Design for Verification of a PCI Bus in SystemC

Karim Oumalou, Ali Habibi, and Sofiène Tahar
Department of Electrical and Computer Engineering
Concordia University
1455 de Maisonneuve, West
Montreal, Québec, H3G 1M8, Canada
Email: {rimka,habibi,tahar}@ece.concordia.ca

Abstract—**In this paper, we present an approach to design and verify SystemC intellectual properties (IPs). We considered as illustrative case a PCI bus modeled as a monitor module that can be interfaced to existent SystemC IPs. We defined three design steps where we first model the bus in UML; then, design it completely with Abstract State Machines (ASM); and, finally, translate the ASM code to SystemC. We considered an intermediate ASM representation in order to extract the finite state machine of the bus that can be used to for model checking of PSL properties. The final SystemC monitor block can be seen as a stand-alone PCI IP as well as a verification module to validate other SystemC PCI compatible devices. Besides, our design offers a flexible and easy to configure IP that supports a large number of master/slave devices.**

I. Introduction

SystemC [9] is a relatively new system level language for embedded system design and verification. It is expected to make a strong effect in the area of architecture, co-design and integration of hardware and software [8]. The SystemC library of classes and simulation kernel extend C++ to enable the modeling of System-on-a-Chip (SoC). The extensions include support for concurrent behavior, a notion of time sequential operations, data types for describing hardware, structure hierarchy and simulation.

In order for SystemC to model complex yet real SoC, it must be equipped by a library of Intellectual Properties (IP) including in particular bus structures, which will be used to to interconnect devices such as processors, memories, etc. In this paper, we present a design methodology to implement a PCI (Peripheral Component Interconnect) [10] Local Bus in SystemC as a stand-alone monitor. The PCI bus is a high performance bus for interconnecting chips, expansion boards, and processor/memory subsystems. It was adopted as an industry standard administered by the PCI Special Interest Group (PCI SIG) [10].

The verification of SystemC designs is a serious bottleneck in the system design flow. Classical simulation does not guarantee the absence of errors. On the other hand, formal techniques, in particular model checking cannot handle neither the object-oriented (OO) nature of the library nor the complexity of its simulator. In order to overcome these two problems, we propose to use an intermediate level in the design flow, where we model the system in terms of Abstract State Machines (ASMs) [7]. At the ASM level, it is possible to check formally a set of the properties of the design that guarantee its correctness when translated to SystemC. The proposed approach is illustrated through the PCI bus.

An ASM model by definition encodes only those aspects of the system's structure that affect the behavior being modeled. ASM provides a variety of features that allow the description of the relevant state of a system in a very economical, high-level way. Each abstract state machine represents a particular view of the distinct operational steps that occur in the real system being modeled.

AsmL [7] is the language used to model systems in ASM. It is integrated with Microsoft's software development environment including Visual Studio, MS Word, and Component Object Model (COM), where it can be compiled and connected to C# or to the .NET framework [7].

As related work to ours, we cite the approach proposed by Bruschi *et* al. [2] to design a PCI bus in SystemC and the work of Kanna *et* al. [11] to implement a PCI bus as a Verilog monitor and to verify its properties using SMV [6]. In [2], the design was directly defined in a synthesible subset of SystemC, which made the bus structure look more like a Verilog module rather than a high level design. Besides, there was no verification of the functionalities of the bus which does not guarantee the correctness of the design. In [11], the bus was implemented in Verilog with all the properties embedded as part of the code. This makes its modification or upgrade a very complex task. Besides, the Verilog model they verified includes only 2 agents (one master and one slave), which does not allow the verification of the properties related to the bus arbitration.

The rest of the paper is organized as follows: Section II describes the PCI bus. Section III presents our approach to design and verify the PCI bus. Section IV presents the experimental results. Finally, Section V concludes the paper.

II. The PCI Bus

The PCI bus boasts a 32-bit data path, 33MHz clock speed and a maximum data transfer rate of 132MB/sec. A 64-bit specification exists for future PCI designs, which will double data transfer performance to 264MB/sec. In Figure 1, we show a generic structure of the PCI bus with a single master and a slave. We added also an external monitor module that will be used to track the signals at the input and output ports of the bus in order to validate the good functioning of the bus.

0-7803-8558-6/04/$20.00 ©2004 IEEE.

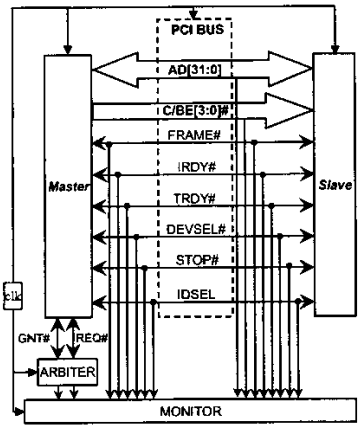

Fig. 1. PCI Bus Structure.

Each PCI master has a pair of arbitration lines that connect it directly to the PCI bus arbiter. When a master requires the use of the PCI bus, it asserts its device specific REQ# line to the arbiter. When the arbiter has determined that the requesting master should be granted control of the PCI bus, it asserts the GNT# (grant) line specific to the requesting master. In the PCI environment, bus arbitration can take place while another master is still in control of the bus.

In PCI terminology, data is transferred between an initiator, which is the bus master, and a target, which is the bus slave. The initiator, drives the C/BE[3:0]# signals (Figure 1) during the address phase to signal the type of transfer (memory read, memory write, I/O read, I/O write, etc.). During data phases, the C/BE[3:0]# signals serve as byte enable to indicate which data bytes are valid. Both the initiator and target may insert wait states into the data transfer by de-asserting the IRDY# and TRDY# signals. Valid data transfers occur on each clock edge in which both IRDY# and TRDY# are asserted. A target may terminate a bus transfer by asserting STOP#. When the initiator detects an active STOP# signal, it must terminate the current bus transfer and re-arbitrate for the bus before continuing. If STOP# is asserted without any data phases completing, the target has issued a retry. If STOP# is asserted after one or more data phases have successfully completed, the target has issued a disconnect.

III. Design for Verification Approach

Our approach is shown in Figure 2, where we start with an informal specification for the intended design, developed in UML. This step provides a better view of the design components and their interactions. We also specify the properties on the system under verification. Based on the UML model (class diagram, sequence diagram, etc.), we derive an ASM model, which will enable the verification of the design using formal verification tools (model checkers for our case). When the verification results show an error, we go back to the UML specification, update it and redo the verification at the ASM level. Finally, when the verification passes, we directly map our ASM model into a SystemC implementation.

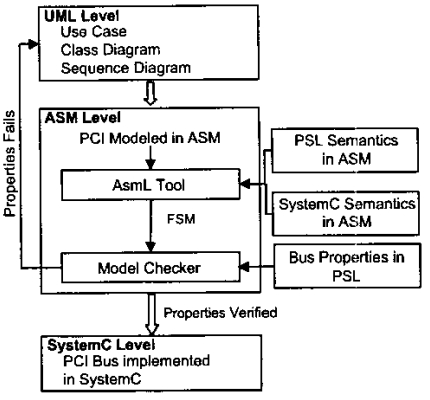

Fig. 2. PCI Bus Design and Verification Methodology.

A. UML Specification

In order to design the PCI bus in UML, we considered a structure based on four classes: bus, arbiter, slave and master. What is specific in our design approach is the definition of two classes to model a master and a slave. This way the bus can be used in two different modes:

- A stand-alone IP: the master and slave classes provide an interface to connect external devices to the bus.
- A testing monitor module: to validate other PCI IPs by checking the state of the signals at the inputs and outputs of the bus and comparing them to the expected values (as described in Figure 1).

B. Properties Specification

In order to model the bus properties we used and embedding of the Accellera Property Specification Language (PSL) [1] in ASM [5]. PSL was developed to address the lack of information about properties and design characteristics in RTL modeling. It provides means of specifying design properties using a concise syntax with clearly defined formal semantics. PSL permits specifying a large class of real design properties that range from simple to complex ones since it consists of four layers: Boolean, temporal, verification and modeling layers. It is intended to be used for functional specification to capture requirements regarding the overall behavior of a design in one hand, and as an input to verification tools using simulation or formal verification on the other hand.

In addition to the properties defined in [11], we considered several other more complex properties, which define a complete sequence of transactions over the bus. In what follows are presented three sample properties:

$Property\ P_1$:

$forall\ Master\ in\ \{Master0, \ldots, Master4\}$
$\quad if\ (!Master.REQ == true)\ then$
$\quad\quad eventually\ (!Master.GNT == true)$

meaning that if a master requests the bus ($!Master.REQ == true$) it will get access to it in the future ($!Master.GNT == true$), which guarantees that no master will use the bus indefinitely.

Property P_2 :

$forall\ Master\ in\ \{Master0, \ldots, Master4\}$
$forall\ Slave\ in\ \{Slave0, \ldots\ Slave4\}$
$if\ (!Master.GNT == true)\ and$
$(!Master.DEST == Slave.ID)$
$then\ eventually\ (!Bus.FRAME == true)\ and$
$(!Master.TRDY == true)\ and$
$(!Slave[ID].TRDY == true)\ and$
$(!Master.GNT == false)$

meaning that if a master is selected by the arbiter, then it will be able to get access to the bus by setting $!Bus.FRAME$. Thereafter, its destination slave will be activated by setting its $!Slave.TRDY$. Finally, the master will release the bus once $!Master.GNT$ is set to false.

Property P_3 :

$forall\ Master\ in\ \{Master0, \ldots, Master4\}$
$if\ (!Master.STOP == true)\ and$
$(!Master.GNT == true)\ then$
$eventually\{(!Bus.FRAME == false)\ and$
$forall\ Slave\ in\ \{Slave0, \ldots\ Slave4\}$
$(!Slave.TRDY == false)\ and$
$(Slave.IDSEL == false)\}$

meaning that if a master stops a request, then the bus and all the slaves will be released.

C. ASM Model

In order to model the bus structure in ASM, we used an existing embedding of SystemC in ASM [3]. The bus properties (in PSL) are added directly to the ASM model using an embedding of PSL in ASM [5]. We then compiled the ASM model, including both the design and the properties, using the AsmL tool and generate its Finite State Machine (FSM). This FSM is translated into the input language of the model checking tool, which will verify the correctness of the model. Similarly, the AsmL compiler can generate test scenarios in the form of .NET or C# models for verification by simulation.

The generation of the FSM from ASM is performed using the algorithm given in [4]. Unfortunately, this algorithm is not openly available as the AsmL tool is provided as a black–box. To solve this problem, we embed the state of every property (as Boolean) in every system's state. Therefore, once the FSM is generated, it will include, by construction, a Boolean state variable giving the state of the property. The last step in the verification process is to translate the FSM to a format supported by a model checker. Note that there is no restriction on the model checker as the final FSM is concrete and includes only Boolean variables to represent the state of the PSL properties.

D. Model Checking

As shown in Figure 2, the bus verification is performed using a model checking tool. In fact, we translate the FSM representation, output of the AsmL tool, into a code supported by the model checker (SMV [6] here). All the model checker is required to do is to make a state exploration of all the possible states of the system and to verify that the embedded properties are always true. In case an error is found, the model checker generates a counter-example that can be used to localize the error and to guide the needed changes on the UML model.

E. Translation to SystemC

The translation of the ASM code to SystemC is a critical path in our approach. In fact, all properties are being verified at the ASM level. Therefore, the SystemC code must be identical to the verified ASM code otherwise additional errors could be added to the final SystemC design. We were able to solve this problem thanks to a deep embedding of the SystemC library in ASM [3], where we defined a one to one mapping between the SystemC library components (modules, signals, etc.) and their embedding in ASM. For now, we are doing the direct translation manually, however, as a future work we are planning to build a tool to perform this task automatically.

IV. Experimental Results

In following, we describe our results on the verification of the PCI bus using the proposed methodology. All experiments [1] were performed on a 2.4 GHz Pentium IV and 512 MB of RAM (PC 2700).

A. FSM Generation

Figure 3, shows the FSM of a bus structure including a single master and two slaves. State S1 is used to initialize the parameters of the bus. States S2 and S17 illustrate the non-determinism of the machine as the master may send to any slave randomly. The FSM displays a symmetry property because the scenario of sending to one slave is exactly identical to sending to the other slave.

The CPU times required for the generation of the FSM for different numbers of masters and slaves are given in Table I. We note that the numbers of states and transitions increase exponentially as a function of the numbers of masters and slaves connected to the bus. This is due, for instance, to the FSM generation algorithm used internally by the AsmL tool [7].

B. Model Checking

We used the SMV model checker in order to verify the bus properties. We combined the three properties P_1, P_2 and P_3 in a single property, P, defined as: $P_1\ and\ P_2\ and\ P_3$. Table II shows the model checking time and the number of BDDs as function of the number of slaves and masters connected to the bus.

[1] The UML design, SystemC code, ASM code, AsmL configuration files, and the generated FSMs for the PCI Bus are available at http://hvg.ece.concordia.ca/Research/SoC/ASM/PCI-Bus/

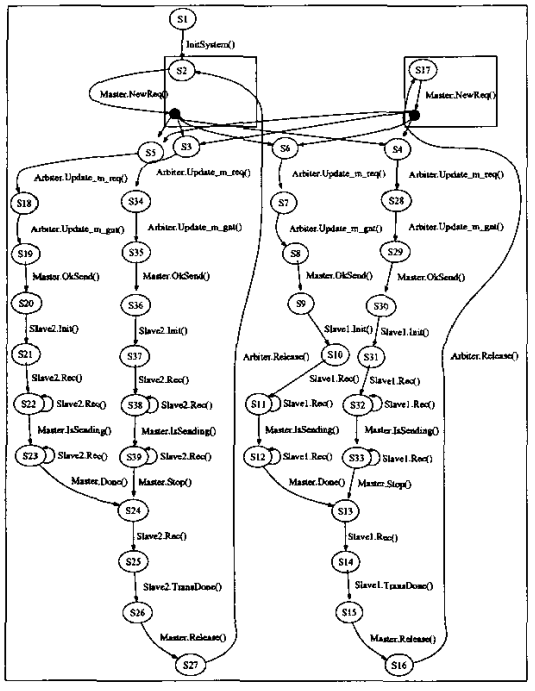

Fig. 3. PCI Bus FSM for one Master and two Slaves.

TABLE I
FSM GENERATION STATISTICS.

Number of		CPU	Number of FSM	
Masters	Slaves	Time (s)	Nodes	Transitions
1	1	2.3125	20	25
1	2	2.9375	39	53
3	1	26.015625	236	341
2	2	26.84375	293	449
2	3	101.375	658	1117
3	2	574.1875	1881	3153
3	3	6836.015625	6346	12097

C. Simulation

Table III shows a simulation evaluation of the PCI bus when implemented in SystemC. We display the average execution time per clock cycle as a function of the number of masters and slaves connected to the bus.

V. CONCLUSION

In this paper, we presented a design and verification approach using UML and ASM to create SystemC IPs. We used the PCI bus as illustrative case. We first defined a model of the bus in UML that we translated to ASM in order to verify, using model checking, a set of PSL properties of the bus. Finally, we mapped the verified ASM code to SystemC. Experimental results showed a short verification time for the properties (less than a few minutes for all the properties). Besides, the final

TABLE II
MODEL CHECKING RESULTS.

Number of		Model Checking	
Masters	Slaves	CPU Time (s)	BDDs Allocated
1	1	0.125	7699
1	2	9.406250	27916
3	1	11.40625	49008
2	2	12.875	52777
2	3	50.484375	68491
3	2	85.015625	110662
3	3	155.125	253420

TABLE III
SIMULATION RESULTS.

Number of		Average Execution
Masters	Slaves	Time per Clock Cycle (10^{-9}s)
1	1	24.31
1	2	29.32
3	1	29.766
2	2	30.891
2	3	32.744
3	2	34.032
3	3	36.828

PCI SystemC IP was deeply and formally verified, which makes it very suitable for use as external monitor to validate existent PCI compatible IPs. We believe that our approach improves the design and verification methodologies used for SystemC models by integrating formal methods as part of the design flow.

REFERENCES

[1] Accellera Organization. Accellera Property Specification Language Reference Manual, version 1.01. http://www.accellera.org, 2004.

[2] F. Bruschi and F. Ferrandi. Synthesis of Complex Control Structures from Behavioral SystemC Models. In *Proc. Design, Automation and Test in Europe*, pages 20112–20119, Munich, Germany, March 2003.

[3] A. Gawanmeh, A. Habibi, and S. Tahar. An Executable Operational Semantics for SystemC using Abstract State Machines. Technical report, Department of Electrical and Computer Engineering, Concordia University, March 2004.

[4] W. Grieskamp, Y. Gurevich, W. Schulte, and M. Veanes. Generating Finite State Machines from Abstract State Machines. *Software Engineering Notes*, 27(4):112–122, 2002.

[5] A. Habibi, A. Gawanmeh, and S. Tahar. Embedding of PSL in ASM with an Application to SystemC Verification. Technical report, Department of Electrical and Computer Engineering, Concordia University, May 2004.

[6] K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[7] Microsoft Corporation. AsmL for Microsoft .Net Framework (version 2.1.5.7). http://www.research.microsoft.com/foundations/asml, 2004.

[8] Open SystemC Initiative. *SystemC 2.0.1 Language Reference Manual*. 2003.

[9] Open SystemC Initiative. http://www.systemc.org, 2004.

[10] PCI-Sig. PCI Special Interest Group. http://www.pcisig.com, 2004.

[11] K. Shimizu, D. L. Dill, and A. J. Hu. Monitor-Based Formal Specification of PCI. In *Proc. Formal Methods in Computer-Aided Design*, pages 335–353, Austin, Texas, USA, November 2000.