Orpheus: A Self-Checking Translation Tool Arrangement for Flight Critical Hardware Development

David Greve* Matthew Wilding* Mark Bickford† David Guaspari†

Abstract

We describe *Orpheus*, our vision for a development and verification environment for flight critical hardware devices. Orpheus provides an arrangement of translation tools that are self-checking and that integrate synthesis, high-speed simulation, and formal analysis. Implementation of the Orpheus architecture would allow tight integration of these formerly distinct activities and facilitate the use of formal analysis in flight-critical system certification. Further, flexibility in the choice of design representation provided by Orpheus would support both current design practice and hardware/software codesign. This paper describes the notion of self-checking tools, the Orpheus tool architecture, and how commercially-available tools could be used to implement such a system.

1 Current Practice

1.1 Background

Certification of flight critical systems is today a labor-intensive, manual process. Verification and certification of flight critical software and application-specific integrated circuits (ASICs) require an almost heroic effort of intense inspections and process documentation. The complexity of systems and devices will increase, because increases in cockpit automation and application integration offer important safety benefits, and because astonishing improvements in digital computing technology can potentially improve performance and decrease cost. The current approach to verification and certification may not be adequate in the face of this increased complexity. In order to reap fully the safety benefits of these technological advances we must develop new methods for verification and certification of flight critical devices.

Several recent developments permit a superior approach to verification and certification. First, flight critical ASICs can now be developed using standard hardware description languages (HDLs) because recent advances in equivalency-checking tools provide an independent check that synthesis preserves functional correctness. Second, theorem proving tools have emerged that enable mechanical formal analysis of device properties. Third, translation tools are emerging that allow the integration of mathematical analysis into the conventional fabrication/simulation-based development environment.

1.2 Flight Critical HDL Use

Modern hardware devices are typically developed using one of several hardware description languages (HDLs), such as Verilog or VHDL.

*Rockwell Collins, Inc. Advanced Technology Center, Cedar Rapids IA

†Odyssey Research Associates, Ithaca NY

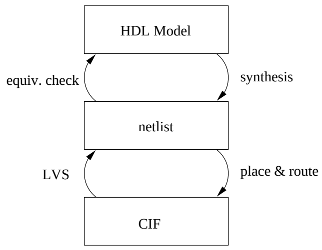

Figure 1: Fabrication toolsets for flight critical HDL provide self-checking

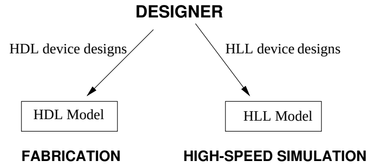

Figure 2: Designers typically build two device models

In the area of flight critical hardware, however, this has been the case only within the last few years. The delay in adopting these design techniques has been a result of concerns about the reliability of the process by which an implementation expressed in an HDL is used to fabricate the actual device. The complexity of HDLs means that tools that manipulate HDL designs are complex. As a result, the move toward using standard HDLs was hindered because requirements could not be traced to the device without trusting the synthesis tools and supporting libraries.

Fortunately, tools now exist that allow highly-dependable HDL fabrication. Figure 1 shows how 4 fabrication-oriented tools can be used to make the fabrication process immune from corruption by a fault in any single tool. A synthesis tool converts an HDL design into a netlist, and a place-and-route tool converts the netlist into CIF data that can be fabricated. The CIF data is checked against the netlist using an LVS (layout-versus-schematic) tool. The netlist is checked against the VHDL model using equivalence-checking tools.

The dependability of the connection between the design and physical device afforded by an independent tool chain as presented in Figure 1 has changed how flight critical hardware is developed. Incorporation of this innovation into the development process has allowed developers of airborne hardware to benefit from modern design practices such as synthesis and optimization.

1.3 Device simulators

It is commonly the case that a high-speed simulator is developed in parallel with an HDL model of a device. There are several reasons for this.

- Execution of the VHDL model is often too slow to support testing activities. This is especially true for large test suites such as are typical for regression testing.
- Software or other parts of the system that rely on the device must be developed before the HDL model is complete.

High performance is critical for device simulators, so simulators of this type are typically constructed using a high-level language (HLL) such as C or C++ for which there are compilers that generate efficient code[1].

[1] Multiple simulators are routinely built during device development. For example, a microcoded microprocessor's simulators would typically include both an instruction-level simulator and a microarchitecture simulator. The device simulator we are describing here is a low-level, cycle-accurate simulator.

The required functionality of complex computational devices is typically implemented using a combination of hardware and software, and an early design decision in the development of these systems is where to draw the line between these two kinds of implementations. The distinction between hardware and software in implementions adds complexity to these systems, since it requires that an interface be defined. Furthermore, this interface between hardware and software can change during a design cycle as implementation issues make clearer the tradeoffs between implementing various functions in hardware or in software. It would therefore be desirable to develop hardware and software using the same languages and tools, and delay decisions about the exact form in which they will be implemented. Designing could be done, for example, using C. Functions whose design will ultimately appear in hardware can be fabricated using the HDL representation. This has the potential to simplify development efforts since no hardware/software interface need be considered during development.

Figure 2 shows the artifacts resulting from current practice: two models that are expected to be identical in substance but that are written in different languages. This is typical of the current state-of-the-art design practice for airborne hardware devices.

2 Formal Analysis

Current certification processes provide some hard-to-quantify assurance that critical airborne hardware devices meet their requirements. Teams of inspectors "walk through" a design, assessing whether the implementation indeed meets the stated requirements. This process generates a paper trail that documents the level of effort of the inspectors and ensures that all relevant parts of the design have in fact been examined against the requirements. For complex designs this type of examination is very labor-intensive, but there is currently no viable alternative. Even so, the quality of the device is, to a large extent, measured indirectly via the inspection process.

Several aspects of the current process for developing and certifying safety-critical devices are not ideal. It would be better if certification practice measured the quality of the device directly, rather than measuring the effort applied to the verification. Further, as the trend is toward using more complex devices for critical airborne activities, current verification and certification threaten to become increasingly inadequate. It has long been hoped that mathematical reasoning — rather than careful documentation of the efforts of inspectors — could ferret out design flaws more effectively than manual inspections. The potential for establishing by direct, formal reasoning that a device meets its requirements has obvious appeal, and is increasingly recognized as a viable verification methodology by certification authorities.

Mathematical proofs about computing devices tend to be very complex and detail-laden, which makes them impractical to develop or check by hand. There has been considerable research applied to the development of automated theorem provers that are capable of checking and/or generating mathematical proofs. Leading tools include ACL2, HOL, and PVS, and each is increasingly finding application in industrial settings where safety or wide product distribution makes establishing design correctness imperative. Various verification projects have used theorem provers to analyze computer system models [1, 2, 4, 6, 8, 13, 14, 17]. A dramatic recent example of the possibilities of applying formal analysis to computing systems is the ACL2-checked verification of AMD's Athlon (formally "K7") floating-point operations [16].

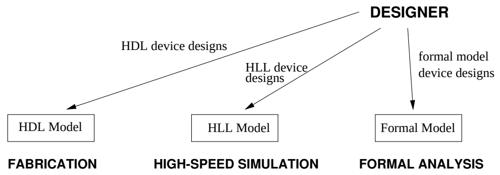

Figure 3: Formal analysis requires designers build yet another model.

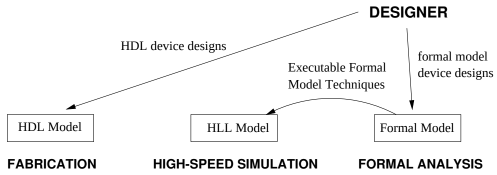

Figure 4: Executable formal models reduce the number of models

The increased industrial use of automated theorem provers results from improvements in the tools themselves and increased availability of reusable libraries of results [7, 9, 16]. Although we expect these tools will be increasingly common, poor integration with other aspects of the design environment remains an impediment to their adoption [12]. We believe that formal analysis will become pervasive *only when the tools are properly integrated with other aspects of the design environment.*

Figure 3 identifies the artifacts resulting from a process augmented to support formal verification: three models of the same device written in three different languages each supporting its own development or verification activity. In a recent effort Rockwell Collins developed three separate device models — one each for fabrication, simulation, and formal analysis — in order to benefit from each of these activities [11]. However, the high cost of building and maintaining models alone makes this approach unsustainable. Even more troublesome is that the multiple models might be inconsistent with each other, so a property proved about the formal model or the observable behavior of the simulator used to develop other parts of the system might not be reflected in the actual fabricated device.

3 Orpheus

We propose a comprehensive development and verification environment for safety-critical hardware devices called *Orpheus*. In Greek mythology, Orpheus subdues the fearsome, three-headed, dog-like Cerebus. As we have seen, verification and certification of increasingly-complex safety-critical devices requires us to overcome another three-headed challenge: to support device fabrication, high-speed simulation, and formal analysis in an integrated way. Orpheus does so without requiring the development of multiple models that are expensive and possibly inconsistent. The Orpheus approach can be integrated into current approaches for flight critical device development. The Orpheus tools are self-checking, so as to guarantee that no single translation tool can introduce an error into the verification process. The approach allows flexibility of design paradigm: it supports HDL development, hardware/software codesign, and designs derived from formal specification.

3.1 Reducing Three Models to Two

In part to address the issue of multiple distinct models, Rockwell Collins recently developed techniques that allow formal models written in a particular style in the ACL2 logic to be compiled into C for use as a high-speed simulator [10, 18]. This work effectively combines the formal and simulator models, thereby reducing the number of models from three to two. Figure 4 shows the impact of this innovation. The integration increases confidence in the validity of the unified model, since the *same model* is used both as a simulator and as a target of formal analysis. This important capability—high speed execution of formal logic definitions—has since been added to two theorem proving systems:

- A recent PVS extension provides a translator from PVS functions into Common Lisp. Rockwell Collins' preliminary tests using a version of the benchmark from [18] in PVS 2.3 [15] indicate that execution speeds are within an order of magnitude of the speed of a model written conventionally in C. We expect that PVS will ultimately develop the capability to integrate models expressed in the PVS logic into other tools.

- *Single-threaded objects* have been added to ACL2 2.4 and provide for high-speed execution of certain definitions [5]. Single-threaded objects are an extension of the notion of ACL2 "state" that permits the introduction of user-defined state elements. ACL2 enforces syntactic restrictions on the use of single-threaded objects to guarantee that the optimizations are legitimate. Rockwell Collins' experiments suggest that complex device models can be expressed despite the syntactic restrictions enforced on single-threaded objects, indicating that these restrictions do not make the ACL2 language impractical. Rockwell Collins has recently shown that ACL2 code can be integrated with other tools [18].

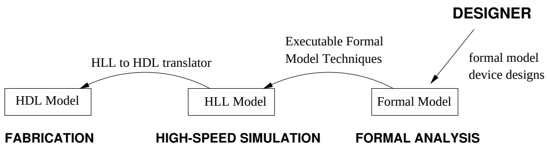

Figure 5: Formal models could provide a single, unified model

3.2 Reducing Two Models to One

An approach has recently emerged that potentially allows the integration of high-speed simulation models and device designs written in HDL. Several commercial tools are now available to translate high-level language (HLL) models into HDL models suitable for fabrication. Among the leading tools of this type are CynApps' C++-to-Verilog converter and C level's C-to-HDL converter, which generates either Verilog or VHDL. These tools promote an HLL-based design methodology that integrates simulation and fabrication. The existence of such tools and the emerging push for system level design and hardware/software codesign practices suggest that the commercial world will continue to develop and improve these tools.

The ability to compile a formal model into a simulation model, as described above, reduces the three models to two. Figure 5 suggests an obvious way to reduce the two models to one, by compiling the simulation model into a fabrication model expressed in an HDL. We discuss in Section 3.4 our initial testing of one of these tools, C level's C-to-HDL tool, and this experience suggests that Orpheus may be a realistic path for some applications.

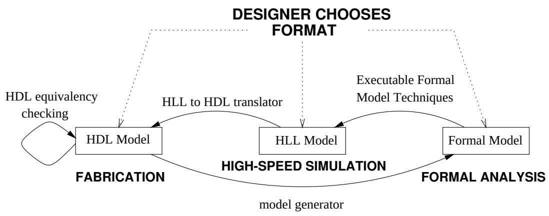

Figure 6: The Orpheus translation circle uses a single model to combine fabrication, high-speed simulation, and formal analysis

3.3 Closing the Loop with Orpheus

Although the tools outlined above allow the translation of high-level artifacts to HDL, and while such a process supports methodology changes that could reduce design errors, there are problems with using these tools for flight critical applications. First, the process outlined above requires that device development be accomplished by constructing a model or specification in formal logic. This is impractical, as hardware development is most appropriately done in an HDL or, in the case of hardware/software codesign, in an HLL.

The second issue is tracibility. Specifically, there must be a way to trace the requirements to the device through the tools. This issue is analogous to the one discussed in Section 1.2 that has until recently bedeviled those who wished to use an HDL for flight critical hardware design. Note that, unlike the fabrication tools of Figure 1, the compilation tools described in Figure 5 are not arranged to be self-checking. As a result the two compilers employed in this process would have to be thoroughly vetted before they could be used in a process for developing flight critical devices, which is problematic.

The Orpheus system addresses these two important issues by adding to the chain another tool, a *model-generator*, that converts an HDL design into a formal model. Figure 6 shows how the Orpheus tools are arranged. The translators form a circle in which a representation is converted in turn into each of the other representations and ultimately back into its original representation language. For example, a device model could be developed in an HDL that supports fabrication. The model-generator then creates a formal model that can be analyzed using a theorem prover. Using executable formal models techniques, the formal model is translated into an HLL model that supports high-speed simulation. Finally, the HLL model is translated back into HDL, and shown to be equivalent to the initial HDL model using an equivalency checker of the kind used in the HDL fabrication process.

There is only a single model, yet three distinct device representations are involved to support the three different uses: fabrication, high-speed simulation, and formal analysis. These three activities support each other, both for model validation and in the fabrication/verification process, because they involve the single model in different ways.

As previously discussed, although the necessary formats can be generated without completing the circle of translations, the question of translation correctness remains open. The certification of flight critical devices must address this issue. If the circle is completed, and the initial design and final design are shown equivalent, then each representation of the design is guaranteed correct so long as at most one of the tools has erred. Much as the fabrication tools diagrammed in Figure 1 are arranged to be self-checking, so too are the Orpheus translation tools. Even if more than one tool errs, the probability of catching the error is still very high since otherwise the multiple mistaken tools would have to fail in ways that mask each other's errors.

This kind of self-checking tool arrangement provides a very strong argument for the absence of translator-induced errors, and makes this kind of development practical just as modern self-checking fabrication tools permit HDL use in safety-critical devices. Orpheus therefore provides a framework for tight and highly reliable integration of formal analysis, simulation, and fabrication.

3.4 Orpheus Translation Circle Example

To assess the technical feasibility of the Orpheus approach, we have done a small experiment with using current versions of Orpheus components in a manner consistent with the tool arrangement of Figure 6.

As discussed previously, one of the advantages of Orpheus is that it allows a developer to use any of the representations for his device. We might expect VHDL to be the language of choice for hardware designers, while C might be preferred for hardware/software codesign. This experiment begins from a formal ACL2 model of an interrupt controller that forms part of a proprietary device developed by Rockwell Collins. We will navigate around the Orpheus circle to generate a simulation model, a VHDL model, and a second formal model. We have already discussed the benefits accruing from these different representations. The point of the experiment is to observe that the two formal models have sufficient similarities in structure, complexity, and level of detail to indicate that a proof of their equivalence — and therefore a self-check of all the translations — is feasible.

The Common Lisp model of this device uses a macro package developed by Rockwell Collins to ease modeling in Common Lisp. The line

```
(ST. SYNC1 = (& (ST. SYNC0) (HxFFDF)))
```

expresses the following behavioral detail: SYNC1 is an element of the machine state, a register. It is updated each clock tick with the result of applying a constant bit-mask to another state variable, SYNC0.

We also wish to simulate this device. We might choose merely to execute the Common Lisp code. However, there would be two disadvantages to that approach. First, it would be slow. Our experiments with running applicative Common Lisp models indicates that these models execute roughly 100 times slower than equivalent C language models [18]. Second, it is difficult to integrate raw, applicative Common Lisp into other tools.

Rockwell Collins has been working on this challenge for two years and, as described above, has sped applicative Common Lisp execution and integrated this code into other applications. This approach, broadly called "executable formal models," is outlined in two recent publications [10, 18]. Using these optimizations and a Lisp compiler, we generate a C program that executes at roughly the same speed as hand-coded C, and can be integrated with other software. Rockwell Collins in the past has integrated code of this type into various simulation and development environments [18].

We apply this technique to the example above. The line of the resulting C code that corresponds to the given line of Common Lisp reads as follows:

```
V12= (D.SYNC1 = ((((((V11)), Q.SYNC0)) &
  ((-(33))))),((V11)));
```

We also wish to fabricate this device. To do so we have applied a C-to-HDL tool (developed by C level) to convert the auto-generated C program produced into VHDL. Many transformations are done, such as converting variables in the C code that maintain state into registers in the VHDL. The line of C code shown above translates into the following line of VHDL:

```
D_var (SYNC1_2'range) := (Q_var (SYNC0_2'range)
  and "1111111111011111");
```

Ultimately, we wish to fabricate devices from VHDL using the approach outlined in Figure 1. We applied a Synopsys VHDL synthesizer to this VHDL code, and the result appears correct. As described in Section 1.2, it is this synthesis step from VHDL that current-available tools such as the Chrysalis equivalence checker can verify.

We really want to check much more than this final step. We want to verify that the synthesized design implements the formal model with which we began, so we complete the circle with a model-generator developed by ORA [2, 3]. This tool currently generates a description in first-order logic, rather than ACL2 code, and there are other modest problems related to differences between the VHDL libraries assumed by the C level tool and the libraries assumed by the ORA tool. However, with minor manual changes to the VHDL needed to overcome the library issue, we were able to use the model-generator to construct a specification in first-order logic. In this notation, the value assigned to SYNC1 is:

```
(slice(s.q, 79, 64) and
  flip(shift(vector("1111111111011111"), 78))))
```

The "`slice`" expression denotes the 16-bit slice of vector `q` that, by definition, represents `SYNC0`. The "`flip`" expression is of course the mask. (It is "flipped" because `q` has been defined to run down from 79 to 64 rather than up from 64 to 79.) Although it is expressed in a different syntax (i.e. Larch/VHDL rather than ACL2) the generated formal model corresponds term-by-term to the original Common Lisp (ACL2) model.

Sophisticated digital design, simulation and test-generation, and machine-checked formal analysis, each individually pose technical challenges that are not solved by using the Orpheus approach. However, Orpheus provides a framework for integrating these separate domains, and we believe that the simple experiment reported here indicates that this novel technical approach can succeed.

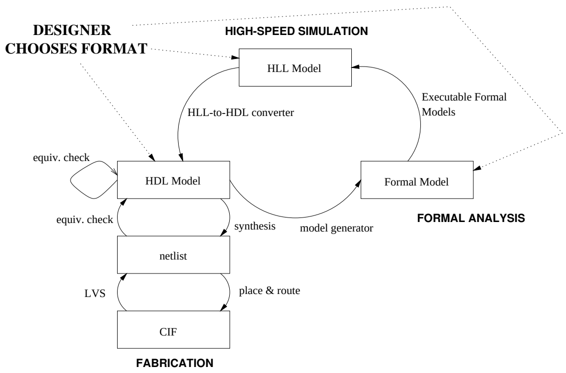

Figure 7: Orpheus Supports and Integrates Each Design Activity

4 Summary

Current verification and certification of devices appears increasingly inadequate in the face of increasing complexity of flight critical systems. Figure 7 summarizes the Orpheus approach. Orpheus supports hardware development and hardware/software codevelopment in a way that allows for formal analysis, fabrication, and high-speed simulation. The Orpheus tools are self-checking, just as modern HDL fabrication tools are, to insure their reliability. Orpheus supports a verification approach that forms the basis of a superior certification approach that provides a way to meet this looming challenge.

References

[1] William R. Bevier, Warren A. Hunt Jr., J Strother Moore, and William D. Young. An approach to systems verification. *Journal of Automated Reasoning*, 5(4):411–428, December 1989.

[2] Mark Bickford. Technical Information Report - Final Report for Formal Verification of VHDL Design. Technical Report TM-96-0025, ORA, July 1996. Delivered to Rome Lab under contract F30602-94-C-0136.

[3] Mark Bickford and Damir Jamsek. Formal specification and verification of VHDL. In Mandayam Srivas and Albert Camilleri, editors, *Formal Methods in Computer-Aided Design – FMCAD*, volume 1166 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[4] Robert S. Boyer and J Strother Moore. Mechanized formal reasoning about programs and computing machines. In R. Veroff, editor, *Automated Reasoning and Its Applications: Essays in Honor of Larry Wos*. MIT Press, 1996.

[5] Robert S. Boyer and J Strother Moore. Single-threaded objects in ACL2, 1999. `http://www.cs.utexas.edu/users/moore`.

[6] Robert S. Boyer and Yuan Yu. Automated proofs of object code for a widely used microprocessor. *Journal of the ACM*, 43(1):166–192, January 1996.

[7] Bishop Brock. ACL2 integer hardware specification (IHS) books, 1998. Standard ACL2 distribution at `http://www.cs.utexas.edu/users/moore`.

[8] Bishop Brock, Matt Kaufmann, and J Strother Moore. ACL2 theorems about commercial microprocessors. In Mandayam Srivas and Albert Camilleri, editors, *Formal Methods in Computer-Aided Design – FMCAD*, volume 1166 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[9] Ricky Butler, Paul Miner, et al. PVS libraries for arithmetic, sets, and graphs. `http://shemesh.larc.nasa.gov/fm`.

[10] David Greve, Matthew Wilding, and David Hardin. High-speed, analyzable simulators. In *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Publishers, to appear. `http://www.pobox.com/users/hokie/docs/hsas.ps`.

[11] David A. Greve. Symbolic simulation of the JEM1 microprocessor. In *Formal Methods in Computer-Aided Design – FMCAD*, Lecture Notes in Computer Science. Springer-Verlag, 1998.

[12] David Hardin, Matthew Wilding, and David Greve. Transforming the theorem prover into a digital design tool: From concept car to off-road vehicle. In Alan J. Hu and Moshe Y. Vardi, editors, *Computer-Aided Verification – CAV '98*, volume 1427 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998. `http://pobox.com/users/hokie/docs/concept.ps`.

[13] Steven P. Miller, David A. Greve, Matthew M. Wilding, and Mandayam Srivas. Formal verification of the AAMP-FV microcode. Technical report, Rockwell Collins, Inc., Cedar Rapids, IA, 1996.

[14] Steven P. Miller and Mandayam Srivas. Formal verification of the AAMP5 microprocessor: A case study in the industrial use of formal methods. In *WIFT'95: Workshop on Industrial-Strength Formal Specification Techniques*, Boca Raton, FL, 1995. IEEE Computer Society.

[15] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS System Guide*. Computer Science Laboratory, SRI International, Menlo Park, CA, September 1998.

[16] David M. Russinoff. A mechanically checked proof of IEEE compliance of the floating point multiplication, division, and square root algorithms of the AMD-K7 processor, January 28 1998. `http://www.onr.com/user/russ/david`.

[17] Matthew Wilding. A mechanically verified application for a mechanically verified environment. In Costas Courcoubetis, editor, *Computer-Aided Verification – CAV '93*, volume 697 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993. `ftp://ftp.cs.utexas.edu/pub/boyer/nqthm/wilding-cav93.ps`.

[18] Matthew Wilding, David Greve, and David Hardin. Efficient simulation of formal processor models. *Formal Methods in System Design*, to appear. Draft TR available as `http://pobox.com/users/hokie/docs/efm.ps`.