Formal Interface Compliance Verification

C. Norris Ip

ip@tempusf.com, Tempus Fugit, Inc., Fremont, CA

Abstract

This paper discusses the role of formal interface compliance verification in a design flow, and how it can significantly improve the design and verification process of complex SoC.

1 Introduction

Most complex SoCs and reusable IPs being designed today work around standard interfaces. And as design complexity increases, proprietary interfaces are also being developed more vigorously within large companies to partition a complex design and for reuse across multiple product generations.

The verification problem for designs using these interfaces is critical [1]. Frequently, during the design stage, it is not known which specific systems the SoC or the reusable IP will be interacting with. Furthermore, it is possible that different designers reading the same English specification may interpret various scenarios differently.

Since 1999, the team at Tempus Fugit has been formally verifying complex interfaces, for a wide range of designs. To prepare for these verification efforts, we first captured the interface requirements in an executable verification module. We will use PCI [2], AGP [4], and PCI-Express as examples in this paper. Each of these interfaces contain more than a hundred requirements, some come from the actual compliance checklist from PCI-SIG [3], and some come from the English specification documents.

Once the requirements were captured, we reused them for a wide range of designs. Using TempusQuest, an advanced formal verification tool developed at Tempus Fugit, we analyzed each complex design for compliance with these requirements, exhaustively catching all corner cases that violate these requirements. These efforts can fit into any existing methodology and lessen the effort in writing simulation test cases to hit these corner cases.

Finally, to obtain the maximum benefit of formal technologies, we also apply our tool to the interfaces at the individual blocks of a design, utilizing *an assume-guarantee* approach to verify the interaction among multiple components. We also applied formal techniques on some of the end-to-end requirements for the components. In doing so, we can replace a significant portion of ad-hoc unit-level simulation tests, resulting in shortened integration time, and perform formal regression to make sure last minute changes have not broken the design.

PCI, AGP, PCI-Express and other standard or proprietary interfaces contain complex requirements. While there are several commercial assertion-based verification products, their focus is typically simple local assertions instead of complex interface requirements. If a requirement is too complex for the automated analysis to formally verify, the requirement is validated in simulation instead (as stimulus generator and event checker). However, we have been able to consistent verify them formally with our verification engine developed at Tempus Fugit. In fact, the tool is powerful enough to verify not only interface compliance, but also end-to-end functional requirements.

In section 2, we summarize various aspects of functional verification of these complex designs, and then in section 3 we move on to discuss the potential productivity gain through the consistent use of formal techniques.

2 Functional Verification of RTL Designs

The various aspects of functional verification can be summarized as follows:

Local Assertions

A local assertion is a statement to declare the implementation decision with respect to the code in the surrounding vicinity. Examples include one-hot encoding, queue-overflows, etc.

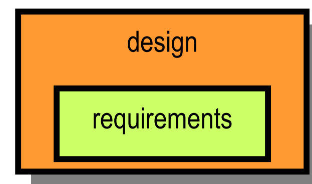

Figure 1. Local Assertions

Interface Requirements

An interface requirement is a statement to declare how two entities communicate with each other. Each

entity can be a complete ASIC design, an SoC design, a reusable IP, or a block within a design.

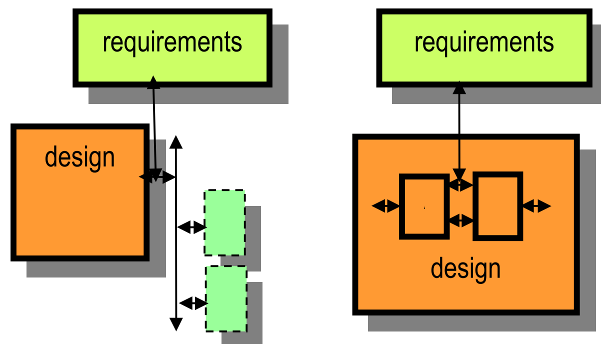

Figure 2. Interface Requirements

Let's take the PCI and AGP interfaces as examples. PCI has more than a hundred requirements, and one requirement is that the master must always assert byte enables on the upper bits of CBE when the master asserts REQ64# and the slave has not responded. AGP has a similar number of requirements, and one of the requirements is that, if any AGP read or write grants are pending, the core logic must not start PCI or fast-write transaction.

End-to-End Functional Requirements

An end-to-end functional requirement is a statement to declare what an entity should generate with the supplied data. The entity can be a complete ASIC design, an SoC design, a reusable IP, or a block within a design.

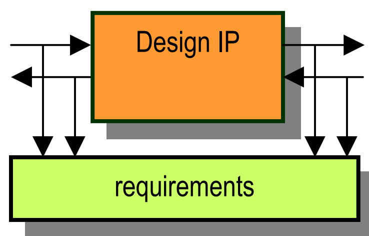

Figure 3. End-to-End Functional Requirements

Let's take the link layer of the PCI-Express interface as an example. For each packet going into the design, we need to verify proper sequence number assignment to the corresponding outgoing packets, their data integrity, proper ordering, and that the design does not drop or duplicate packets.

System-level Requirements

A system-level requirement is a statement to declare the overall behavior of the design across time. Sometime it involves less tangible items such as typical throughput, etc.

The complexity of these requirements increases as they move from local assertions to system requirements. Because local assertions cover the code in the surrounding vicinity, these local assertions tend to catch local problems.

Because interface requirements describe how two entities communicate to each other, they typically involve the control logic of the design. These requirements become the contract between the two entities. The verification of these requirements on each entity is challenging because of potential ambiguity in the specification and because of the complex corner cases due to the possibly arbitrary behavior of the other entity.

While the implementation within a module may change from time to time, a well-designed interface is less likely to change. This is especially true for standard interfaces and reusable cores. Furthermore, it is usually difficult to manually create simulation test benches to hit all the corner cases with respect to the interface requirements, so it naturally calls for a formal solution.

On the other hand, end-to-end functional requirements are related to the data-path and the overall functionality of the design. Formal verification of these requirements is often more difficult, but it has the potential of significantly contribute to the ASIC and IP sign-off process.

Finally, less tangible system requirements are probably best done by system-level simulation, which is outside the topic of this paper.

3 Using Formal Interface Compliance Verification

The ability to formally verify interface requirements (and end-to-end functional requirements) can have a big impact in the design process. Without it, a design flow may look like the diagram in Figure 4.

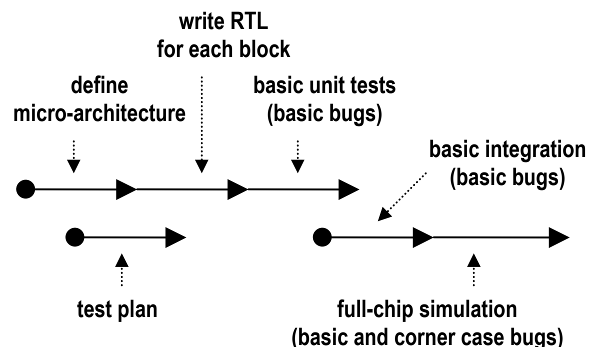

Figure 4. Basic Design Process

In this process, the designer would write ad-hoc unit-level simulation tests to verify some functionality of the blocks, but the main effort in finding corner case bugs would have to wait until full-chip simulation is up and running. Because of the loose interaction among the development of individual blocks, a significant amount of time is also spent in the integration of the units before full-chip simulation is ready.

With a formal tool capable of verifying complex interface requirements, significant improvement in the development process can be achieved with minimal changes to the existing methodologies. For example, it may be shortened into something like the diagram in Figure 5.

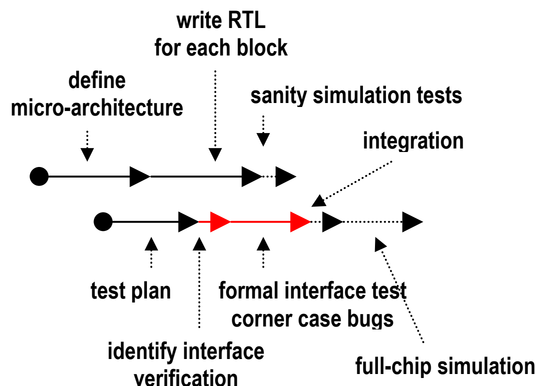

Figure 5. Formal Interface Verification

The steps relevant to the use of formal interface compliance verification in this diagram are:

1) Work out the overall architecture of the design.
2) While defining individual components in the design, create or buy appropriate *interface verification module* for formal analysis.
3) While developing the RTL code for individual components, debug the code interactively and formally with respect to the requirements in the verification modules.
4) During the process of developing the RTL code, exchange assumptions among the designers and verify them formally.
5) When full-chip simulation is up and running, maintain the day-to-day correctness by running formal regressions.

In this process, the interface requirements are used as a contract among the blocks in the design. If each block has been verified using the same interface verification module before integrating into full-chip simulation, the integration step would be much shorter.

Furthermore, the full-chip simulation step would also be shorter, because corner case bugs with respect to the interfaces and blocks are detected early during the unit-level verification. While the designers are busy writing the RTL, the verification engineers can write or obtain appropriate interface verification modules. We have also designed our tool so that the verification of the unit may start even before the RTL of the block is complete, because our tool will regard the missing pieces as black boxes.

Finally because formal verification achieves 100% coverage for the requirements, the proofs performed during the early steps can be executed again as an effective formal regression during the later steps, while enhancements and non-interface bug fixes are made to the design description.

The remainder of this section will go over each step in more details.

3.1 Interface Verification Modules

Very often, it is possible that different designers reading the same English specification may interpret various scenarios differently. It is important that the designers or the verification engineers consider all possible interpretation of the other side of the interface.

Writing the interface requirements in an executable format would reduce the possibility of misunderstanding. We tested our verification modules for various standard interfaces in a wide range of designs, and therefore, have resolved most of the misunderstanding. For proprietary interfaces, the architects or the verification engineers responsible for the design may write the verification modules. Because these verification modules are written independently from the design efforts, discrepancy in the interpretation of specification by the architects and the designers can be resolved through the formal verification process.

The actual languages in which these requirements are specified are not critical to the design flow, as long as they have clear semantics associated with them. In many cases, a hardware description language such as Verilog and VHDL is sufficient. Open Verification Library (OVL) can be used a convenient layer on top of Verilog for a more concise assertion specification.

There are other advanced languages being standardized in the industry. A declarative property specification language is being standardized by the Accelera Formal Verification Technical Committee, and a procedural assertion construct proposal is being evaluated by the Accelera SystemVerilog Committee. Both of them contain constructs that are more expressive than Verilog and VHDL, useful for liveness properties that describe infinite behavior.

One of the difficulties in getting the designer to use assertion-based verification is to persuade them to write assertions. By supporting HDL languages, they can start capturing requirements without learning a new language.

By concentrating on interfaces, the verification module can be created by anyone, not necessarily the designer. Using it as a contract between two components can avoid misunderstanding, and free up precious simulation time for validating the transaction-level activities of the design.

3.2 Formal Block-level Verification Before Simulation

Because the formal analysis does not depend on the creation of a test bench, designers can start verifying their blocks without waiting for the simulation environment to be up and running. Furthermore, formal verification may start even when the RTL code has not been completely developed. Any missing code in the RTL can be regarded as a black box that generates arbitrary outputs. If the existing RTL can be formally verified with these arbitrary behaviors, the RTL is guaranteed to behave correctly when the missing pieces are added.

3.3 Simplifying the Integration Process

Traditional unit-level simulation tests are usually done in an ad-hoc way with restricted stimulus. Therefore, they usually test the basic behavior of the block only. Furthermore, as shown in Figure 6, the stimulus generator and the event checker for individual blocks are typically developed independent of each other, and it is difficult to determine whether the generator for block A generates all legal outputs from block B and vise versa.

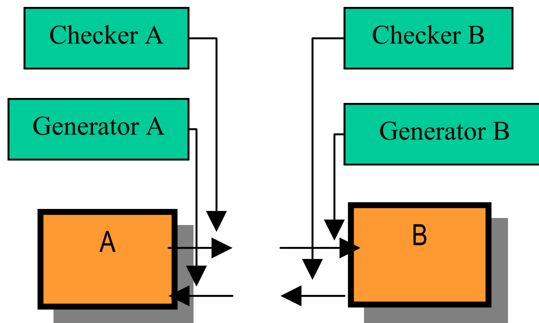

Figure 6. Unit-level Simulation Tests

When formal verification is used, the same interface verification module can be used for the verification for both block A and block B, as shown in Figure 7.

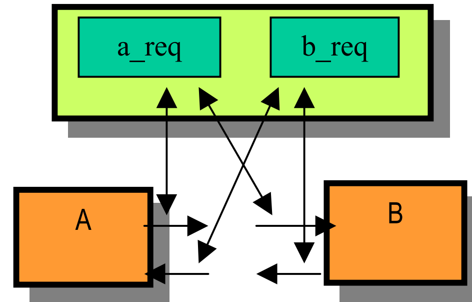

Figure 7. Formal Unit Tests

During the verification of block A, the proof can be set up as follows:

% assume b_req
% prove a_req

and during the verification of block B, the proof can be can be flipped as follows:

% assume a_req
% prove b_req

This style of proof structure is known as the *assume-guarantee approach*. Because formal verification is exhaustive, the verification of block A may also detect extra assumptions about block B in order for block A to function correctly. These assumptions can be appended to the verification module so that they can be verified in block B. A semi-formal or simulation methodology would be less effective

The fact that these requirements can act as a contract has serious implications. As shown in Figure 8, it can be used to verify an IP core while capturing its assumption about the environment. The same requirement can be packaged with the IP so that the IP user to verify those assumption (formally or informally) to make sure the SoC is using the IP correctly.

3.4 Trust-worthy Regressions

The verification modules and the set up scripts for the proofs at the unit level can also be reused as a formal regression through out the later steps in the design process. Because the addition of a new feature or the fix for other functional requirements may introduce new corner cases, it may break existing features. Because of the new corner cases, existing simulation tests may not be able to detect this problem. New tests have to be written for both of the old and the new features. With formal verification, the formal proofs can automatically

adapt to the new design description and verify the new corner cases as well, so the designers only need to verify the new features.

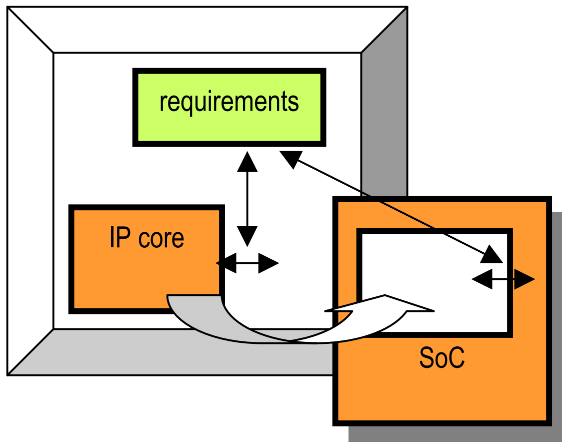

Figure 8. IP and Its Operating Environment

4 Conclusion

In this paper, we identified the various aspects of functional verification and described how formal verification can significantly impact the design process. Our experience indicates that applying formal tools for the interface compliance verification (and end-to-end functional requirement) can lead to a significant productivity gain.

Reference

[1] Vigyan Singhal and Joseph Higgins, Compliance Verification for SoC and IP interfaces. DesignCon 2002.

[2] PCI Special Interest Group. PCI Local Bus Specification Rev 2.2. December, 1998.

[3] PCI Special Interest Group. PCI 2.2 Compliance Checklist. www.pcisig.com.

[4] Intel Corporation. Accelerated Graphic Port Interface Specification, Revision 2.0.