Verifying a Virtual Component Interface-based PCI Bus Wrapper with FormalCheck

Annette Bunker and Ganesh Gopalakrishnan

UUCS-01-006

School of Computing
University of Utah
Salt Lake City, UT 84112 USA

June 14, 2001

Abstract

The Virtual Sockets Interface Alliance (VSIA) recently released the Virtual Component Interface (VCI) Standard. This paper reports recent experiences in formally verifying a few properties of a VCI-compliant PCI 2.1 bus wrapper model in the formal verification tool, FormalCheck. Though we chose to only verify three liveness properties and three safety properties, the verification highlighted issues buried deeply within the model, quickly. We found eight issues in our model in four person-weeks of verification effort.

1 Introduction

The Virtual Sockets Interface Alliance (VSIA) recently released the Virtual Component Interface (VCI) Standard [Gro00]. The VCI is designed to allow compliant virtual components to communicate with one another easily, even if developed in different design organizations. Communication may take place via the VCI directly, or over an integrator-selected bus, via VCI-compliant bus wrappers, such as the one we modeled and verified in this case study.

[1]This work was supported by National Science Foundation Grants CCR-9987516 and CCR-0081406

As a part of our joint work with the VSIA, we created a wrapper that takes VCI transactions as input and generate PCI 2.1 [Gro95] transactions as output, await their results and retranslate them into the VCI protocol for consumption by a hypothetical VCI initiator component. The FormalCheck[2] formal verification tool was then used to verify six properties about the wrapper.

We discuss the key insights gained in this case study in Sections 4 and 5, but summarize them here for reader convenience:

- We present a case study in verifying an implementation of the newly-released Virtual Component Interface Standard.
- We show that even loosely-defined formal verification efforts can yield valuable design results in a short time.
- We quantify the overall verification effort.

The rest of this section introduces the VCI and PCI protocols and the FormalCheck verification tool. We examine related work in Section 2. We describe our PCI bus wrapper model in Section 3 and our verification methods and results in Section 4. Finally, we draw conclusions and outline plans for future work in Section 5.

1.1 The Virtual Component Interface

The Virtual Component Interface Standard specifies a family of point-to-point communication protocols aimed at facilitating communication between virtual components, possibly those created by separate design organizations. Three protocols currently belong to the family: the Peripheral Virtual Component Interface (PVCI), the Basic Virtual Component Interface (BVCI) and the Advanced Virtual Component Interface (AVCI).

The AVCI is a superset of the BVCI which is a superset of the PVCI. The PVCI is not a split-transaction protocol; request and response data transfers occur during a single control handshake. The BVCI, on the other hand, is a split transaction protocol. The only constraint placed on responses by the standard is that they arrive at the initiator in the same order in which the initiator generated matching requests. The AVCI is also a split-transaction

[2] FormalCheck is a trademark of Lucent Technologies and Cadence Design Systems.

protocol. AVCI requests may be tagged to allow request threads to be interleaved and transactions reordered.

All VCI standards require separate address and data busses. They allow for multiple addressing modes enabling integrators to take advantage of memory access optimizations and bus optimizations.

1.2 The PCI standard

The Peripheral Component Interconnect (PCI) Standard defines a chip-level interface for connecting I/O devices to the system's processor/cache/memory subsystem via an acyclic network of busses and bus bridges. The PCI standard is a split-transaction protocol based on two types of transactions, posted and delayed. A posted transaction completes on the originating bus before it completes on the destination bus. Delayed transactions, on the other hand, complete remotely before completing locally. They do so by leaving markers in each bridge along the path from source to destination which must be matched at each step by the transaction acknowledge. Only when the marker at the source is matched is a delayed transaction completed.

In an attempt to obey the producer/consumer property and remain deadlock free, PCI allows certain requests and responses to be reordered while in flight in the network. Address and data share the same bus.

1.3 FormalCheck

The FormalCheck model checker, marketed by Cadence Design Systems, Inc., is based on language inclusion test for ω-automata [Bel98]. FormalCheck compiles the Verilog or VHDL design to build its system model. It provides the user with a series of templates for writing constraints (environmental assumptions) and properties. The user may choose from several verification styles ranging from bug-hunt to rigorous state-space exploration. A variety of model reduction techniques can be employed, including the default one-step reduction, iterative reduction, clock extraction, seeded and non-seeded reductions.

If the verification property holds on the system under investigation, the tool supplies the user with a message indicating this, depending on the verification style employed. If the model violates the property, FormalCheck supplies the user with a waveform describing the violation and the events leading to it. Violations may stem from either improperly defined properties/constraints or from issues in the RTL model.

2 Related work

Work related to our case study generally resides in two categories, that involving specifying and verifying standards at the RTL abstraction level and that involves the use of the FormalCheck model checker. We treat each category of work, here.

Most of the work previously done in the area of standards specification and verification involves the PCI standard. Shimizu, et al [SDH00], specify the PCI standard using monitors written in HDLs. Monitor-based specifications can be checked for consistency and for receptivity, however they lack conceptual cohesion and we expect them to be difficult to understand. [CCLW99] and [Wan99] demonstrate the verification of the PCI specification at roughly the same level of abstraction as the one reported here.

In [XCS$^+$99], Xu, et al, describe the verification of a proprietary frame mux/demux chip by Nortel Corporation using FormalCheck. The authors present the properties verified as well as their experiences using the model reduction features of FormalCheck. In [XCSH97], the verification of an ATM fabric switch using various theorem provers as well as model-checkers is described. The paper presents techniques to handle large queues as well as addresses the verification of latency properties specific to this chip in FormalCheck.

To the best of our knowledge, our case study is the first involving the Virtual Components Interface standard, a wrapper component of the kind that will typically be used with this standard and FormalCheck, the most widely-used commercial formal verification tools.

3 PCI bus wrapper model

The PCI bus wrapper[3] consists of eight fifos and six state machines, as shown in 3. Each fifo is responsible for storing one element of the request or response, while earlier transactions proceed on the buses. The address, byte enables (BE), command (CMD[1:0]), write data (WDATA[31:0]) and end-of-packet (EOP) fifos contain the input transaction information from the VCI, as indicated by the name of each fifo. The response error (RERR), read data

[3] The complete Verilog model may be viewed at http://www.cs.utah.edu/~abunker/vci/vlog.

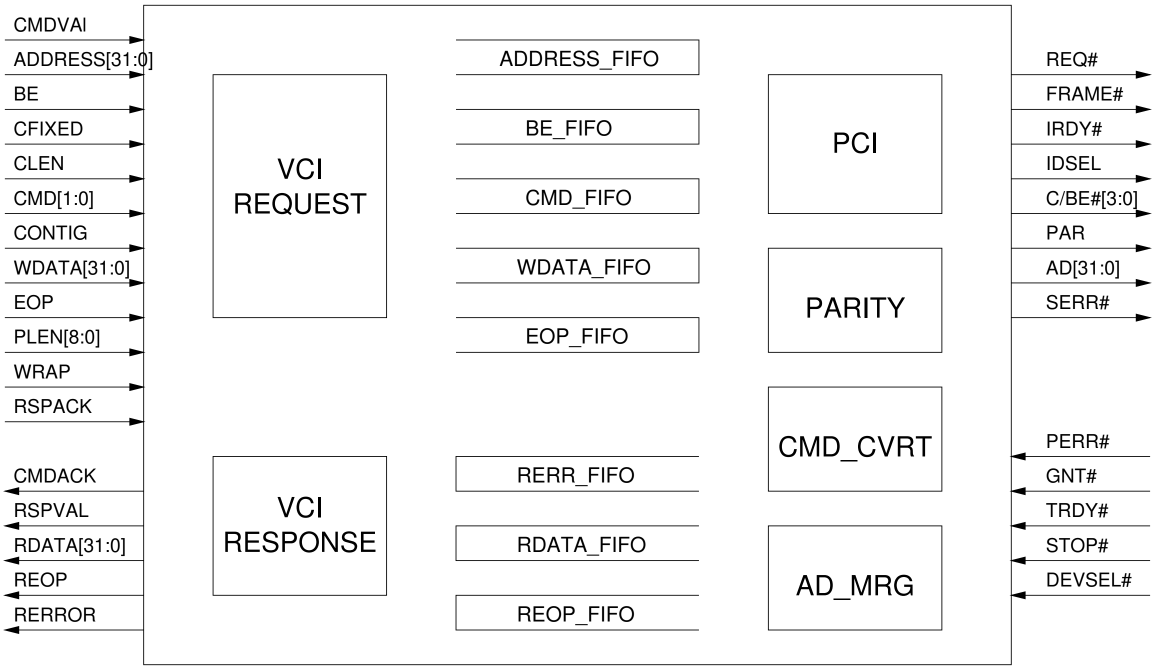

Figure 1: Structure of the PCI Bus Wrapper

(RDATA[31:0]) and response end-of-packet (REOP) fifos return response information to the VCI. Note that all information is stored in the wrapper in its VCI-compliant format. This design decision localizes the complexity in the PCI state machine.

Of the six state machines, three make up the actual bus interfaces: two for the VCI and one for the PCI interface. The VCI Request machine reads input transactions from the VCI and inserts them into the appropriate fifos. Likewise, the VCI Response machine reads response transactions from the appropriate fifos and drives them onto the VCI. The PCI machine handles all timing relative to transacting on the PCI bus, but it is aided in certain aspects by the remaining three state machines. The Parity machine calculates even parity to be output on the PCI bus, as VCI does not use a notion of parity checking. The CMD_CVT (command convert) machine converts the VCI transaction command and byte enables into their PCI-compliant format while multiplexing them onto the the PCI command/byte enable bus (C/BE#[3:0]). Similarly, the AD_MRG (address/data merge) machine multiplexes the address and write data onto the PCI AD[31:0] bus, though it does not do any data reformatting.

If the PCI network is able to service a request without errors, the PCI state machine reformats and loads the response data into the response queues immediately. However, if the PCI network returns a retry, the PCI state machine leaves the current transaction at the head of the fifos and immediately attempts the transaction on the PCI bus again, essentially busy-waiting on the retried PCI transaction. We made this decision to simplify the wrapper's design. The only PCI errors that must be supported by this style of wrapper are target aborts, which mean that the addressed device is unable to service the request due to a fatal error. In this case, we remove the request from the fifos and return an error to the VCI initiator.

4 Verifying with FormalCheck

This section discusses the process we used to verify the wrapper model in FormalCheck. Though we present our process in a linear fashion for clarity, here, we used an iterative process in which, model reductions, constraint formulation and bug tracking interacted and fed back to one another. Subsection 4.1 explains the model reductions necessary to make the model model-checkable. Subsection 4.2 enumerates the environmental constraints necessary to complete the model checking and Subsection 4.3 enumerates the properties we verified. We discuss the issues found with the design in this case study in Subsection 4.4.

4.1 Model reductions

The primary reductions that had to be made to the model to allow FormalCheck to handle the wrapper design were to reduce the counter sizes and the data bus widths. Since we were aware that we would likely make these reductions at design time, the fifos and state machines that know about the bus sizes and counter widths were all designed using Verilog parameters. Using this Verilog feature allowed us to change the size parameter at the highest level of module instantiation and the Verilog compiler managed change propagation to all necessary modules. (Later versions of FormalCheck make these reductions automatically, without the aid of parameterized designs.)

The original design contained 32-bit address and data busses as one would expect to use when interfacing with a standard PCI network. These busses were all reduced to two bits to allow for a nontrivial number of address and data element possibilities.

The original design also specified 9-bit fifo counters (for the head and tail pointers). Nine bits allows enough slots to store an entire VCI packet in the fifos at once. Though our model does all processing on the cell-level, we chose this design for flexibility. We wished to make this design easy to modify to allow full packet-based processing, if we choose to do so, later. For purposes of the verification, however, these counters were also reduced to two bits, allowing for only 4 slots in our fifos.

4.2 Constraints

It is our experience that precisely defining environmental constraints is the most difficult and time-consuming portion of a FormalCheck-based verification. We began our verification with a minimal set of constraints and added a new constraint only when it was necessary in order to avoid a false negative. We chose this approach for three reasons: to keep the verification as simple as possible, to allow our results to be as strong as possible and to mimic the industrial procedures of which we are aware.

Besides the usual clock and reset constraints, we created eight constraints during the verification project. The final set of constraints we used and their English semantics are enumerated below.

1. `Assume Never: (xlator.reset_l == 0) && (xlator.cmdval == 1)`

   CMDVAL may never be asserted while the wrapper is in reset.

2. ```
   After: (xlator.cmdval == 1) && (xlator.clk == rising)
   Assume Always: (xlator.cmdval == 1)
   Unless: (xlator.cmdval == 1) && (xlator.cmdack == 1) &&
     (xlator.clk == rising)
   ```

   CMDVAL must remain asserted until it is properly acknowledged.

3. ```
   After: (xlator.req_l == 0) && (xlator.clk == rising) &&
     (xlator.reset_l == 1)
   Assume Eventually: (xlator.gnt_l == 0) && (xlator.clk == rising)
   ```

   The PCI arbiter will eventually grant ownership of the PCI bus to the bus wrapper.

4. ```
   After: (@retry) && ((xlator.ul.curr_state == 4) ||
     (xlator.ul.curr_state == 3)) && (xlator.clk == rising)
   Assume Eventually: (xlator.stop_l == 1) &&
     (xlator.trdy_l == 0) && (xlator.ul.curr_state == 4) ||
     (xlator.ul.curr_state == 3)) && (xlator.clk == rising)
   ```

   The PCI environment will not give the wrapper a PCI retry response in all future states when the wrapper samples the PCI response.

5. ```
   After: (xlator.rspval == 1) && (xlator.clk == rising)
   Assume Eventually (xlator.rspack == 1) && (xlator.clk == rising)
   ```

   The VCI environment acknowledges every response, eventually.

6. ```
After: (xlator.reset_l == 1) && (xlator.frame_l == 0)
Assume Eventually: (xlator.trdy_l == 0) && (xlator.clk == rising)
```

   The PCI environment will eventually take ownership of every transaction the wrapper drives onto the PCI bus.

7. ```
Assume Never: (xlator.rspack == 1) && (xlator.clk == rising)
Unless: (xlator.rspval == 1) && (xlator.clk == rising)
```

   The VCI environment does not generate an acknowledge to a response unless the response has been driven.

8. ```
After (xlator.req_l == 0) && (xlator.clk == rising)
Assume Always: (xlator.req_l == 0)
Unless: (xlator.req_l == 0) && (xlator.gnt_l == 0) &&
  (xlator.clk == rising)
```

   The PCI request remains asserted until the PCI environment grants the wrapper ownership of the bus.

Constraint 4 deserves more comment. The `@retry` term in the constraint represents a FormalCheck macro. In this case, the macro expands to `(xlator.stop_l == 0) && (xlator.devsel_l == 0) && (xlator.trdy_l == 1)`, the PCI signaling that indicates a retry response. Ironically, the complexity of this constraint is a direct result of our goal to keep the verification simple. We chose to model the PCI environment with as few constraints as possible. As a result, the PCI environment is allowed to drive responses at anytime, whether it is actually a moment at which the PCI machine samples the PCI inputs or not. Hence, merely stating that the PCI environment cannot give a retry forever is not strong enough. Instead, we must stipulate that retries may not be returned at all future states when the PCI transaction is sampled.

## 4.3 Properties

We verified six properties of the PCI bus wrapper. Without having done any formal reasoning, we believe that three of these properties, together, imply that the wrapper is live, given the liveness of both the PCI environment and the VCI environment. The other three properties, we believe, imply that the wrapper does not generate any spurious PCI transactions or VCI responses. The formulation of the properties in FormalCheck follows.
```

Liveness properties:

1. `After: (xlator.reset_L == 1) && (xlator.cmdval == 1) &&`
   `(xlator.clk == rising)`
   `Eventually: (xlator.cmdack == 1) && (xlator.clk == rising)`

   CMDACK must be raised when the clock ticks, sometime after CMDVAL is raised when the clock ticks and the wrapper is not in reset.

2. `After: (xlator.reset_L == 1) && (xlator.cmdval == 1) &&`
   `(xlator.cmdack == 1) && (xlator.clk == rising)`
   `Eventually: (xlator.frame_l == 0) && (xlator.clk == rising)`

   FRAME# is asserted when the clock ticks some time after a VCI transaction has been asserted and acknowledged when the clock ticks and the module is not in reset.

3. `After: (xlator.reset_L == 1) && (xlator.cmdval == 1) &&`
   `(xlator.cmd == 1) && (xlator.clk == rising)`
   `Eventually: (xlator.rspval == 0) && (xlator.clk == rising)`

   A VCI response must be made sometime after the VCI request has been properly asserted and acknowledged.

Safety properties:

1. `Never: (xlator.cmdack == 1) && (xlator.clk == rising)`
   `Unless: (xlator.cmdval == 1) && (xlator.clk == rising)`

   A VCI transaction is never acknowledged unless it is first requested.

2. `Never: (xlator.frame_l == 0) && (xlator.clk == rising)`
   `Unless: (xlator.cmdval == 1) && (xlator.cmdack == 1) &&`
   `(xlator.clk == rising)`

   A PCI transaction is never generated unless a VCI transaction was first input.

3. `Never: (xlator.rspval == 1) && (xlator.clk == rising)`
   `Unless: (xlator.cmdval == 1) && (xlator.cmdack == 1) &&`
   `(xlator.clk == rising)`

   A VCI response is never generated unless a VCI request is first input.

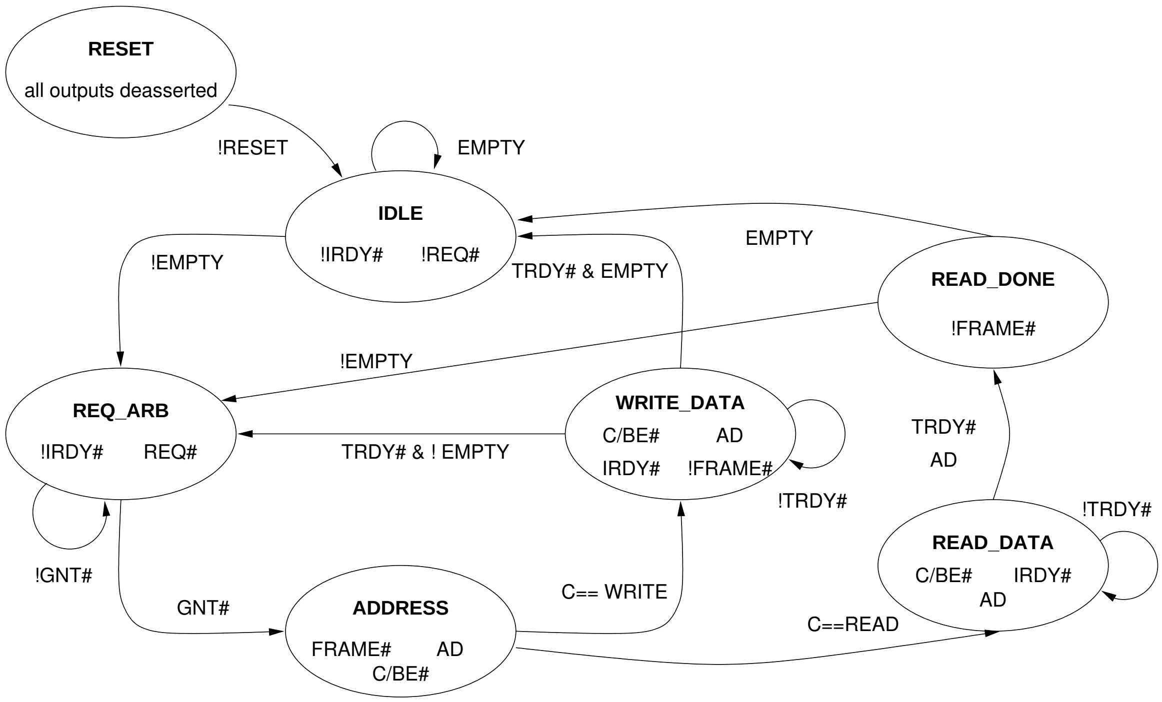

Figure 2: Original PCI State Machine Design

4.4 Issues

We found eight issues with our model during the verification portion of the case study. We found three issues as a direct result of model checking activities, while the other five were identified as a result of deeper examination of the code inspired by feedback obtained from the model-checker.

Of the eight issues, six resided in the PCI state machine. One of these six issues corrected difficult-to-understand coding, but was not a functional correctness issue. Another of the six was an issue related to the fix for an earlier-identified issue.

Two of the six PCI state machine issues were closely related. When the model checker found the first, we knew that the second must exist, but we chose not to correct it to see if the model checker would find it without our assistance, which it did. We examine these two issues, which we call *last transaction target abort* and *last write* in more depth, here.

4.4.1. Last transaction target abort. Figure 2 shows the original design of the PCI state machine. The problem first identified by the model checker in this design results when the

PCI network returns a target abort during the last transaction in the fifo. Target abort results occur when the target device is unable to service the request due to invalid addressing, invalid command, or other signaling or service problem. When this occurs, we should drop the current VCI request from the request fifos and immediately generate an error response for the VCI initiator on the response fifos.

The wrapper learns of a target abort result while in the READ_DATA or WRITE_DATA state. As shown in the state diagram, we immediately check the empty status of the queues and transition directly to bus arbitration, if another transaction awaits service. Unfortunately, this does not give the fifos enough time to discard the aborted transaction and properly update the empty status bit. When a target abort occurs on the last transaction, the state machine transitions to REQ_ARB and begins a new transaction before the fifos signal that they are empty. This causes a garbage transaction to be generated on the PCI and the fifo read and write pointers to behave improperly.

What is surprising about this issue is that we found it before we began verification of the safety properties. It was, in fact, Liveness 2 that identified this issue. Because the read pointer in the fifo would pass the write pointer, the fifo signaled that it was full, denying all future incoming VCI operations. This violated the property, because CMD_VAL could be asserted forever without FRAME# ever becoming asserted.

The solution to this problem, was to remove the optimizations between READ_DONE and REQ_ARB and insert a wait state through which the state machine must pass to give the fifos enough time to recover from the ejection of the transaction and update the empty status bit if necessary.

4.4.2. Last write. A second look at Figure 2 indicates that a problem similar to *last transaction target abort* would exist for successful write transactions. This was the second issue of this nature we identified. Though we knew that successful writes must be directed through the newly created recovery state, we were interested to see if the model checker would find this issue as well.

While the model checker did highlight this issue, as well, we were surprised that it did not do so until we attempted to verify Safety Property 2. The reason for this is unclear. The effects of the *last write* design flaw on the fifo pointers should be the same as those resulting from the *last transaction target abort* design flaw. We expected the model checker to show us this issue before it allowed Liveness Property 2 to pass.

Property	State Vars	Time	Memory
Liveness 1	67	1:22	202
Liveness 2	70	7:24	238
Liveness 3	69	22:35	303
Safety 1	44	1:26	205
Safety 2	45	1:23	205
Safety 3	64	1:25	205

Table 1: FormalCheck statistics summary

5 Conclusions and future work

The verification experiences reported in this paper took about four person-weeks of continuous effort. The verifier designed the module and had previous FormalCheck experience, so no learning time is included in this esimate. Table 1 summarizes the verification statistics reported by FormalCheck on the final, passing runs of each property. Times are reported in minutes:seconds and memory is reported in megabytes.

The main conclusion we draw from the experiences described here is that even a cursory formal verification can yield valuable results. In this case study, we chose to verify properties that we believe lead to a conclusion that the bus wrapper is live, but we have not done any formal reasoning showing this is actually the case. We also verified properties that we believe imply that the wrapper will not generate any spurious PCI transactions or VCI responses. Again, no formal reasoning supports this belief. However, despite this lack of rigor, our verification provided us with useful results which directly affected the wrapper design. We found eight issues through model checking and resulting reviews.

However, we also believe that in order to conclude that an implementation conforms to a standard specification, the verification effort must establish a wide variety of facts relating to the implementation. We expect that a systematic approach to establishing these facts will aid the verification effort. We plan to investigate modeling languages and approaches that will facilitate formal verification of standards compliance in the future.

Specifically, our future plans for this work include:

- Fully categorizing the properties necessary for standards-compliance verification.
- Formalizing the VCI specification in an appropriate modeling language. [BG01]

References

[Bel98] Bell Labs design Automation and Lucent Technologies. *FormalCheck User's Guide*, v2.1 edition, 1998.

[BG01] Annette Bunker and Ganesh Gopalakrishnan. Formal specification of the virtual component interface standard in the unified modeling language. Technical Report UUCS-01-007, University of Utah, June 2001.

[CCLW99] Pankaj Chauhan, Edmund M. Clarke, Yuan Lu, and Dong Wang. Verifying ip-core based system-on-chip designs. In *IEEE International ASIC/SOC Conference*, pages 27–31, September 1999.

[Gro95] PCI Special Interest Group. *PCI Local Bus Specification*. PCI Special Interest Group, 1995.

[Gro00] OCB Design Working Group. *VSI Alliance Virtual Component Interface Standard*. Virtual Sockets Interface Alliance, November 2000.

[SDH00] Kanna Shimizu, David L. Dill, and Alan J. Hu. Monitor-based formal specification of PCI. In Warren A. Hunt Jr. and Steven D. Johnson, editors, *Formal Methods in Computer-Aided Design*, volume 1954 of *Lecture Notes in Computer Science*, pages 335–352. Springer-Verlag, November 2000.

[Wan99] Dong Wang. Formal verification of the PCI local bus: A step towards ip core based system-on-chip design verification. Master's thesis, Carnegie Mellon University, May 1999.

[XCS+99] Y. Xu, E. Cerny, A. Silburt, A. Coady, Y. Liu, and P. Pownall. Practical application of formal verification techniques on a frame mux/demux chip from nortel semiconductors. In Laurence Pierre and Thomas Kropf, editors, *Proceedings of the 10th IFIP WG 10.5 Advanced Research Working Conference, CHARME '99*, volume 1703 of *Lecture Notes in Computer Science*, pages 110–124. Springer Verlag, September 1999.

[XCSH97] Ying Xu, Eduard Cerny, Allan Silburt, and Roger B. Hughes. Property verification using theorem proving and model checking. www.isdmag.com, November 1997.