Model Based Verification of SystemC Designs

Haja Moinudeen, Ali Habibi and Sofiène Tahar
Department of Electrical and Computer Engineering
Concordia University
1455 de Maisonneuve, West,
Montreal, Quebec H3G 1M8
Email: {haja_m, habibi, tahar}@ece.concordia.ca

Abstract— **Recent advancement in System-on-Chip design leads to the promotion of system level languages such as SystemC. This latter enables rapid prototyping and fast simulation in comparison to the classical Register Transfer Level (RTL) based approach. Intuitively, from a verification point of view, faster simulation induces better coverage results. In this paper, we propose a methodology to verify SystemC designs. We propose an automatic generation procedure of the system's finite state machine (FSM) from SystemC. The generated FSM is then used to produce test suites allowing functional testing of SystemC designs. Furthermore, the same FSM is used to perform conformance testing to validate lower abstraction levels of the design (e.g., RTL). We illustrate the feasibility and efficiency of our approach on a PCI bus standard.**

I. Introduction

Due to the ever increasing complexity of today's System-On-Chip (SoC) and time-to-market pressure, the abstraction level in the design flow has been raised to the system level. Until recently, modeling architectures required pin-level hardware descriptions, typically Register Transfer Level (RTL). Great effort is required to design and verify the models, and simulation at this level of detail is tediously slow. System level is eventually the best solution to address these issues. Moreover, the benefits of adopting system level are derived from early software development, early functional verification, and higher system quality.

At the system level, we only model the level of detail that is needed by the designers developing the system components and sub-system for a particular task in the development process [6]. SystemC [13] fulfills the requirements of system-level languages. It is expected to make an adverse effect in the arena of architecture, co-design and integration of hardware and software. The SystemC library of classes and simulation kernel extend C++ to support concurrent behavior, a notion of time sequential operations, data types for describing hardware, structure hierarchy, etc.

Nevertheless, the verification of SystemC is a bottleneck due to the object-oriented nature of the language. Two verification techniques could be used: formal verification and simulation. The first technique, has its own limitation of state space explosion in the case of model checking and non-automation in the case of theorem proving. In contrast, the second technique, is widely used in industry, where test suites are generated to validate the design functionalities. Generally, in order to have efficient test suites, a finite state machine (FSM) model of the design is to be used. For this reason, it is crucial to provide an FSM generation algorithm for SystemC that produces a correct FSM of the system which can be used for verification purposes.

In this paper, we propose to generate FSM from SystemC designs using two algorithms [8] originally proposed for the generation of FSM from Abstract State Machines (ASM) [5] [3]. The two algorithms that we use show good performance and they are less prune to state space explosion. We propose to use the system's state model (generated FSM) to generate test suites that can be used to verify the functionality of system through simulation. In addition, our methodology facilitates the conformance checking [1] between RTL and system level of SystemC designs. We evaluate the performance of the proposed algorithm on the case study of the PCI standard bus [17].

The rest of the paper is organized as follows: Section II discusses some related work. Section III elucidates our proposed methodology. In section IV, we provide a PCI bus as a case study to substantiate our methodology. Finally, Section V concludes the paper.

II. Related Work

In [5], Grieskamp *et al.* proposed an FSM-generating algorithm from ASM. In this work, given an AsmL [7] model, the algorithm generates an FSM. This algorithm is specific for ASM based languages. Campbell *et al.* extended the work of [5] in [3] to lessen the number of states in the generated FSM. FSM generation algorithm from AsmL (or Spec# [11]) in this work is based on multiple state groupings which is an extension of the concept of hyperstates.

FSM based testing was initially driven by problems arising in functional testing of low level hardware circuits. The bulk of the work in this area has dealt with deterministic FSMs [10] [14]. Then, Extended Finite State Machine (EFSM) approach has been introduced to cope with the state explosion problem of the FSM approach [2]. More work related to finite state machine based software testing can be found on the homepage of Model-Based Testing [15].

As far as we know, Dick *et al.* [4] were the first to introduce automated technique for extracting FSMs from model-based specifications for the purpose of test case generation. The approach of [4] is based on a finite partitioning of the state space of the model using full disjunctive normal forms (full

DNFs) of the conditions in the specification and is called the DNF approach. The DNF approach suffers from two problems: (1) state explosion and (2) the use of theorem proving (time-consuming).

In [9], Habibi *et al.* presented a top-down approach to verify SystemC designs where the verification was integrated as part of the design process and AsmL model was first designed. From the AsmL model, an FSM was generated and model checking was performed. The verified AsmL model is translated to SystemC. The approach of [9] assumes that the original design models are defined in the AsmL language which is a major drawback for the verification of actual SystemC designs. Furthermore, it requires that the designer defines two different models of the same design: the first in AsmL and the second in SystemC. On the other hand, using a general purpose FSM generation algorithm does not take advantage from the SystemC simulation semantics where processes are being executed in a pre-defined and deterministic way.

III. Proposed Methodology

Figure 1 describes our methodology of verifying SystemC designs. We propose two algorithms (the first is called *direct algorithm* and the second as *grouping algorithm*). The original SystemC code is used to feed the direct algorithm with the information required for the FSM generation. This information includes: the list of methods, the list of state variables and the list of trigger events. In the the algorithms, side, these entities correspond, respectively, to actions (A), state variables (V) and action's pre-conditions (A_{pre}). In addition to the previous information, the grouping algorithm requires as input a set of grouping conditions (G_C). Both algorithms perform a state exploration procedure in order to discover all possible system's state starting from a set of initial state. The generated FSMs are used to perform two applications: (1) test sequences generation; and (2) conformance checking.

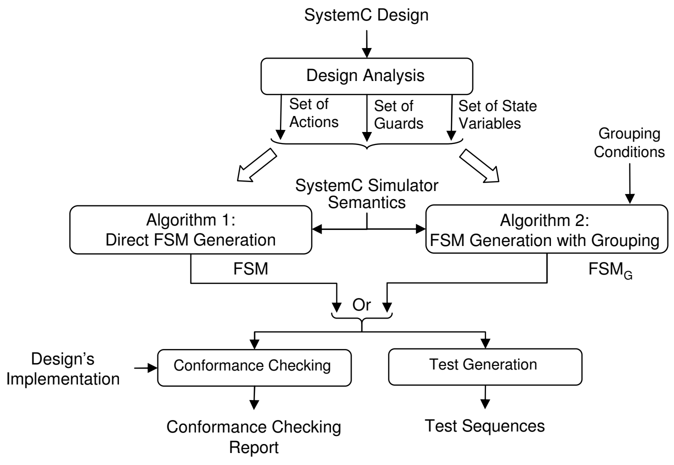

Fig. 1. Methodology and Motivation.

Both FSM generation algorithms are fed with the SystemC simulation semantics. This latter provides a valuable information about the order of execution of the processes. For instance, this execution is constrained by the nature of the process (timed or un-timed) and by the order in which the processes have been initialized in the elaboration phase.

The SystemC language reference manual [12] does not define the formal semantics for SystemC simulation. For this reason, we formalized the current implementation of the SystemC library which follows the evaluate-update paradigm that is common in HDLs. For instance, the concept of delta cycles, where multiple evaluate-update phases can occur at the same simulation time, is supported. Following are the steps of SystemC simulator:

1) *Initialization:* Execute all processes to initialize the system.
2) *Evaluate:* Execute a process that is ready to run. Iterate until all ready processes are executed. Events occurring during the execution could add new processes to the ready list.
3) *Update:* Execute any update calls made during step
4) If delayed notifications are pending, determine the list of ready processes and proceed to the Evaluate phase (Step 2).
5) Advance the simulation time to the earliest pending timed notification. If no such event exists, simulation is finished, else determine ready processes and proceed to Step 2.

Using the constructed state model, we can use various techniques to choose which paths we want our tests to take through it. One of the most popular choices is to allow the tests to move randomly through the state model, taking any available action out of a state. Given enough time, these random walks can cover a good part of the application. The random nature of such choices means that they tend to produce unusual combinations of actions that human testers would not bother to try.

A more advanced path generation technique, called a "Chinese Postman tour", touches every action in the state model as efficiently as possible [16]. These sequences of actions can be stored in an external file (such as "TestSeq.txt"). This action sequence file then serves as the instructions to the test execution phase.

For the specific case of SystemC a partial "Chinese Postman tour" technique can be used in order to ensure visiting specific states of the system. This is quite important when only a subset of the design's actions is of interest for the verification. This is the case if we want to test a specific mode for a bus model for example.

Conformance checking [1] represents a very important application for SystemC verification. For instance, the constructed state model from the SystemC design can represent a golden model to validate lower levels implementations (such as RTL). A specific execution can be performed where we can check if actions are enabled and if they return correct values.

IV. Application: PCI Bus Standard Verification

The PCI [17] 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 2, 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.

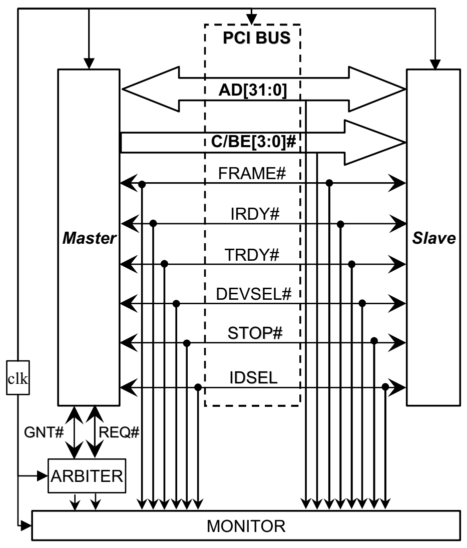

Fig. 2. PCI Bus Structure.

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 2) 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.

To evaluate the performances of both algorithms we consider as criteria: the CPU time needed for generating the FSM and the number of its states and transitions. The grouping condition (see Figure 3) is a conjunction of the *destination slave* and the *final status of the transmission* (completed or stopped). For example, in case of two slaves, the number of grouped states is four. Table I includes the experimental results, (platform: Pentium IV/1GB memory/WinXP-SP2) for multiple numbers of masters and slaves. For both algorithms, the number of states and transitions increase exponentially as a function of the number of masters and slaves connected to the bus. However, we can note that the CPU time required for the FSM generation using the direct algorithm is shorter than the one required for the grouping algorithm. For both algorithms the CPU time does not exceed 13 minutes even for the case of three masters and three slaves. Finally, as expected the grouped FSM is smaller than the general FSM in terms of number of states and transitions. We also note that the number of states in the grouped FSM depends on the grouping condition itself, that is why it is reduced to four when we have two slaves and two possible termination conditions for the transaction (succeeded or stopped).

TABLE I
FSM Generation Evaluation.

Number of		**Algorithm 1**			**Algorithm 2**		
		CPU	Num. of FSM		CPU	Num. of FSM	
Ma.	Sl.	Time (s)	Nod.	Tran.	Time (s)	Nod.	Tran.
1	1	0.34	20	25	0.36	2	4
1	3	0.80	58	85	0.86	6	23
3	1	4.44	236	341	4.56	2	4
2	2	4.31	293	449	4.61	4	14
3	2	108.03	1881	3153	148.47	4	14
3	3	727.01	3880	7542	772.60	6	23

Figure 4 gives a snapshot of the grouped FSM for the case of two masters and two slaves where the grouped condition is the one described in Figure 3. For example, group 0 (node *G0* in Figure 4) corresponds to the case when the destination slave is *Slave1* and the transaction succeeds. Figure 5 provides a possible test sequence solution for the “Chinese Postman tour” problem using the FSM of Figure 4.

V. Conclusion

We proposed in this paper a methodology to verify SystemC designs by generating a state model of the design. For instance, we used two algorithms for the FSM generation: the first one explores all possible states, while, the second uses a grouping condition that helps to reduce the states and transitions in the final FSM. The generated FSMs can be of great importance for various verification techniques such as model checking, test generation, coverage evaluation and conformance testing. Finally, we illustrated our approach for the case of the PCI bus standard.

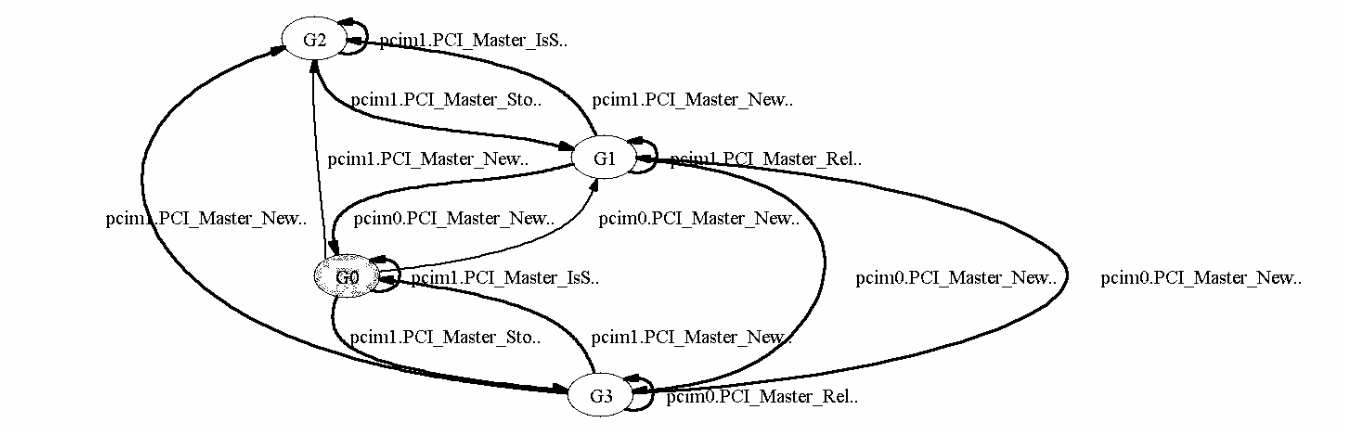

Fig. 4. Generated FSM using the Grouping Algorithm: 2 Masters / 2 Slaves.

```
GroupingCondition()=
    if ((exists x in MASTERS where x.m_dest = 1
                        and x.m_stop = true)= true) then
        Return 0
    else
        if ((exists x in MASTERS where x.m_dest = 1
                        and x.m_stop = false) = true) then
            Return 1
        else
            if ((exists x in MASTERS where x.m_dest = 2
                        and x.m_stop = true)= true) then
                Return 2
            else
                if ((exists x in MASTERS
                    where x.m_dest = 2
                    and x.m_stop = false) =true) then
                    Return 3
                else Return -1
```

Fig. 3. Grouping Condition.

```
Test Sequence:
    From G0 to G0 using InitSystem()
    From G0 to G0 using PCI_Master_NewReq(False,2)
    From G0 to G1 using PCI_Master_NewReq(True,1)
    From G1 to G0 using PCI_Master_Stop(True,1)
    From G0 to G2 using PCI_Master_NewReq(False,1)
    From G2 to G3 using PCI_Master_NewReq(True,2)
    From G3 to G0 using PCI_Master_NewReq(False,1)
```

Fig. 5. Test Sequence for the "Chinese Postman Tour"

References

[1] M. Barnett and L. Nachmanson W. Schulte. Conformance Checking of Components Against Their Non-deterministic Specifications. Technical report, Microsoft Research, MSR-TR-2001-56, June 2001.

[2] C. Bourhfir, R. Dssouli, and E. M. Aboulhamid. Automatic Test Generation for EFSM-based Systems. Technical report, Publication departementale 1043, Departement IRO, Universite de Montreal, August 1996.

[3] C. Campbell and M. Veanes. State Exploration with Multiple State Groupings. In *Proc. International Workshop on Abstract State Machines*, pages 119–130, Paris, France, January, 2005.

[4] J. Dick and A. Faivre. Automating the Generation and Sequencing of Test Cases from Model-Based Specifications. In *Proc. of the First International Symposium of Formal Methods Europe on Industrial-Strength Formal Methods*, pages 268–284, London, UK, 1993. Springer-Verlag.

[5] 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.

[6] T. Grotker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Kluwer Academic Publishers, 2004.

[7] Y. Gurevich, B. Rossman, and W. Schulte. Semantic Essence of AsmL. Technical report, Microsoft Research, MSR-TR-2004-27, March 2004.

[8] A. Habibi, H. Moinudeen, and S. Tahar. Towards a Faster Simulation of SystemC Designs. In *Proc. of the IEEE Computer Society Annual Symposium on VLSI*, pages 418–419, Karlsruhe, Germany, March 2006.

[9] A. Habibi and S. Tahar. Design for Verification of SystemC Transaction Level Models. In *Proc. Design Automation and Test in Europe*, pages 560–565, Munich, Germany, March 2005.

[10] D. Lee and M. Yannakakis. Principles and Methods of Testing Finite State Machines - A Survey. In *Proceedings of IEEE Computer Society*, volume 84, pages 1090–1123, Berlin, Germany, August, 1996.

[11] Microsoft Corp. Spec#. http://research.microsoft.com/specsharp/, 2005.

[12] Open SystemC Initiative. *SystemC 2.0.1 Language Reference Manual.* 2005.

[13] Open SystemC Initiative. www.systemc.org, 2005.

[14] A. Petrenko. Fault Model-driven Test Derivation from Finite State Models: Annotated Bibliography. pages 196–205, 2001.

[15] H. Robinson. Model-based testing. website: http://www.geocities.com/model_based_testing/, 2005.

[16] H. Robinson. Graph Theory Techniques in Model-Based Testing. In *Proc. International Conference on Testing Computer Software*, Washington, D.C, USA, June, 1999.

[17] PCI Special Interest Group. www.pcisig.com, 2004.