowen. *Handel-C Language Reference Manual*. Embedded Solutions Ltd, 2.0 edition, October 1998.
4. Rod Chapman. SPARK Examiner release note - release 7.0. Technical report, Praxis Critical Systems Ltd., August 2003.
5. Jon Connell and Bruce Johnson. Early HW/SW integration using SystemC v2.0. In *Proceedings of the Embedded Systems Conference*. ARM and Synopsys Inc., 2002.
6. Don Davis. Forge: High performance hardware from high-level software. Technical report, Xilinx, September 2002.
7. Adrian J. Hilton. *High Integrity Hardware-Software Codesign*. PhD thesis, The Open University, December 2003.
8. Adrian J. Hilton and Jon G. Hall. Mandated requirements for hardware/software combination in safety-critical systems. In *Proceedings of the workshop on Requirements for High-Assurance Systems 2002*. Software Engineering Institute, Carnegie-Mellon University, September 2002.
9. Adrian J. Hilton and Jon G. Hall. Refining specifications to programmable logic. In John Derrick, Eerke Boiten, Jim Woodcock, and Joakim von Wright, editors, *Proceedings of REFINE 2002*, volume 30 of *Electronic Notes in Theoretical Computer Science*. Elsevier, November 2002.
10. International Electrotechnical Commission. *IEC Standard 61508, Functional Safety of Electrical / Electronic / Programmable Electronic Safety-Related Systems*, March 2000.
11. George Romanski. Review of 'Safer C' (by Les Hatton). Technical report, Thomson Software Products, January 1996.
12. RTCA / EUROCAE. *RTCA DO-178B / EUROCAE ED-12B: Software Considerations in Airborne Systems and Equipment Certification*, December 1992.
13. RTCA / EUROCAE. *RTCA DO-254 / EUROCAE ED-80: Design Assurance Guidance for Airborne Electronic Hardware*, April 2000.
14. R. J. Sheraga. ANSI C to behavioural VHDL translator, Ada to behavioural VHDL translator. *The RASSP Digest*, 3, September 1996.
15. UK Ministry of Defence. *Defence Standard 00-56 Issue 2*, December 1996. Safety Management Requirements for Defence Systems.
16. UK Ministry of Defence. *Interim Defence Standard 00-54 Issue 1*, March 1999. Requirements for Safety Related Electronic Hardware in Defence Equipment.
17. M. Ward and N. C. Audsley. Hardware implementation of programming languages for real-time. In *Proceedings of the Eighth IEEE Real-Time Embedded Technology and Applications Symposium (RTAS'02)*, pages 276–284. IEEE, September 2002.
18. M. Ward and N. C. Audsley. 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.
19. M. Ward and N. C. Audsley. Language issues of compiling Ada to hardware. In *11th International Real Time Ada Workshop*, April 2002.
20. M. Ward and N.C. Audsley. Hardware compilation of sequential Ada. In *Proceedings of CASES 2001*, pages 99–107, 2001.