Interface-Based Design

James A. Rowson
Alta Group of Cadence Design Systems Inc.
jimr@altagroup.com

Alberto Sangiovanni-Vincentelli
University of California at Berkeley
alberto@eecs.berkeley.edu

Abstract

A new system design methodology is proposed that separates communication from behavior. To demonstrate the methodology we applied it to a simple ATM design. Since verification is clearly a major stumbling block for large system design, we focussed on the verification aspects of our methodology.

In particular, a simulator was developed that is based on the communication paradigm typical of our methodology. The simulator gives substantial performance improvements without sacrificing user access to detail.

Finally, the potential for this methodology to improve verification, modeling and synthesis is explored.

1. Introduction

The design of large electronic systems such as an ATM network, a computer network, an automotive engine control unit, a multiprocessor system, is indeed very complex. Complexity arises not only from the ever increasing functionality of the systems but also from more and more stringent requirements imposed upon them: time-to-market constraints, safety and performance requirements are populating the nightmares of system designers.

Time-to-market pressure together with the multitude of components that are needed to implement the required functionality make it impossible for a single company to design and manufacture an entire electronic system in time and within reasonable cost. Hence, design re-use and Intellectual Property (IP) trading should now be considered a necessity. The recent Virtual Socket Initiative (VSI) [1] is a fist step towards a methodology supporting the trade of IP blocks. However, present design methodologies are at a loss when IP blocks coming from different design groups are mixed and matched to create a new product. In particular, verifying whether a design satisfies all constraints and requirements is today a most difficult design step. To address this problem, we emphasized that design tools are not enough, new methodologies have to be put in place [2]. This paper is concerned with an important aspect of a design methodology that favors design re-use and verification: **interface-based design**.

The design methodology originally proposed in [2] was based on three cornerstones:

DAC 97, Anaheim, California
(c) 1997 ACM 0-89791-920-3/97/06 ..$3.50

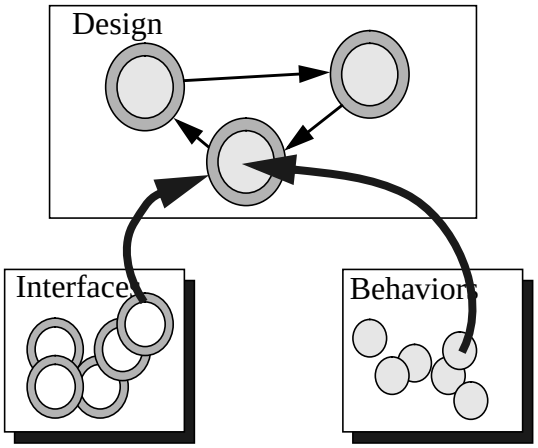


Figure 1: Separating Behavior from Communication

- Formalization, which consists of capturing the design and its specification in an unambiguous, formal "language" with precise semantics.
- Abstraction, which eliminates details that are of no importance when dealing with high-level design or checking whether a design satisfies a particular property.
- Decomposition, which consists of breaking the design at a given level of the hierarchy into components that can be designed and verified almost independently.

These three theoretical tools can be used to simplify design and verification in many different ways. An important one consists of "orthogonalizing" the properties of a design. For example, decomposing the verification problem into a functional verification phase, where the timing aspects of the design are ignored, and into a timing verification phase, where only limited information about functionality is considered, has been a major design methodology improvement.

In this paper, we propose a way of orthogonalizing an electronic design along different dimensions: **behavior and communications**. This idea is based on the realization that there is a common structure of electronic systems consisting of a set of entities (either hardware, software or both) that are connected together. Each of the entities may either be a similar structure consisting of other inteconnected entities or a basic block of the design, the leaf of the hierarchy. The interconnection is a symbolic way of indicating communication that takes place among entities. In fact, an interconnection of software models is certainly an abstraction since there are no wires there to use for the communicating entities!

Examining most of the methods used to design such systems shows that communication is often intertwined with behavior and/or with its physical carrier so that it is difficult to talk about the "abstract" communication aspects of a design. This is particularly true at the RTL level and below.

Inspired by [3,4,5,6,7,8], we believe that it is possible to clearly

identify what role communication plays at **all levels** of the design and keep it separate from component behavior (see Figure 1). It is our intention to show how the design of an electronic system can be carried out into **two almost independent steps**: the design and selection of the functionality of the components of the design and the way in which these components interact through a communication mechanism. In particular, we believe that a specific methodology can be used to carry out the design of communications in a top-down, constraint-driven fashion.

2. Interface-Based Design

We propose a new way of describing designs that includes communication abstraction. Because the focus of such a method would be on how the modules interface with each other, we'll call this new method interface-based design.

Telecommunication developers have long used protocol stacks to provide an abstraction mechanism for their complex communication systems. Protocols use hierarchical principles to hide complex time or encoding behavior of the lower levels of the communication stack. Each layer of the stack provides System Access Points (or SAPs), which are points of contact between that stack layer and the one below. These SAPs are analogous to ports on a module.

As a more pragmatic example, a bus could be described as a simple hierarchical protocol. The top layer of the protocol is the set of possible bus transactions: read, write, burst read, burst write, read-modify-write. Each of these transactions can be further described using some protocol on a given set of pins, say PCI bus or EISA bus.

In order to generalize these concepts to all levels and types of design, we need to formalize better what is the basic communication mechanism that we propose and how to refine it.

2.1 Models of Computation

Lee [11] has proposed over the years that systems be designed using a heterogeneous collection of models of computation. A model of computation is a self-consistent set of rules or laws of physics that are useful for modeling at an abstract level. Commonly used models include Dynamic Data Flow, Communicating Finite State Machines, Synchronous Data Flow and Discrete Event. Each of these models has certain properties that are quite useful in design.

If the differences between the most popular system level models are analyzed, it becomes clear that the major difference is in the method of communication between concurrent objects. Dataflow [9] (both dynamic and static) communicate using queues, never losing a token. Discrete Event [10] (DE) level performance modeling uses a single entry queue to hold tokens until the receiver can be invoked. Each model of computation relies on specific properties to be guaranteed by the communication mechanism. In DE, there is no guarantee that every event will be seen by the receiver because the receiver may not be sensitive to that event when it is emitted. On the other hand, there is a global order that can be established between events in unrelated portions of the design. Dataflow guarantees the safe arrival of every token and that the sequence of tokens on each communication path is defined. There is, however, no global ordering for tokens in unrelated design parts.

Others have also proposed models of computation. Chiodo et al. [12] have proposed a model (Codesign Finite State Machines) based on FSMs communicating among themselves with event broadcasting that allows for asynchronous communication. Note that in this case the composition of the FSMs under the event broadcasting communication model cannot be claimed to have the same property as an FSM. Both DE and event broadcasting were selected because of their simplicity and generality. However, while other more complex models of communication could be expressed in terms of event broadcasting, it was not clear how to use this model to do top-down design of the communication part of the design.

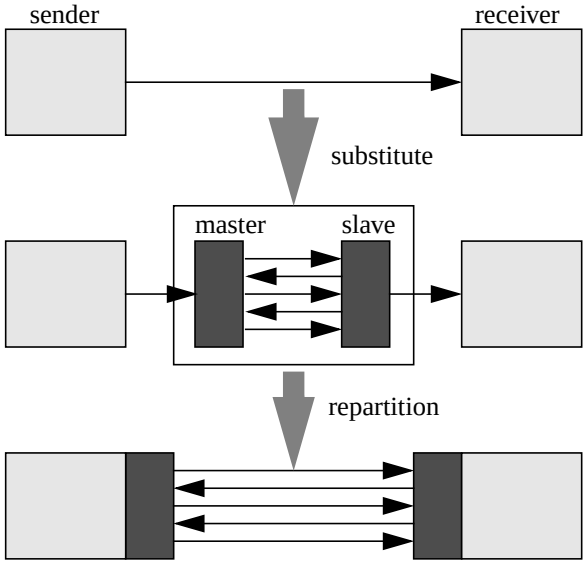


Figure 2: Refining Communication

2.2 Incrementally Refining a Model of Computation

The interface-based system design methodology that we propose adopts the token passing methodology from dataflow and discrete event system level while providing a method to refine communication mechanisms incrementally and hierarchically.

A token represents a complete communication between two or more design entities. Some examples of tokens include bus writes, bus reads, bursts, and variable length encoded data.

The process of refining token passing down to an implementation is similar to the successive refinement concept proposed by several researchers especially in the formal verification community. One simple method of communication refinement could be implemented by replacing the abstract, simple token exchange in the model of computation by another design that has two parts: a master and a slave (see Figure 2).

The master side initiates the communication by accepting the token from the sender and breaks the token down into a series of data and handshaking events on some new communication paths.The slave recognizes the handshaking events and gathers the sent data, reconstructing the token for delivery to the receiver. The master part of this new communication design will be synthesized with the sender, the slave synthesized with the receiver.

This methodology is clearly not limited to hardware. It is appropriate for describing communication between multiple software threads (shared memory, queues, posted events, etc.) or between hardware-software (polling, interrupts, etc.) or software-hardware (I/O instructions, memory mapped registers, coprocessor).

3. Interface-based Design Example

Hierarchical stepwise refinement is an essential part of interface-based design. To illustrate the concepts, we use a mixed hardware/

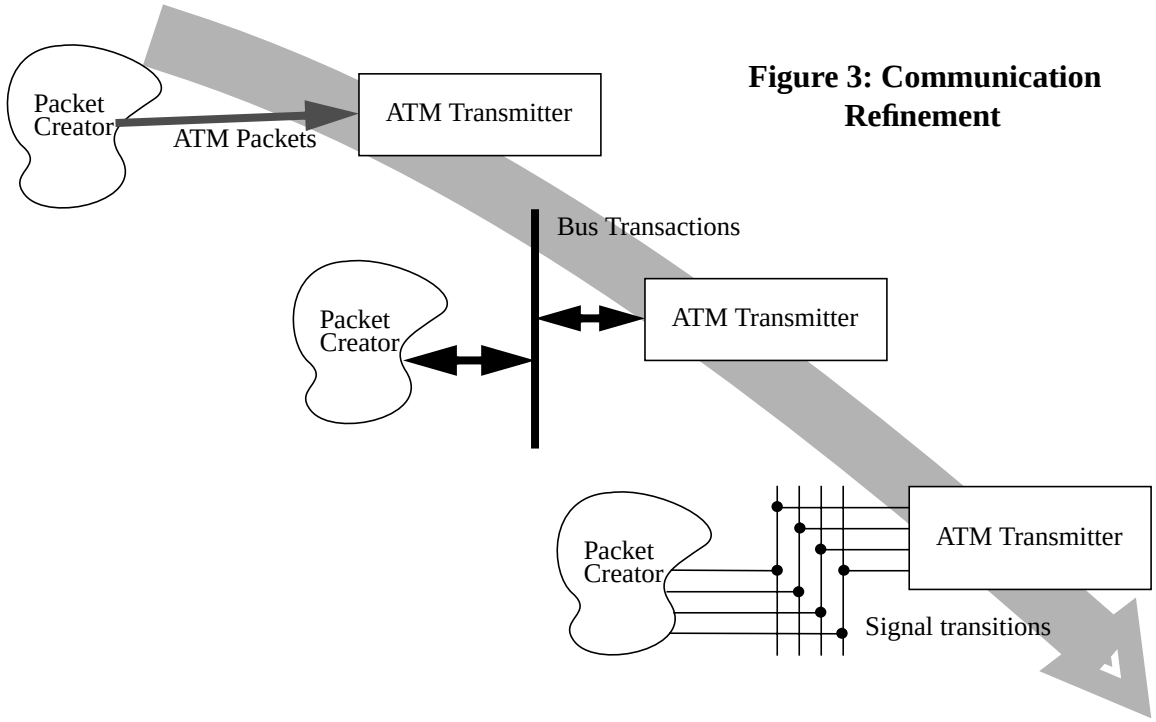


Figure 3: Communication Refinement

software design that is communicating across a network using ATM packets. For example purposes we will only look at a small portion of the overall system, namely a software thread that is sending packets through a hardware transmitter (see Figure 3). We implemented this example in an experimental simulator named Cheetah that is described in more detail in the next section.

At the most abstract level, the ATM packets will be modeled as a token. The token contains 53 bytes of data (5 header bytes and 48 payload). To debug system functionality, we can pass these tokens around without regard to how long they take to be transmitted. At first, it is immaterial that the packet creation is software and packet transmission is hardware, and in fact the exact line between hardware and software should remain fuzzy as long as possible to keep implementation options open. Simulation at this abstract level can be done in the DE domain by passing tokens between abstract behaviors simply by exchanging pointers.

Simulation results from Cheetah at this level consist of a simple trace of the ATM packets crossing through the interface between the blocks.

As we refine the system we need to make design implementation choices. We can now explicitly choose to put the packet creation in software and transmitter in hardware. The communication between them will be across the processor's bus. At this point we have not chosen the processor bus (or the processor, for that matter), so we will model this refinement using an abstraction of a bus that handles reads, writes, burst reads, and burst writes. Each of these transactions will, for the moment, take a constant amount of time.

Refining the token passing into a series of bus transactions can be done in many different ways. In this example, we'll only consider the differences between individual writes and burst writes. Since each ATM packet has 53 bytes, we can transmit it in several different ways, from one byte per write to all 53 bytes in a single burst transaction. To make the protocol a bit more robust, we will add a write to a special location as the first bus transaction, followed by writes of the real data to another address as shown in Figure 4. The data transactions can be burst transactions of various lengths or could be individual writes. In a real system, we would probably be packing bytes into words for efficiently, but for this example we will just put each byte into a separate write (or separate part of a burst write).

Of course, we are dealing with a more complex system than just the packet creator and transmitter, so other communication paths will also be sharing the processor bus. We need to make sure that our protocols are being refined correctly, that the transmitter is faithfully reconstructing the packets and that other traffic on the bus is also getting through. At this point, we cannot yet do accurate performance studies (although if we made a slightly more accurate bus model, we could get a first feel of bottlenecks). Arbitration of the bus has to be modeled at this level, and we can start to assign memory maps and get the programmer's model of the system decided upon.

Simulation results from Cheetah at this level of refinement allow

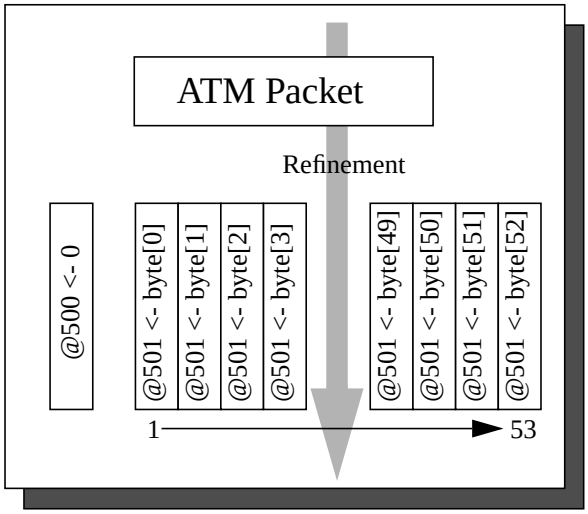
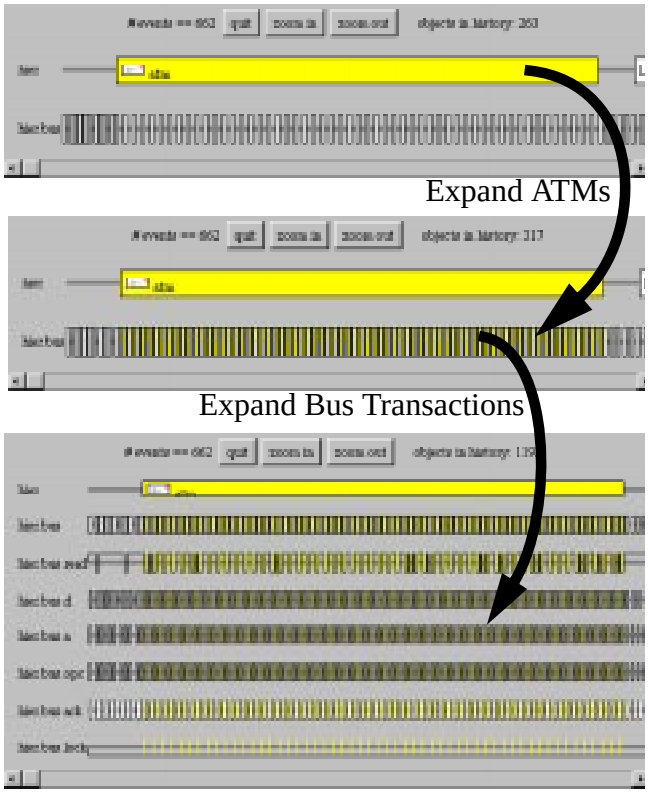


Figure 4: Refining the ATM cell

Figure 5: ATM packets sent over a PI Bus



the user to "expand" the ATM packets into the more detailed bus transactions that implement the protocol.

The next refinement is to select a bus implementation. We might choose a PCI bus or an EISA bus, but for this example we chose the PI-bus from the Open Microprocessor Initiative in Europe [15]. The PI-bus is a synchronous bus intended for on-chip work. Now we can get cycle accurate knowledge of how each bus transaction is happening. Buses can have multiple masters, so the arbitration of bus access is important. Here, arbitration is modeled to the cycle accurate level, as is the performance.

Simulation results here allow the user to "expand" down through the bus transaction level and into the cycle accurate pin transition detail as shown in Figure 5. Within Cheetah this last level of detail isn't simulated, but is instead recreated on demand from the user.

Now that we have selected a bus, it is possible to do detailed throughput versus latency studies. The burst mode we can use to send ATM packets will offer high throughput for the ATM packets at the cost of higher latency for normal bus transactions for others using the bus. Only with models that show a more detailed timing model of the bus can we make these studies. Optimally we would like a cycle accurate detail, but we at least need to be statistically correct (what might be called a cycle approximate model).

4. The Cheetah Simulator

We have written a simulator, named **Cheetah**, that is designed to support interface-based design. Cheetah was developed to explore two areas: simulation speed and modeling style. Simulation speed comes from abstracting the interface, in particular avoiding detailed simulation of the interface internals. Modeling style exploration included investigations into how to simulate an interface abstractly and configure different interface implementations into the simulator easily.

The Cheetah simulator is an event driven simulator. Events are used to trigger actions in modules and interfaces. Full objects (instances of a Java class, in this case) are passed from module to module through abstract interfaces. We can substitute different interface implementations without changing the module code. The ATM example shown in Figure 3 and described in the next section has three different interface implementations that satisfy the same abstract interface.

Speed of simulation in Cheetah is obtained by avoiding unnecessary events and computations within the interfaces. During simulation, the interface simply passes tokens through, delaying them by a quickly computed delay before passing them onto the receiver and before letting the sender know the transaction has finished. No detailed cycle accurate simulation is done although the delay represents the cycle accurate time.

Synchronization, as for arbitrating access to a bus, is performed using a typical DE resource object, which knows how to allocate a shared resource to multiple requestors. No events are posted to the main event queue for handling synchronization, instead a local algorithm awakens the waiting requestor.

By abstracting synchronization and avoiding unnecessary events and computation, we can get performance similar to that of other token passing, queuing level performance analysis simulators.

For analysis purposes, designers would like to be able to expand this token level simulation into its constituent parts so that they can analyze resource usage, see opportunities for optimization, and understand how a bus protocol is working. As with most simulators, Cheetah allows probes on the interfaces between modules. The probed data captures the transaction, its beginning and end, and any synchronization or parameterization data necessary to reconstruct the cycle accurate behavior of that interface for that transaction.

On demand from the user, the probe data can then be expanded by the interface into what appears to be more probed data of assignments to the lower level details within the interface. In effect, we are resimulating each probed transact ion in isolation to create new probed data for display.

The interface, then, consists of a type specification (what type of data is carried by a single transaction across this interface), a delay calculation that is cycle accurate for the given transaction, and a procedure to expand or resimulate a previously handled transaction.

Interfaces can be hierarchical, allowing the parent interface to break a big transaction up into a series of smaller transaction on the children interfaces.

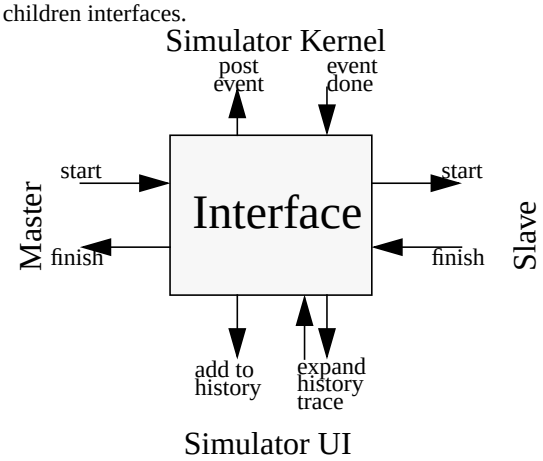


Figure 6: Interface interactions

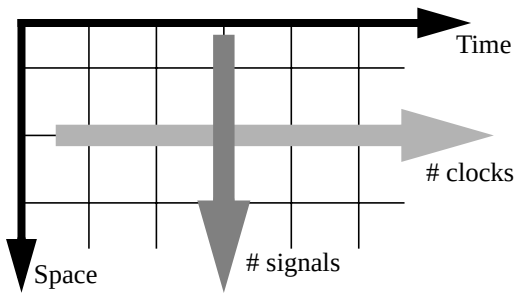


Figure 7: Performance gains from time and space

Each interface can be thought of as serving several purposes as shown in Figure 6.

The master starts a transaction by creating the transaction object and passing it to the interface. The interface then performs its abstract implementation, which for the simplest cases is to just use the global event queue to delay for a computed amount of time. The slave is notified that a new transaction has started. The interface may start building a history trace object using an optionally provided probe. When the abstract implementation is done, both the master and slave are notified that the transaction is finished and the optional trace object is added to the simulation history.

Later on, the user interface can come back to the interface and ask that a given trace be expanded into more detail. In the current implementation of Cheetah, this procedure will then create new history trace objects that show how the original transaction would have simulated if it was being done in detail.

The Cheetah simulator gains performance from abstraction in both time and space as shown in Figure 7. The time abstraction avoids multiple assignments to a single signal because we only deal with the transaction object. Similarly space abstraction avoids assignments to multiple signals. In the case of a bus, the space expansion includes the address and datalines, as well as many handshaking, request, acknowledge, and state signals. The best simulator performance is at the most abstract level, and was approximately 25x the slowest simulation we saw during the ATM methodology experiment.

Cheetah is in an experimental condition, so it is difficult to get an accurate performance comparison against an RTL level simulator. To get some sort of relative performance comparison we use events as our metric. Many designs are event queue limited, which means that their performance (in clocks simulated per second of wallclock time) is proportional to the number of events required to simulate a clock cycle. Our bus model avoids using events to simulate the detailed cycle level behavior, so each bus transaction only requires one event. An HDL simulator would require at least one event per bus signal transaction. Our bus has a minimum 7 signal assignments per bus transaction.

Based on this relative performance argument, our most accurate simulation would be about 7x faster than an equivalent RTL simulation. Our most abstract simulation would then be approximately 175x the performance of a cycle accurate RTL.

5. Summary

We have proposed a new methodology for system level design that separates communication from behavior. This interface-based design methodology seems to provide numerous advantages in design modeling and exploration, synthesis, and verification.

A simulator (Cheetah) was built to explore the performance and modeling implications of this methodology and was found to given cycle accurate simulation with substantially fewer events posted to an event queue than would be required in an HDL simulator. Full cycle accurate detail can be reconstructed post-simulation.

By adopting this methodology, we expect to see improvements in three major areas: modeling and design exploration, synthesis, and verification.

5.1 Modeling and Design Exploration

Modeling is improved in two important ways:

- better design reuse
- easier exploration of the design space

By separating communication from behavior, we have orthogonalized two extremely important design considerations. This separation makes it possible to mix and match communication techniques with behaviors. We can model some behavior and then snap on different bus interfaces. Alternatively, we can create a communication architecture and plug in different IP.

Design exploration is enhanced by providing a new place to configure the design. Using VHDL configurations as the archetype, we can switch in different communication architectures without modifying the original design.

5.2 Synthesis

By separating communication and behavior, new opportunities for synthesis are also created:

- improving the productivity of logic synthesis
- solving a major stumbling block for high level synthesis.

Despite the use of synthesis for a decade, design reuse is still cited as the most important desired productivity step. By incrementally inserting the communication logic into the behavior, blocks and communication schemes can be more easily reused.

Coelho [13] has shown that having an abstraction of the communication between modules can provide important *don't care* information that leads to better synthesis results.

High level synthesis suffers from a lack of composition methodology. Each block can be synthesized using high level synthesis, but no methodology exists to easily compose separately synthesized modules together. By imposing a communication mechanism on a behavior, the designer is also imposing a set of high level constraints on the two modules, which is suitable for guiding the high level synthesis of each for smooth composition.

5.2.1 Verification

Perhaps the most important impact could be felt in the verification area. Important improvements could include

- verification performance
- enabling formal verification
- testbench generation and reuse
- measurement of functional coverage
- block-based design verification methodology

Higher level abstractions provide better performance, in general, but at the cost of accuracy or visibility into the fine detail. If we separate communication from behavior, we can avoid simulating the communication once it has been verified and instead abstract the communication into a minimal set of delays. On demand, we reconstruct the fine detail if desired by the user. A simulator that relies on these techniques is briefly described in Section 4.

Formal verification is an extremely powerful technique that requires abstract descriptions coupled with a formal description of

the environment around the design. By keeping the modules behavioral and separate from the communication we can help keep them simpler and more likely to be amenable to formal techniques. The formal specification of the communication protocols can provide the necessary environment description that can help isolate a block, further simplifying the design to be verified. In addition, formal verification can be used to verify that an interface refinement continues to uphold the original abstract properties of the original communication (each model of computation will have different properties that need to be checked). Balarin [16] has some initial work in this area for CFSMs.

Testbenches are an unsung but extremely important design task. Most designers will admit to the testbench being at least as much work as the design. A formal, perhaps declarative, description of the communication protocol being used on a block could be used to automatically generate test fixtures that stimulate and check a design. Some tools like this, from companies such as InSpec, already exist as a separate add-on to an HDL design description. By putting the communication descriptions right into the design language, test generation and checking can be a natural fallout of the design process. Further, by creating the test generation at a high level and keeping it separate from the communication mechanism, any manual work creating tests can be more easily reused from design to design.

Code coverage [14] has emerged as a technique to answer the difficult question: Have I simulated enough yet? The known techniques can be used to analyze coverage of behavior. With a formal protocol description of a communication mechanism, a new kind of coverage analysis is possible, checking to see that all the types and temporal combinations of transactions have happened.

Designing complex systems by reusing large building blocks is partly difficult because of verification. By separating communication and behavior it should be possible for the author to verify the IP block against an abstract interface definition. The IP user could then use a high level description of the block, checking for interactions between the blocks and using functional coverage to make sure that all interface interactions had been investigated, but not needing to dive deeply into the block to verify its functionality. Different blocks can be tried out, with the correct interfaces synthesized as needed between them. The separation of communication and behavior can help segment an unsolvable verification problem into a collection of more manageable ones.

5.2.2 Future Work

Future work will involve several types of research:

- Theoretical: What are the types of properties in different models of computation and how might they be preserved during refinement?
- Language: What kind of description language would embody this methodology most effectively? How can an interface be described declaratively so a single description can be used to generate, recognize, and check its protocol?
- Formal verification: Can we formally prove refinement of a communication style?
- Simulation: With more realistic examples, what performance improvement will we get?
- Modeling: How can we mix together large complex IP without knowing how they are built inside?
- Refinement: How do we refine a collection of interfaces onto a communication architecture that involves shared resources, addressing, arbitration, etc.?
- Synthesis: How can an interface description be used to set constraints on the synthesis of a module? How can we minimally incorporate the interface behavior into a module?

Acknowledgments

This work has evolved partly because of many valuable conversations with Dan Yoder, Chris Ditzen, A. Richard Newton, Patrick Scaglia, Luciano Lavagno, Cary Ussery, Misha Burich, Rick McGeer, Felice Balarin, Ken McMillan, Christopher Hoover, and Wendell Baker.

References

[1] Virtual Socket Interface (VSI) website: http://www.vsi.org

[2] A. Sangiovanni-Vincentelli, P.C. McGeer, A. Saldanha, "Verification of Electronic Systems," *Proc. of 33rd IEEE/ ACM Design Automation Conference*, Las Vegas, 1996.

[3] G. Borriello "A New Interface Specification Methodology and its Application to Transducer Synthesis," University of California Technical Report, UCB/CSD 88/430, 1988.

[4] C. A. Wood, P. D. Gray, A. C. Kilgour, "Experience with Chisl, a Configurable Hierarchical Interface Specification Language," Computer Graphics Forum, vol. 7, no. 2, pp. 117-127, 1988.

[5] A. Seawright, F. Brewer, "Clairvoyant: A Synthesis System for Production-Based Specification," IEEE Trans. on VLSI Systems, vol. 2, pp 172-185, June 1994.

[6] L. Lavagno, A. Sangiovanni-Vincentelli, H. Hsieh, "Embedded System Co-Design," *Hardware/Software Co-Design*, pp. 213-242, Kluwer Academic Publishers, 1996.

[7] J. Oberg, A. Kumar, A. Hemani, "Grammar-based Hardware Synthesis of Data Communication Protocols," presented at the International Symposium on System Synthesis, La Jolla, November 1996.

[8] S. Vercauteren, B. Lin, H. De Man, "Constructing Application-Specific Heterogeneous Embedded Architectures from Custom HW/SW Applications," *Proc. of 33rd IEEE/ ACM Design Automation Conference*, Las Vegas, 1996.

[9] E. A. Lee, D. G. Messerschmitt, "Synchronous Data Flow," IEEE Proceedings, September, 1987.

[10] P. A. Fishwick, *Simulation Model Design and Execution*, Prentice Hall, New Jersey, 1995.

[11] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, "Ptolemy: A Framework for Simulation nd Prototyping Heterogeneous Systems," Int. Journal of Computer Simulation, special issue on "Simulation Software Development," vol. 4, pp. 155-182, April, 1994.

[12] M. Chiodo, P.Giusto, H. Hsieh, A. Jurecksa, L. Lavagno, and A. Sangiovanni-Vincentelli, "A Formal Methodology for Hardware/Software Codesign of Embedded Systems," IEEE Micro, August 1994.

[13] C. Coelho Jr, "Analysis and Synthesis of Concurrent Digital Systems Using Control-Flow Expressions," Stanford Technical Report, CSL-TR-96-690, 1996.

[14] A. Hosseini, D. Mavroidis, P. Konas, "Code Generation and Analysis for the Functional Verification of Microprocessors," *Proc. of 33rd IEEE/ACM Design Automation Conference*, Las Vegas, 1996.

[15] Open Microprocessor Initiative web site: http://www.omimo.be/index.htm

[16] F. Balarin, H. Hsieh, A. Jureckska, L. Lavagno, A. Sangiovanni-Vincentelli, "Formal Verification of Embedded Systems based on CFSM Networks," *Proc. of 33rd IEEE/ ACM Design Automation Conference*, Las Vegas, 1996.