Design for Verification of a PCI-X Bus Model

Haja Moinudeen

A Thesis

in

The Department

of

Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of Master of Applied Science at

Concordia University

Montréal, Québec, Canada

February 2006

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: **Haja Moinudeen**

Entitled: **Design for Verification of a PCI-X Bus Model**

and submitted in partial fulfilment of the requirements for the degree of

Master of Applied Science

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

_____ Dr. Rabin Raut

_____ Dr. Patrice Chalin

_____ Dr. Abdelsalam En-Nouaary

_____ Dr. Sofiène Tahar

Approved by _____

Chair of the ECE Department

_____ 2005 _____

Dean of Engineering

ABSTRACT

Design for Verification of a PCI-X Bus Model

Haja Moinudeen

The importance of re-usable Intellectual Properties (IPs) cores and the system-level design languages have been increasing due to the growing complexity of today's System-on-Chip (SoC) and the need for rapid prototyping. In this respect, the SystemC language is becoming an industrial standard to be used as a modeling language for SoCs at the system level. Nevertheless, it is of paramount importance to have SystemC IPs in particular bus standards in order to facilitate SoC designs using SystemC. PCI-X is the fastest and latest extension of PCI (Peripheral Component Interconnect) technologies that is backward compatible to previous PCI versions. It plays a crucial role in today's SoC since it helps to connect various IP. In this thesis, we provide a design for verification approach of a PCI-X bus model. We use different modeling levels, namely UML, AsmL and SystemC to design and verify the PCI-X. From informal standard specifications of PCI-X, we first represent the PCI-X model in UML where a precise capture of design requirements is possible. From the UML representation, we construct an AsmL model of the bus and define various properties using the Property Specification Language (PSL). Finally, we translate the AsmL model to SystemC. The verification of the PCI-X models has been conducted using both model checking and model based testing (MBT). For the former, we used the

AsmL model which we checked against the PSL properties. For MBT, we used the SystemC model to generate finite state machines (FSMs) from which we produce test cases. To do this, two new FSM generation algorithm have been developed in this thesis. Experimental results allowed us to explore the potential of MBT for SystemC designs in contrast to model checking.

To my parents, and sisters

ACKNOWLEDGEMENTS

First and foremost, I would like to put forward my sincere gratitude to my research advisor, Dr. Sofiène Tahar for his support and encouragement throughout my M.A.Sc. program. His expertise and competent advice have shaped the character of my research.

This thesis would not have come into its entirety without the constant and invaluable technical guidance of Dr. Ali Habibi, Research Associate, Hardware Verification Group (HVG), Concordia University. I extensively benefitted from his deep knowledge and insight in the subject.

I would also like to thank all the members of the HVG for their help and support during my study at Concordia. Special acknowledgements to my examination committee members, for reviewing my thesis and giving me feedbacks.

TABLE OF CONTENTS

LIST OF TABLES x

LIST OF FIGURES xi

LIST OF ACRONYMS xiv

1 Introduction **1**

1.1 Motivation 1

1.2 Methodology 4

1.3 Related Work 7

1.4 Thesis Contributions 11

1.5 Thesis Outline 12

2 Preliminaries **13**

2.1 Unified Modeling Language (UML) 13

2.1.1 Class Diagrams 15

2.1.2 Sequence Diagrams 16

2.2 Abstract State Machines (ASM) 16

2.2.1 ASM Language 16

2.2.2 States 17

2.2.3 Terms 18

2.2.4 Locations and Updates 18

2.2.5 Transition Rules 19
2.3 Abstract State Machine Language (AsmL) 21
2.4 Property Specification Language (PSL) 22
2.5 SystemC 23

3 PCI-X Bus Modeling **27**
3.1 PCI-X Bus 27
3.2 UML Modeling 31
3.3 AsmL Modeling 34
3.4 Translation to SystemC 39

4 Model Checking **42**
4.1 Background 42
4.2 Model Checking Approach 43
4.3 Model Properties 44
4.4 Experimental Results 49

5 Generating FSM from SystemC **51**
5.1 Background 51
5.2 SystemC Syntactical Domain 52
5.3 Collecting Inputs 55
5.4 FSM Generation Algorithm 56
5.4.1 Helper Functions 58

5.4.2 Direct FSM Generation 59
5.4.3 Grouping FSM Generation 61
5.5 Discussion 64

6 Model-Based Testing **66**
6.1 Background 67
6.2 Direct Algorithm 71
6.3 Grouping Algorithm 73
6.4 Test Generation 77
6.4.1 Chinese Postman Tour (CPT) 78
6.4.2 RandomWalk 79
6.4.3 Test Sequences 80
6.5 Discussion 82

7 Conclusion **85**
7.1 Summary 85
7.2 Discussion and Future Work 86

Bibliography **89**

LIST OF TABLES

3.1 PCI-X Signals 29

4.1 Model Checking Results 50

5.1 Link between the SystemC Design and the FSM Generation Algorithm Inputs. 55

6.1 FSM Generation: Direct Algorithm. 72

6.2 FSM Generation: Grouping Algorithm (using *G1*). 78

LIST OF FIGURES

1.1 Design for Verification Methodology of PCI-X. 5

2.1 SystemC Language Architecture [17]. 25

2.2 A Simplified SystemC MetaModel [35]. 26

3.1 General Architecture of PCI-X. 28

3.2 Class Diagram of PCI-X. 32

3.3 Sequence Diagram of Mode 1 Transaction. 33

3.4 Sequence Diagram of Mode 2 Transaction. 34

3.5 Sequence Diagram of Mode 2 Transaction (16 bit). 35

3.6 Used AsmL Enumeration Types. 36

3.7 Arbiter GNT AsmL Method. 36

3.8 Target Assert AsmL Method. 37

3.9 Initiator Termination AsmL Method. 38

3.10 Clock Update AsmL Method. 38

3.11 Data Phase AsmL Method. 38

3.12 Arbiter Class Declaration in SystemC. 40

4.1 Property *P1*. 45

4.2 Property *P2*. 45

4.3 Property *P3*. 45

4.4 Property *P4*. 46

4.5 Property *P5*. 46

4.6 Property *P6*. 47

4.7 Property *P7*. 47

4.8 Property *P8*. 47

4.9 Property *P9*. 48

4.10 Property *P10*. 48

4.11 Property *P11*. 49

4.12 Property *P12*. 49

5.1 Direct FSM Generation Algorithm. 60

5.2 Grouping FSM Generation Algorithm. 63

6.1 Direct Algorithm Results. 71

6.2 Grouping Condition *G1*. 74

6.3 Grouped FSM Using *G1*. 75

6.4 Grouping Algorithm Results (*G1*). 76

6.5 Grouping Condition *G2*. 77

6.6 Grouping Condition *G3*. 79

6.7 Grouped FSM Using *G2*. 80

6.8 Grouped FSM Using *G3*. 81

6.9 Grouping Condition *G4*. 81

6.10 Grouped FSM Using *G4*. 82

6.11 Test Sequences for Grouped FSM of *G1* using CPT. 82

6.12 Test Sequences for Grouped FSM of *G2* using CPT. 83

6.13 Test Sequences for Grouped FSM of *G2* using Randomwalk. 84

6.14 Test Sequences for Grouped FSM of *G3* using CPT. 84

6.15 Test Sequences for Grouped FSM of *G4* using CPT. 84

LIST OF ACRONYMS

AGP	Accelerated Graphics Port
ASM	Abstract State Machines
AsmL	Abstract State Machine Language
BCA	Bus Cycle Accurate
CPP	Chinese Postman Problem
CPT	Chinese Postman Tour
ECC	Error Correction Code
EDA	Electronic Design Automation
EFSM	Extended Finite State Machine
FSM	Finite State Machine
HDL	Hardware Description Language
IC	Integrated Circuit
IP	Intellectual Property
MBT	Model Based Testing
MC	Model Checking
OO	Object Oriented
PCI	Peripheral Component Interconnect
PCI-X	Peripheral Component Interconnect-Extended
PSL	Property Specification Language

RTL	Register Transfer Level
SLL	System Level Languages
SoC	System-On-Chip
UML	Unified Modeling Language
VLSI	Very Large Scale Integration

Chapter 1

Introduction

1.1 Motivation

With the rapid advances in semiconductor processing technologies, the density of gates on the die increased in line with what Moore's law predicted. This helped in the realization of more complicated designs on the same Integrated Circuit (IC). Over the last few years, with the advent of high technology applications, an increasingly evident need has been that of incorporating the traditional microprocessor, memories and peripherals or in other words the whole system on a single silicon. This is what has marked the beginning of the System-on-Chip (SoC) era.

An SoC can be viewed as a collection of various Intellectual Property (IP) cores, with interconnecting buses running among them. Since IPs are obtained from different vendors, and are often meant to be re-used at various stages of the SoC

design, there is a dire need for standard buses to connect them. One such and latest bus standard is PCI-X [40]. PCI-X is a high performance bus for interconnecting chips, expansion boards, and processor/memory subsystems. It has the performance to feed the most bandwidth-hungry applications and helps to alleviate the I/O bottleneck problem while at the same time maintaining complete hardware and software backward compatibility to previous generations of PCI [40]. PCI-X was adopted as an industrial standard administered by the PCI Special Interest Group (PCI-SIG) [40]. Nevertheless, bus protocols in general are not easy to model and verify correctly due to their complex nature and PCI-X is no exception.

Until recently, modeling IP architectures required pin-level hardware descriptions, typically Register Transfer Level (RTL). Design and simulation at this level of detail is tediously slow. Therefore, designers turned to the system level, where only details needed for developing the system components and sub-system for a particular task in the development process are considered [17]. In other words, system and sub-system communication and functionality can be developed and refined independently. By modeling only the necessary details, design teams can realize huge gains in modeling speed. Moreover, the benefits of adopting system level are derived from early software development, early functional verification, and higher system quality.

In this thesis, we propose to design and verify a PCI-X model at the system level by partially using the top-down methodology presented in [19]. We start with an informal specification of PCI-X and model it with the Unified Modeling Language

(UML) [42] in order to have a clear view of the design modules and their interactions. Then, we manually construct an Abstract Machine Language (AsmL) [18] model from the UML representation. AsmL is an object-oriented, executable specification language developed at Microsoft Research [28]. It can be used to capture the abstract structure and step-wise behavior of discrete systems such as integrated circuits, software components, and devices that combine both hardware and software.

We define a set of properties of the PCI-X in the Property Specification Language (PSL) [1]. PSL is a language developed by Accellera for specifying properties or assertions about hardware designs. PSL is an implementation independent language to define properties and addresses the lack of information about properties and design characteristics. PSL has become IEEE standard in September 2004. The PSL properties can be simulated or formally verified using model checking. We embed the defined properties with the AsmL model of the PCI-X model and run model checking [9] against the properties defined.

Finally, we translate the AsmL model to SystemC [32]. SystemC is among a group of system level languages (SLL) being proposed to raise the abstraction level for embedded system design and verification. The SystemC library of classes and simulation kernel extend C++ to include the support for concurrent behavior, a notion of time sequential operations, data types for describing hardware, structure hierarchy and simulations.It is expected to make an adverse effect in the arenas of architecture, co-design and integration of hardware and software. As SystemC

is becoming a main player in SoC design, it is becoming mandatory to have large collection of IPs in SystemC especially bus standards.

To validate the SystemC model of the PCI-X, we use model based technique (MBT) [36]. MBT is is an evolving technique in the arena of testing. MBT is an approach in which the behavior of a system is defined in terms of actions that change the state of the system. Such a model of the system results in a well-defined Finite State Machine (FSM) which helps understand and predict the systems behavior. For this reason, it is inevitable to provide an FSM generation algorithm from SystemC that produces a correct FSM of the system. In this thesis, we extend two FSM generation algorithms from SystemC that have been originally proposed for the generation of FSM from Abstract State Machines (ASM) [3]. Using existing graph traversing technique, test cases are obtained from the generated FSMs to validate the PCI-X model.

1.2 Methodology

Figure 1.1 depicts the methodology we want to adopt for the design and verification of a PCI-X model. This methodology is partially derived from a top down approach for verifying SystemC designs proposed by Habibi [19]. We first model the PCI-X in UML using class and sequence diagrams in order to capture all the design requirements efficiently. Class diagram is used to represent all the core components of the bus and the sequence diagram is used to model all the sequences of various

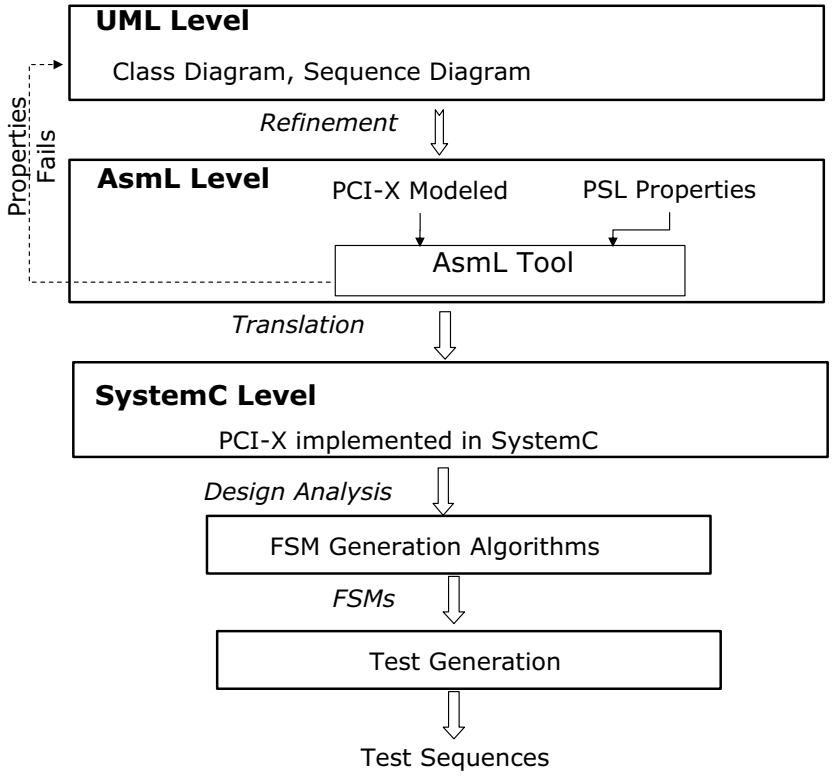

Figure 1.1: Design for Verification Methodology of PCI-X.

transactions. Having a clear UML representation of the system makes the AsmL modeling easy. Then, from the UML representation we design the PCI-X in AsmL. The properties of the bus design are specified in PSL and embedded with the AsmL model. They are basically extracted from the sequence diagrams and encoded in the PSL syntax. Next, model checking is performed on the AsmL model using the algorithm provided in AsmL tester (Asmlt) [30]. The model checking process ends to: (1) a completion either with success or failure of the property; or (2) a state explosion. The updates of UML representation and UML to AsmL translation tasks

are repeated until all the properties pass with success or do not complete. The model checking approach of the PCI-X and the results are vividly discussed in Chapter 4. Once the model checking of AsmL model is completed, we perform the translation of AsmL to SystemC.

Nevertheless, it is not always feasible to verify all the properties that we define due to state explosion problem and also writing properties is time consuming that eventually hinders the verification process. Also, the SystemC model has to be validated since the translation of AsmL to SystemC can be erroneous. In order to address this issues, we perform model based testing of the PCI-X model in SystemC. To accomplish this, we develop two algorithms to generate FSMs from SystemC models. These algorithms have been originally proposed for ASM in [15] [5]. The first algorithm (*direct algorithm*) performs a direct state exploration procedure to generate FSM and the second (*grouping algorithm*) involves a notion of state grouping to group states in the FSM. The formalization of these two algorithms is presented in Chapter 5. The generated FSMs are used to generate test cases using available techniques such as Chinese Postman Tour(CPT) [36] and RandomWalk [36]. The details and the experimental results of PCI-X MBT are explained in detail in Chapter 6. Furthermore, the generated FSM can also be used for conformance checking [2] between two design abstraction levels (AsmL and SystemC for example).

1.3 Related Work

In his PhD thesis, Habibi [19] proposed a combination of various techniques to verify system level languages in particular SystemC. In [19], a top-down approach to verify SystemC designs was proposed. In that approach, the verification was integrated as part of the design process and AsmL model was first designed. The AsmL was mapped from UML representations of the systems. From the AsmL model, model checking was performed against some system properties. The verified AsmL model is translated to SystemC. The SystemC model is also verified through assertion-based verification.

Similar to the work of [19], Habibi *et al.* proposed in [21], a bottom-up verification methodology where starting from an existent Accelerated Graphics Port (AGP) IP in SystemC, an AsmL model was developed. Like in [19], an FSM was generated from the AsmL model and then the systems properties were verified at the ASM level.

Our work is an application of the top-down methodology presented in [19] to model and verify the PCI-X bus standard. In addition, in our work, we made an attempt to explore MBT techniques to validate our PCI-X design when model checking is not feasible due to state explosion. Furthermore, in [19], MBT technique was not considered in the verification approach.

There exists some other related works to ours in the context of PCI technologies design and verification environment. For instance, Shimizu *et al.* [39] presented a

specification of the PCI bus as a Verilog monitor. This approach, however, makes any modification or refinement of the model complex since the level of specification of the PCI is very low. Oumalou *et al.* [33] implemented the PCI bus in SystemC [32]. First, they specified the bus in UML; then, mapped the UML representation to AsmL and finally translated the AsmL code into SystemC. In [20], Habibi *et al.* specified and implemented the Look-Aside Interface [29] using the same approach as in [33]. Our approach is similar to the work of [33] and [20], but distinguishes itself by correctly designing and verifying the latest high-speed bus standard (PCI-X) including its very complex transaction rules and validating the PCI-X model through MBT.

In [8], a bridge for PCI Express and PCI-X was designed in Verilog at RTL and also verified and synthesized. Chang *et al* [6] proposed a simulation based PCI-X verification environment using C and Verilog. Wang *et al.* [43] proposed a similar verification environment as in [6] for PCI-X bus functional models using VERA. Yu *et al.* [44] extended verification environment in [6] to support PCI, PCI-X and PCI-Express in a single platform. In all the afore-mentioned works [6, 43, 44], the authors defined some pseudo model of masters and slaves to substantiate their approach. But, they failed to include some of the PCI-X specific transactions such as Split transactions. The merit of our work is on the modeling of the PCI-X and verifying using various techniques such as model checking and MBT.

For work related to FSM generation, we cite Grieskamp *et al.* [15], in which,

given a model in given a model in the Abstract State Machine Language (AsmL) [18], they provide an algorithm to generate an FSM. The state transitions of the ASM are used to generate a link between hyperstates which is based on single state groupings. The notion of hyperstates in here is analogous to data abstraction. But, the drawback of this algorithm is the use of single state grouping, which is not adequate to solve the state space explosion problem. Moreover, this algorithm is specific for ASM based languages.

Campbell *et al.* [5] extended the work of [15] to lessen the number of states in the generated FSM. The FSM generation algorithm from AsmL (or Spec# [27]) in [5] is based on multiple state groupings which is an extension of the concept of hyperstates. Using this algorithm, there is a significant reduction of states and transitions in the final FSM. This notion of multiple state groupings is synonymous to predicate abstraction. The problem with this work is that it is only applicable for AsmL and Spec#.

Other related work concerns generating FSM from hardware description languages (HDL). For instance, in [7], a translation procedure from Verilog into timed FSM model was proposed. The resulting timed FSM was used to verify the system being modeled using model checkers. Lohse *et al.* presented in [24] an approach to generate BDD-based FSM from VHDL code. This is performed in two phases: (1) declarations are annotated with BDDs and processes are compiled into control graphs; and (2) the control graphs are then compiled into an FSM and optimization

of the FSM was performed. The generated FSM was mainly used as an input for the model checking tool SMV [26]. However, the approach in both [7] and [24] is restricted to a synthesizable subset of Verilog and VHDL, respectively. The work of Vikram *et al.* [38] was more specific to SystemC, where the design (in SystemC) is first translated to C, then, the FSM is generated from the C code. This approach is problematic in the sense that translating SystemC to C is not always feasible. All previously mentioned techniques could not be applied to SystemC considering the OO nature of the library and that not all SystemC is synthesizable. Ferrandi *et al.* [13] proposed to use FSMs to perform functional verification of SystemC designs. They constructed an FSM directly from the code and then use it to guide the test generation. In that work, the FSM generation algorithm was not precisely explained and does not consider the semantics of the SystemC simulator.

Regarding the application of MBT, as far as we know, Dick *et al.* [11] were the first to introduce automated techniques for extracting FSMs from model-based specifications for the purpose of test case generation. The approach of [11] 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. Heuristics are used in the DNF approach as part of theorem proving, whereas, in the approach we propose in this thesis uses heuristics (grouping conditions) to prune the search space. The DNF approach suffers from two problems: (1) state explosion and (2) the use of theorem proving (time-consuming).

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 [23] [34]. Then, Extended Finite State Machine (EFSM) approaches have been introduced to cope with the state explosion problem [4]. More work related to FSM-based software testing can be found on the home page of Model-Based Testing [36].

All the afore-mentioned work regarding FSM based testing are not proposed for SystemC. FSM based testing could be a promising techniques and well suitable for SystemC as it is meant to be used in higher abstraction level.

1.4 Thesis Contributions

In light of the above related work review and discussions, we believe the contribution of the thesis are as follows:

1. We model the PCI-X in UML, AsmL and SystemC. Thus, providing a PCI-X IP in SystemC to accommodate the dire need of bus standard IPs in SystemC.

2. We define properties in PSL and verify the PCI-X using model checking.

3. We provide a formalization of two FSM generation algorithms for SystemC.

4. We perform model based testing of the PCI-X using the formalization of the FSM generation algorithms provided.

5. We provide a better understanding of the problem of state explosion in model checking and a good comparison with MBT.

1.5 Thesis Outline

The thesis is organized as follows: In Chapter 2, we provide an overview of UML, ASM, ASML, PSL and SystemC. This chapter lays a foundation for the better understanding of the thesis. Chapter 3 presents the PCI-X and included the modeling of PCI-X in UML, AsmL and SystemC. In Chapter 4, we show the verification of PCI-X using model checking of the AsmL model against the properties defined in PSL. In Chapter 5, we present the FSM generation algorithms from SystemC. It includes the formalization of the two FSM generation algorithms: a) Direct FSM Generation b) Grouped FSM Generation. Chapter 6 discusses the model based testing of PCI-X model using the algorithms presented in Chapter 5. Chapter 7 provides the summary of the thesis with some discussions regarding the FSM generation algorithms and PCI-X IP. It also draws major conclusions and lessons learned from this work. In addition, it highlights some future research directions that can be stemmed from this thesis.

Chapter 2

Preliminaries

In this chapter, we give an insight on UML, ASM, AsmL, PSL and SystemC language and its architecture. This chapter would provide a good foundation for the understanding of the rest of thesis.

2.1 Unified Modeling Language (UML)

The Unified Modeling Language (UML) [42] is the standard visual modeling language used for modeling businesses, software application, and system architecture. In other words, it includes a standardized graphical notation that may be used to create an abstract model of a system: the UML model [25].

While the UML is used to specify, visualize, construct, and document software-intensive systems, it is not restricted to modeling software. It has its strengths at higher, more architectural levels and has been used for modeling hardware and

is commonly used for business process modeling, systems engineering modeling, and representing organizational structure. In UML, a *diagram*, is the layout and visualization of different modeling elements as described within the UML. Each UML diagram is used for specific purposes, typically to visualize a certain aspect of your system.

The UML contains two different basic diagram types: structure diagrams and behavior diagrams. Structure diagrams depict the static structure of the elements in your system. The various structure diagrams are as follows [25]:

- Class diagrams
- Component diagrams
- Object diagrams
- Deployment diagrams
- Composite Structure
- Package diagrams

Behavior diagrams depict the dynamic behavior the elements in your system. The various behavior diagrams are as follows [25]:

- Activity diagrams
- Use case diagrams

- Statechart diagrams
- Collaboration diagrams
- Sequence diagrams
- Timing diagrams
- Interaction overview diagrams

In our approach we mainly use class diagrams and sequence diagrams and they are explained below.

2.1.1 Class Diagrams

Class diagrams are a type of static structure diagrams that describes the structure of a system by the showing the system's classes and the relationships between them. A class in the software system is represented by a box with the name of the class written inside it. A compartment below the class name can show the class's attributes. Each attribute is shown with at least its name, and optionally with its type, initial value, and other properties. The class's operations (i.e. its methods) can appear in another compartment. Each operation is shown with at least its name, and optionally also with its parameters and return type. Different types of associations in a class diagram is used to show the relationship between classes.

2.1.2 Sequence Diagrams

In contrast to class diagrams, sequence diagrams can depict the dynamic behavioral aspects of the system being modeled. It is used to show the processes that execute in sequence. The sequence diagram shows the sequence of messages, which are exchanged among roles that implement the behavior of the system, arranged in time. It shows the flow of control across many objects that collaborate in the context of a scenario.

2.2 Abstract State Machines (ASM)

Abstract State Machines (ASM) [3] is a specification method for software and hardware modeling, where a system is modeled by a set of states and transition rules which specifies the behavior of the system. Transition rules specify possible state changes according to a certain condition. The notation of ASM is efficient for modeling a wide range of systems and algorithms as the number of case studies demonstrates [22].

2.2.1 ASM Language

The ASM notation includes *static functions* and *dynamic functions*. Static functions have the same interpretation in every state, while dynamic functions may change their interpretation during a run. There are also *external functions* which cannot

be changed by the system itself, but by the outer environment.

2.2.2 States

An ASM model consists of states and transition rules. States are given as many sorted first-order structures, and are usually described in terms of functions. A structure is given with respect to a signature. A signature is a finite collection of function names, each of a fixed arity. The given structure fixes the syntax by naming sorts and functions, and provides carrier sets and a suitable symbol interpretation on the carrier sets, which assigns a meaning to the signature. So a state can be defined as an algebra for a given signature with *universes* (*domains* or *carrier sets*) and an interpretation for each function symbol.

States are usually described in terms of functions. The notion of ASM includes *static functions*, *dynamic functions* and *external functions*.

- ***Static functions*** have a fixed interpretation in each computation state: that is, static functions never change during a run. They represent primitive operations of the system, such as operations of abstract data types (in software specifications) or combinational logic blocks (in hardware specifications).

- ***Dynamic functions*** whose interpretation can be changed by the transition occurring in a given computation step, that is, dynamic functions change during a run as a result of the specified system's behavior. They represent the internal state of the system.

- ***External functions*** whose interpretation is determined in each state by the environment. Changes in external functions which take place during a run are not controlled by the system, rather they reflect environmental changes which are considered uncontrollable for the system.

- ***Derived functions*** whose interpretation in each state is a function of the interpretation of the dynamic and external function names in the same state. Derived functions depend on the internal state and on the environmental situation (like the output of a Mealy machine). They represent the view of the system state as accessible to an external observer.

2.2.3 Terms

Variables and *terms* are used over the signature as objects of the structure. The syntax of terms is defined recursively, as in first-order logic:

- A variable is a term. If a variable is Boolean, the term is also Boolean.

- If f is an $r-ary$ function name in a given vocabulary and $t_1, \ldots t_r$ are terms, then $f(t_1, \ldots t_r)$ is a term. The composed term is Boolean if f is relational.

2.2.4 Locations and Updates

States are described using functions and their current interpretations. The state transition into the next state occurs when its function values change. *Locations* and

updates are used to capture this notion.

A *location* of a state is a pair of a dynamic function symbol and a tuple of elements in the domain of the function. For changing values of locations the notion of an *update* is used. An *update* of state is a pair of a location and a value. To fire an update at the state, the update value is set to the new value of the location and the dynamic function is redefined to map the location into the value. This redefinition causes the state transition. The resulting state is a successor state of the current state with respect to the *update*. All other locations in the next state are unaffected and keep their value as in the current state.

2.2.5 Transition Rules

Transition rules define the changes over time of the states of ASMs. While terms denote values, transition rules denote *update sets*, and are used to define the dynamic behavior of an ASM. ASM runs starting in a given initial state are determined by a closed transition rule declared to be the *program*. Each next state is obtained by firing the update sets at the current state. Basic transition rules are *skip*, *update*, *block*, and *conditional rules*.

The *skip* rule is the simplest transition rule. This rule specifies an "empty step". No function value is changed. It is denoted as

skip

The *update* rule is an atomic rule denoted as

$$f(t_1, t_2, \ldots, t_n) := t$$

It describes the change of interpretation of function f at the place given by $(t_1, t_2, \ldots, t_n)$ to the current state value of t.

A *block* rule is a group of sequence of transition rules. The execution of a block rule is the simultaneous execution of the sequence of the transition rules. All transition rules that specify the behavior of the ASM are grouped into a block indicating that all of them are fired simultaneously.

```
block
    R1
    R2
endblock
```

In *conditional rules* a precondition for updating is specified.

if g

then R_1 else R_2

endif

where g is a first order Boolean term. R_1 and R_2 denote arbitrary transition rules. The condition rule is executed in state S by evaluating the guard g, if *true* R_1 fires, otherwise R_2 fires.

2.3 Abstract State Machine Language (AsmL)

Abstract State Machine Language (AsmL) [18] is an executable specification language based on the theory of ASM. It is fully object-oriented and has strong mathematical constructs in particular, sets, sequences, maps and tuples as well as set comprehension, sequence comprehension and map comprehension. ASMs steps are transactions, and in that send AsmL programming is transaction based. AsmL is fully integrated into the .NET framework and Microsoft development tools providing inter-operability with many languages and tools. Although the language features of AsmL were chosen to give the user a familiar programming paradigm, the crucial features of AsmL, intrinsic to ASMs are massive synchronous parallelism and finite choice. These features give rise to a cleaner programming style than is possible with standard imperative programming languages. Synchronous parallelism and inherently AsmL provide a clean separation between the generation of new values and the committal of those values into the persistent state.

AsmL 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 the .NET framework. AsmL effectively supports specification and rapid prototyping of different kinds of models. The AsmL tester [30] can also be used for FSM generation or test case generation.

2.4 Property Specification Language (PSL)

Property Specification Language (PSL) [1] is an implementation independent language to define properties (also called assertions). It does not replace, but complements existing verification methodologies like VHDL and Verilog test benches. PSL presents a different view of the design and may imply FSMs in the implementation. The syntax of PSL is very declarative and structural which leads to sustainable verification environments. Both VHDL and Verilog flavors are provided. PSL consists of four layers based on the functionality:

The Boolean layer to build expressions which are used in other layers, specifically the temporal layer. Boolean expressions are evaluated in a single evaluation cycle.

The temporal layer is used to describe properties of the design, in addition to simple properties, this layer can describe properties that involve complex temporal relations. Temporal expressions are evaluated over a series of evaluation cycles.

The verification layer is used to tell the verification tool what to do with the properties described by the temporal layer.

The modeling layer is used to model behavior of design inputs for formal verification tools, and to model auxiliary parts of the design that are needed for verification.

This layered approach allows the expressing of complex properties from simple primitives. A property is built from three types of building blocks: Boolean expressions, sequences, which are themselves built from Boolean expressions, and finally

subordinate properties. Sequences, referred to as SEREs (Sequential Extended Regular Expressions), are used to describe a single or multicycle behavior built from Boolean expressions. In this thesis, we use only the first three layers of PSL.

2.5 SystemC

SystemC is an open standard controlled by a steering group composed of thirteen major companies in the Electronic Design Automation (EDA) industries [32]. It is one of the most promising system-level design languages and has been recently standardized by IEEE (IEEE 1666) [41] for system-level chip design. It has evolved in response to a pervasive need for a language that improves overall productivity for designers of electronic systems. SystemC offers real productivity gains by letting engineers design both the hardware and software components together at a high level of abstraction. This gives the design team a fundamental understanding early in the design process of the intricacies and interactions of the entire system and enables better system trade-offs, better and earlier verification, and overall productivity gains through reuse of early system models as executable specifications.

SystemC provides mechanisms crucial to modeling hardware while using a language environment compatible with software development. It offers several hardware-oriented constructs that are not normally available in a software language but are required to model hardware. All of the constructs are implemented within the context of the C++ language. The major hardware-oriented features implemented

within SystemC include:

- Timed models
- Hardware data types
- Module hierarchy to manage structure and connectivity
- Communications management between concurrent units of execution
- Concurrency models

The SystemC language architecture is shown in Figure 2.1. The language is built on top of standard C++. The first layer of shaded gray blocks are part of the so called core layer (or layer 0) of the standard SystemC language. The second shaded gray layer immediately after the kernel layer is the layer 1 of SystemC; it comes with a predefined set of interfaces, ports and channels. Finally, the layers of design libraries above the layer 1 are considered separate from the SystemC language. The user may choose to use them or not. Over time other standard- or methodology-specific libraries may be added and conceivably be incorporated into the standard language.

Figure **??** depicts a simplified metamodel [35] of the main SystemC terms and abstractions. SystemC has a notion of a container class, called *module*, that provides the ability to encapsulate structure and functionality of hardware/software blocks for partitioning system designs. A system is essentially broken down into a containment hierarchy of *modules*. Each module may contain *variables* as simple data members,

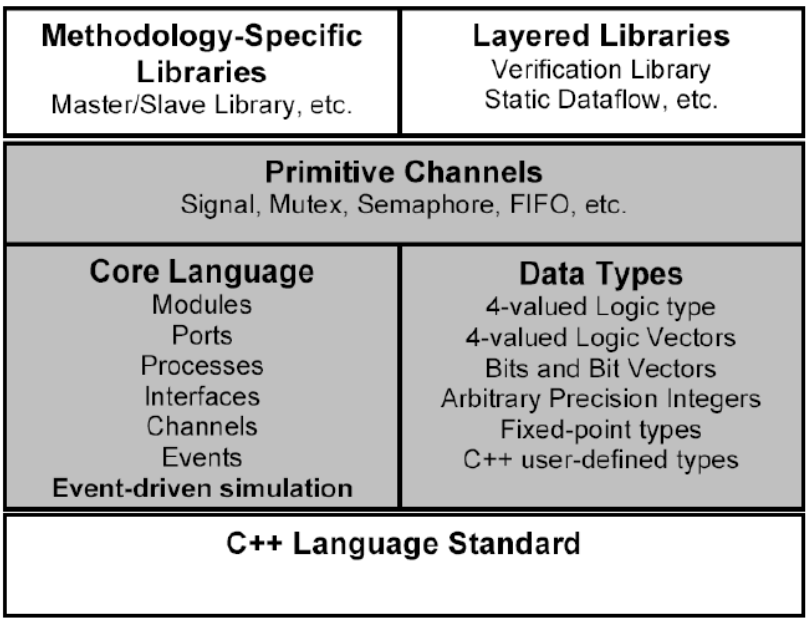

Figure 2.1: SystemC Language Architecture [17].

ports for communication with the outside environment and *processes* for performing modules functionality and expressing concurrency in the system. Three kinds of processes are available: *method* processes, *thread* processes, *clocked thread* processes. They run concurrently in the design and may be sensitive to events which are notified by *channels*. A port of a module is a proxy object through which the process accesses a channel interface. The *interface* defines the set of access functions (methods) for a channel, while the channel provides the implementation of these functions to serve as a container to encapsulate the communication of blocks. There are two kinds of channels: *primitive* channels and *hierarchical* channels. Primitive channels do not exhibit any visible structure, do not contain processes, and cannot (directly) access other primitive channels. A hierarchical channel is a module, i.e., it can have

structure, it can contain processes, and it can directly access other channels.

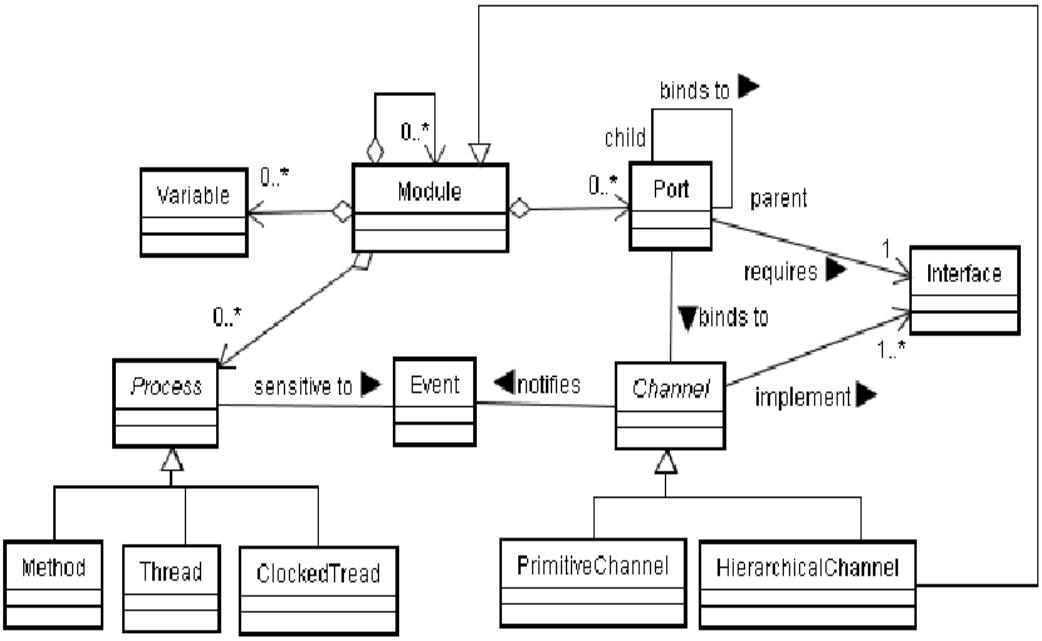

Figure 2.2: A Simplified SystemC MetaModel [35].

As mentioned earlier, SystemC permits to model a system at different levels of abstraction: functional untimed, functional timed, transactional, behavioral, bus cycle accurate (BCA) and register transfer level (RTL) [17]. These description levels may be freely mixed in the same design, however, hardware modeling is made at RTL or behavioral level. Typically, hardware modeling in SystemC is obtained by restricting the description style to the following rules: (i) modeling the communication by signals; (ii) modeling the functionality by processes sensitive to clocks and signals; (iii) use of modules to describe hierarchical designs.

Chapter 3

PCI-X Bus Modeling

In this chapter, we elucidate the PCI-X architecture and its modeling using the methodology that we presented in Chapter 1. It includes UML modeling, AsmL modeling and translation of AsmL to SystemC of the PCI-X.

3.1 PCI-X Bus

Improvements in processors and peripheral devices have caused conventional PCI technology to become a bottleneck to performance scalability. The introduction of the PCI-X technology [40] has provided the necessary bandwidth and bus performance needed to avoid the I/O bottleneck, thus achieving optimal system performance. For instance, version 2.0 of PCI-X specifies a 64-bit connection running at speeds of 66, 133, 266, or 533 MHz, resulting in a peak bandwidth of 533, 1066, 2133 or 4266 MB/s, respectively.

PCI-X provides backward compatibility by allowing devices to operate at conventional PCI frequencies and modes. Moreover, PCI-X peripheral cards can operate in a conventional PCI slot, although only at PCI rates and may require a 3.3 V conventional PCI slot. Similarly, a PCI peripheral card with a 3.3 V or universal card edge connector can operate in a PCI-X slot, however the bus clock will remain at a frequency acceptable to the PCI card. Figure 3.1 shows the general architecture of PCI-X with one initiator (master) and target (slave) and in Table 3.1, we show a list of PCI-X signals and it meanings. There is an arbiter that performs the bus arbitration among multiple initiators and targets. Unlike the conventional PCI bus, the arbiter in PCI-X systems monitors the bus in order to ensure good functioning of the bus.

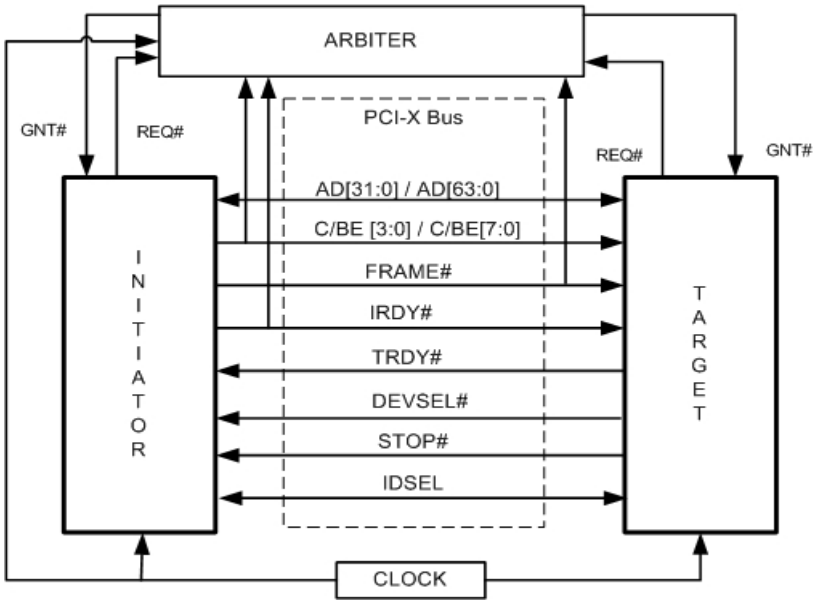

Figure 3.1: General Architecture of PCI-X.

Table 3.1: PCI-X Signals

Signals	Meanings
AD	Address of the data read or write
C/BE	Specifies the type of the command
FRAME#	Transaction initiation
IRDY#	Signals the readiness of initiators
TRDY#	Signals the readiness of targets
DEVSEL#	Enables to allow minimum decoding speed
STOP#	Signals the termination of the transaction
IDSEL#	Configuring transactions timing

Both PCI-X initiator and target have a pair of arbitration lines that are connected to the arbiter. When an initiator requires the bus, it asserts REQ#. If the arbiter decides to grant the bus to that initiator, it asserts GNT#. FRAME# and IRDY# are used by the Arbiter to decide the granting of an initiator request for the bus. Unlike PCI, the targets can only insert wait states by delaying the assertion of TRDY#. TRDY# and IRDY# have to be asserted for a valid data transfer. An initiator can abort the transaction either before or after the completion of the data transfer by de-asserting the FRAME# signal. In contrast, a target can terminate a bus transaction by asserting STOP#. If STOP # is asserted without any data transfer, the target has issued a retry and if STOP# is asserted after one or more data phases, the target has issued a disconnect. Unlike PCI, the target has also REQ# and GNT# that are connected to the arbiter. This facilitates the Split

Transaction of PCI-X which does not exist in conventional PCI. In Split Transaction, the initiators and targets switch their roles. Split Transaction is very useful if a target can not be able to continue the current transaction. In this case, the target will memorize the transaction and signal a Split- telling the initiator not to retry IO read. When the data is ready, the target will send the initiator a Split Completion containing the data. The addition of PCI-X Split Completion frees up the bus for other transactions, making PCI-X more efficient than PCI. This notion of split transaction of PCI-X and its high bandwidth capacity makes the PCI-X bus pretty complex.

PCI-X supports two modes of operations: Mode 1 and Mode 2. Mode 1 operates at either 66 or 133 MHz and uses a parity error checking scheme and Error Correction Code (ECC) as optional. In Mode 1 operation, data transfers always use common clock. The higher transfer rates (266 or 533 MHz) in PCI-X 2.0 are defined in Mode 2, which uses ECC as its error correcting scheme. Mode 2 operation of PCI-X also supports 16 bit bus interface which facilitates low cost interface. Split Transactions in PCI-X replace Delayed Transactions in conventional PCI [40]. If a target cannot complete the transaction within the target initial latency limit, the target must complete that transaction as Split Transaction. If the target meets its initial latency limits, it can optionally complete the transaction immediately. All these aforementioned transactions' sequences are described vividly using sequence diagrams in Section 3.2.

3.2 UML Modeling

UML representation is helpful for the programmers who talk different languages. From the specification of PCI-X, we identify the core components of the bus viz, initiators, targets, arbiters, PCI-X bus, which will be represented as classes, where specific instances of the components are called as objects. In addition to these four components, we also added another component, the Simulation Manager (SimManager), in order to have a notion of updates. Figure 3.2 shows these five classes, where each has its own data members and methods with their access types. It also shows the relationship among each classes. For instance, the relationship between the arbiter and the initiators is one-to-many (1 - *). It is because there can be only one arbiter which is connected to many initiators. For another example, the relationship between the SimManager and arbiter is one-to-one (1 - 1) as there can be only one SimManager and one arbiter in the system.

We modeled different modes and types of operations of PCI-X using sequence diagram. Sequence diagrams enable us to model the bus in AsmL easily and efficiently. Also, it helps to extract the properties of the system being modeled, which can be used to verify using model checking. In addition, UML modeling helps to close the gap between informal specification and formal models in AsmL.

In Figure 3.3, we show the protocol sequence of a typical Mode 1 transaction of the PCI-X using a sequence diagram. Figure 3.3 is a best case scenario of Mode 1 transactions, with one initiator and one target and without any wait states. In the

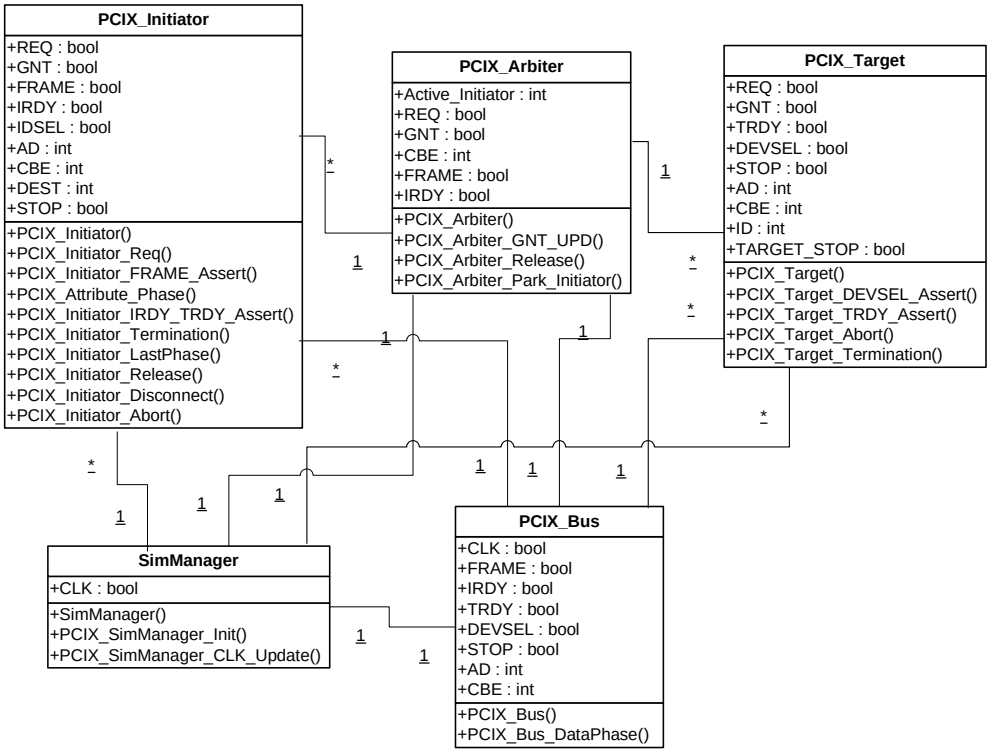

Figure 3.2: Class Diagram of PCI-X.

first clock cycle, the initiator asserts the REQ# signal to get the control of the bus. The arbiter asserts the GNT# signal to that initiator. In the third clock cycle, the address phase takes place and also the FRAME# signal is asserted by the initiator to signal the start of the transaction. In the next clock cycle, the attribute phase takes place, where additional information included with each transaction is added. In clock cycle N+4, the DEVSEL# signal is asserted by the target and in the next clock cycle, the data phase is started with the assertion of the IRDY# and TRDY# signals by the initiator and the target, respectively. Before the last data phase, the FRAME# signal is de-asserted to signal the completion of the data transfer and in the termination phase all the other signals are de-asserted. In order to represent the

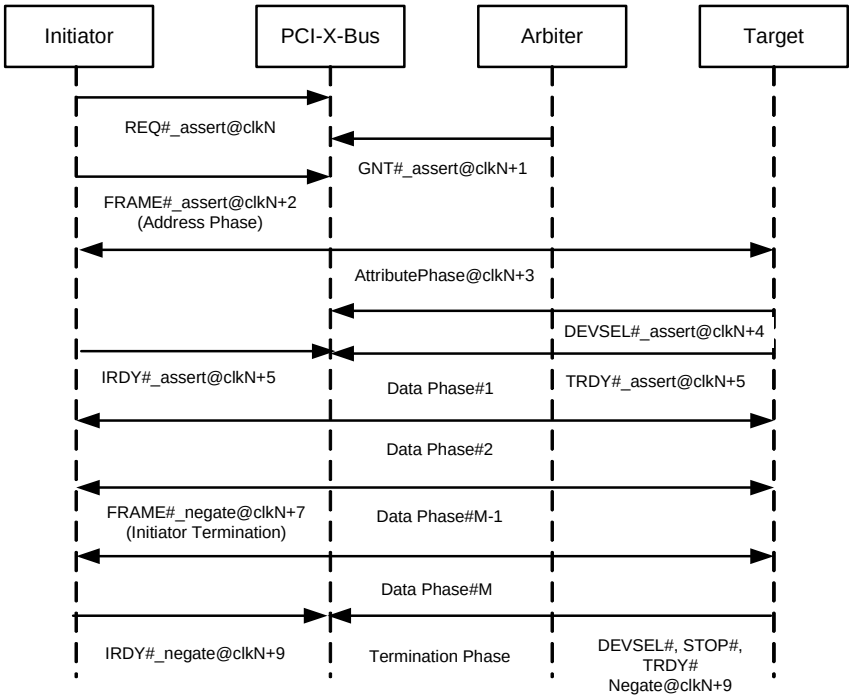

Figure 3.3: Sequence Diagram of Mode 1 Transaction.

clock constraints of the PCI-X transaction, we added an additional operator, "@", to specify at which clock cycle a particular action should occur.

Figure 3.4 shows the protocol sequence of a typical Mode 2 transaction of PCI-X using a sequence diagram. Mode 2 transaction is pretty similar to Figure 3.3, except that there is an additional delay of one clock cycle (*Target Response Phase*)and one additional idle clock between any two transactions.

In Figure 3.5, we show the transaction sequences of a 16 bit interface of Mode 2 transaction. In this transaction, the attribute phase takes two cycles unlike the transaction types. Other signal activities of this transaction are the same as the generic Mode 2 transaction. In the coming section, we present the AsmL modeling of PCI-X from UML.

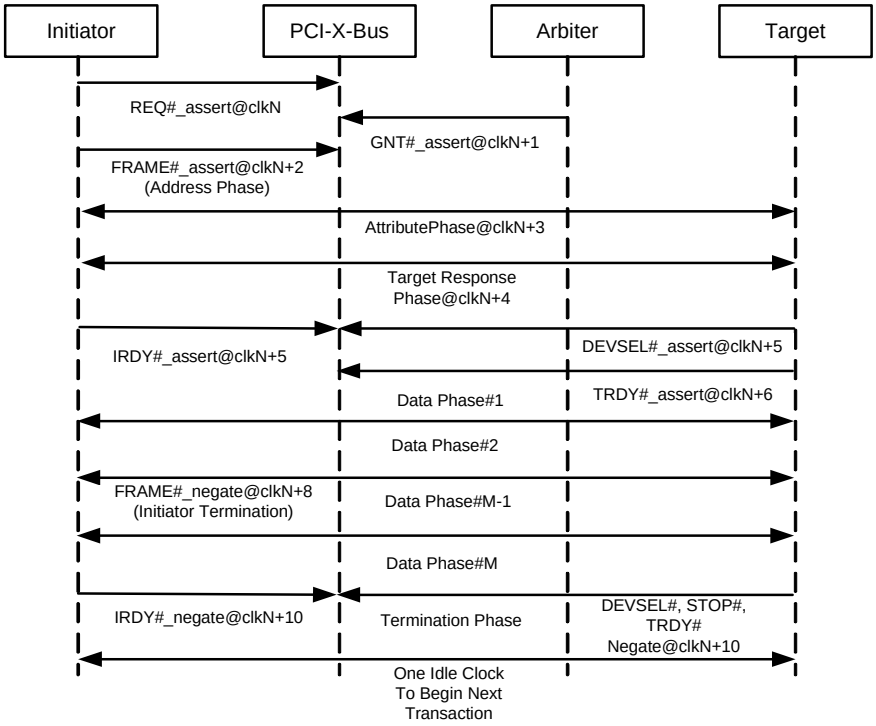

Figure 3.4: Sequence Diagram of Mode 2 Transaction.

3.3 AsmL Modeling

In this section, we show our approach of modeling the PCI-X in AsmL from UML representation by showing some of the important *methods* used. By having class diagrams in UML, it is easy to implement the classes in any language. We use AsmL's class features to model all the five core components of PCI-X discussed in the previous section. Each of these has its own data members (signals) and methods (behavior) in addition to the constructors. We also use enumeration features (*enum*) of AsmL to model different modes of PCI-X, different types of transaction phases, the state of the system and the clock (see Figure 3.6).

In addition, we exhaustively use the *require* and "*:=*" statements of AsmL in

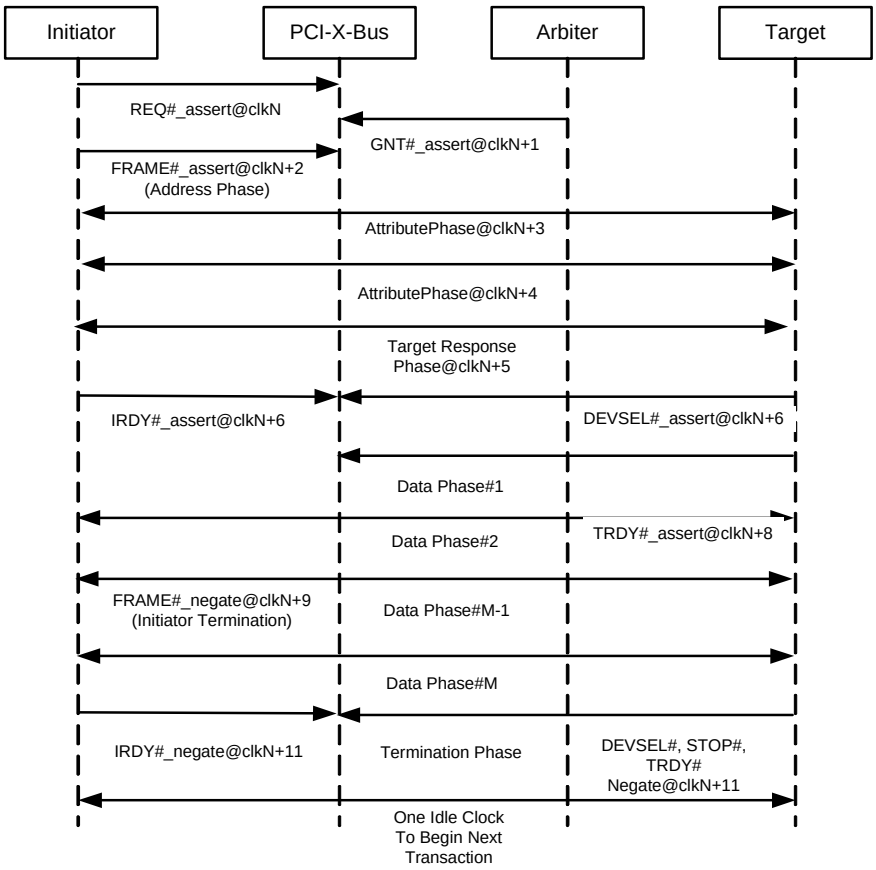

Figure 3.5: Sequence Diagram of Mode 2 Transaction (16 bit).

our design approach. *require* is the pre-condition statement in AsmL used to check if a certain condition is satisfied in order to enable a method, and the operator "*:=*" represents an update statement used to change the system state. Figure 3.7 shows the AsmL model of the arbiter grant method (*PCIX_Arbiter_GNT()*). As the name of the method says it acts as an arbiter for granting the bus to the requesting initiator. In order to grant the bus to the requested initiator, this method has the following pre-conditions (*require*) to be met: (1) there must be at least one initiator requesting the bus and that initiator has not been granted the bus at the time of

```
enum SystemStatus          enum Clock
 CLK_UP                      INIT
 CLK_DOWN                    STARTED
enum Transaction_Phase     enum Transaction_Mode
 IDLE_PHASE                  MODE_1
 ADDR_PHASE                  MODE_2
 ATTR_PHASE
 TGT_RES_PHASE
 DATA_PHASE
 INR_TER_PHASE
 TURN_AROUND_PHASE
```

Figure 3.6: Used AsmL Enumeration Types.

the request; (2) the clock is on the rising edge; and (3) the mode can be either Mode 1 or Mode 2. If these pre-conditions are met, the arbiter updates the GNT# signal.

```
public PCIX_Arbiter_GNT()
  require (exists x in Initiators where x.REQ =  true and
          x.GNT = false) and  me.GNT = false and Smanager.CLK = CLK_UP
            and (Mode = MODE_1 or Mode = MODE_2)
  me.Active_Initiator := min y | y in Initiators_Range where
           (Initiators(y).REQ = true)
  me.GNT := true
  Initiators(Active_Initiator).GNT := true
```

Figure 3.7: Arbiter GNT AsmL Method.

In Figure 3.8, we show how a target can signal its readiness using TRDY# signal. We call this method as *PCIX_Target_TRDY_Assert()*. The pre-conditions are the following: TRDY# is *false*, FRAME# and DEVSEL# are *true*, CLK is *CLK_UP*, Phase is *DATA_PHASE_FIRST* and the *AD* of the Bus should be the *ID* of the target. If the pre-conditions are true, then TRDY# will be asserted.

```
public PCIX_Target_TRDY_Assert()
    require me.TRDY = false and Bus.FRAME = true and
            Bus.AD = me.ID and Bus.DEVSEL = true and
            Smanager.CLK = CLK_UP and Phase = DATA_PHASE_FIRST
    me.TRDY := true
    me.AD := Bus.AD
    Bus.TRDY := true
    Phase := DATA_PHASE
```

Figure 3.8: Target Assert AsmL Method.

Figure 3.9 shows the initiator termination method (*PCIX_Initiator_Termination*). This method basically signals the end of a transaction if *BYTECOUNT* is less than 2. *BYTECOUNT* indicates the number of bytes of data transfer in a transaction. This signaling is done by de-asserting the FRAME# signal and updating the transaction phase to the initiator termination phase (INI_TER_PHASE). This method has a specific pre-conditions that need to be true so that it can terminate the transactions. The pre-conditions (*require*) are the following: initiator's REQ#, GNT#, FRAME#, IRDY#, DEVSEL#, TRDY# are asserted, *BYTECOUNT* is less than 2, and the clock is on the rising edge. If all the above pre-conditions are true, this method updates the FRAME# signal to false and the phase to *INI_TER_PHASE*. After this FRAME# signal de-assertion, the initiator's last phase method (*PCIX_Initiator_LastPhase()*) is invoked.

Figure 3.10 shows the clock update method. It is a method of the SimManager class, which checks if the system and simulation are started in order to update the clock.

```
public PCIX_Initiator_Termination()
  require me.GNT = true and me.REQ = true and me.FRAME = true and
          me.IRDY = true and Bus.TRDY = true and Bus.DEVSEL = true
          and BYTECOUNT < 2 and me.STOP = false and
          Smanager.CLK = CLK_UP
  me.FRAME := false
  Bus.FRAME := false
  Phase := INR_TER_PHASE
```

Figure 3.9: Initiator Termination AsmL Method.

```
 public PCIX_SimManager_CLK_Update()
    require SystemFlag = STARTED and SimStatus = RUNNING
    if CLK = CLK_UP then
        CLK := CLK_DOWN
    else
        CLK := CLK_UP
```

Figure 3.10: Clock Update AsmL Method.

Figure 3.11 shows the data phase method of the AsmL code. The method decreases the *BYTECOUNT* by 1 based on the following pre-conditions: IRDY#, DEVSEL#, TRDY# are asserted, the phase has the value *DATA_PHASE*, the clock is on the rising edge, and the system started running.

```
public PCIX_BUS_DataPhase()
  require SystemFlag = STARTED and SimStatus = ON_INIT and
          Bus.IRDY = true and Bus.TRDY = true and Bus.DEVSEL = true
          and Smanager.CLK = CLK_UP and Phase = DATA_PHASE
  BYTECOUNT := BYTECOUNT - 1
```

Figure 3.11: Data Phase AsmL Method.

3.4 Translation to SystemC

After the ASM model of PCI-X is verified against the PSL properties, we translate it to SystemC as this is the ultimate goal. The verification of the AsmL using model checking is discussed in the next chapter. The transformation of AsmL to SystemC is syntactical based on a set of rules developed in [19] and summarized as follows:

- Basic types: AsmL basic types are mapped to their equivalent SystemC types (e.g. *Integer* to *int*, *Byte* to *unsigned char*)
- Class members are translated into SystemC signals having the same basic type. (e.g. *var val as Integer* to *sc_signal <int> val*)
- Class methods in AsmL contain two parts: first one defining the pre-/post conditions for its invocation and the second is the method itself. The first part is integrated in SystemC module's *constructor*. The method itself is integrated as is in the SystemC module.
- Global methods are integrated in the SystemC's main procedure *sc_main*.

To illustrate the translation procedure, we show a sample code in Figure 3.12, the *Arbiter* class of the PCI-X in SystemC. All classes in AsmL modeling are represented as module (*SC_MODULE*) in SystemC and the data members in AsmL are mapped as signals (*sc_signal*) in SystemC. It is also possible to represent them as ports (*sc_port*). The methods in AsmL class are represented as *SC_METHOD*.

```
SC_MODULE (PCIX_Arbiter) {

  sc_signal <int> Active_Initiator
  sc_signal <bool> REQ
  sc_signal <bool> GNT
  sc_signal <int>  CBE
  sc_signal <bool> FRAME
  sc_signal <bool> IRDY
  sc_signal <bool> STOP

  SC_CTOR (PCIX_Arbiter) {

    Active_Initator = 0;
    REQ = false;
    GNT = false;
    CBE = 0;
    FRAME = false;
    IRDY = false;
    STOP = false;

    SC_METHOD(PCIX_Arbiter_REQ_UPD);
    sensitive << Initiators.REQ << Initators.GNT
              << REQ << GNT << Smanager.CLK;
    SC_METHOD(PCIX_Arbiter_GNT_UPD);
    sensitive << REQ << GNT << Smanager.CLK;
    SC_METHOD (PCIX_Arbiter_Release);
    sensitive << Initiators(me.Active_Initiator).REQ
              << Initiators(me.Active_Initiator).GNT
              << REQ << GNT << Smanager.CLK;
  }
};
```

Figure 3.12: Arbiter Class Declaration in SystemC.

SC_METHOD is one type of modeling process in SystemC. In some sense, it is similar to the Verilog *always@* block or the VHDL process. The SystemC simulation engine usually calls the *SC_METHOD* process repeatedly based on the static or dynamic sensitivity. Static sensitivity establishes the parameters during elaboration (i.e., before simulation begins). It is established with a call to the *sensitive()* method or the overload stream operator << that is placed just following the registration of a process.

Dynamic sensitivity can also be used by means of the *next_trigger()* method. It is somewhat similar to the traditional *wait* statement in HDLs. In our translation procedure, we use the static sensitivity scheme and the stream operator << for modeling the pre-condition statement (*require*) of AsmL. The method registration with sensitivity list is done in the constructor method (*SC_CTOR*).

In Figure 3.12, we first show the data members of the class using *sc_signal*. Then, we show the constructor definition of the class (*SC_CTOR (PCIX_Arbiter)*). In the constructor, we first do the initialization of the data members which is one of the purposes of having constructor in OO modeling. After that, we include the method registration using *SC_METHOD* with the use of static sensitivity (*sensitive*) in order to fire that particular method.

The correctness of the above translation rules from AsmL to SystemC and vice versa, has been provided in [19] using abstract interpretation [10].

Chapter 4

Model Checking

In the last chapter, we have seen the AsmL modeling of a PCI-X model. In this chapter, we are going to define a set of properties in PSL in order to verify the AsmL model through model checking. The properties are easy to obtain from the sequence diagrams of the UML representation and also from the specifications. This chapter also includes the model checking results and its limitation.

4.1 Background

Model checking [9] is an FSM based formal verification technique that can be used to determine the validity of the properties written in some temporal logic with respect to a behavioral model of a system but it is impossible to determine whether the given specification covers all the properties that the system should satisfy. Model checking is based on the state space exploration technique, and uses the reachability state

graph as a Kripke structure, which encodes the set of all possible sequences of states for a system over computation trees. Model checking tools are effective as debugging aids for industrial designs, and since they are fully automated, minimal user effort and knowledge about the underlying technology is required to be able to use them. However, there are two drawbacks with model checking. The first drawback is the state space explosion and how to avoid it, and the second is the difficulty of judging whether the verified property completely characterizes the desired behavior of the system.

4.2 Model Checking Approach

The way we perform model checking is different than the classical approach. In the classical approach, the properties and the design being modeled are given separately to the model checking tool. But, in our approach, once the modeling of PCI-X in AsmL is done, we include the defined PSL properties in the AsmL design. Then, the AsmL design with the properties is fed to Asmlt tester that generates an FSM of the model. Using Asmlt, we perform state exploration which gives the notion of model checking. We encode the properties evaluation in every state, which enables checking its correctness on-the-fly while executing the FSM generation algorithm [15] (part of the Asmlt). Any incorrect property detection halts the reachability algorithms and outputs a sub-portion from the complete FSM, which can be used to identify counter-examples.

Properties are embedded in every state in the FSM generated by the Asmlt and is represented by two Boolean state variables P_{eval} and P_{val}. P_{eval} represents whether a property can be evaluated or not and P_{val} denotes the state of the property in the current state. A violated property is detected if $P_{eval} = true$ $P_{val} = false$. A correct verification process results on the generation of the system's FSM. If the property is not verified, then an error trace would be generated. In the case of state explosion, the FSM generation will become unsuccessful. In the next section, we show the properties that we defined.

4.3 Model Properties

We define various properties of the PCI-X bus in PSL. The properties are obtained from the sequence diagrams and the informal specification. Some of the properties that we show are generic to any bus protocols and others are specific to PCI-X. The first property *P1* shown in Figure 4.1, is a common behavior for any bus standard. It states that if an initiator is requesting the bus *(!Initiator.REQ == true)*, it will eventually be granted *(!Initiator.GNT == true)*. This also makes sure the fact that no initiator will be using the bus indefinitely. It is to be noted that all signals of PCI-X are active-low.

Property *P2* in Figure 4.2 is about the termination of a PCI-X transaction be it Mode 1 or Mode 2. It means that if an initiator is terminating the bus by asserting the *STOP* signal, then eventually the bus will be released by de-asserting the

```
Property P1:
  forall Initiator in {Initiator0, ..., Initiator4}
    if(!Initiator.REQ == true) then
      eventually (!Initiator.GNT == true)
```

Figure 4.1: Property *P1*.

FRAME signal and targets will be released by de-asserting *TRDY* and *DEVSEL*.

```
Property P2:
  forall Initiator in {Initiator0, ..., Initiator4}
    if((!Initiator.STOP == true) and
       (!Initiator.GNT == true)) then
      eventually {(!Bus.FRAME == false) and
        forall Target in {Target0, ..., Target4}
          (!Target.TRDY == false) and
          (!Target.DEVSEL == false)}
```

Figure 4.2: Property *P2*.

```
Property P3:
  forall Initiator in {Initiator0, ..., Initiator4}
    if(!Initiator.GNT == true) then
      next[0] (!Initiator.FRAME == true)
```

Figure 4.3: Property *P3*.

Property *P3* in Figure 4.3 is for the assertion of *FRAME* signals. This property is important for the start of the transaction. If an initiator is granted to the bus, then in the next clock cycle the *FRAME* signal has to be asserted.

Property *P4* (Figure 4.4 is to check the phase *(ADDR_PHASE, ATTR_PHASE, IDLE_PHASE)* of the transaction. If the initiator FRAME signal is asserted, then in the next clock cycle, the phase of the transaction will be the attribute phase (ATTR_PHASE).

```
Property P4:
  forall Initiator in {Initiator0, ..., Initiator4}
    if(!Initiator.FRAME == true) then
      next[0] (Transaction_Phase == ATTR_PHASE)
```

Figure 4.4: Property *P4*.

```
Property P5:
  forall Initiator in {Initiator0, ..., Initiator4}
  forall Target in {Target0, ..., Target4}
   if((!Target.GNT == true) and
       (!Initiator.DEST == Target.ID)) then
    eventually {(!Bus.FRAME == true) and
        (!Initiator.TRDY == true) and
        (!Target[ID].TRDY == true) and
        (!Target.GNT == false)}
```

Figure 4.5: Property *P5*.

Property *P5* in Figure 4.5 is regarding the arbitration of the bus. If an initiator is selected by the arbiter, then it will be able to get access to the bus by setting *!Bus.FRAME*. Then, its destination target will be activated by setting its *!Target.TRDY* and the initiator will release the bus once *!Initiator.GNT* is set to false.

Property *P6* in Figure 4.6 is to check the initiation of a transaction termination when data transfer is about to be completed. If $BYTECOUNT$ is equal to one (data transfer is in the penultimate cycle), then in the current cycle $FRAME$ will be de-asserted.

Property *P7* (Figure 4.7 is somewhat similar to P6. The property says that if the data transfer is over *(BYTECOUNT == 0)*, then in the next clock cycle, initiator's $IRDY$ will be de-asserted.

```
Property P6:
  forall Initiator in {Initiator0, ..., Initiator4}
     {if(BYTECOUNT == 1) then
           next[0] (!Initiator.FRAME == false) }
```

Figure 4.6: Property *P6*.

```
Property P7:
  forall Initiator in {Initiator0, ..., Initiator4}
    always {if(BYTECOUNT == 0) then
            next (!Initiator.IRDY == false) }
```

Figure 4.7: Property *P7*.

Property *P8* in Figure 4.8 is to check the response of a target when it has been selected as destination. This property is specific for Mode 1 transaction. The property says that if the transaction mode is MODE_1 and the initiator's FRAME is asserted, then eventually the *IRDY* and *TRDY* of the initiator and target respectively, will be asserted.

```
Property P8:
  forall Initiator in {Initiator0, ..., Initiator4}
  forall Target in {Target0, ..., Target4}
    if((!Initiator.FRAME == true) and
      (Transaction_Mode == MODE_1)) then
      eventually {(!Initiator.IRDY == true) and
        (!Target.TRDY == true)}
```

Figure 4.8: Property *P8*.

Property *P9* in Figure 4.9 is similar to *P8* but it is related to the Mode 2 transaction. If *FRAME* is asserted and transaction is Mode 2 then eventually *IRDY* and *DEVSEL* will be asserted together.

Property *P10* shown in Figure 4.10 is about the target response property of

```
Property P9:
  forall Initiator in {Initiator0, ..., Initiator4}
  forall Target in {Target0, ..., Target4}
    if(!Initiator.FRAME == true) and
      (Transaction_Mode == MODE_2) then
      eventually {(!Initiator.IRDY == true) and
        (!Target.TRDY == true)}
```

Figure 4.9: Property *P9*.

Mode 2 transaction. If *FRAME, IRDY, DEVSEL* are asserted and transaction is Mode 2, then in the next clock cycle, *TRDY* will be asserted.

```
Property P10:
  forall Initiator in {Initiator0, ..., Initiator4}
  forall Target in {Target0, ..., Target4}
    if(!Initiator.FRAME == true) and
      (!Initiator.IRDY == true) and
      (!Target.DEVSEL == true) and
      (Transaction_Mode == MODE_2) then
       next (!Target.TRDY == true)
```

Figure 4.10: Property *P10*.

Property *P11* in Figure 4.11 checks the idle phase of Mode 2 transaction after a transaction has completed. It says that if *IRDY, STOP, DEVSEL, TRDY* are de-asserted then in the next clock cycle, the phase of the transaction will be idle. In other words, the bus will be idle.

Property *P12* in Figure 4.12 is about the initiation of Split transaction. In split transaction, the target can request for the bus *(!Target.REQ == true)* and it can be eventually granted *(!Target.GNT == true)*.

```
Property P11:
  forall Initiator in {Initiator0, ..., Initiator4}
  forall Target in {Target0, ..., Target4}
    if((!Initiator.IRDY == false) and
      (!Initiator.STOP == false) and
      (!Target.DEVSEL == false) and
      (!Target.TRDY == false) and
      (Transaction_Mode == MODE_2)) then
       next (Transaction_Phase == IDLE_PHASE)
```

Figure 4.11: Property *P11*.

```
Property P12:
  forall Target in {Target0, ..., Target4}
    if(!Target.REQ == true) and
      (Transaction_Mode == SPLIT) then
      eventually (!Target.GNT == true)
```

Figure 4.12: Property *P12*.

4.4 Experimental Results

Table 4.1 details the results of model checking of PCI-X model (5 initiators and 5 targets). In the table, we show the CPU time, number of states and transitions for the PCI-X model with the various properties that we defined. The experiments were performed on a Pentium IV processor (2.4 GHz) with 768 MB of memory. As can be seen from the table, all the properties have been verified except Property 6 and Property 7 due to a state explosion problem. This is due to the fact the these properties are related to the successful completion of a data transfer. When the *BYTECOUNT* is large, the action *PCIX_BUS_DataPhase()* will be enabled many times and would lead to state explosion. This is a classical problem of model checking and many techniques have been proposed in the recent past to alleviate this

Table 4.1: Model Checking Results

Property	CPU Time (s)	States	Transitions
P1	385.24	2169	3250
P2	194.23	1800	2563
P3	150.52	1578	2156
P4	130.45	1489	2096
P5	156.35	1478	2265
P6	–	–	–
P7	–	–	–
P8	173.50	1925	2439
P9	174.47	2013	2698
P10	178.42	1873	2359
P11	256.63	2192	2980
P12	143.52	1356	1923

problem but not to eradicate. Even though, the CPU time to verify the properties is relatively shorter, we also have to consider the time spent to write the properties and to learn the tool.

Chapter 5

Generating FSM from SystemC

In the last chapter, the model checking of the PCI-X in AsmL model was shown. Once the model checking of the AsmL model is done, we translate it to SystemC. In order to the validate the SystemC model, we provide two FSM generation algorithms from which MBT can be applied. In this chapter, we provide the formalization of these two algorithms.

5.1 Background

The work in [15] derives an FSM from the ASM languages like AsmL. The FSM-generating algorithm is a particular kind of graph reachability algorithm. It starts from the initial state and builds up a labeled state transition graph by invoking actions that are parts of the ASM. The FSM generation allows them to integrate with the existing tools (Asmlt) to derive test suites to achieve model based testing.

In [5], the FSM generation algorithm is extended from [15] to have compact FSMs using the notion of state groupings. The work in [5] considers both AsmL and Spec# as the source language. Nevertheless, these two works can be only applicable for ASM languages. For SystemC, these algorithms can not be used directly as its syntax is different and the notion of SystemC simulator has to be considered.

To tackle this issue, we provide a formalization of two algorithms to generate FSM from SystemC. These two algorithms are extended from [15, 5]. We adapt the work in [15, 5] based on the existing SystemC semantics [19, 14] to generate FSM from SystemC. At first, the given SystemC design is analyzed to collect the necessary inputs for the FSM generation. Then, we apply one of the two algorithms that we propose to generate FSMs. We call the first *Direct Algorithm* and the second one as *Grouping Algorithm*. Both algorithms perform a state exploration procedure in order to discover all possible system's state starting from a set of initial states. The grouping algorithm's exploration procedure is very similar to the direct one but it has grouping conditions as an additional input. More details about grouping algorithm will be discussed in Section 5.4.3. In the coming sections, we detail the SystemC syntactical domains that are needed to collect the inputs for the algorithms.

5.2 SystemC Syntactical Domain

The SystemC language has a large number of syntactical domains. These, however, are based on the single `SC_Module` domain [19]. Hence, the minimum representation

for a general SystemC design is a set of modules.

In SystemC, the structural decomposition is specified with modules, which are the basic building blocks. A SystemC description consists of a set of connected modules, each encapsulating some behavior or functionality. Modules can be hierarchical, containing instances of other modules. The nesting of hierarchy can be arbitrarily deep, which is an important requirement for structural design representation. The formal definition of an `SC_Module` is given in the following:

Definition 5.2.1. SystemC Module: (`SC_Module`)
A SystemC Module is a tuple ⟨`DMem`, `Ports`, `Chan`, `Mth`, `SC_Ctr`⟩, where `DMem` is a set of the module data members, `Ports` is a set of ports, `Chan` a set of SystemC channels, `Mth` is a set of methods (functions) definition and `SC_Ctr` the module constructor.

The simplest means of connecting together different SystemC modules is by using ports (`SC_Port`) and signals. Formally, a SystemC port is a tuple ⟨`IF`, `N`, `SC_In`, `SC_Out`, `SC_InOut`⟩, where `IF` is a set of the virtual methods declarations, `N` is the number of interfaces that may be connected to the port, `SC_In` is an input port (provides only a `Read` method), `SC_Out` is an output port (provides only a `Write` method) and `SC_InOut` is an input/output port (provides `Read` and `Write` functions).

In contrast to default class constructors for OO languages, the SystemC module constructor `SC_Ctr` contains the information about the processes and threads that will be executed during simulation, and their sensitivity lists, `SC_SL`, specifying

which events can affect their states. SC_Ctr is defined as a tuple ⟨Name, Init, SC_Pr, SC_SSt⟩, where Name is a string specifying the module name, Init is a default class constructor, SC_Pr a set of processes and SC_SSt is a set of sensitivity statements (to set the process sensitivity list SC_SL). The following definition provides the formal description of SC_Pr.

Definition 5.2.2. SystemC Process (SC_Pr)

A SystemC process is a tuple ⟨PMth, PTh, PCTh⟩, where PMth is a method process (defined as a set ⟨Mth, SC_SL⟩ including the method and its sensitivity list), PTh is a thread process (accepts a wait statement in comparison to the method process), PTh is a clocked thread process (sensitive to the clock event).

A SystemC design is a connection of multiple modules (SC_Mod) [19]. Restricting our model to modules does not affect the validity of the results since modules are the default syntactical domain for SystemC. All other domains are built on top of it.

Definition 5.2.3. SystemC Design (SC_Design)

A SystemC design is a tuple ⟨L_{SC_Mod}, SC_main⟩, where L_{SC_Mod} is a set of SystemC modules and SC_main is the main function in the program that performs the simulator initialization and contains the modules declarations.

5.3 Collecting Inputs

The original SystemC design is analyzed to collect the needed components in order to feed the algorithm with the information required for the FSM generation. Table 5.1 shows the relation between the SystemC design and our algorithms inputs. At first, we collect the processes `SC_Pr` in the SystemC design and add to actions (A). Then, data members and ports (`SC_Port`) are considered as state variables for the FSM generation algorithm. Finally, actions pre-conditions (A_{pre}) are collected from the sensitivity lists of the processes in SystemC designs. In addition to these three entities, the grouping algorithm requires as input a set of grouping conditions (G_C).

Table 5.1: Link between the SystemC Design and the FSM Generation Algorithm Inputs.

Algorithm Inputs	SystemC Design: `SC_Design`
Actions: A	Set of all the processes in `SC_Design`: $\{\texttt{sc\_pr}, \texttt{sc\_pr} \in \texttt{SC\_Design}\}$.
State Variables: V	Set of all data members and ports in `SC_Design` $\{\texttt{dmem}, \exists \texttt{sc\_mod} \in \texttt{SC\_Design} \mid \texttt{dmem} \in \texttt{sc\_mod}\}$ $\cup \{\texttt{port} \mid \exists \texttt{sc\_mod} \in \texttt{SC\_Design} \mid \texttt{port} \in \texttt{sc\_mod}\}$
Action's preconditions: A_{pre}	Set of sensitivity lists of all the processes in `SC_Design`: $\{\texttt{sc\_sl}, \exists \texttt{sc\_mod} \in \texttt{SC\_Design} \mid$ $\exists \texttt{sc\_pr} \in \texttt{sc\_mod} \mid \texttt{sc\_sl} \in \texttt{sc\_pr}\}$

5.4 FSM Generation Algorithm

The FSM generation algorithms need the following entities:

- *State Variables (V)*
- *State Space (S)*
- *Initial States (S_{init})*
- *Actions (A)*
- *Transition Relation (R)*

V is a set of state variables and for each v_i in V, there is a corresponding domain d_{vi} in D, where D is a set of domains for every type of state variables. $S = \{s_1, s_2, s_3...s_n\}$ is a total state space of the design being modelled where each state s_i in S is an instantaneous description of the system that captures the value of state variables (V) at particular instant of time. $S_{init} \subseteq S$ is a set of initial states from where the state exploration starts. $A = \{a_1, a_2, a_3...a_n\}$ is a set of actions in the model defined as follows.

Definition 5.4.1. *Action* (a)

a is a four-tuple $\langle a\_M, a\_Pre, a\_Post, a\_Cst \rangle$ where,

- a_M is a method
- a_Pre is a set of pre-conditions

- *a_Post* is a set of post-conditions
- *a_Cst* is a set of constraints

a_Pre is a set of Boolean propositions that have to be true in the beginning of an action $a_i \in A$ execution, in a state s. *a_Post* is also a set of Boolean propositions that must be true at the end of an action $a_i \in A$ execution and *a_Cst* is a set of Boolean propositions that must to be true at certain part of an action $a_i \in A$ execution. Next, we define the transition, R, from one state to another during the action execution and it is defined as follows:

Definition 5.4.2. Transition Relation (R)

Let S be a set of states and Abe a set of actions then the transition relation R is defined as

$R : S \times A \to S$

$(s_{current}, a) \mapsto s_{next}$

where $s_{current}$ is the current state and s_{next} is the next state obtained after executing the action a

During the exploration phase of the algorithm, relevant states are stored in S_{FSM}. Starting from S_{init}, new discovered states are added to S_{FSM} together with the new transition T. $T = \{t_1, t_2, t_3...t_n\}$ is a set of transitions included in the FSM where t_i is a three-tuple, $\langle s_{current}, a, s_{next} \rangle$.

5.4.1 Helper Functions

Some helper functions are needed for the FSM generation algorithms. They include:

- *enabled* - used to know for a specific state s if an action $a \in A$ is enabled in it
- *nonExp* - used to know if a state s is fully explored.
- *Sort* - used to sort the actions in A based on the the SystemC simulator semantics

Definition 5.4.3. *enabled*

Let S be a set of states and A be a set of actions, then the *enabled* function is defined as:

$S \times A \rightarrow \{true, false\}$

$(s, a) \mapsto enabled(s, a)$

where:

$$enabled(s,a) = \begin{cases} true, & (a\_Pre = true); \\ false, & (a\_Pre = false) \end{cases}$$

Definition 5.4.4. *nonExp*

Let S be a set of states and T be a set of transitions, then the *nonExp* function is defined as:

$nonExp(s)$:$S \rightarrow T \times T \times ... \times T$

$s \mapsto \{ \ (t_1, \ t_2, \ ... \ t_n) \mid ((t_i = \langle \ s, a, R(s,a) \ \rangle \ ) \wedge (enabled(s,a) = true))\}$

Definition 5.4.5. *Sort*()

Let A be a set of actions, $Sort(A)$ uses the $SC\_Simulator()$ given in [14] after initializing the system with the values in the current system's state. It extracts then the list of actions to be considered in the exploration from the list of active processes provided by the SystemC simulator.

Definition 5.4.6. *Frontier* (F)

Let S be a set of states and A be a set of actions, then the Frontier $F = \{s \mid s \in S \mid \exists a \; enabled(s, a) = true\}$ is a set of states that have not yet been fully explored during the FSM generation process. The exploration of the state space stops, if F is empty. Initially, F contains S_{init}.

5.4.2 Direct FSM Generation

The direct FSM generation algorithm performs a state exploration procedure in order to discover all possible system states starting from a set of initial states. It starts from initial states and constructs a labelled state transition graph by invoking actions that are parts of the design. If a new state is encountered, it is added to the *Frontier* of unexplored states. The output of the algorithm are two sets that correspond to states S_{FSM} and transition T which form the FSM.

In the direct algorithm of Figure 5.1, we first sort the actions A based on the SystemC simulation semantics using the helper functions *Sort* (line 4). The

```
1:  S_FSM = {S_init}
2:  F = {S_init}
3:  T = {∅}
4:  Sort(A)
5:  while(F ≠ ∅) {
6:     current := F.Head
7:     foreach a ∈ A{
8:        if(enabled(current, a)) {
9:           next := R(current,a)
10:          if (next ∉ S_FSM) {
11:             S_FSM := S_FSM ∪ {next}
12:             T := T ∪ {(current, a, next)}
13:             if (exists a in A where enabled(next,a) = true) {
14:                F := F ∪ {next}
15:             }
16:          }
17:          elseif ((current,a,next) ∉ T) {
18:             T := T ∪ {(current, a, next)}
19:          }
20:    }}
21:     F := F.Tail
22:    Sort(A)
23: }
```

Figure 5.1: Direct FSM Generation Algorithm.

algorithm starts exploring the states if the Frontier (F) is not empty (line 5). In the beginning, F contains S_{init}. For each action $a_i \in A$, the algorithm checks if an a_i is enabled in the current state (*current*) using a helper function *enabled* (line 8). Thereafter, the new state (*next*) is discovered using the transition relation R (line 9). If the new discovered state (*next*) is not in S_{FSM}, it is added to S_{FSM} and the transition t (*current,a,next*) is added to T (lines 11 and 12). The new state *next* is also added to F if there exists an action enabled in this new state. If *next* is already in S_{FSM}, t is still added to T if it does not exist in it. The algorithm terminates

once the F becomes empty (line 5).

5.4.3 Grouping FSM Generation

We extend the algorithm presented in [5] to generate FSM using the notion of state grouping for SystemC designs. State grouping is a technique for controlling scenarios by selecting representative states with respect to an equivalent class. Also, it is an efficient way to prune exploration to distinct cases of interest for testing, in particular to avoid exploring symmetric configuration. Typically, the state space is very large and it is always important to prune it as much as possible. The state grouping technique is one of the potential candidates for this problem. It uses a state-based grouping condition to group the states. The grouping condition G_C is a Boolean proposition that uses state variables (V) and functions defined in the design. It is defined as follows.

Definition 5.4.7. Grouping Condition (G_C)

Let v_1, v_2, ... v_n be a set of state variables in V, then a grouping condition G_C is defined as:

$G_C : V \rightarrow B$

$$(v_1, v_2, ..., v_n) \mapsto G_C(v_1, v_2, ..., v_n) \in \{true, false\}$$

Having defined G_C, we need to define the grouped state s_g which is a set of states that are equivalent under one grouping condition G_{Ci}, formally defined as follows.

Definition 5.4.8. Grouped State (s_g)

Let S be a set of states, G_C be a set of grouping condition then the grouped state s_g is defined as:

$s_g = \{s \mid s \in S \mid G_{Ci} \ = true\}$

In order to group the states, we need to have a way to map the states s to grouped state s_g. This is achieved by a grouping function *g*. g maps states of the model to concrete values defined by a state-dependent grouping condition (G_C). The value *g(s)* is called the *g-label* of s. Two states are *g-equivalent* if they have the same *g-label*. g is defined as follows.

Definition 5.4.9. Grouping Function (g)

Let S be a set of states, S_G be a set of grouped states then grouping function g is defined as:

$g : S \rightarrow S_G$

$\quad s \mapsto s_g$

R_g is transition relation that provides the transition of one grouped state to another during the action execution. Unlike R where it deals with individual state s, R_g deals with grouped state s_g and is defined as follows.

Definition 5.4.10. Transition Relation in Grouped FSM (R_g)

Let S_G be a set of grouped states and A be a set of actions then the transition relation R_G is defined as:

$$R_g : S_G \times A \rightarrow S_G$$

$$(s_g^{current}, a) \mapsto s_g^{next}$$

where $s_g^{current}$ is the current grouped state and s_g^{next} is the next grouped state obtained after executing the action a.

The output of the grouping algorithm will be FSM_G that has two entities viz., S_{FSM}^G and T_G. S_{FSM}^G is a set of grouped states that have been discovered and T_G is a set of transitions among the grouped states in the generated FSM.

1: $S_{FSM}^G = \{\emptyset\}$
2: $F = \{S_{init}\}$
3: $T_G = \{\emptyset\}$
4: $g = \{g1, g2...gk\}$
5: $Sort(A)$
6: $while(F \neq \emptyset)$ {
7: $current := F.Head$
8: $foreach\ a \in A$ {
9: *if (enabled (a, current))* {
10: $next := R(current,a)$
11: *if* $(g(next) \notin S_{FSM}^G)$ {
12: $S_{FSM}^G := S_{FSM}^G \cup \{g(next)\}$
13: $T_G := T_G \cup \{(current, a, g(next))\}$
14: *if (exists a in A where enabled(next,a) = true)* {
15: $F := F \cup \{next\}$
16: }
17: }
18: *elseif* $((current,a,next) \notin T)$ {
19: $T_G := T_G \cup \{(current, a, g(next)\}$
20: }
21: }}
22: $F := F.Tail$
23: $Sort(A)$
24: }

Figure 5.2: Grouping FSM Generation Algorithm.

Like the direct FSM generation algorithm in Figure 5.1, this grouping algorithm starts with F containing S_{init} and then explores all the possible actions in A that can be enabled in the current state (*current*) (line 9). Then, the new state (*next*) is discovered using the transition relation R (line 10). Thereafter, the new discovered state is mapped to a grouped state based on the grouping function g which in turn depends on the state-dependant grouping condition G_C. If the new grouped state (*g(next)*) is not in S^G_{FSM}, it is added to it and the transition (*current,a,g(next)*) is added to T_G. The new state *next* is also added to F to explore further. At last, the algorithm terminates when F is empty.

5.5 Discussion

Both FSM generation algorithms provided perform an exploration of the state space. The first one, considering the current state explores all possible actions that can be enabled. It collects all possible new states and links them to existing ones using the actions as transition labels. The second one uses the same exploration procedure, however, it maps the states into groups having the same value for a specific grouping condition. For this reason, the second algorithm is more complex and is expected to require little more CPU execution time than the direct one.

The advantage of using the grouping algorithm in comparison to the direct one is the reduction of the number of states in the final FSM. For instance, the number of grouped states will depend on the grouping condition. For example, considering

a condition with five possible values will result in an FSM with at most five states. This grouping notion of such an algorithm is suitable and important for SystemC in particular when it comes to grouping all the actions for a specific SystemC module (*sc_module*) [31]. This way the whole model of the module will be represented as a single node in the final FSM.

Chapter 6

Model-Based Testing

In the last chapter, we showed the formalization of two FSM generation algorithm in order to perform MBT of PCI-X model in SystemC. In this chapter, we make an earnest attempt to verify the PCI-X model using MBT technique at the SystemC level. MBT is an evolving technique mainly being used for software testing. But, it has as its roots in hardware testing too. In order to facilitate the MBT technique, it is inevitable to have a model of the system. We prefer to use FSM as our model. For this reason, we presented two FSM generation algorithms for SystemC in previous chapter. Using the generated FSM model, the test cases are generated by traversing the FSM using available techniques such as Chinese Postman Tour [36] and Randomwalk [36].

6.1 Background

Model-based testing (MBT) is a general term that signifies an approach that bases common testing tasks such as test case generation and test result evaluation on a model of the system under test. MBT has as its roots applications in hardware testing, most notably telephone switches, and recently has spread to a wide variety of software domains. The wealth of published work portraying case studies in both academic and industrial settings is a sign of growing interest in this style of testing [16]. Specific steps of the MBT are the following:

1. Build the model
2. Generate expected inputs
3. Generate expected outputs
4. Run tests
5. Compare actual outputs with expected outputs
6. Decide on further actions (whether to modify the model, generate more tests, or stop testing)

Build the model:

Forming a mental representation of the systems functionality is a prerequisite to building a model for testing purposes. Testers need to understand not only the system, but also the environment in which it operates. The model should be a

depiction of the systems behavior, which can be described in terms of the input sequences accepted by the system; the actions, conditions, and output logic; or the flow of data through the applications, modules, and routines. There are many formal modeling techniques (ways to depict behavior) from which to choose. A variety of techniques/methods exist for expressing models of system behavior. These include, but are not limited to [12]:

1. **Decision Tables -** Tables used to show sets of conditions and the actions resulting from them

2. **Finite State Machines -** A computational model consisting of a finite number of states and transitions between those states, possibly with accompanying actions

3. **Grammars -** describe the syntax of programming and other input languages

4. **Markov Chains (Markov process)-** A discrete, stochastic process in which the probability that the process is in a given state at a certain time depends only on the value of the immediately preceding state

5. **Statecharts-** Behavior diagrams specified as part of the Unified Modeling Language (UML). A statechart depicts the states that a system or component can assume, and shows the events or circumstances that cause or result from a change from one state to another.

Finite State Machines and Markov chains are the two most popular techniques in MBT for modeling system behavior [12]. Finite State Machines can ensure that generated test cases cover the model. When a Markov chain model is used, a random process generates test cases, making coverage criteria more difficult to ensure in some specified number of test cases. The mathematics of Markov chains, however, provides analytical formulas to determine expected values useful in test planning.

Generate expected inputs:

Using the model, we need to generate test cases, which consist primarily of specifying the inputs and expected outputs. The difficulty of generating tests depends on the nature of the model. In the case of finite state machines, it is a matter of traversing the state transition diagram (a directed graph) [37]. Tests are, by definition, the sequence of inputs along the generated paths. Thus, if the model is well defined, the tests can be generated automatically.

Generate expected outputs:

MBT involves execution of a program under test using some fault-revealing input data and examination of the output to determine success or failure. A fundamental assumption of this testing is that there is some mechanism, a test oracle, that will determine whether or not the results of a test execution are correct, something that defines/identifies the expected outputs. A test oracle is the criterion used to check the correctness of the output.

Run tests:

Most MBT environments are supported with test generation tools that generate test cases which can easily be translated into executable test scripts, or produce the test script directly from the test data contained within the tool. Although tests can be run as soon as they are created, in most testing groups it is policy to run the tests only after a complete suite that meets certain adequacy criteria is generated. Typically, there is a coverage plan that is being addressed. In some instances, only a small number of tests relating to a particular feature or component would be run, even though the complete suite has been generated.

Compare actual with expected outputs:

In this step, the comparison of actual to expected outputs is performed, and tester are alerted about the failures. This is dependent on the quality and completeness of the test oracle. MBT cannot make good information out of bad data. It should provide an efficient means to drill down into the particular test cases that failed.

In our approach, we first generate the FSM of the PCI-X model in SystemC based on the algorithms that we formalized in the previous chapter. Once the FSM is generated, we apply existing test generation techniques to generate test cases.

6.2 Direct Algorithm

We use the direct algorithm that we presented in Section 5.4.2 to generate the FSM of the PCI-X design. We generate the FSM for various combinations of initiators and targets to show the robustness of our approach.

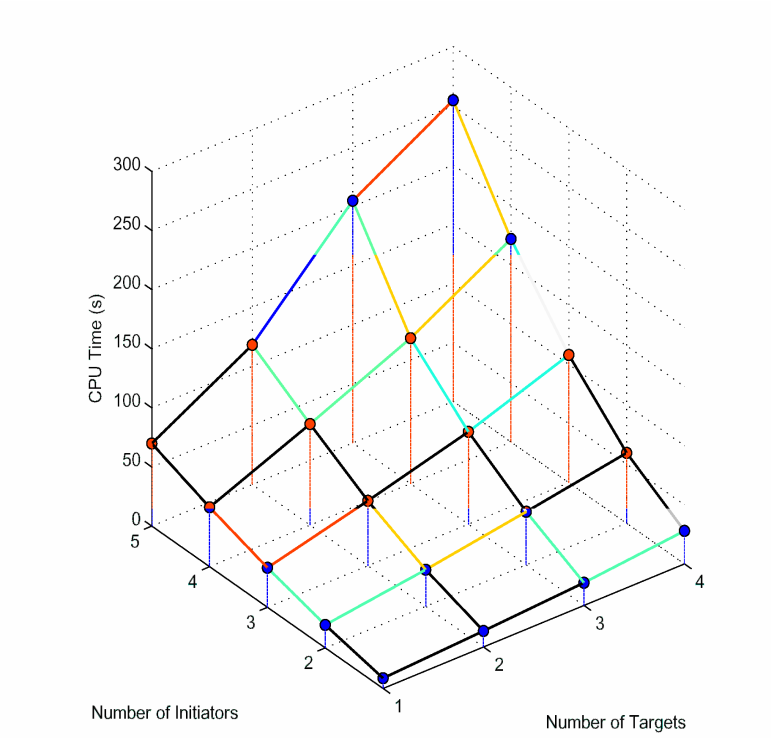

Figure 6.1: Direct Algorithm Results.

Table 6.1 shows the CPU time, states and number of transitions in the generated FSM. From the table, we notice that as the number of initiators and targets increases, the states and the transitions increase and of course the CPU time too. In order to infer more on the results we have plotted a 3-D graph in Figure 6.1

Table 6.1: FSM Generation: Direct Algorithm.

Number of		CPU	States	Transitions
Initiators	Targets	Time (s)		
1	2	8.75	96	103
1	3	13.60	142	153
1	4	19.21	188	203
1	5	27.95	234	253
2	2	19.42	190	205
2	3	30.98	282	305
2	4	44.93	374	405
2	5	59.50	466	505
3	2	33.43	284	307
3	3	54.90	422	457
3	4	78.21	560	607
3	5	108.04	698	757
4	2	50.25	378	409
4	3	85.54	562	609
4	4	123.01	746	809
4	5	171.73	930	1009
5	2	69.89	472	511
5	3	118.20	702	761
5	4	204.93	932	1011
5	5	254.82	1162	1261
10	10	2925.31	4622	5021

involving CPU time, initiators and targets. From the graph, it can be inferred that the increase in CPU time is more for the increase in the number of targets than the increase in the number of initiators. This is due the fact that as the number of targets is larger, the choice of selecting a target is many and this will lead to the exploration of many paths.

6.3 Grouping Algorithm

Using the grouping algorithm that we presented in Section 5.4.3, we define some grouping condition in order to group the states. This approach enables us to group the states and test a particular part of the system. We apply these grouping algorithm for various numbers of initiators and targets of the PCI-X model.

Grouping Conditions

In the following, we show some grouping conditions that are used to group the states using the grouping algorithm presented Chapter 5. In Figure 6.2, a grouping condition is shown. *G1()* groups the individual states based on the conjunction of the *destination target* and the *final status of the transaction* (completed or stopped). The procedure *G1()* returns an integer value in {0, 1, 2, 3} or -1 when an error happens. Each of these values identifies a grouped state. Therefore, considering the definition of *G1()*, the maximum number of grouped states is four. The variable *Initiators* in Figure 6.2 refers to the set of the initiators connected to the bus. The integer data member *DEST* identifies for the owner's object (initiator) the target destination for the current transaction. Finally, the Boolean data member *STOP* specifies if the transaction can be stopped by the target before it is fully completed.

Figure 6.3 shows the grouped FSM of the PCI-X using the grouping condition *G1* defined in Figure 6.2 for the case of two initiators and two slaves. For example, group 0 (node *G0* in Figure 6.3) corresponds to the case when the destination

```
Grouping Condition G1:
 G1() as Integer
 if ((exists x in  Initiators where x.DEST = 1 and
      x.STOP = true)= true) then
        return 0
 else
   if ((exists x in  Initiators where x.DEST = 1 and
        x.STOP = false) = true) then
          return 1
   else
     if ((exists x in  Initiators where x.DEST = 2 and
          x.STOP = true) = true) then
            return 2
     else
       if ((exists x in Initiators where x.DEST = 2 and
            x.STOP = false) = true) then
             return 3
       else
            return -1
```

Figure 6.2: Grouping Condition *G1*.

target is *Target1* and the transaction succeeds. In Table 6.2, we show the grouping algorithm results of PCI-X model using the grouping condition *G1*. From the table, we note that an increasing trend in CPU time as we increase the number of targets like in the direct algorithm. In addition, for all the cases, the value of CPU time of grouping algorithm is relatively higher than the direct algorithm. This is because the grouping algorithm has the notion of grouping conditions and this facilitates the mapping of individual states to a grouped states. However, as expected the grouped FSM is smaller than the original 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

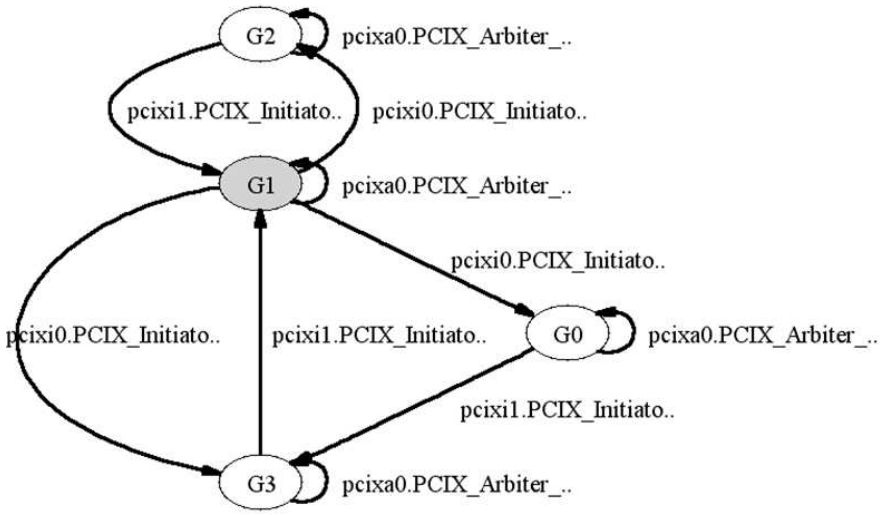

Figure 6.3: Grouped FSM Using *G1*.

targets and two possible termination conditions for the transaction (succeeded or stopped). Figure 6.4 shows a 3D-Plot involving initiators, targets, and the CPU time of Grouped FSM generation results using G1. 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.

Next, we present another grouping condition (*G2()*) in Figure 6.5 that groups the states based on the phase of the transaction. There are nine phases in a typical PCI-X transaction. As it can be seen in the Figure 6.5, the maximum number of grouped states can be 10 including the error state. This grouping condition is common for all the number of initiators and targets unlike the previous one. The grouped FSM using *G2* is shown in Figure 6.7.

Grouping Condition 3 (G3()) in Figure 6.6 groups the states based on the mode of operation of PCI-X. As it is mentioned earlier, PCI-X has two modes of

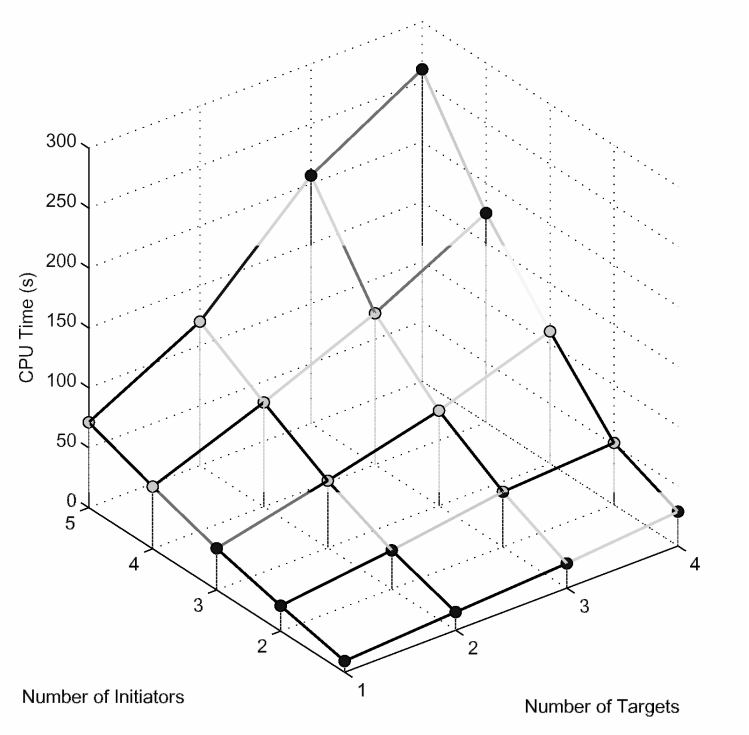

Figure 6.4: Grouping Algorithm Results (*G1*).

operation and eventually the grouped FSM will have three grouped states including an error states.

In Figure 6.9, we show a grouping condition that groups the states based on the status of the *SystemFlag* variable. The system can be in one of the three states (*INIT, STARTED, IDLE*). From this grouping condition, the maximum number of states that we have is 3 and the grouped FSM is shown Figure 6.10.

```
Grouping Condition G2:
 G2() as Integer
    if (Phase = IDLE_PHASE) then return 0
    else
      if (Phase = ADDR_PHASE) then return 1
      else
        if (Phase = ATTR_PHASE) then return 2
        else
          if (Phase = TGT_RES_PHASE) then return 3
          else
            if (Phase = DATA_PHASE_FIRST) then return 4
            else
              if (Phase = DATA_PHASE) then return 5
              else
                 if (Phase = INR_TER_PHASE) then return 6
                 else
                    if (Phase = LAST_PHASE) then return  7
                     else
                       if (Phase = TURN_AROUND_PHASE) then return 8
                        else return -1
```

Figure 6.5: Grouping Condition *G2*.

6.4 Test Generation

Using the generated FSM, we can use various techniques to choose which paths we want our tests to take through it. FSM traversing is the same as traversing the directed graphs. There are several efficient graph traversing techniques in the open literature. We don't provide any new technique for traversing the FSM and it is also not the scope of the thesis. In our approach, we consider the Chinese Postman Tour (CPT) method and Randomwalk .

Table 6.2: FSM Generation: Grouping Algorithm (using *G1*).

Number of		CPU	States	Transitions
Initiators	Targets	Time (s)		
1	2	9.37	4	10
1	3	15.12	6	14
1	4	20.56	8	18
1	5	28.65	10	26
2	2	21.18	4	10
2	3	32.35	6	14
2	4	46.02	8	18
2	5	51.69	10	26
3	2	34.78	4	10
3	3	56.01	6	14
3	4	79.35	8	14
3	5	110.35	10	26
4	2	51.98	4	10
4	3	87.05	6	14
4	4	126.27	8	14
4	5	174.72	10	26
5	2	71.25	4	10
5	3	120.12	6	14
5	4	206.84	8	18
5	5	260.36	10	26
10	10	2952.56	20	58

6.4.1 Chinese Postman Tour (CPT)

Finding the shortest route for a traveling salesman, who wishes to visit every city, is a well known problem. Less well known is the Chinese postman who wishes to travel along every road to deliver letters. The Chinese Postman Problem (CPP) is interesting because it has many applications, is a simply-stated problem, but for which there is no simple algorithm. There are many variations to the CPP, most notably whether the roads are one-way (this is the Directed CPP or DPP)

```
Grouping Condition G3:
 G3() as Integer
  if (MODE = 1) then
    return 0
  else
    if (MODE = 2) then
      return 1
    else
       return -1
```

Figure 6.6: Grouping Condition *G3*.

and whether the postman has to return back to where they started (closed or open CPP). A postman delivering letters in a village may wish to know a circuit that traverses each street (in the appropriate direction if one-way streets), starting and returning to their office. This is a graph theoretic problem: roads are directed edges (transitions), and road junctions are vertices (states). The postman requires a Chinese Postman Tour(CPT). In summary, CPT touches every action in the state model as efficiently as possible [37].

6.4.2 RandomWalk

Randomwalk produces test cases that consist of a single sequence of invocations that start in the initial state. At each state one of the outgoing transition will be randomly selected. 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 testers would not bother to try.

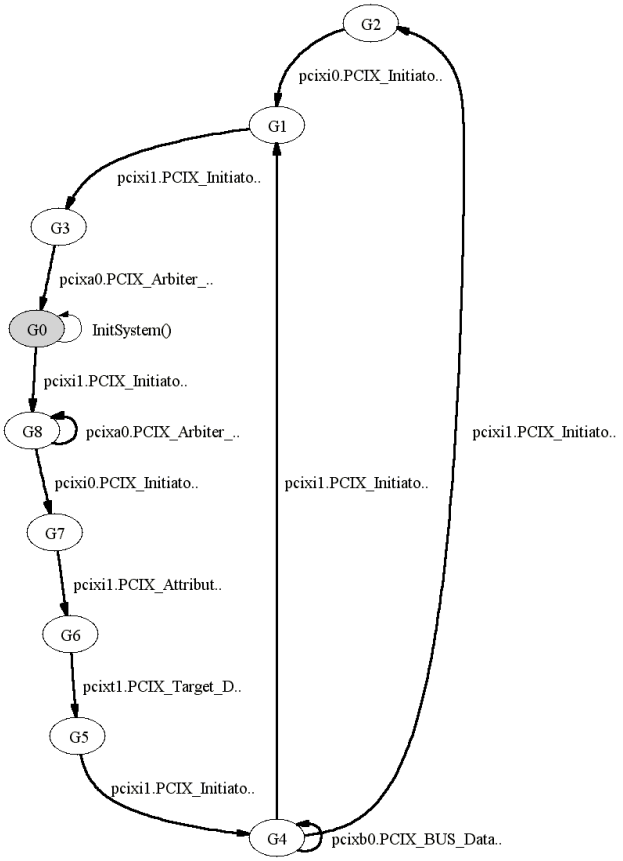

Figure 6.7: Grouped FSM Using *G2*.

6.4.3 Test Sequences

Figure 6.3 shows a grouped FSM of PCI-X (2 initiators and 2 targets) using the grouping condition *G1* which is defined in Figure 6.2. In Figure 6.11, we provide a possible test sequences using Chinese Postman Tour (CPT). The CPT technique tries to cover all the possible transitions (actions). Figure 6.12 provides a possible test sequences using CPT of the grouped FSM shown in Figure 6.7. In Figure 6.13 we show possible test sequences using Randomwalk of the grouped FSM shown in Figure 6.7. In this, we traversed 10 transitions randomly from the initial state

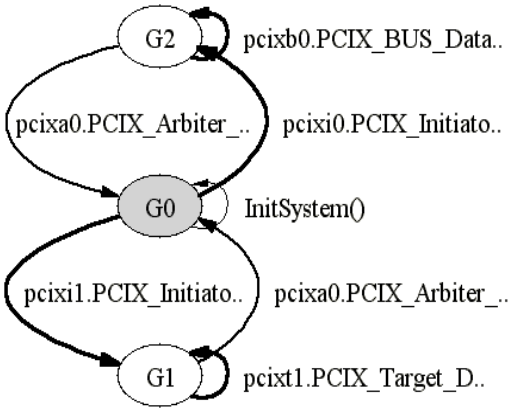

Figure 6.8: Grouped FSM Using *G3*.

```
Grouping Condition G4:
 G4() as Integer
  if (SystemFlag = INIT) then
    return 0
  else
    if (SystemFlag = STARTED) then
      return 1
    else
      if (SystemFlag = IDLE) then
        return 2
      else
         return -1
```

Figure 6.9: Grouping Condition *G4*.

(*G0*). Figure 6.14 provides a possible test sequence using CPT of the grouped FSM shown in Figure 6.8. Figure 6.15 provides a possible test sequence using CPT of the grouped FSM shown in Figure 6.10.

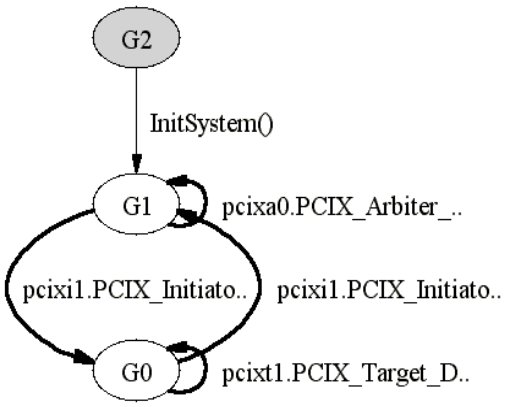

Figure 6.10: Grouped FSM Using *G4*.

```
From G1 to G1 using InitSystem()
From G1 to G2 using PCIX_Initator_Req()
From G2 to G2 using PCIX_Arbiter_GNT_UPD()
From G2 to G1 using PCIX_Initiator_Disconnect()
From G1 to G3 using PCIX_Initator_Req()
From G3 to G3 using PCIX_Arbiter_GNT_UPD()
From G3 to G1 using PCIX_Initiator_Disconnect()
From G1 to G0 using PCIX_Initator_Req()
From G0 to G0 using PCIX_Arbiter_GNT_UPD()
From G0 to G3 using PCIX_Initiator_Disconnect()
```

Figure 6.11: Test Sequences for Grouped FSM of *G1* using CPT.

6.5 Discussion

MBT is more suitable for SystemC designs due to the limitations of the classical simulation. In our approach, using the system's FSM, we generate test cases to perform coverage evaluation. Furthermore, we can use conformance checking [2] between different abstraction levels. This technique 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

```
From G0 to G0 using InitSystem()
From G0 to G8 using PCIX_Initator_Req()
From G8 to G8 using PCIX_Arbiter_GNT_UPD()
From G8 to G7 using PCIX_Initiator_FRAME_ASSERT()
From G7 to G6 using PCIX_Attribute_Phase()
From G6 to G5 using PCIX_Target_DEVSEL_Assert()
From G5 to G4 using PCIX_Initiator_IRDY_TRDY_ASSERT()
From G4 to G4 using PCIX_Bus_DataPhase()
From G4 to G2 using PCIX_Initiator_Termination()
From G2 to G1 using PCIX_Initiator_LastPhase()
From G1 to G3 using PCIX_Initiator_Release()
From G3 to G0 using PCIX_Arbiter_Release()
From G0 to G8 using PCIX_Initator_Req()
From G8 to G8 using PCIX_Arbiter_GNT_UPD()
From G8 to G7 using PCIX_Initiator_FRAME_ASSERT()
From G7 to G6 using PCIX_Attribute_Phase()
From G6 to G5 using PCIX_Target_DEVSEL_Assert()
From G5 to G4 using PCIX_Initiator_IRDY_TRDY_ASSERT()
From G4 to G1 using PCIX_Initiator_Disconnect()
```

Figure 6.12: Test Sequences for Grouped FSM of *G2* using CPT.

implementations (such as RTL). A specific execution can be performed where we can check if actions are enabled and if they return correct values.

In MBT, choosing a correct method to represent a model for the system under verification is not easy. In the case of finite state models, working knowledge of the various forms of finite state machines and a basic familiarity with formal languages, automata theory, and perhaps graph theory and elementary statistics are needed. Also, testers need to possess expertise in tools, scripts, and programming languages necessary for various tasks. Nevertheless, MBT, in general, seems to be gaining favor, particularly in domains where quality is essential and less-than-adequate system is not an option.

```
From G0 to G0 using InitSystem()
From G0 to G8 using PCIX_Initator_Req()
From G8 to G8 using PCIX_Arbiter_GNT_UPD()
From G8 to G7 using PCIX_Initiator_FRAME_ASSERT()
From G7 to G6 using PCIX_Attribute_Phase()
From G6 to G5 using PCIX_Target_DEVSEL_Assert()
From G5 to G4 using PCIX_Initiator_IRDY_TRDY_ASSERT()
From G4 to G4 using PCIX_Bus_DataPhase()
From G4 to G1 using PCIX_Initiator_Disconnect()
From G1 to G3 using PCIX_Initiator_Release()
```

Figure 6.13: Test Sequences for Grouped FSM of *G2* using Randomwalk.

```
From G0 to G0 using InitSystem()
From G0 to G2 using PCIX_Initator_Req()
From G2 to G2 using PCIX_Bus_DataPhase()
From G2 to G0 using PCIX_Arbiter_Release()
From G0 to G1 using PCIX_Initator_Req()
From G1 to G1 using PCIX_Target_DEVSEL_Assert()
From G1 to G0 using PCIX_Arbiter_Release()
```

Figure 6.14: Test Sequences for Grouped FSM of *G3* using CPT.

```
From G2 to G1 using InitSystem()
From G1 to G0 using PCIX_Initator_Req()
From G0 to G0 using PCIX_Target_DEVSEL_Assert()
From G0 to G1 using PCIX_Initiator_Release()
From G1 to G1 using PCIX_Arbiter_Release()
```

Figure 6.15: Test Sequences for Grouped FSM of *G4* using CPT.

Chapter 7

Conclusion

7.1 Summary

In this thesis, we provided a design for verification method applied on the latest high speed PCI-X standard bus in SystemC. The modeling was done at various levels: UML, AsmL, SystemC. At first, the UML representation of PCI-X was developed in terms of class diagrams and sequence diagrams. Then, an AsmL model was designed from the UML representation. In order to perform the model checking, we defined various properties of the PCI-X standard in PSL and embedded them in the AsmL model to be verified using model checking. Once, the verification is done, the translation of AsmL to SystemC was performed.

We also investigated the potential of the model based testing approach. To accomplish this, we formalized two FSM generation algorithms from SystemC: direct

algorithm and grouping algorithm. The formalization of these algorithms was done based on an existing SystemC semantics. The direct algorithm generates an FSM and the grouping algorithm generates the grouped FSM which has the notion of state groupings. State groupings is a good technique to reduce the size of the FSM. It enables us to validate (test) a particular part of the system which is very much applicable in the case of SystemC. The traversal of the FSM was performed to generate test cases. We used existing traversal techniques such as Chinese Postman Tour and Randomwalk to generate the test cases.

7.2 Discussion and Future Work

Model checking is a good technique but it is still immature to be used for complex designs in particular SystemC designs. Model checking is reliable in catching bugs as the correctness of a formally verified design implicitly involves all cases regardless of the input values. Nevertheless, the state space explosion problem limits its wide use in a complex systems in industry.

On contrary, MBT is a good compromise between the blind simulation and model checking. In MBT, a model serves as a unifying point of reference that all teams and individuals involved in the development process can share, reuse, and benefit from. It has an upper hand in terms of early bug detection, time savings, easy test case maintenance, enhanced communication between developers and testers, etc., However, MBT demands certain skills from the testers. Testers

need to be familiar with the model and its underlying and supporting mathematics and theories.

The real work that remains for the foreseeable future is fitting specific models (finite state machines, grammars or language-based models) to specific application domains. Often this will require new inventions as mental models are transformed into actual models. Perhaps, special purpose models will be made to satisfy very specific testing requirements and more general models will be composed from any number of pre-built special-purpose models.

Regarding the grouping algorithm, no proof of soundness has been given. This means we need to prove that the FSM from the grouping algorithm preserves the original FSM. This may sound trivial but it is important to provide these proofs to show the efficiency of our approach. In order to show the soundness of the grouping algorithm, we would have to prove the following.

- For every state s in S_{FSM}, there is a corresponding grouped state s_g in S^G_{FSM} with respect to a grouping condition.
- For every transition t in T, there is a corresponding transition t_g in T_G
- For every path p in FSM, there is a corresponding path p_g in grouped FSM_G

The proofs of the above steps could be a good future work of this thesis. In addition to this, providing a complexity and performance evaluation of the two algorithms could also be a another future work. To be specific, we would have to

perform the space and time complexity analysis of both the algorithms.

Regarding the PCI-X IP of SystemC, it is not synthesizable. It would be a good future work to make it synthesizable and implement in RTL. Also, some of the advanced but often not used transactions of PCI-X is not being supported in our model. Once such transaction is split transaction using the notion of PCI-X bridges but the split transaction without a bridge is supported in our model. Other future work that could be stemmed from this research is the investigation of automatic generation of HDL from SystemC and automatic translation of AsmL to SystemC.

Bibliography

[1] Accellera Organization. Accellera Property Specification Language Reference Manual, version 1.1. www.accellera.org, 2005.

[2] 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.

[3] E. Boerger and R. Staerk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer Verlag, 2003.

[4] 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.

[5] 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.

[6] K. H. Chang, Y. C. Su, W. T. Tu, Y. J. Yeh, and S. Y. Kuo. A PCI-X Verification Environment Using C and Verilog. In *Proc. VLSI Design/CAD Symposium*, Taiwan, 2003.

[7] S. Cheng, R .K. Brayton, G. York, K. Yelick, and A. Saldanha. Compiling Verilog into Timed Finite State Machines. In *Proc. 4th International Verilog HDL Conference*, pages 32–44. IEEE Computer Society Press, 1995.

[8] M. Chong. A PCI Express to PCI-X Bridge Optimized for Perfromance and Area. Master's thesis, Department of Electicial Engineering and Computer Science, Massachussets Institute of Technology, March 2004.

[9] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, London, England, 1999.

[10] P. Cousot and R. Cousot. Abstract Interpretation Frameworks. *Journal of Logic and Computation*, 2(4):511–547, August 1992.

[11] 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.

[12] I. K. El-Far and J. A. Whittaker. Model-based Software Testing. In *Encyclopedia of Software Engineering*. Wiley, 2001.

[13] F. Ferrandi, M. Rendine, and D. Scuito. Functional Verficiation for Systemc Descriptions Using Constraint Solving. In *Proc. Design Automation and Test in Europe*, pages 744–751, Paris, France, March, 2002.

[14] A. Gawanmeh, A. Habibi, and S. Tahar. Enabling SystemC Verification using Abstract State Machines. In *Proc. Forum on Specification and Design Languages*, pages 306–421, Lille, France, 2004.

[15] 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.

[16] I. Gronau, A. Hartman, A. Kirshin, K. Nagin, and S. Olvovsky. A Methodology and Architecture for Automated Software Testing. Technical report, IBM Research Laboratory, MATAM Advanced Technology Center, Haifa, 2000.

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

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

[19] A. Habibi. *A Framework for System Level Languages: The SystemC Case*. PhD thesis, Dept of ECE, Concordia University, Montreal, Quebec, Canada, December 2005.

[20] A. Habibi, A.I. Ahmed, O. Ait-Mohamed, and S. Tahar. On the Design and Verification of the Look-Aside Interface. In *Proc. Design Automation and Test in Europe*, pages 290–295, Munich, Germany, March 2005.

[21] A. Habibi and S. Tahar. An Approach for the Verification of SystemC Designs using AsmL. In *International Symposium on Automated Technology for Verification and Analysis,* LNCS 3707, pages 69–83, Taipei, Taiwan, October, 2005. Springer.

[22] J.K. Huggins. Abstract state machines home page.
website: http://www.eecs.umich.edu/gasm/, 2005.

[23] 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.

[24] J. Lohse, J. Bormann, M. Payer, and G. Venzl. VHDL-Translation for BDD-based Formal Verification. Technical report, Siemens Corporate R&D, 1994.

[25] R. A. Maksimchuk and E. J. Naiburg. *UML for Mere Mortals.* Addison-Wesley, 2005.

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

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

[28] Microsoft Research. http://www.research.microsoft.com.

[29] Network Processing Forum. *Look-Aside (LA-1) Interface, Implementation Agreement, Revision 1.1.* Kluwer Academic Publishers, April 15, 2004.

[30] Microsoft Research Foundations of Software Engineering. Asml for microsoft .net. http://www.research.microsoft.com/foundations/asml.

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

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

[33] K. Oumalou, A. Habibi, and S. Tahar. Design for Verification of a PCI Bus in SystemC. In *Proc. Symposium on System-on-Chip*, pages 201–204, Tampere, Finland, November 2004.

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

[35] E. Riccobene, P. Scandurra, A. Rosti, and S. Bocchio. A SoC Design Methodology Involving a UML 2.0 Profile for SystemC. In *Proc. Design Automation and Test in Europe*, pages 704–709, Munich, Germany, March 2005.

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

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

[38] V. S. Saun. FSM Derivation from SystemC. Technical report, CSE Dept., Indian Instititute of Technology, Delhi, May 2004.

[39] K. Shimizu, D. L. Dill, and A. J. Hu. Monitor-based formal specification of PCI. In *Formal Methods in Computer-Aided Design*, pages 335–353. LNCS 1954, Springer-Verlag, 2000.

[40] PCI Special Interest Group. www.pcisig.com, 200.

[41] IEEE Standards. IEEE 1666 - SystemC Standardization. http://standards.ieee.org/announcements/pr_p1666.html.

[42] Unified Modeling Language. http://www.uml.org, 2005.

[43] R. Wang and Z. Wen. A Verification Environment for PCI-X BFMs in VERA. Technical report, Synopsys Inc., 2002.

[44] C. C. Yu, K. Chang, Y. Yeh, and S. Kuo. System Level Assertion-Based Verification Environment for PCI/PCI-X and PCI-express. In *Proc. VLSI Design/CAD Symposium*, Taiwan, 2004.