Automatic Generation of Transactors in SystemC

Tareq Hasan Khan

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 (Electrical & Computer Engineering)

at

Concordia University

Montréal, Québec, Canada

September 2007

Your file Votre référence
ISBN: 978-0-494-34700-3
Our file Notre référence
ISBN: 978-0-494-34700-3

ABSTRACT

Automatic Generation of Transactors in SystemC

Tareq Hasan Khan

System-on-chip (SoC) is a major revolution taking place in the design of integrated circuits due to the unprecedented levels of integration possible. To specify, design, and implement complex SoC systems, the need arises to move beyond existing register transfer level (RTL) of abstraction. A new modeling method, transaction level modeling (TLM) has been proposed recently to fulfil this need. TLM modules communicate with each other through function calls and allow the designers to focus on the functionality, while abstracting away implementation details. At the RTL, however, different modules communicate through pin level signaling. SoC design methodologies involve the integration of different intellectual property (IP) blocks modeled at different levels of abstraction. Therefore a special module or channel is needed in order to link modules, IPs, designed at different levels of abstraction. This module, called transactor can be modeled using a finite state machine (FSM) providing a functional specification of the protocol's behavior. In this thesis, we propose to specify TLM-RTL transactor behaviors using the Abstract State Machine Language (AsmL). Based on AsmL specification, we have developed a methodology and tool that automatically generates SystemC code for the transactors. SystemC is a system level description language, which became IEEE standard recently. Along with the AsmL specification approach, we also proposed another approach where the transactor behavior can be described by drawing FSMs graphically and the tool will then generate SystemC code from the graphical FSM description automatically. The proposed approaches have been implemented and applied on several case studies including an UTOPIA standard protocol.

ACKNOWLEDGEMENTS

I would like to start by praising Allah, the creator and the sustainer of the universe. I feel fortunate and grateful working with my thesis supervisors Dr. Sofiène Tahar and Dr. Otmane Ait Mohamed who not only helped me to overcome the difficulties, but also taught me how to think critically. I would like to give special thanks to Dr. Ali Habibi at MIPS Technologies who came up with the idea of this thesis and helped me to get started by his close supervision. Thanks also to Dr. Ghiath Al Sammane for his suggestions and constructive critics. The Hardware Verification Group (HVG) members were very nice and friendly. I would like to give thanks to Haja, Essam, Saleem, Suliman, Bahador, Naeem, Osman and Kamran for their help and support. I am grateful to my honorable parents, sweet and caring sisters, brother, and my oldest friend Ashraf Zaman for their enthusiasm and inspiration.

To My Parents

TABLE OF CONTENTS

LIST OF TABLES ix

LIST OF FIGURES x

LIST OF ACRONYMS xii

1 Introduction **1**

1.1 Motivation 1

1.2 Methodology 5

1.3 Related Work 7

1.4 Thesis Contribution 9

1.5 Thesis Outline 10

2 Preliminaries **11**

2.1 Abstract State Machines (ASM) 11

2.1.1 States 11

2.1.2 Terms 12

2.1.3 Locations and Updates 13

2.1.4 Transition Rules 13

2.1.5 Abstract State Machine Language (AsmL) 15

2.2 SystemC 16

2.2.1 SystemC Language Structure 17

2.2.2 SystemC Simulator 18

2.3 Register Transfer Level Modeling 20

2.4 Transaction Level Modeling 21

3 Specifying Transactors **23**

3.1 Specifying Transactor in AsmL 23

3.1.1 AsmL Subset 24

3.1.2 Hardware Data Types in AsmL 25
3.1.3 The Step Rule 27
3.1.4 Guidelines for Specifying Transactor 28
3.1.5 Limitations of AsmL 29
3.2 Translation from AsmL to SystemC 29
3.2.1 Data Type Mapping 30
3.2.2 Semantic Translation 30
3.2.3 Syntax Translation 33
3.3 Specifying Transactor in Graphical FSM 40
3.3.1 Guidelines for Specifying Transactors in Graphical FSM . . . 41
3.3.2 FSM Representation in ASF Format 43
3.3.3 FSM Objects 44
3.3.4 AsmL Code Generation 44

4 SystemC Transactor Generator Tool **47**
4.1 Input Interface 50
4.1.1 TLM Interface 50
4.1.2 RTL Interface 51
4.1.3 Code Settings 53
4.2 Generating Transactor from AsmL 53
4.2.1 AsmL Template Generator 53
4.2.2 XML to DOC conversion and vise versa 55
4.2.3 AsmL to SystemC Translator 56
4.2.4 Integrator 58
4.3 Generating Transactor from Graphical FSM 59
4.3.1 FSM Drawing Template 59
4.3.2 FSM to AsmL Code Generator 60
4.4 Generating Transactor Code Template 60
4.5 Library Generation 60

5 Case Studies **62**

5.1 UTOPIA Transactor 62

5.1.1 Signal Description 62

5.1.2 Protocol Description 64

5.1.3 Modeling in SystemC 64

5.1.4 Generating SystemC Transactor 65

5.1.5 Test Case Generation 68

5.1.6 Simulation of the Generated Code 69

5.1.7 Experimental Results 72

5.2 Memory Interface Transactor and Library Generation 74

6 Conclusion **77**

6.1 Summary 77

6.2 Discussion and Future Work 78

A **79**

A.1 AsmL Code for UTOPIA 79

A.2 SystemC Transactor Code 80

A.3 SystemC Transactor Template 85

A.4 AsmL Code for Memory Access 86

Bibliography **87**

LIST OF TABLES

3.1 Subset of AsmL Data Types 24
3.2 AsmL Data Type Mapping with SystemC 30
3.3 Translation of AsmL Variable to SystemC 32
3.4 Translation of AsmL Update Semantic to SystemC 33
3.5 Translation of Conditional Statements 34
3.6 Translation of Iteration Statements 35
3.7 Translation of Assertion Statements 35
3.8 Translation of Symbols and Operators 36
3.9 Translation of Assignment Statements 37
3.10 Translation of Enumeration and Constant Declarations 38
3.11 Translation of Blocks 40

4.1 Analyzed AsmL line ID 57
4.2 Analyzed AsmL token ID 58

5.1 UTOPIA Interface Signals (optional signals are not listed 63
5.2 TLM Functions Called by the ATM Module 65
5.3 Experimental Results 73

LIST OF FIGURES

1.1 TLM-RTL Transactor 3

1.2 Methodology for Transactor Generation and Verification 6

2.1 SystemC Language Structure 17

2.2 SystemC Simulation Methodology 19

3.1 AsmL Subset 25

3.2 Hardware Data Types in AsmL 25

3.3 Hardware Constants in AsmL 26

3.4 Binary String to Decimal and Vise-Versa Conversion Functions 26

3.5 Binary String to Decimal Conversion Algorithm 27

3.6 Decimal to Binary String Conversion Algorithm 27

3.7 Variable in C language 31

3.8 Variable in AsmL 31

3.9 Variable in SystemC 32

3.10 Algorithm for Generating Block 39

3.11 A Sample Graphical FSM (Snapshot from State Editor) 41

3.12 ASF Format 43

3.13 FSM Objects 44

3.14 AsmL Code Generation Algorithm 45

4.1 Block Diagram of SystemC Transactor Generator Tool 48

4.2 SystemC Transactor Generator Tool GUI 49

4.3 TLM Interface Input Window 50

4.4 Data Structure for TLM Functions 51

4.5 RTL Interface Input Window 52

4.6 Data Structure for RTL Ports 53

4.7 Format for Writing TLM function 55
4.8 SystemC Transactor Generation 59
4.9 Adding Transactor Library 61

5.1 UTOPIA Transactor 63
5.2 Graphical FSM Specification of the Function GetCell() 67
5.3 Test Case Generation by the AsmL Tester 69
5.4 Scenario 1: Simulation Timing Diagram 70
5.5 Scenario 2: Simulation Timing Diagram 71
5.6 Scenario 3: Simulation Timing Diagram 72
5.7 Relationship between AsmL and SystemC lines of code 73
5.8 Memory Access Transactor 74
5.9 Graphical FSM for the function mem_write() 75
5.10 UTOPIA Transactor after Adding Memory Access Protocol Library . 76

A.1 AsmL Code for function SendCell() 80
A.2 Automatically Generated SystemC Code 84
A.3 Automatically Generated SystemC Code Template 85
A.4 AsmL Code for function mem_read() 86

LIST OF ACRONYMS

AHB	Advanced High-performance Bus
AMBA	Advanced Microcontroller Bus Architecture
ASCII	American Standard Code for Information Interchange
ASF	Active HDL State machine Format
ASM	Abstract State Machines
ATM	Asynchronous Transfer Mode
AsmL	Abstract State Machine Language
BCA	Bus Cycle Accurate
CPU	Central Processing Unit
DLL	Dynamic Link Library
EDA	Electronic Design Automation
FIFO	First In First Out
FSM	Finite State Machine
GUI	Graphical User Interface
GNU	GNU's Not Unix
HDL	Hardware Description Language
IEEE	Institute of Electrical and Electronics Engineers
I2C	Intelligent Interface Controller
IP	Intellectual Property
OO	Object Oriented
PHY	PHisical laYer
PSL	Property Specification Language
PV	Programmer's View
RTL	Register Transfer Level
SERE	Sequential Extended Regular Expressions
SoC	System-On-Chip

SVA	System Verilog Assertions
TLM	Transaction Level Modeling
VCD	Value Change Dump
VHDL	VHSIC Hardware Description Language
UTOPIA	Universal Test and Operations PHY Interface for ATM

Chapter 1

Introduction

1.1 Motivation

When systems were composed primarily of discrete parts such as microprocessors, memory chips, analog devices, and application specific integrated circuits (ASICs), the design process usually started with one or two system design experts who would partition the functionality into hardware and software, and further partition the hardware part to standard parts and ASICs.

In contrast, modern era system-on-chip (SoC) may contain one or more processors including 32-bit microcontrollers and digital signal processors (DSPs) or specialized media processors. On-chip memory, accelerated hardware units for dedicated functions, and peripheral control devices, linked together by a common complex on-chip communication network that incorporates on-chip buses. Software and its architecture, layering, and complexity are inherent in such a design [19].

To specify, design, and implement such complex systems, incorporating functionality implemented in both software and hardware forms, the need arises to move beyond existing register transfer level (RTL) of abstraction. The new modeling method transaction level modeling (TLM) [42] has been proposed recently to fulfil this need. TLM allows the designers to focus on the functionality of the design, while abstracting away implementation details that will be added at lower abstraction levels [13]. Transaction level models use software *function calls* to model the communication between blocks in a system. This is in contrast to hardware RTL and gate level models, which use *signals* to model the communication between blocks. For example, a transaction level model would represent a burst read or write transaction using a single function call, with an object representing the burst request and another object representing the burst response. An RTL hardware description language model would represent such a burst read or write transaction via a series of signal assignments and signal read operations occurring on the wires of a bus [46].

SoC design methodologies involve the integration of different intellectual property (IP) blocks communicating between each other modeled at different levels of abstraction. The ultimate goal in developing an SoC is to find a perfect match between all system blocks in order to satisfy a set of predefined requirements (cost, power, performance, etc.). In this process, it is inescapable to face the problem of integrating IPs designed at different levels of abstraction. This, however, creates a major concern about the communication mechanisms among the system elements. For example, data transfer between an un-timed block and a clocked module requires the definition of an explicit interface. In order to be able to link modules modeled at different levels of abstraction, the notion of *transactor* has been recently introduced [8, 11]. A TLM-RTL transactor as shown in Figure 1.1 would have two interfaces, one at TLM side and another at RTL side. The TLM interface consists of virtual declarations of the TLM functions. The RTL interface consists of the declaration of the RTL ports. The implementation of each TLM function is done inside the transactor module. When a TLM function is called from the TLM module, signal activities take place between the transactor and the RTL module. To accomplish the task of a TLM function on the RTL side, there can be a finite state machine implemented inside the transactor [42].

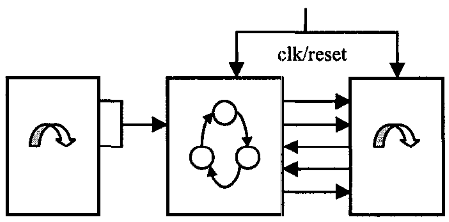

Figure 1.1: TLM-RTL Transactor

Inside a TLM-RTL transactor, we need to implement one or more RTL hardware protocols to accomplish a particular task on the RTL module. These protocols are generally specified by the protocol designers in natural languages such as in English texts. But natural languages are often incomplete and ambiguous. Also, informal specification causes verification problems which stems from the fact that there is no mathematical means to prove its correctness. Moreover, a naturally expressed specification cannot be executed or simulated in different relevant scenarios thus creating the problem of validation. These problems may cause more bugs and faults in the product, delays for time to market, etc.

On the other hand, if we write the transactor in a hardware description language such as VHDL or Verilog or even SystemC [29], we will not have the feasibility to use high level abstract constructs to specify the protocol early. In this thesis, we propose to create formal models of the transactor protocol taking the natural language text as reference. We will use the Abstract State Machine Language (AsmL) [37] as a formal means for specification and communication and then translate it to SystemC. The main advantage of using AsmL and translating it to SystemC instead of using directly SystemC is the possibility to specify the transactor on a very high level of abstraction enabling the customer and the design team members to understand the specification. AsmL models are precise, concise and readable to a wide range of people who have different areas of expertise due to its simple and intuitive language constructs [12]. This model removes the language and communication problem of natural languages and also provides efficient ways of verification and validation. So, once the AsmL model is completed and verified, it can be used to automatically generate the transactors in other languages. In this work, we have generated SystemC code from AsmL specification according to developed syntax and semantic translation rules developed in this thesis.

An AsmL specification presents the following advantages [12]:

- Precise at appropriate level of detailing yet flexible and modifiable
- Simple and intuitive to be understandable by people of different background, culture and expertise
- Concise specification which replaces hundreds of pages of tedious specification expressed in natural languages
- Verifiable model using model checking, mechanized or manual proofs
- Validation can be done for different scenarios due to the machine executability of AsmL models

ASMs (Abstract State Machines) [12] are used to specify both software and hardware. In hardware circuits, simultaneous multiple operations like sending signals to different pins, may occur during a single clock cycle. ASM supports this kind of parallelism due to its update semantic feature [12]. A TLM-RTL transactor deals with transaction level model where the model is described from the programmer's point of view (PV) and also with register transfer level where the model is described from hardware design point of view. Thus ASM fits properly to specify transactor as it has the ability to describe both points of view.

Inside a TLM-RTL transactor, the hardware protocol can be modeled as a finite state machine. ASM languages like AsmL (Abstract State Machine Language) [37] provide powerful constructs and language features to model finite state machines such as *step*, *update* semantics, etc. which are very useful in modeling transactors.

Along with the AsmL specification, we also provide another approach where the behavior of the transactor can be described by Finite State Machines (FSM) graphically. Graphs are frequently used in computer applications as a general data structure to represent objects and relationships between them [16, 26]. They are used to implement hierarchies, dependency structures, networks, configurations, data flows, etc. Usually graph visualization tools support the following options: directed, undirected, and mixed graphs, hyper graphs, hierarchical graphs and graphical representations [47]. Hardware designers are familiar with graphical FSM and thus it removes the overhead to learn a new specification language. Furthermore, a visual representation of FSM simplifies the access of the protocol description. Graphical FSMs are intuitive, easy to follow, and understand. After the FSM specification is drawn, we translate it to AsmL according to an AsmL code generation algorithm. Finally, the AsmL code is translated to SystemC according to the translation rules proposed in this thesis.

It may be required to specify the transactor directly in SystemC if the specification writer is unfamiliar with ASM language or graphical FSM. In that case to help the specification writer to ease the job, we provide another approach where a SystemC template is automatically generated by our proposed transactor generator tool. The specification writer can use the template to write the transactor directly in SystemC manually.

1.2 Methodology

In the proposed methodology shown in Figure 1.2, we create a formal model of the transactor protocol in AsmL based on natural language text. It can play a significant role among the SoC design team members as an unambiguous, precise, and concise specification. Syntax and semantics of AsmL is formalized and thus it gives us the opportunity to verify formally the transactor protocol at an early stage of

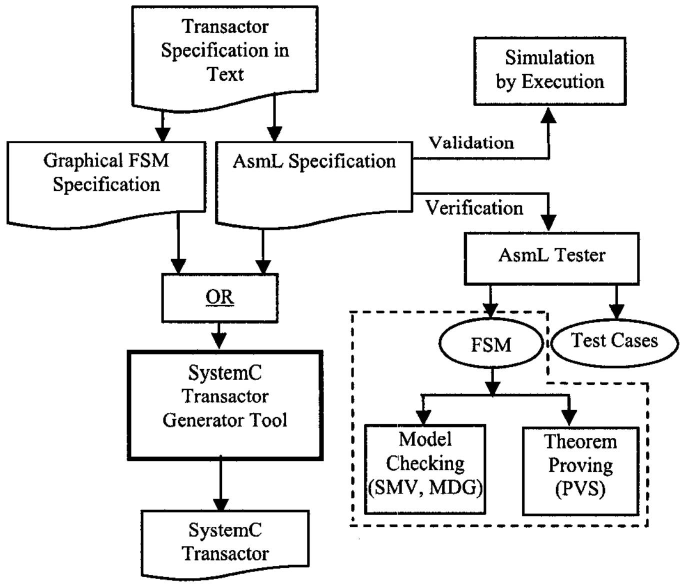

Figure 1.2: Methodology for Transactor Generation and Verification

the SoC design process. AsmL specifications are executable, thus can be validated by simulation for different scenarios using the *asmlc* compiler [37]. Using the Microsoft's AsmL Tester (Asmlt) tool, the FSM of the AsmL model can be generated [37]. The generated FSM can be verified formally using model checking tools such as SMV [14] or MDG [18]. Manual or mechanized theorem proving of ASM models can be done by tools like PVS [17, 41] and Isabelle [31]. These formal verifications will enhance the confidence in the correctness of the finally generated transactor. Moreover, the AsmL Tester can generate test cases which can be used to simulate the transactor model for different scenarios. Once the AsmL model is completed and verified, it can be used as input to the proposed *SystemC Transactor Generator Tool* to automatically generate the SystemC transactor. Another approach to specify the transactor is by drawing Finite State Machine graphically. The FSM description will be drawn in a State Diagram Editor and then the FSM description will be given as input to the *SystemC Transactor Generator Tool.* The tool at the first stage will generate AsmL code from the FSM description and then at the next stage, the

AsmL code is translated to SystemC.

In the methodology shown in Figure 1.2, the blocks inside the dashed line (i.e. formal verification of the AsmL models by model checking and theorem proving) are not implemented in this thesis. The remaining blocks of the methodology are implemented and discussed in the rest of the thesis.

1.3 Related Work

Regular expressions and temporal logic [39] are the two main formalisms that have been used for formal interface specifications. Both formalisms can be expressed with finite-state automata [27]. More recently, standard languages have been proposed to specify system properties (in particular, the Property Specification Language (PSL) [3] and the System Verilog Assertions (SVA) [30]. These languages are based on temporal logic, but both of them also include a capability to specify regular expressions. In PSL, such an extension is called Sequential Extended Regular Expressions (SEREs). Balarin *et al.* [8] proposed to specify TLM-RTL transactors using PSL. They took advantage from the SEREs aiming at generating synthesizable transactors. This approach is limited by the expressivity of SEREs and by the fact that the final transactor has to be synthesized. Hence, it presents a critical limitation of the use of transactors in the SystemC design flow only at RTL. Many commercial tools include features to generate SystemC transactors, for example: SystemC Transactor Generation Wizard from Aldec's Active HDL [4], Catapult C from Mentor Graphics [33], TransactorWizard from Structured Design Verification [45], and Cohesive from Spiratech [44]. The Cohesive tool uses the CY language as transactor specification. In Active HDL v7.1, the SystemC Transactor Generation Wizard creates the interfaces and a template for the transactor. Then the users have to write the transactor code in SystemC by hand. In contrast to above related work, we do not restrict our method to certain abstraction level. We also propose a tool that automatically generates SystemC codes for transactors.

We will now discuss some related work on the graphical representation of FSMs. Different formats have been proposed as input to visualization tools. They usually consist of a language core to describe the structural properties of a graph and a flexible extension mechanism to add application-specific data. In our work, we used the new and rich Active HDL State Machine Format (ASF) to represent FSMs. The Active HDL tool uses the ASF format to store graphical information to textual form and vise-versa. It also generates VHDL and Verilog code from the FSM. Similar visualization tools include the daVinci graph visualization [15] program and the VCG tool [48] which automatically computes the most optimal way to view the finite-state automaton by minimizing the number of crossing edges. Another visualization tool is AiSee [2], which is a part of the Absint static analyzer tool suite and was developed initially to visualize the internal data structures found in compilers. Today it is widely used in many different areas including visualizing FSMs. AiSee automatically calculates a customizable layout of graphs specified in GDL (graph description language) [15]. This layout is then displayed, and can be printed or interactively explored. Xilinx company provides a commercial tool for the rapid prototyping of an FSM design directly from the state diagram. Xilinx ISE tools [49] include an editor, named StateCAD, which allows users to graphically input state diagrams and translated them into a Verilog behavioral HDL model. In [1], the authors implemented a tool that takes dot FSM format, then converted it to Kiss format [40] and then generated VHDL code. In this work, we have developed a tool that takes FSM description in ASF format and generates AsmL code. This AsmL code is then used to automatically generate SystemC codes for the transactors.

1.4 Thesis Contribution

In this thesis, we have developed a methodology and implemented a tool to automatically generate SystemC transactors both from AsmL specifications and from graphical FSMs. We have also done several case studies.

In summary, the thesis contributions are as follows:

1. We have defined a subset, rules and guidelines to specify transactors in AsmL. Also, we have defined hardware data types and constants in AsmL to declare RTL ports and to represent hardware oriented information.

2. We have defined a set of semantic and syntax translation rules to translate AsmL specification to SystemC.

3. We have defined a set of rules to specify transactors by graphical FSM and developed an algorithm to generate AsmL code from graphical FSM description.

4. We have developed a *SystemC Transactor Generator Tool* for automatic generation of SystemC transactors both from AsmL specification and from graphical FSM description. The tool consists of Graphical User Interface (GUI), FSM to AsmL Code Generator, AsmL to SystemC Compiler and other necessary modules. The tool also provides features to generate transactor libraries for standard protocols.

5. We have done a case study with the UTOPIA transactor. We wrote AsmL specifications and also drew graphical FSMs to specify the transactor. Then we generated the SystemC transactors using our developed *SystemC Transactor Generator Tool.* We also modeled a TLM ATM module and an RTL PHY module in SystemC. A transactor library for memory access was also generated.

1.5 Thesis Outline

This thesis is made up of six chapters. In Chapter 2, we provide an overview of ASM, AsmL, SystemC, RTL and TLM modeling. This chapter lays a foundation for better understanding of the thesis. In Chapter 3, we discuss the method for specifying transactor in AsmL and its syntax and semantics translation to SystemC. Also, specifying transactor using graphical FSM and algorithm for generating AsmL code from FSM description is discussed. In Chapter 4, we describe different modules of the *SystemC Transactor Generator Tool.* In Chapter 5, we discuss the case study of UTOPIA transactor and transactor library generation. In Chapter 6, we provide a summary of the thesis, some concluding discussions and future work hints. Finally, Appendix A contains some AsmL and SystemC source codes for the case studies.

Chapter 2

Preliminaries

In this chapter, we give a brief insight into ASM, AsmL, SystemC, RTL and TLM modeling. This chapter would provide a good foundation for the understanding of the rest of the thesis.

2.1 Abstract State Machines (ASM)

Abstract State Machines (ASM) 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 [12]. 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.

2.1.1 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* which 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* which interpretation is determined in each state by the environment. Changes in external functions that 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* which 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.

2.1.2 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 $t_1 ... t_r$ are terms, then $f(t_1 ... t_r)$ is a term. The composed term is Boolean if f is relational.

2.1.3 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 [21, 22, 23].

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.

2.1.4 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
    R1
else
    R2
endif
```

where g is a first-order Boolean term. *R1* and *R2* denote arbitrary transition rules. The condition rule is executed in state S by evaluating the guard g, if *true* *R1* fires, otherwise *R2* fires.

2.1.5 Abstract State Machine Language (AsmL)

Abstract State Machine Language (AsmL) [37] 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 sense AsmL programming is transaction based. 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 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. Microsoft is distributing AsmL with MS Spec Explorer [36] recently. AsmL effectively supports specification and rapid prototyping of different kinds of models. The AsmL tester [37] can also be used for FSM generation or test case generation.

An AsmL model (or program) is defined using a fixed vocabulary of symbols of our choice. It has two components: the names of its state variables and a fixed set of operations of an abstract state machine [34]. Values are simple, immutable elements like numbers and strings. State can be seen as a particular association of variable names to values, in the style of a dictionary: (name1, val1), (name2, val2), .

A run of the machine is a series of states connected by state transitions. Each state transition, or step, occurs when the machine's control logic is applied to an input state and produces an output state. "Control logic" is a synonym for the machine's set of operations.

The program consists of statements. A typical statement is the conditional update "if *condition* then *update.*" Each update is in the form "$a := b$" and indicates that variable name a will be associated with the value b in the output state.

The program never alters the input state. Instead, each update statement adds to a set of pending updates. Pending updates are not visible in any program context, but when all program statements are invoked, the pending updates are merged with a copy of the input state and returned as the output state.

An inconsistent update error occurs if the update set contains conflicting information. For example, the program cannot update a variable to two different values in a single step.

2.2 SystemC

SystemC, one of the proposals of the electronic design automation (EDA) community has become the IEEE standard (IEEE1666-2005) [29] for system level design [32]. SystemC aims at bridging the gap between hardware and software design flows. Furthermore, it promotes the integration of different levels of abstraction in a unique design process. 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. SystemC provides hardware-oriented constructs within the context of C++ as a class library implemented in standard C++. Its use spans design and verification from concept to implementation in hardware and software. SystemC provides an interoperable modeling platform which enables the development and exchange of very fast system-level C++ models. It also provides a stable platform for development of system-level tools.

2.2.1 SystemC Language Structure

The SystemC language architecture is shown in Figure 2.1. The language is built on top of standard C++. The layer above it is the so called core layer (or layer 0) of the standard SystemC language. It contains constructs and data types for simulating hardware oriented features. It also contains an event driven simulation kernel. Then the layer above 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.

SystemC has a notion of a container class, called *module*, that provides the ability to encapsulate structure and functionality of hardware/software blocks for

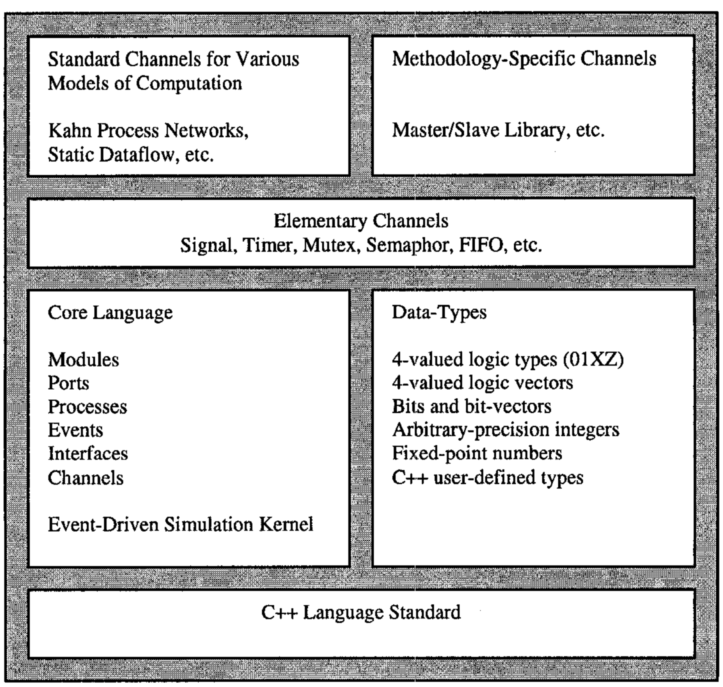

Figure 2.1: SystemC Language Structure

partitioning system designs. A system is essentially broken down into a containment *hierarchy of modules.* Each module may contain *variables* as simple data members, *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 [10, 19].

2.2.2 SystemC Simulator

The simulation kernel for SystemC follows the evaluate-update paradigm that is common in HDLs. The concept of delta cycles, where multiple evaluate-update phases can occur at the same simulation time, is supported [19]. A simplified version of the simulation algorithm is as follows:

1. *Initialization:* Execute all processes to initialize the system.

2. *Evaluate:* Execute a process that is ready to run. Iterate until all ready processes are executed. Events occurring during the execution could add new processes to the ready list.

3. *Update:* Execute any update calls made during any step.

4. If delayed notifications are pending, determine the list of ready processes and proceed to Evaluate phase (step 2).

5. *Advance the simulation time* to the earliest pending timed notification. If no such event exists, the simulation has finished, else determine ready processes and proceed to step 2.

Figure 2.2 illustrates a generic simulation methodology in the SystemC environment. The SystemC model can be written at different levels using C/C++ augmented by the SystemC class library. The class library serves two important purposes. First, it provides the implementation of many types of objects that are hardware-specific, such as concurrent and hierarchical modules, ports, and clocks. Second, it contains a kernel for scheduling the processes. The design's SystemC code can be compiled and linked together with the class library with any standard C++ compiler (such as GNU's gcc, Microsoft Visual C++), and the resulting executable serves as the simulator of the user's design. The testbench for verifying the correctness of the design is also written in SystemC and compiled along with the design.

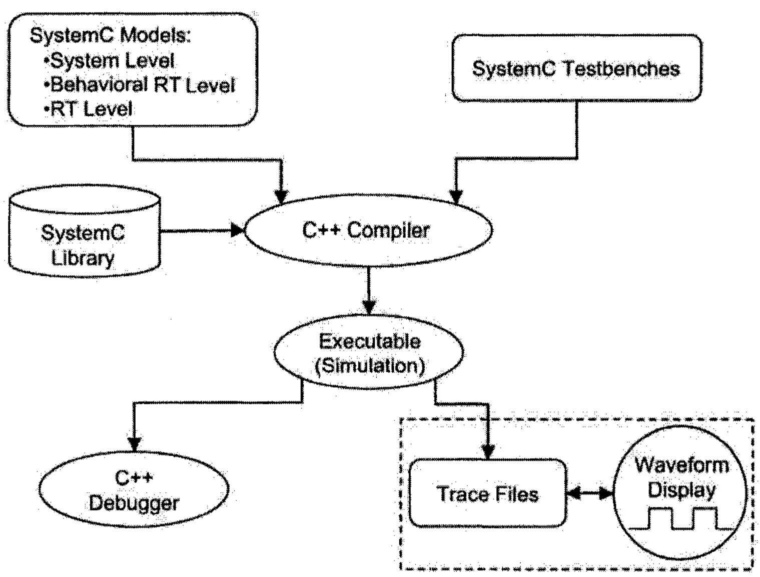

Figure 2.2: SystemC Simulation Methodology

The executable can be debugged in any familiar C++ debugging environment (such as GNU's gdb). Additionally, trace files can also be generated to view the history of selected signals using a standard waveform display tool.

The import of a traditional software development environment into the hardware design and system design scenario entails some powerful advantages. The sophisticated program development infrastructure already in place for C/C++ can be directly utilized for the SystemC verification and debugging tasks. For hardware designers traditionally used to view simulation data in the form of waveform displays, the trace file generation facility provides a familiar interface. Conceptually, the most powerful feature is that the hardware, software, and testbench parts of the design can be simulated in one simple and unified simulation environment without the need for clumsy co-simulations of disparate modeling paradigms.

2.3 Register Transfer Level Modeling

The Register Transfer Level (RTL) is a modeling style that corresponds to digital hardware synchronized by clock signals. This modeling style is widely used within languages such as Verilog and VHDL, and it is widely supported by commercial hardware synthesis tools. In the RTL style, all communications between processes occur through signals. Processes may either represent sequential logic, in which case they are sensitive to a clock edge, or they may represent combinational logic, in which case they will be sensitive to all inputs. RTL modules are pin accurate. This means that the ports of an RTL module directly correspond to wires in real-world implementation of the module. RTL modules are also cycle-accurate [19].

2.4 Transaction Level Modeling

Transaction level modeling (TLM) is becoming a usual practice for simplifying architecture exploration and system-level design. It allows designers to focus on the functionality of the design, while abstracting away implementation details that will be added at lower abstraction levels [13]. Transaction level models use software *function calls* to model the communication between blocks in a system. This is in contrast to hardware RTL and gate level models, which use *signals* to model the communication between blocks. For example, a transaction level model would represent a burst read or write transaction using a single function call, with an object representing the burst request and another object representing the burst response. An RTL HDL model would represent such a burst read or write transaction via a series of signal assignments and signal read operations occurring on the wires of a bus [46].

Complexity management, particularly at the highest level of design, has led to the emergence of TLM. The primary goal of TLM is to dramatically increase simulation speed, while offering enough accuracy for the design task at hand. The increase in speed is achieved by the TLM abstracting away the number of events and amount of information that have to be processed during simulation to the minimum required. In summary, the benefits of Transaction Level Modeling are as follows [13]:

- Allows to tackle complexity by hiding implementation details
- Faster simulation (up to 1000x) than RTL
- System level design exploration and verification are simplified
- Early platform for software development can be quickly developed
- Deterministic test generation is more effective and less tedious than at RTL,

since tests are written without taking care of the communication protocol between components

Chapter 3

Specifying Transactors

Inside a TLM-RTL transactor, we need to implement one or more RTL hardware protocols to accomplish a particular task on the RTL module. These protocols are generally specified by the protocol designers in natural languages such as in English texts. According to the proposed transactor generation methodology described in Section 1.2, we create a formal model of the transactor protocol taking the natural language text as reference. The formal models can be either in AsmL or in graphical FSM.

In this chapter, we will describe how to specify transactors in AsmL and its translation to SystemC. Also, specifying transactor in graphical FSM and AsmL code generation from FSM is also discussed.

3.1 Specifying Transactor in AsmL

We propose to create formal models of the transactor in AsmL based on the natural language text specification. AsmL models are precise, concise and readable to a wide range of people who have different areas of expertise due to its simple and intuitive language constructs. Also, syntax and semantics of AsmL is formalized and thus it gives us the opportunity to verify formally the transactor protocol at early stage of the SoC design process. This verification will enhance the confidence in the correctness of the finally generated transactor. So, once the AsmL model is completed and verified, it can be used to automatically generate the transactors in other languages such as in SystemC. The AsmL specification is translated to SystemC based on our proposed syntax and semantics translation rules.

3.1.1 AsmL Subset

We have chosen a subset of AsmL for transactor specification. The subset contains constructs and symbols that can be used for RTL hardware protocol specification. Figure 3.1 shows the chosen subset of AsmL keywords for transactor specification. Enumeration declaration, variable declaration, constant declaration, comment lines, step statements, iteration statements, conditional expressions, assignment statements, assertion statements, mathematical and logical symbols, etc. are included in the subset. Non-deterministic and high level software specification related keywords are not handled. To specify transactor in AsmL, we choose the data types shown in Table 3.1.

Table 3.1: Subset of AsmL Data Types

Data Type	*Meaning*
`Boolean`	The type containing the values `true` and `false`
`Byte`	8-bit unsigned integer type
`Short`	16-bit signed integer type
`Integer`	32-bit signed integer type
`Long`	64-bit signed integer type
`Char`	Unicode character type. Used to represent a single bit which consist of '1', '0', 'X' or 'Z'
`String`	Unicode string type. Used to represent bit vectors which consist of '1', '0', 'X' and 'Z'
`Seq of A`	The type of all sequences formed with elements of type `A`

=	:	and	not
<=	<	as	or
<>	=	const	otherwise
>=	>	else	public
(	//	elseif	require
)	:=	enum	skip
*	ensure	false	step
+	result	if	then
,	return	import	true
-	var	match	until
/	ref	mod	while

Figure 3.1: AsmL Subset

3.1.2 Hardware Data Types in AsmL

AsmL is mostly designed for software specification and it lacks hardware related data types to specify hardware. As we are proposing to describe RTL protocols in AsmL, we need to add data types, constants and functions to specify hardware in AsmL.

We declare the data types for RTL ports as shown in Figure 3.2

```
__Set4Val = {'1','0','X','Z'}
public type Logic = Char where (value in __Set4Val)
public type Lv_2 = String where (Size (value) <= 2)
public type Lv_4 = String where (Size (value) <= 4)
public type Lv_8 = String where (Size (value) <= 8)
public type Lv_16 = String where (Size (value) <= 16)
public type Lv_32 = String where (Size (value) <= 32)
public type Lv_64 = String where (Size (value) <= 64)
```

Figure 3.2: Hardware Data Types in AsmL

For instance, we have declared *Logic* type as an alias of *Char* type to represent single wire RTL port. Here, we have put a type constrain so that variables declared as *Logic* type can only have the value '1', '0', 'X' or 'Z'. These letters are used to represent 4 valued data type as logic 1, logic 0, unknown and high impedance, respectively. To represent logic vectors, we have used *String* types which are groups of characters. We also put length constraints on the *String* types according to the port bus-width. For example, variables declared as type *Lv_2* can hold information for 2 bit width RTL port. Assigning a string of length larger than 2 with the variable will cause a constrain violation error.

The constants shown in Figure 3.3 are also declared for assigning them with single bit RTL ports.

public const LOGIC_0 = *'0'*
public const LOGIC_1 = *'1'*
public const LOGIC_X = *'X'*
public const LOGIC_Z = *'Z'*

Figure 3.3: Hardware Constants in AsmL

We also declare two functions (toInt and toLv) to convert binary strings composed of '1', '0', 'X' and 'Z' to their equivalent decimal value and vice-versa namely. These functions are frequently used inside a transactor specification because they deal with both binary strings for the RTL ports and decimal values for the TLM function parameters.

public toInt (s **as** String) **as** Integer
public toLv (n **as** Integer) **as** String

Figure 3.4: Binary String to Decimal and Vise-Versa Conversion Functions

The algorithm for binary string to decimal and vise-versa conversion functions are shown in Figures 3.5 and 3.6, respectively.

```
Procedure toInt (S: null terminated string consist of characters '1', '0', 'X' or 'Z') as
Integer
N := 0
L := Length ( S )
For I := 0 to L – 1
        If ( S [ L - 1 – I ] = 'X' or S [ L - 1 – I ] = 'Z' ) then
                ShowMsg ( "Cannot Convert to Decimal" )
                Exit Procedure
        Else
                N := N + S [ L - 1 – I ]  * ( 2 ^ I )
Return (N)
End Procedure
```

Figure 3.5: Binary String to Decimal Conversion Algorithm

```
Procedure toLv (N: Integer containing Decimal value) as String
I := 0
Q := 0
Do
        S [ I ] := ToChar (Q mod 2)
        Q := Q div 2
        I := I + 1
Loop until Q = 0
ReverseString (S)
Return (S)
End Procedure
```

Figure 3.6: Decimal to Binary String Conversion Algorithm

3.1.3 The Step Rule

In AsmL, we describe the behavior of a system in a step-by-step correspondence. So, to describe an RTL protocol in AsmL, the steps to perform the task are determined first. We define *"each step corresponds to one clock cycle. It means the codes between two consecutive steps are considered to be executed in a single simulation clock cycle in SystemC"*. We will refer to this rule as the *step rule*.

3.1.4 Guidelines for Specifying Transactor

1. AsmL, as the name implies is a state machine. So, when we want to describe an RTL protocol in AsmL, the first task is to find the distinct steps or states to perform the operation and then assign state names with them. An enumerated data type with the state names can be declared as a type for state variables (say, *CurrentState*). Then initialize the state variable with *Initial State.*

2. If there is more than one clock signal port then specify the clock name which will be used as the clock signal for the state machine using the function *SetClockSignal (<ClockSigName:Logic>, < isPosEdge:Boolean>).*

3. *The step rule* must be followed when writing the specification. The keywords and constructs used in the specification must be in the AsmL subset defined in Section 3.1.1

4. Sometimes it is necessary to repeat a segment of code for a specific number of times. To do this, a state containing the segment of code is made and a variable is declared for counting the number of times that the state machine has reached that state. The state machine will come to the same state, repeat the segment of code and increment the counter variable. When the counter variable reaches its final value then we update the state variable to next state.

5. In SystemC simulation, the transactor is connected with an RTL unit. The RTL unit responses, according to the requests from the transactor ports. But in AsmL specification, the transactor's RTL port are not connected with any RTL unit. So, for proper execution of the AsmL specification, we can use

standard input (i.e. *keyboard*), to give RTL responses or use a response file from where the transactor will read the RTL responses. It also gives us the opportunity to run the specification for different scenarios. For debugging AsmL specification, we can use standard output (i.e. *display*), or output file where different values of the transactor variables will be shown or stored for different states. These standard input/output or file operation functions that are used for giving responses and debugging are removed when the specification is translated to SystemC.

3.1.5 Limitations of AsmL

1. AsmL supports only one dimensional array (sequences) and the array elements cannot be modified at run time. At run time they can only be read, but can not be updated with new values.

2. AsmL does not support function parameters to pass by reference. Though AsmL has a keyword **ref**, but it is not yet implemented for passing function parameters by reference. In AsmL, a function parameter can only be read. Assigning any value with function parameter (which is normally done when it is passed by reference) in the body of the function will generate an error. But in SystemC, we use pass by reference frequently and assign values to it when its value needs to be read by the caller function.

3.2 Translation from AsmL to SystemC

The translation from AsmL to SystemC is done based on several rules so that the original behavior of the AsmL code is preserved in the translated SystemC code. Ali Habibi [24, 25] proposed some rules for AsmL to SystemC translation. We have expanded and in some cases modified some of these rules according to our definitions.

3.2.1 Data Type Mapping

AsmL basic data types are mapped to their equivalent SystemC data types as shown in Table 3.2

Table 3.2: AsmL Data Type Mapping with SystemC

AsmL	*SystemC*	*Comments*
`Boolean`	`bool`	Data types for user defined variables
`Byte`	`unsigned char`	
`Short`	`short`	
`Integer`	`int`	
`Long`	`long`	
`Char`	`char`	
`Logic`	`sc_logic`	Data types for user defined hardware orientated variables.
`Lv_n` (*where, n=2,4,…,64*)	`sc_lv < n >`	

3.2.2 Semantic Translation

The execution semantics of a language defines how and when the various constructs of a language should produce a program behavior. For example, the semantics may define the strategy by which expressions are evaluated to values, or the manner in which control structures conditionally execute statements. Semantics of a programming language tell how the program is executed. AsmL program executes differently compared to other sequential languages due to its update semantics. On the other hand, SystemC has its own semantics for simulating hardware like process scheduling, simulation timing, etc. In the following sections, we will discuss some aspects of the semantic translation from AsmL to SystemC.

AsmL Variable

In sequential languages like C/C++, when we assign a value with a variable, it is assigned immediately. If we read the variable in the next statement, we get the assigned value [28] as shown in Figure 3.7.

```
// C code
int a = 1;
a = 2 ;
printf ( ''a=\%i'' , a ) ; // prints a=2
```

Figure 3.7: Variable in C language

In contrast, variables declared in AsmL, behave different from sequential programming languages. If we assign a value with an AsmL variable then read it in the same step, we will get its old value, not the newly assigned value. Whenever there is a step statement, the variables are *updated* with the newly assigned values as shown in Figure 3.8.

```
// AsmL code
Var a as Integer = 1
step
  a := 2
  WriteLine ( ''a='' + a ) //prints a=1
step
  WriteLine ( ''a='' + a ) //prints a=2
```

Figure 3.8: Variable in AsmL

In SystemC, the signals declared as `sc_signal` `<type>` also behave similarly to the AsmL variable. If we write a value to a SystemC signal, it is not *updated* at that simulation (δ) cycle. If we read that signal at the same simulation cycle, we will get its old value, not the newly written value. For thread process, the signals are *updated* with newly written values whenever the program reaches a `wait ()`

statement as shown in Figure 3.9

```
// SystemC code
sc_signal <unsigned int> a ;
a.write(1) ;
wait (SC_ZERO_TIME) ;
a.write (2) ;
cout << ``a='' << a.read() << endl ; //prints a=1
wait (clk->posedge_event());
cout << ``a='' << a.read() << endl ; //prints a=2
```

Figure 3.9: Variable in SystemC

So, we found that there is a semantical similarity between AsmL variable and SystemC signals. We translate variable declaration in AsmL to SystemC as shown in Table 3.3

Table 3.3: Translation of AsmL Variable to SystemC

AsmL	*SystemC*	*Comments*
`Var a as Integer`	`sc_signal <int> a ;`	Variable Declaration
`Var x as Short = 4`	`sc_signal <short> x ;` `x.write ( 4 ) ;` `wait (SC_ZERO_TIME) ;`	Variable Declaration with Initial Value

sc_signal cannot be declared inside a member function of a SystemC module. To solve this problem, we declare all the local AsmL variables globally in the SystemC class using the naming convention: *<FunctionName>_<VariableName>*. This naming convention solves the problem of multiple declarations of variables which have the same name in different member functions.

Update Semantics and Step Statement

In AsmL, all variables are *updated* whenever the program reaches any `step` statement. In SystemC, all signals are *updated* whenever the program reaches a `wait ()` statement.

We translate AsmL `step` in SystemC to `wait (clk->posedge_event())` where `clk` is the clock signal name and `posedge_event` indicates the positive edge event of the clock signal. This translation satisfies the *update* semantics and also respects the *step rule* as this `wait` statement will cause the SystemC scheduler to increment its simulation time by one clock cycle [19].

For transactors that communicate with cycle accurate RTL models through request-grant protocols, sometimes it is necessary for them to *update* the RTL ports a little time (*setup time*) before the clock event occurs. In that case, we put a statement `wait` (t_{bs}) before the `wait(clk->posedge_event())` where t_{bs} = T − t_{su} [T=Clock period, t_{su} = setup time]

Table 3.4: Translation of AsmL Update Semantic to SystemC

AsmL	*SystemC*
`step`	`Update () ;` *where,* `Update ()` `{ wait (`t_{bs}`) ;` ` wait (clk->posedge_event()); }`

3.2.3 Syntax Translation

The syntax of a language describes the possible combinations of symbols that form a syntactically correct program. The meaning given to a combination of symbols is handled by semantics. Each programming language has its own syntax and it describes *how a program statement or construct can be written.* For instance, sequential languages like C and Pascal, both have different syntax for integer variable declaration, but their semantics are the same. AsmL has its own syntax for writing code. SystemC has similar syntax like C++. In the following sections, we will discuss syntax translation from AsmL to SystemC.

Conditional Statement

Conditional statements allow the execution of a statement or a block of statements depending upon conditions which truth value may change while the program is running. The mapping between AsmL and SystemC conditional statements is shown in Table 3.5

Table 3.5: Translation of Conditional Statements

AsmL	*SystemC*	*Comment*
`if (condition_1) then` `statement_1` `elseif (condition_2) then` `statement_2` `else statement_3`	`if (condition_1)` `{` `statement_1 ;` `}` `else if (condition 2)` `{` `statement_2 ;` `}` `else` `{` `statement_3 ;` `}`	*condition* is a 1st order Boolean expression or an integer expression. The block is executed if the *condition* is evaluated as *true* or any *non-zero value*. If the *condition* is evaluated as *false* or *zero*, the *else* block is executed.
`match (exp)` `val_1:` `statement_1` `val_2:` `statement_2` `otherwise` `statement_3`	`switch (exp)` `{` `case val_1:` `statement_1;` `break;` `case val_2:` `statement_2;` `break;` `default:` `statement_3;` `}`	*exp* is an integer expression. Val_1,Val_2 are integer constants. If the evaluated *exp* matches with the case constants listed, then the corresponding case block is executed. Otherwise, *default* block is executed. The *break* statement is used to execute the blocks in a mutually exclusive manner.

Iteration Statement

Iteration is the repetition of a statement or a block of statements in a program. The mapping between AsmL and SystemC iteration statements is shown in Table 3.6

Table 3.6: Translation of Iteration Statements

AsmL	*SystemC*	*Comment*
`step while (condition)` `statement_1` `...` `statement_n`	`while (condition)` `{` `statement_1 ;` `...` `statement_n ;` `Update () ;` `}` *where,* `Update ()` `{` `wait (t_bs) ;` `wait (clk-posedge_event();` `}`	*condition* is a Boolean expression or an integer expression. The loop is executed if the *condition* is evaluated as *true* or any *non-zero value.*
`step until (condition)` `statement_1` `...` `statement_n`	`while !(condition)` `{` `statement_1 ;` `...` `statement_n ;` `Update () ;` `}`	The loop is executed if the *condition* is evaluated as *false* or *zero.*

Assertion Statement

AsmL supports both pre-condition and post-condition assertion statements. We translate them to SystemC as shown in Table 3.7

Table 3.7: Translation of Assertion Statements

AsmL	*SystemC*	*Comment*
`require (condition)`	`assert (condition) ;`	*Pre-condition* assertion statement. The assertion fails and generates exception message if the condition is evaluated as false.
`ensure (P(result))`	`assert ( P( ret_val ));`	*Post-condition* assertion statement. P (*result)* is a propositional function where *result* keyword contains the return value of a function. In SystemC, *result* is replaced by the return expression (`ret_val`) of the function.

Symbols and Operators

Different symbols are used in a programming language to represent special meanings. Operators are symbols that operate on one or more expressions and thus produce a value. The mapping between AsmL and SystemC symbols and operators are shown in Table 3.8

Table 3.8: Translation of Symbols and Operators

Symbols

AsmL	*SystemC*	*Comment*
(	(or [	Opening bracket, translated to '[' for arrays
)	) or]	Closing bracket, translated to ']' for arrays
//	//	Comment

Arithmetic Operators

AsmL	*SystemC*	*Comment*
+	+	Add
-	-	Subtract
*	*	Multiply
/	/	Division

Conditional Operators

AsmL	*SystemC*	*Comment*
=	==	Is equal to
<>	!=	Is not equal to
>	>	Is greater than
<	<	Is less than
>=	>=	Is greater than or equal to
<=	<=	Is less than or equal to

Logical Operators

<table>
<tr><th>AsmL</th><th>SystemC</th><th>Comment</th></tr>
<tr><td>and</td><td>&&</td><td>Logical AND</td></tr>
<tr><td>or</td><td>||</td><td>Logical OR</td></tr>
<tr><td>not</td><td>!</td><td>Logical NOT</td></tr>
</table>

Assignment Statement

In AsmL, assignment of a value with a variable is done by using the assignment operator ':='. Inside a transactor, there are several kinds of variables. Some are

TLM function parameters, some are RTL ports and some are user defined AsmL variables. When an RTL port variable value needs to be assigned with a TLM function parameter or any user defined *Integer* type variable, it must be converted to its corresponding decimal value using the function *toInt*. In the same way, when an *Integer* type variable needs to be assigned with an RTL port, it should be converted to its corresponding binary string using the function *toLv*. The translation for assignment statements between different variables to SystemC is shown in Table 3.9

Table 3.9: Translation of Assignment Statements

AsmL	*SystemC*	*Comment*
`a := b`	`a = b ;`	Where *a, b* are TLM function parameters
`a := toInt (b)`	`a = toInt(b.read()) ;`	Where, *a* is TLM function parameter; *b is* RTL port or user defined variable
`a := toLv (b)`	`a.write ( toLv (b) ) ;`	Where, *a* is RTL port or user defined variable; *b* is TLM function parameter
`a := b`	`a.write( b.read());`	Where *a, b* are RTL ports or user defined variables

Enumeration and Constant Declaration

Enumerations are user defined integer types which instance objects can be assigned with constants declared as the enumerators. Constants are objects which values do not change while the program runs. The mappings between AsmL and SystemC enumeration and constant declarations are shown in Table 3.10

Table 3.10: Translation of Enumeration and Constant Declarations

AsmL	*SystemC*	*Comment*
`enum typeName` `enumurator1` `enumurator2` ... `enumuratorN`	`enum typeName` `{` `enumerator1 = 0 ,` `enumurator2 ,` ... `enumuratorN` `} ;`	Enumerated type declaration. The variables of the user defined type *typeName* can only have the values of the *enumerators* listed.
`const X as Integer = 4`	`const int X = 4 ;`	Constant declaration

In SystemC, enumeration declaration cannot be done inside member function of a module. So, we declare enumerations and their enumerators globally in the class module using naming convention *<FunctionName>_<Enumuration>* and *< FunctionName>_<Enumurators>*. This naming convention solves the problem of multiple declarations of enumerations and their enumerators which have the same name.

Generating Blocks

AsmL, in contrast to other programming languages does not use braces or keywords like `begin` or `end` to specify a block. AsmL uses appropriate *number of white space* at the left of the line to determine a block. Blocks can be nested. We have developed a stack based algorithm to generate blocks in SystemC as shown in Figure 3.10.

If an AsmL line starts more to the right than its previous line then we push its number of left white space and a reason ID which indicates the previous AsmL line. If an AsmL line starts more to the left than its previous line then the previous block or blocks should be closed first. Here we start popping the number of left white spaces from the stack and depending on the reason ID we close blocks until the popped number of left white space is aligned with the current line number of left white space. Table 3.11 shows an example of block generation.

NoOfLeftWS: Holds the number of white space character at the left of an AsmL line

GetNoLeftWS () as Integer: Returns the number of left white space of the next AsmL line

PrevNoLeftWS: Stores the number of left white space of the previous AsmL line

Push (x, y): Push data *x, y* to stack

AsmLReason: Holds information about the AsmL line whether it is an *if statement*, *step*, *loop statement*, etc.

PrevAsmLReason: Holds *AsmLReason* for the previous AsmL line

OpenBlock (AsmLReason, NoLeftWS): Put '{'or block starting characters based on *AsmLReason*

Pop (x, y): Pop data to *x, y* from stack

CloseBlock (AsmLReason, NoLeftWS): Put '}' or block ending characters based on *AsmLReason*

```
Do
        NoLeftWS = GetNoLeftWS ()

        If (NoLeftWS > PrevNoLeftWS)
                Push ( PrevAsmLReason, NoLeftWS)
                OpenBlock (PrevAsmLReason, NoLeftWS)

        If (NoLeftWS < PrevNoLeftWS)
                Do
                   Pop (AsmLReason, PopedNoLeftWS)
                   If (NoLeftWS < PopedNoLeftWS )
                           CloseBlock (AsmLReason, PopedNoLeftWS)
                   Else Push (AsmLReason, PopedNoLeftWS)
                While (PopedNoLeftWS > NoLeftWS and Stack Not Empty)

        PrevNoLeftWS = NoLeftWS

Loop until (End of Specificaion)

CloseAllBlocks()
```

Figure 3.10: Algorithm for Generating Block

Table 3.11: Translation of Blocks

AsmL	*SystemC*
<pre>if a = 1 Then x := 20 y := 30 else x := 50 if a = 2 then y := 60 z := 10 // z is out of the block of else // as it started aligned with // the else statement</pre>	<pre>if (a.read() == 1) { x.write(20) ; y.write(30) ; } else { x.write(50) ; if (a.read () == 2) { y.write(60) ; } } z.write(10) ;</pre>

3.3 Specifying Transactor in Graphical FSM

An FSM can be represented graphically, which would help the designer to visualize and design in a more efficient way. In this section, we will discuss specifying transactor behavior by drawing graphical FSMs. We propose to create a formal model of the transactor protocol by drawing FSMs based on the natural language text specification. After the protocol is drawn, we translate the FSM description to AsmL using our developed AsmL code generation algorithm.

Figure 3.11 represents an FSM for an UTOPIA transactor protocol. The detailed protocol description is done in Chapter 5. The gray circles represent *state.* The state label is shown in the middle of the circles. The triangular ending arrow indicates the *initial state* as shown in state S_CheckCellAvailable. A state may have *action* statements. They are shown in white boxes which contain the actions of the assigned state. The arrows are the *transition lines.* It tells us about what is the next state in the successive clock cycles. A transition line may have a *guard* or a Boolean *condition* associated with it. In that case, the transition only occurs if the guard is evaluated as *true.* We can also set *transition line priorities* if more than one

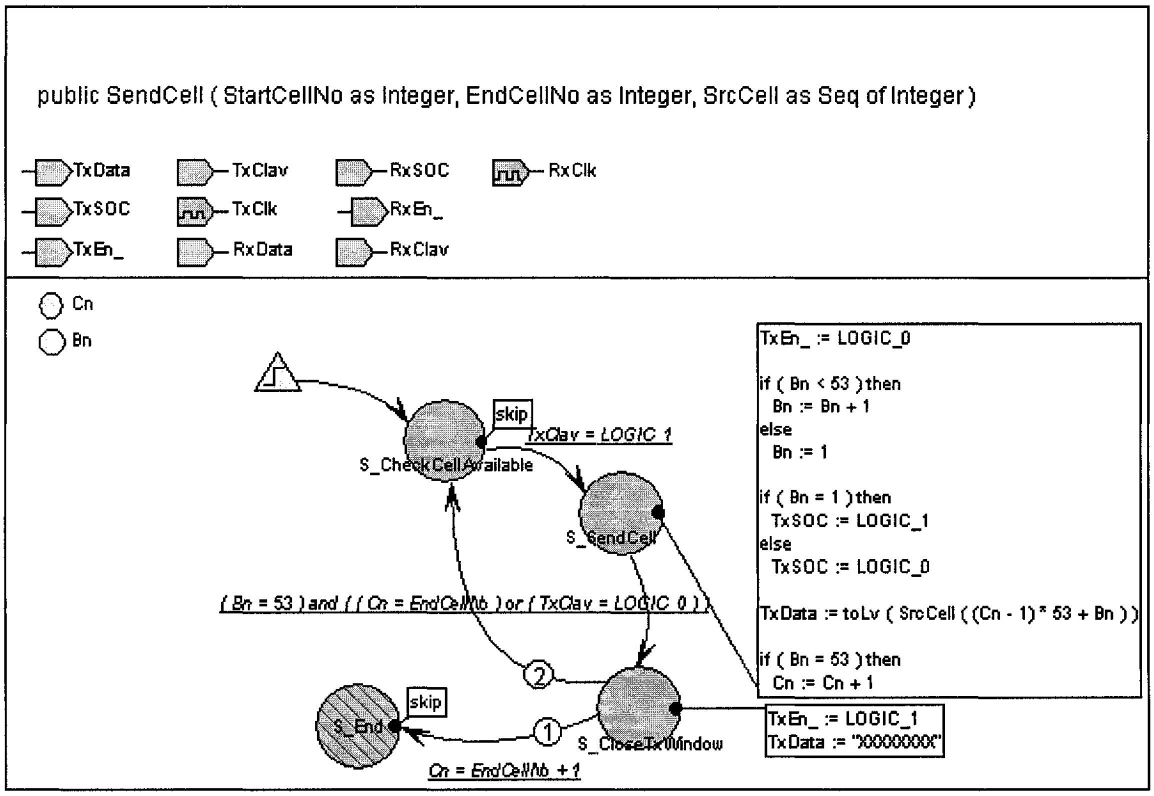

Figure 3.11: A Sample Graphical FSM (Snapshot from State Editor)

transition line comes out from a state as shown in state *S_CloseTxWindow.* This sets the order in which the transition conditions will be evaluated.

We can also set a state as *trap state.* It is the state that the machine will reach if the variable which contains the information of the current state is assigned with any illegal value. *default state* is the state which the machine will reach next, if all the next state transition condition for a state is evaluated as *false.*

3.3.1 Guidelines for Specifying Transactors in Graphical FSM

An FSM drawing consists of *states, actions, transition lines, conditions,* etc. Our code generation algorithm imposes that the following rules and guidelines must be followed when specifying a transactor protocol using FSM.

1. The FSM is drawn using *state, action, transition line, condition or guard* components in the *Active HDL State Editor* tool. The graphical description is saved in *(Active HDL State Machine Format) ASF* format [4]

2. The *initial state* is indicated with the *Initial State Indicator* component.

3. Variables of integer type can be declared using the *variable/signal* component. Also, integer constants can be declared using the *constant* component.

4. The *action* statements in a state must be written in the syntax of AsmL. They are executed simultaneously according to the *update* semantics. If any state has no action, a "skip" statement is written as an empty action.

5. State transition occurs after one clock cycle and updates of the variables and ports are fired.

6. The *conditions* of the transition lines must be also in the syntax of AsmL. If more than one transition line come out from a state, we assign priority to each transition line. This priority sets the order in which the transition conditions will be evaluated. An unconditional transition line must have the least priority.

7. Unlike other FSMs, an FSM inside a transactor must terminate when the operation on the RTL side is completed. So, we indicate the state at which the FSM will terminate by setting it as *trap state.*

8. The graphical FSM approach should be used if the protocol of the transactor is simple and small. If the protocol is large, the FSM drawing can get messy and difficult to understand.

3.3.2 FSM Representation in ASF Format

We used *Active HDL State Editor* to draw the FSM which outputs a textual representation of the FSM in ASF format. The ASF format is a new and reach file format to represent FSM in textual from. A portion of the format is shown in Figure 3.12

State:
    S [ID] [isDefStat|isTrapState]
Label:
    L [ID] [ObjectID] ... [Label Description]....[Label]
Action:
    A [ID] [StateID] .... [Action Statement]
Transition Line:
    W [ID] [Priority] [SrcStateID] [DstStateID]
Condition:
    C [ID] [TranLineID]....[Condition Expression]

Figure 3.12: ASF Format

In the *State* object, the information whether it is set as *Default state* or *Trap state* is stored in LSB 2 bits in the *[isDefStat|isTrapState]* Field. The LSB bit stands for *isDefState* and its next bit stands for *isTrapState.*
In the *Transition Line* object, the priority information is stored in 12 bits starting from LSB in the *[Priority]* field.

A sample line in the ASF file looks like the following:

A 25 10 4 TEXT "Actions" |94596,185504 1 0 0 "D=0;"

An ASF file contains each object's graphical information. They are used to show the FSM graphically in the *Active HDL State Editor.* But they do not have any use in generating code. So, we ignore that information.

3.3.3 FSM Objects

The basic FSM objects are *states, actions, transition lines, conditions*, etc. These objects are read in data structures from the ASF file as shown in Figure 3.13

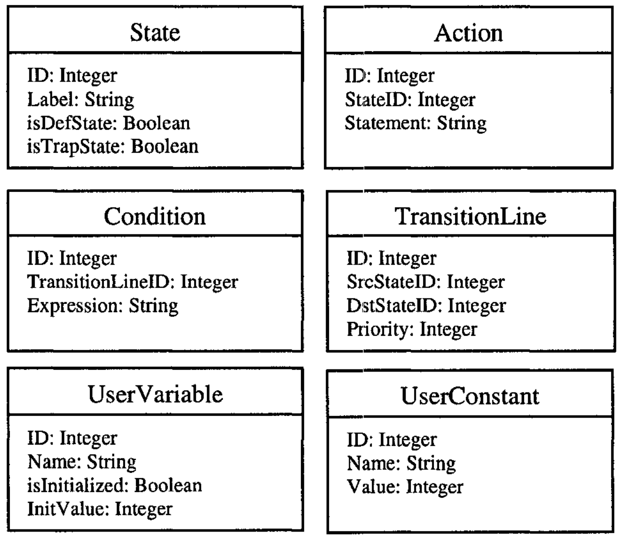

Figure 3.13: FSM Objects

3.3.4 AsmL Code Generation

We generate AsmL code by reading the FSM objects according to the algorithm shown in Figure 3.14. Graphical FSM is a discrete structure consisting of vertices and edges like directional graph. The algorithm is developed with the flavor of directional graph traversing [43].

An enumerated type state variable `CurrentState` is used to hold the present state. A `step while` block [34] is generated with the condition that the loop will terminate if the `CurrentState` is evaluated as the *trap state*. The core FSM code is generated in a `match` block [37] which is used to switch to different states depending on the `CurrentState`. For a State, the code generator writes its Label followed by a

FSM_Drawing: It is an FSM drawing for the transactor protocol.
Write (s: string): Write string s to the code generation file

```
for each UserConstant in FSM_Drawing
  Write ("const " & UserConstant.Name & " as Integer" & " = " & UserConstant.Value )

for each UserVariable in FSM_Drawing
  Write ("var " & UserVariable.Name & " as Integer" & " = " & UserVariable.InitValue )

Write (" step while ( CurrentState <> " & State(TrapStateIndex).Label & " )" )
Write ("match CurrentState")

for each State in FSM_Drawing
  Write ( State.Label & ":")
  for each Action in FSM_Drawing
   if Action.StateID = State.ID then
      Write ( Action.Statement )

 for each TransLine in FSM_Drawing
   if TransLine.SrcStateID = State.ID
      new MultyTransLine
      MultyTransLine.Priority := TransLine.Priority
      Multy TransLine.DstStateLabel:= GetLabel(TransLine.DstStateID)
      for each Condition in FSM_Drawing
        if Condition.TransLineID = TransLine.ID then
           MultyTransLine.Expression := Condition.Expression
           MultyTransLine.isConditional := true

      if Condition not found
         MultyTransLine.isConditional := false

  Sort MultyTransLine objects on Priority in Ascending order

  for each MultyTransLine
   if MultyTransLine.isConditional= true then
     Write ("if " & MultyTransLine.Expression & " then" )
     Write (" CurrentState := "& MultyTransLine.DstStateLabel )
  else Write ("CurrentState := "& MultyTransLine.DstStateLabel)

  if there exist DefState in State and (For all ( MultyTransLine.isConditional) = true ) then
    Write ("else CurrentState := " & DefaultState.Label)

if there exist TrapState in FSM_Drawing
  Write ("otherwise:")
  Write ( " CurrentState := " & TrapState.Label)
```

Figure 3.14: AsmL Code Generation Algorithm

colon ':'. Then the *Action* Statements associated with the state are written. Thereafter, the code generator gathers all the *transition line* and *condition* information of that state. If there are more than one *TransLine* coming out from the state, then *TransLine* is sorted based on the assigned *priority* in ascending order. Then the conditions for determining the next state are written using `if` or `else if` statements. If any state is set as *default state* and there exists no unconditional transition line then assigning *default state* as the next state is done using an `else` statement. To handle any illegal assignments of states, the *trap state* is assigned as next state in the `otherwise` section of the `match` block.

After the AsmL code is generated from the graphical FSM and it is executed and verified, it is then translated to SystemC according to our developed semantics and syntax translation rules described in Section 3.2

Chapter 4

SystemC Transactor Generator Tool

We have developed the *SystemC Transactor Generator Tool* for automatic generation of SystemC transactors both from AsmL specification and from graphical FSM description. The tool consists of a Graphical User Interface (GUI), an FSM to AsmL Code Generator, an AsmL to SystemC Compiler and other necessary modules. The tool also provides features to generate transactor libraries for standard protocols.

The tool is developed for the Microsoft Windows environment. We have used Microsoft Visual Basic 6.0 [9] to develop the tool. The coding method is Object Oriented (OO). OO programming makes the tool (which consists of approximately 10,000 lines of codes) modular, easy to manage, maintain and debug.

The *SystemC Transactor Generator Tool* consists of several modules as shown in Figure 4.1

The tool takes as input the *TLM Interface* which is the declarations of the TLM functions of the TLM module and the *RTL Interface* which is the declarations of RTL ports of the RTL module. In *Code Settings*, the transactor generation method, the clock period & setup time, library generation information, etc., are specified. Then the tool generates an AsmL template which can be edited in the MS Word

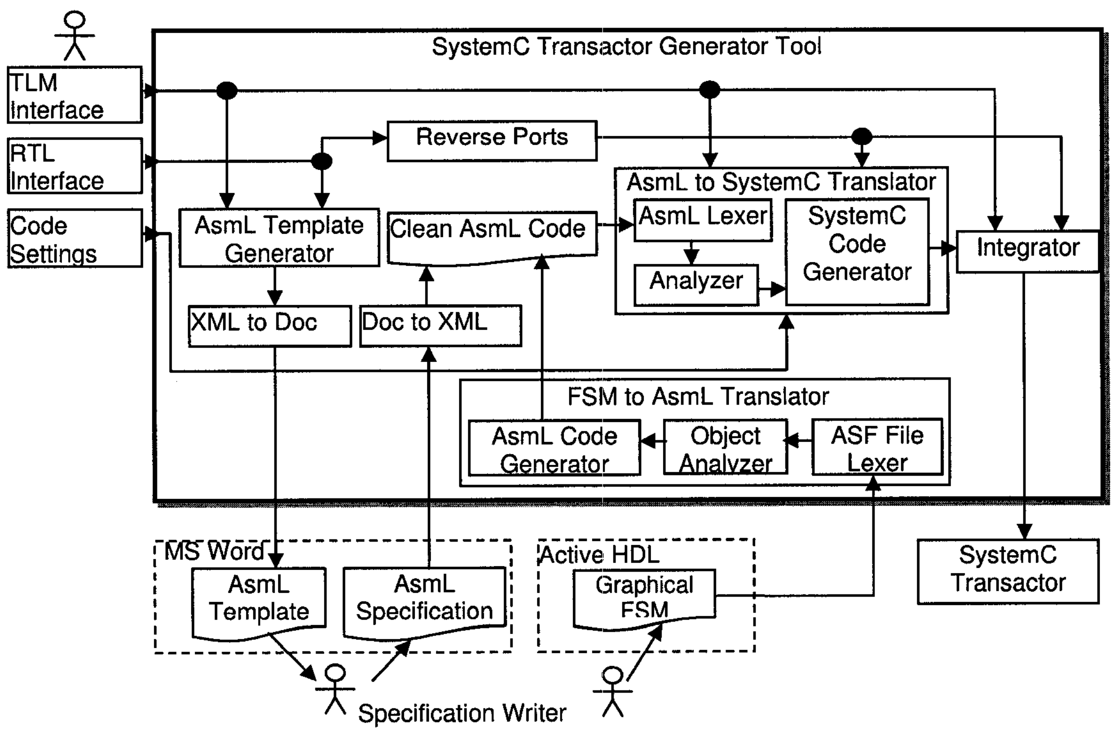

Figure 4.1: Block Diagram of SystemC Transactor Generator Tool

environment. The specification writer then writes the transactor specification in the AsmL template. This specification can be executed and used for validation and verification purposes. Then the specification is given as input to the tool. The tool then extracts unformatted ASCII AsmL code text from it and passes it to the *AsmL to SystemC Translator*. This module translates the AsmL specification to SystemC. Also, the tool supports another approach where the transactor protocol is drawn as graphical FSMs in *Active HDL State Editor* and the ASF files are given as input to the tool. The *FSM to AsmL Translator* generates AsmL code from the FSM descriptions. The generated AsmL code is then passed to the *AsmL to SystemC Translator* to generate SystemC code. The *integrator* integrates the translated SystemC code for all TLM functions and adds other necessary SystemC codes to generate the complete transactor.

The (GUI) consists of menus, toolbars, a status bar, text boxes, check boxes, buttons, etc. A screen shot of the tools GUI is shown in Figure 4.2. To work with a transactor, the user will first create a transactor project with a project name and path. A folder with the project name is created in the specified path. Inside the folder, a project file (*.tp) and three folders /Input , /Temp, /Output are created. The Input folder contains the information of the TLM interface, RTL interface, Code Settings, AsmL specification, ASF files for graphical FSM etc. Inside the Temp folder intermediate files are generated when the tool generates the SystemC transactor. The Output folder contains the generated SystemC transactor (*.cpp, *.h) files. This file system makes any transactor project portable, without losing its dependencies.

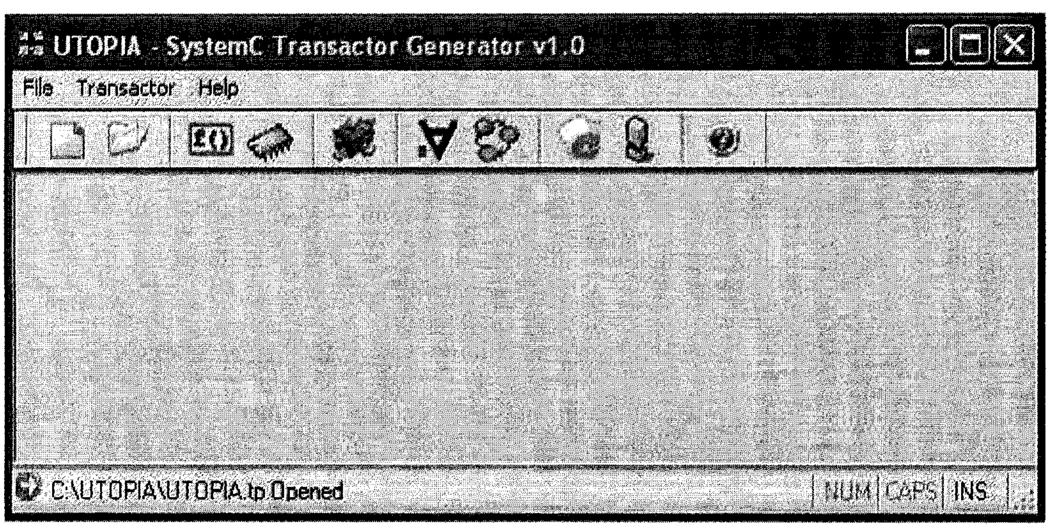

Figure 4.2: SystemC Transactor Generator Tool GUI

4.1 Input Interface

To specify a transactor, we need to give as input the TLM interface, the RTL interface and the protocol to perform the task.

4.1.1 TLM Interface

The TLM interface is the declaration of the TLM functions of the TLM module. A function has return type, function name and its parameters. The information of the TLM functions is taken using the user interface as shown in Figure 4.3. User can Add, Edit, or Remove functions from the project by clicking the corresponding button.

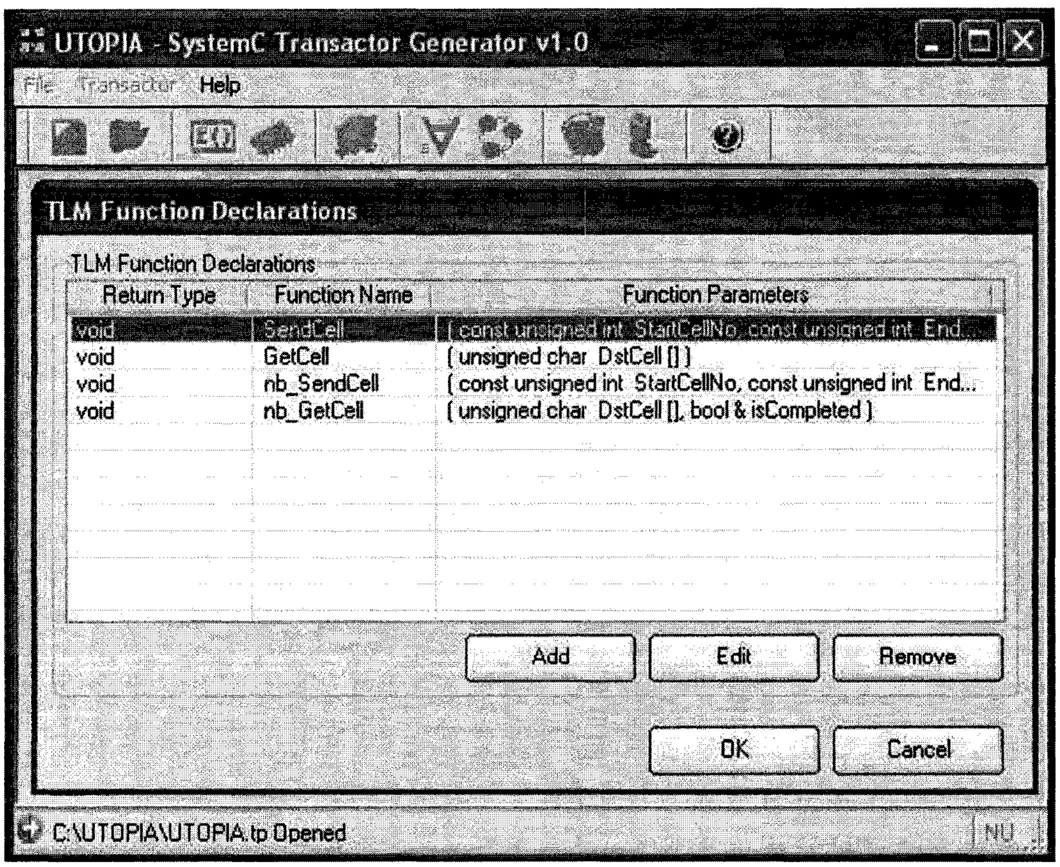

Figure 4.3: TLM Interface Input Window

We used the data structure shown in Figure 4.4 to store the TLM functions information. Arrays of objects of class *clsTlm_Func* are instantiated to hold the function's information.

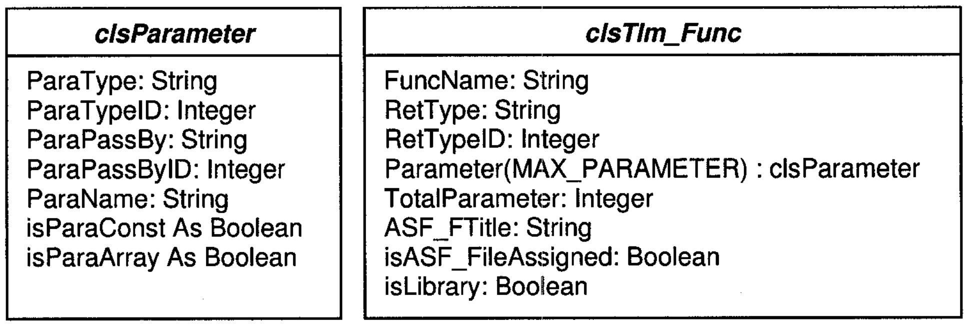

Figure 4.4: Data Structure for TLM Functions

4.1.2 RTL Interface

The RTL interface is the declaration of the RTL ports of the RTL modules. An RTL port has a name, direction, bus width, initial value, etc. The information of the RTL ports is taken using the user interface as shown in Figure 4.5. The user can Add, Edit, or Remove ports from the project by clicking the corresponding button.

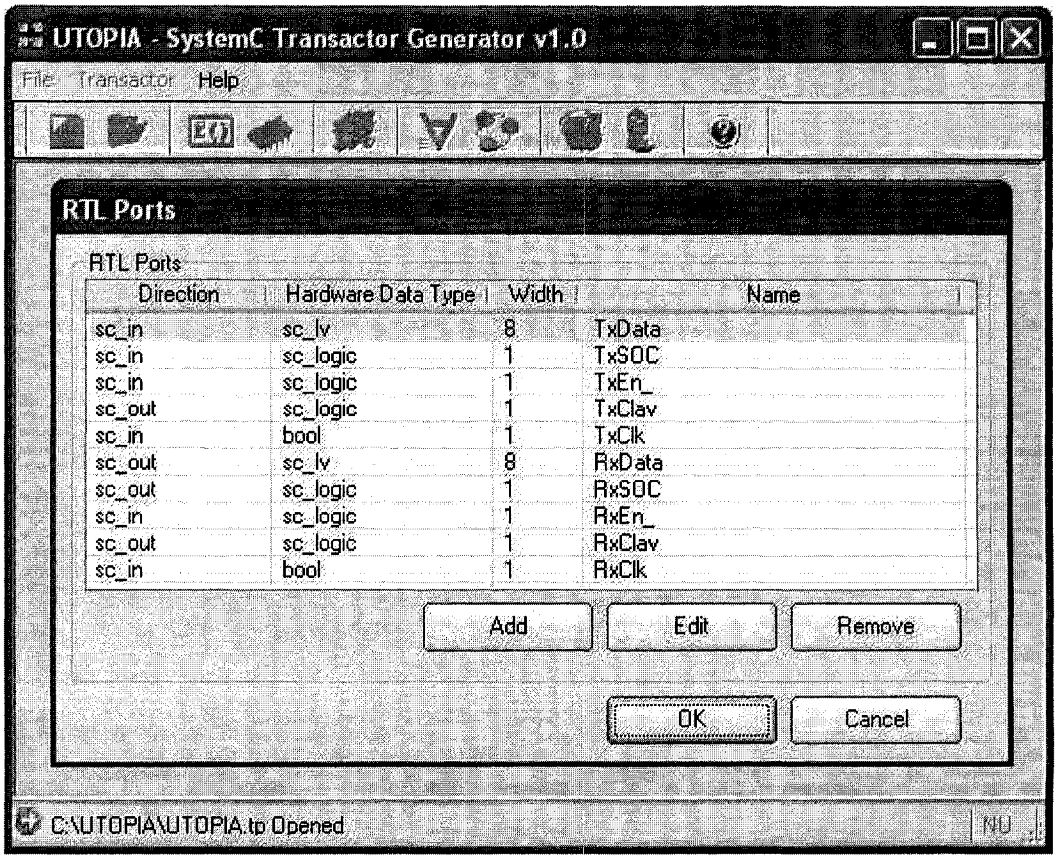

Figure 4.5: RTL Interface Input Window

We used the data structure shown in Figure 4.6 to store the RTL port's information. Arrays of objects of class *clsRtl_Sig* are instantiated to hold the port's information.

clsRtl_Sig
HDType: String HDTypeID: Integer BusWidth: Integer SignalName: String Direction: String DirectionID: Integer InitVal: String isClk: Boolean isLibrary: Boolean

Figure 4.6: Data Structure for RTL Ports

4.1.3 Code Settings

The tool provides three approaches to generate the transactor. They are the AsmL specification approach, Graphical FSM approach and creating a transactor template in SystemC. In Code Settings, the user will tell the tool in which approach the user want to generate the transactor. Also, some other information like clock period, setup time, library transactor generation information, author's information is also taken as input from here.

4.2 Generating Transactor from AsmL

4.2.1 AsmL Template Generator

To help the specification writer to write the AsmL specification of the transactor, the *SystemC Transactor Generator Tool* creates an AsmL Template. The specification writer will then use the template for writing the specification. The AsmL template consists of several *namespaces*. They are briefly described below.

Namespace for Declaring Hardware Data Types: *nsHardwareDatatype*

This namespace contains data types for declaring RTL ports in AsmL. Also, constants for 4 valued hardware data type, binary string to decimal and vise-versa conversion functions are declared here. A detailed discussion on it is done in Chapter 3. This namespace must be added to all AsmL specifications. It is also possible to make a compiled dynamic link library (DLL) of this namespace and link it at compile time when executing the specification using the *asmlc* compiler.

Namespace for Declaring RTL Ports: *nsRTL*

We declare the RTL ports of the transactor in this namespace. We also import the namespace *nsHardwareDatatype* here so that we can use the hardware oriented data types.

Namespace for specifying the TLM functions: *ns<FunctionName>*

We specify each TLM function in a separate namespace in an AsmL specification. For each TLM function, a namespace *ns<FunctionName>* is created by the AsmL template generator. We also import the namespaces *nsHardwareDatatype* and *nsRTL* here. In the template, some comments are included so the specification writer can specify the function protocol in an organized format. The format is shown in Figure 4.7

```
namespace nsFunctionName
        import nsHardwareDatatype
        import nsRTL

        //Declare Enumeration here

        //Declare function here
        public FunctionName ( var1 as type1, var2 as type2 ... )

                //Declare Constants here

                //Declare Local Variables here

                //Specify the RTL Clock Signal

                //Start writing the State Machine from here
```

Figure 4.7: Format for Writing TLM function

Namespace for Calling TLM functions for execution: *nsMain*

A namespace *nsMain* is declared, from where the functions are called. This namespace is used only for execution and debugging of the AsmL specification.

The AsmL specification can also be used to generate transactors in languages other than SystemC.

4.2.2 XML to DOC conversion and vise versa

To make the AsmL template editable and executable using Microsoft Word, the *SystemC Transactor Generator Tool* converts the template from XML format to DOC format using the tool *wordgenerator* which is distributed by Microsoft with the AsmL distribution package. The reverse work is needed, when the AsmL specification in the DOC format is given as input to the tool. The tool extracts ASCII texts from the DOC format using *wordextractor,* which is also supplied with the

AsmL package.

4.2.3 AsmL to SystemC Translator

This module translates the AsmL code to SystemC. It has three sub-modules: *AsmL Lexer, Analyzer* and *SystemC Code Generator.* They are briefly described below.

AsmL Lexer

Lexical analysis is the processing of an input sequence of characters (such as the source code of a computer program) to produce, as output, a sequence of symbols called "tokens" or "words". In our tool, we have developed a lexer to tokenize an AsmL line of code to words. One or more white spaces, double character symbols (<= >= <> := //), and single character symbols (< > =) (+ - * / : ,) are used as punctuator between words. White spaces are not considered as tokens or words, they only act as punctuator. Symbols are considered as tokens as well as punctuator between words. Here, the grammar checking is omitted because it is done once when the AsmL specification is executed by the *asmlc* compiler [37].

Analyzer

After tokenizing, the Analyzer is used to recognize the AsmL line. According to our AsmL subset, the analyzer returns an analyzed ID of the AsmL line as shown in Table 4.1.

Table 4.1: Analyzed AsmL line ID

Analyzed AsmL line ID	*Comments*
ASML_UNRECOGNIZED	Unrecognized line, Error message is generated
ASML_BLANK	Blank line consists of one or more white spaces
ASML_IF	`if` statement block
ASML_ELSEIF	`elseif` statement block
ASML_ELSE	`else` statement block
ASML_STEP	`step` statement
ASML_STEP_WHILE	`step while` loop
ASML_STEP_UNTIL	`step until` loop
ASML_VAR_DECLARATION	Variable declaration statement
ASML_CONST_DECLARATION	Constant declaration statement
ASML_ENUM_TYPE_DECLARATION	Enumerated type declaration
ASML_ENUMURATOR_DECLARATION	Enumerator declaration
ASML_COMMENT	Comment line
ASML_MATCH	`match` block
ASML_CASE	Cases for `match` block
ASML_OTHERWISE	Default case for `match` block
ASML_SKIP	`skip` statement
ASML_SET_CLK_SIG	Specify Clock Signal statement
ASML_ASSIGNMENT	Assignment statement
ASML_NAMESPACE	Declaration of namespace
ASML_IMPORT	Import external namespace
ASML_FUNC_DECLARATION	TLM function declaration
ASML_REQUIRE	Pre-condition assertion statement

A token in an AsmL line is also recognized as one of the following categories as shown in Table 4.2

Table 4.2: Analyzed AsmL token ID

Analyzed AsmL Token ID	*Comments*
ASML_WORD_UNRECOGNIZED	Unrecognized word, Error message is generated
ASML_WORD_TLM_VAR	Function parameter of the TLM function
ASML_WORD_RTL_VAR	RTL port
ASML_WORD_ASML_VAR	User defined AsmL variable
ASML_WORD_CONST	▪ Numeric Constants. (Example: 19, 25 etc.) ▪ String Constants (Example: "1X10Z" etc.) ▪ 4 valued data type constants (LOGIC_0, LOGIC_1, LOGIC_X, LOGIC_Z) ▪ AsmL Constants: `true, false` ▪ User defined AsmL Constants ▪ User defined AsmL Enumurators
ASML_WORD_SYMBOL	"<", ">", "<=", ">=", "<>", "=", ")", "(", "+", "-", "*", "/", "mod", "and", "or", "not"
ASML_WORD_TO_INT	4 valued binary string to decimal and vise-versa conversion functions
ASML_WORD_TO_LV	

SystemC Code Generator

After analyzing and recognizing AsmL line and its words, the *SystemC Code Generator* module translates the AsmL line of code to SystemC according to the discussions made in Section 3.2. Here, the grammar or syntax checking of the source AsmL code is omitted because it is done once when the AsmL specification is executed by the *asmlc* compiler.

4.2.4 Integrator

The *integrator* integrates the translated SystemC codes for all TLM functions and adds other necessary SystemC codes such as setting the initial value of the RTL ports, SystemC representation of the 4 valued binary strings to decimal and vise-versa conversion functions, etc. to generate the complete transactor. A screen shot of the tool after generating a transactor is shown at Figure 4.8

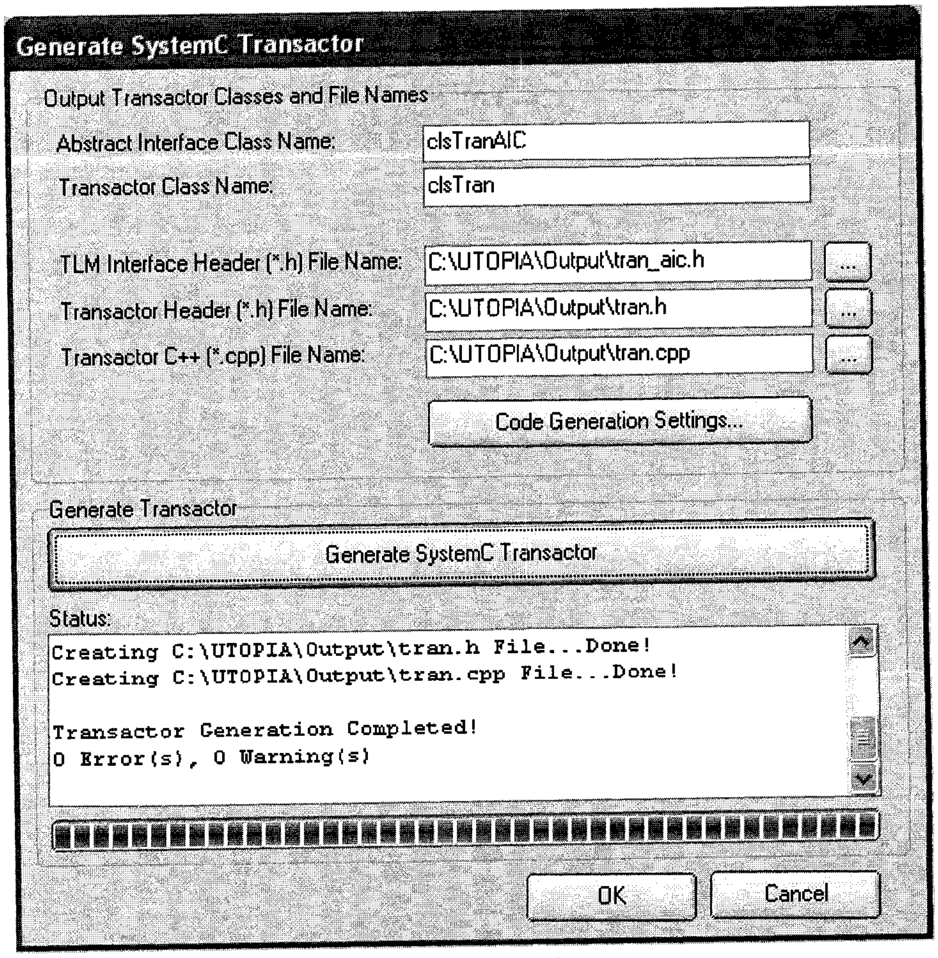

Figure 4.8: SystemC Transactor Generation

4.3 Generating Transactor from Graphical FSM

4.3.1 FSM Drawing Template

The *SystemC Transactor Generator Tool* generates templates for each TLM function for specifying the transactor protocol by FSM graphically. It contains the declaration of a TLM function and the declaration of the RTL ports. The template is opened with the Active HDL State Editor. The specification writer then draws the FSM for each TLM function in the template and then gives the ASF files as input to the *SystemC Transactor Generator Tool.*

4.3.2 FSM to AsmL Code Generator

The *ASF File Lexer* tokenizes the ASF file contents to words. A single white space is used in the ASF format as a punctuator between words. After tokenizing, the information of the FSM objects like *State, Label, Action, Transition Line, Condition* etc. is read in data structures and analyzed by the *Object Analyzer.* Then the *AsmL Code Generator* generates AsmL code according to the algorithm described in Section 3.3.4.

After the AsmL code is generated, it is then passed to the *AsmL to SystemC Translator* to generate SystemC code. The *Integrator* then adds other necessary codes with it and generates the complete transactor.

4.4 Generating Transactor Code Template

The *SystemC Transactor Generator Tool* can generate a transactor template in SystemC. The template contains RTL ports declaration and TLM functions declarations without any implementation code. The specification writer can use the template to specify the transactor writing directly in SystemC.

4.5 Library Generation

The *SystemC Transactor Generator Tool* also supports the feature of generating transactor libraries. We can create libraries for the standard protocol transactors like AMBA [6], AHB [5], UTOPIA [7], I2C [38], etc. and then use them in any project without rewriting the protocol again. To generate a library, the tool archives the information containing the TLM interface, the RTL interface, and the generated SystemC code for the transactor in a single file. The transactor libraries can be distributed independently and can be added to any new transactor project as shown in Figure 4.9. When a library is added to a new transactor project, the

TLM interface and the RTL interface is added with the new transactor. Also, the protocol code for the library transactor is added. Thus user can generate transactors with the help of libraries without rewriting the protocol.

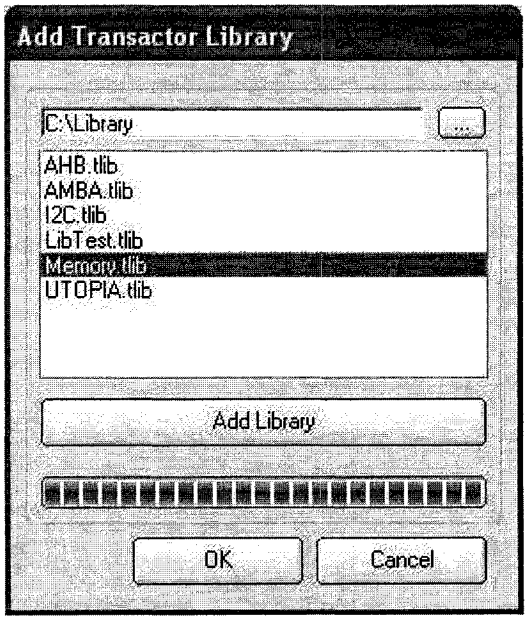

Figure 4.9: Adding Transactor Library

Chapter 5

Case Studies

In this chapter, we will discuss our experiments on the generation of transactors for two case studies, namely UTOPIA protocol [7] and Memory Interface [11]. The latter case study is shown as an example of library transactor.

5.1 UTOPIA Transactor

UTOPIA is a standard protocol used to connect devices implementing ATM and PHY layers. We have modeled the ATM layer at TLM and the PHY layer at RTL in SystemC. These two models are connected through a TLM-RTL transactor as shown in Figure 5.1

5.1.1 Signal Description

By convention, the interface where data flows from ATM to PHY layers is labeled the *Transmit Interface*, and the interface where data flows from PHY to ATM layers is labeled the *Receive Interface.* Table 5.1 describes the essential UTPOIA interface signals. All signals are active high, unless denoted via a trailing "*" after the signal name. Optional signals are not listed.

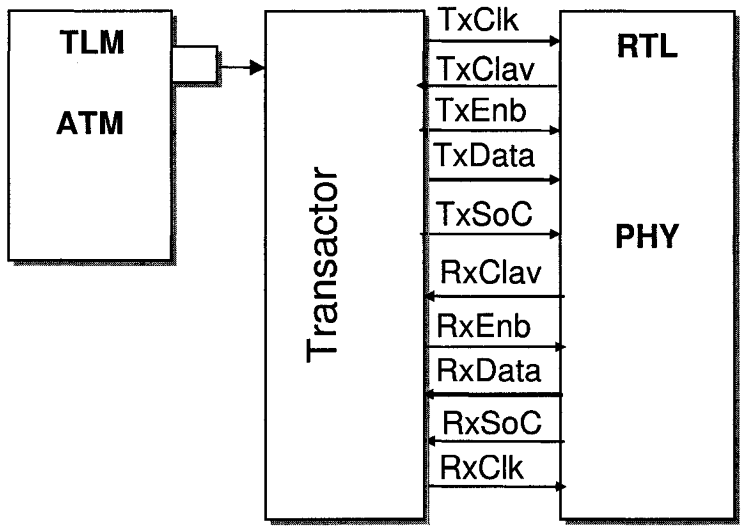

Figure 5.1: UTOPIA Transactor

Table 5.1: UTOPIA Interface Signals (optional signals are not listed

Transmit Interface Signals

Signal	*Direction*	*Description*
TxData[7:0]	ATM to PHY	8 bit Data bus
TxSOC	ATM to PHY	Start of Cell
TxEnb*	ATM to PHY	Enable data transfers
TxClav	PHY to ATM	Cell buffer available
TxClk	ATM to PHY	Transfer/interface byte clock

Receive Interface Signals

Signal	*Direction*	*Description*
RxData[7:0]	PHY to ATM	8 bit Data bus
RxSOC	PHY to ATM	Start of Cell
RxEnb*	ATM to PHY	Enable data transfers
RxClav	PHY to ATM	Cell available
RxClk	ATM to PHY	Transfer/interface byte clock

5.1.2 Protocol Description

Transmit Protocol

The protocol for transmitting one or more cells (each cell consists of 53 bytes) from ATM to PHY in *Cell Level Handshake* mode can be briefly described by the following procedure. The PHY module indicates that it can accept a whole cell by asserting the *TxClav.* Then during a time period termed the *transmit window*, the ATM module drives data on to *TxData* and asserts *TxEnb. TxSoC* is asserted during the transfer of the first byte of the cell. In this way, 53 bytes are sent in the successive 53 cycles of *TxClk.* If the PHY module becomes unable to accept more cells, it deasserts *TxClav* at least 4 cycles before the end of a cell. The ATM module ends its transmission by deasserting *TxEnb.*

Receive Protocol

The protocol for receiving one or more cells from PHY to ATM in *Cell Level Handshake* mode can be briefly described by the following procedure. The PHY layer indicates it has a complete cell by asserting *RxClav.* The ATM layer indicates that it wants to read PHY data by asserting *RxEnb.* The ATM layer may assert and deassert *RxEnb* at any time. The cycles during which *RxEnb* is asserted constitute a *read window.* During a read window the PHY layer reads data from its internal FIFO and presents it on RxData/RxSOC on each low-to-high transition of *RxClk.* Asserting *RxEnb* while *RxClav* is deasserted is not an error but the value of *RxData* is undefined.

5.1.3 Modeling in SystemC

ATM Module

The ATM module was modeled at TLM in SystemC. It connects with the transactor with a port having a TLM interface. The TLM functions that the ATM module calls are shown in Table 5.2

Table 5.2: TLM Functions Called by the ATM Module

TLM Function	*Description*
`void SendCell ( const unsigned int StartCellNo, const unsigned int EndCellNo, const char SrcCell [] )`	Blocking. Transmits one or more cells to PHY module.
`void GetCell ( char DstCell [] )`	Blocking. Receives one cell from PHY module
`void nb_SendCell ( const unsigned int StartCellNo, const unsigned int EndCellNo, const char SrcCell [], bool & isCompleted )`	Non-Blocking. Transmits one or more cells to PHY module.
`void nb_GetCell ( char DstCell [], bool & isCompleted )`	Non-Blocking. Receives one cell from PHY module

PHY Module

The PHY module was modeled at RTL in SystemC. It connects with the transactor through RTL ports. The model is a clock cycle accurate, but not synthesizable. It has a FIFO buffer modeled into it. The transmit interface and receive interface signals are used to write into or read from the FIFO, respectively. In this model, only *Cell Level Handshake* mode is supported.

The clock frequency for both *TxClk* and *RxClk* is set as 25 MHz with 50% duty cycle. So, the clock period T = 40ns. Setup time for other RTL ports is set to 10ns.

5.1.4 Generating SystemC Transactor

We used our developed *SystemC Transactor Generator Tool* to generate the UTOPIA transactor. In the tool, we created a transactor project named UTOPIA. Then we inserted the TLM interface of the ATM module and RTL interface of the PHY module into the tool.

Transactor Generation from AsmL Specification

To specify the transactor protocol in AsmL, the tool generates an AsmL template for the UTOPIA transactor. We used the template to write the specification of the four TLM functions. The AsmL specification for the *SendCell()* is shown in Figure A.1

From the ATM module, when the TLM function *SendCell ()* is called, the transmit protocol must be followed by the transactor to complete the task. We can express the entire procedure of sending cells in three states namely *WaitForCellAvailable*, *TransmitCell*, and *CloseTxWindow.*

At first, the state machine enters the initial state *WaitForCellAvailable.* If *TxClav* is asserted then it sets the next state as *TransmitCell.* At the state *TransmitCell*, the transactor opens the *transmit window* by asserting *TxEnb*. *TxSoC* is asserted when transmitting the first byte of the cell. It also drives *TxData* with the corresponding byte of the *SrcCell* array. Here two user defined variables *Bn* and *Cn* are used to keep track of byte and cell numbers, respectively. When the last byte of the cell is sent, it checks the *TxClav* whether any more cell (if required) can be transmitted. If PHY is unable to accept more cells then it sets the next state as *CloseTxWindow.* At the state *CloseTxWindow*, *TxEnb* is de-asserted and thus the *transmit window* is closed. If all cells are transferred, then the state machine breaks and the SendCell function ends. Otherwise it sets the next state as *WaitForCellAvilable* and so on.

After the AsmL specification is written, we can execute it using the *asmlc* compiler and run for different scenarios, thus we can do validation. Also verification of the specification is possible by model checking and theorem proving.

Once the AsmL specification is executed and verified, we give the specification as input to the *SystemC Transactor Generator Tool* to generate the complete SystemC transactor. We also set different code settings like transactor generation method as AsmL, timing information, etc. A portion of the automatically generated

SystemC code by the tool is shown at Figure A.2

Transactor Generation from Graphical FSM

Another way of specifying transactor is the graphical FSM approach. The *SystemC Transactor Generator Tool* generated template for drawing FSM using the *Active HDL State Editor.* An FSM drawing for the *GetCell()* function is shown in Figure 5.2

We then gave the ASF files as input to the tool and generated the SystemC transactor. We got almost a similar code as shown in Figure A.2

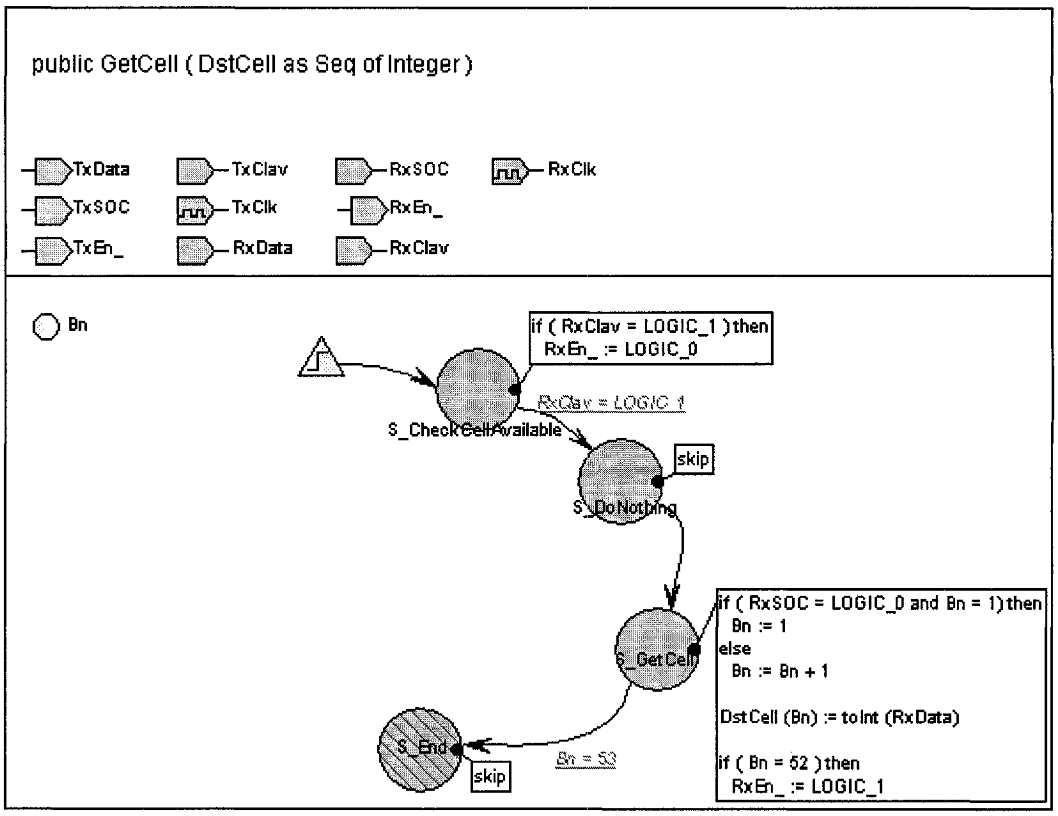

Figure 5.2: Graphical FSM Specification of the Function GetCell()

Transactor Template Generation

The tool can also generate a transactor template where the protocol can be described directly in SystemC. A template for the UTOPIA transactor generated by the tool is shown in Figure A.3

5.1.5 Test Case Generation

We have used the AsmL Tester (asmlt) to automatically generate function parameters for the TLM functions. The AsmL Tester tool checks the pre-condition *require* statement for generating function parameters. A detailed discussion on generating parameters can be found in [35]. In asmlt, we specified the domain of each function parameters and the maximum number of test parameters to be generated. Then the tool generated test function parameters that satisfy the pre-condition *require* statement. For the UTOPIA *SendCell()* function we wrote a pre-condition *require* (*StartCellNo* $\geq$ *1 and StartCellNo* $\leq$ *10 and EndCellNo* $\geq$ *1 and EndCellNo* $\leq$ *10 and EndCellNo* $\geq$ *StartCellNo)* which tells the AsmL Tester to generate the function parameter values for *StartCellNo* and *EndCellNo* in such a way that their range is between 1 to 10 and *EndCellNo* is greater than or equal to *StartCellNo.* A snapshot of the parameter generation of asmlt is shown in Figure 5.3

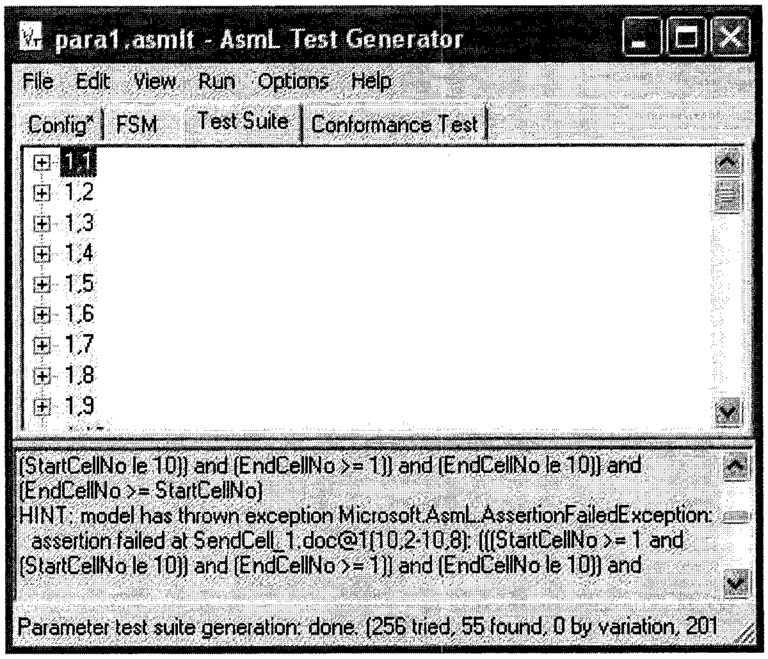

Figure 5.3: Test Case Generation by the AsmL Tester

The test parameter values were stored in an ASCII text file. In SystemC simulation for the UTOPIA model, the ATM module took input from the generated parameter file when calling the TLM function *SendCell()*. We have successfully simulated the UTOPIA model for all the generated function parameters.

5.1.6 Simulation of the Generated Code

After generating the SystemC transactor, we verified the code by SystemC simulation. We placed the generated SystemC transactor between TLM ATM and RTL PHY modules in SystemC as shown in Figure 5.1.

In the ATM module, we declared an array of 530 bytes as the source cell array. Each cell consists of 53 bytes. So, the declared array can hold 10 cells. We initialized all 53 bytes of the first cell (i. e. array index 1 to 53) with '1', all 53 bytes of the second cell (i. e. array index 54 to 106) to '2' and so on.

In the PHY module, a FIFO buffer was modeled which can hold maximum 5 cells.

We simulated the UTOPIA model for different scenarios to check whether the automatically generated transactor is performing correctly or not. Some scenarios are described below.

Scenario 1: The ATM module called the *SendCell()* function to transmit cells from 1 to 2 to the PHY module. After 5000ns, the ATM module calls the function *GetCell()* to receive a single cell from PHY. We generated Value Change Dump (VCD) traces of the UTOPIA RTL signals and used a standard waveform viewer [20] to get the simulation timing diagram as shown in Figure 5.4.

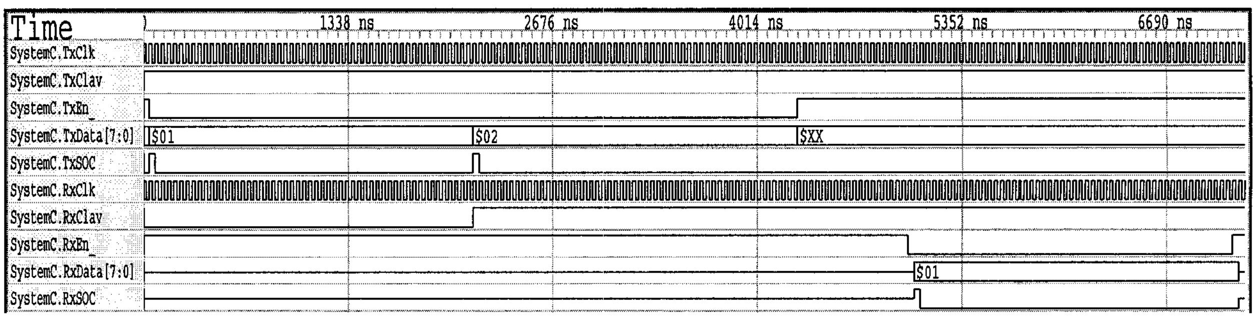

Figure 5.4: Scenario 1: Simulation Timing Diagram

In Figure 5.4, we see that the ATM module sent the two cells to the PHY module consecutively and then closed the transmit window following to the transmit protocol of UTOPIA. At 5000ns, the *GetCell()* got activated and it started to receive a cell from the PHY module. After receiving a cell, it closed the receive window maintaining the accurate receive protocol [7].

Scenario 2: In this simulation, the ATM module calls the *SendCell()* function to transmit cells from 1 to 6 to the PHY module. After 12000ns, the *GetCell()* function gets activated. The simulation timing diagram is shown in Figure 5.5

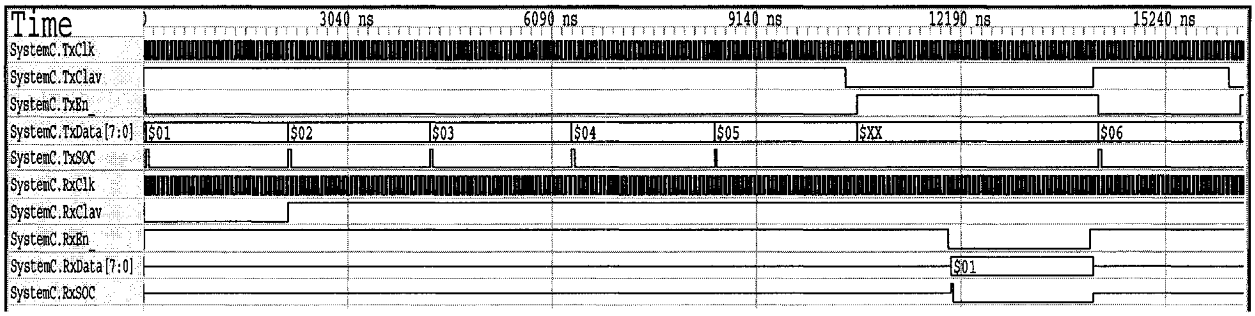

Figure 5.5: Scenario 2: Simulation Timing Diagram

In Figure 5.5, we see that after sending five cells, the PHY FIFO buffer got full and it became unable to receive the remaining cell. The *SendCell()* function was modeled as *blocking* nature which means that the function will not return to the caller until it finishes its task completely. So, the *SendCell()* function waited until there is any empty space in the FIFO to send the remaining cell.

After 12000ns, the *GetCell()* function got activated and it started to receive a cell from the PHY module. After the PHY module sent a complete cell, then an empty space in the FIFO buffer became available. The PHY module then asserted the cell buffer available signal and the waiting *SendCell()* function sent the remaining cell to the PHY.

Scenario 3: In this simulation, the ATM module calls the non-blocking *nb_SendCell()* function to transmit cells from 1 to 6 to the PHY module. After 12000ns, the *GetCell()* function gets activated. The simulation timing diagram is shown in Figure 5.6

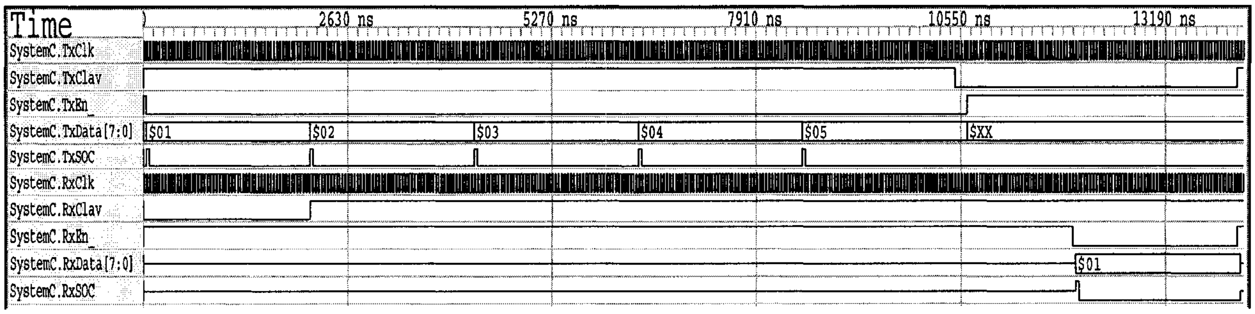

Figure 5.6: Scenario 3: Simulation Timing Diagram

In this scenario, the *nb_SendCell()* function was modeled as *non-blocking* in nature which means that the function will return to the caller whenever it is unable to complete its operation. Non-blocking functions may or may not complete their task which is generally indicated by a boolean return value of the function. So, when the FIFO got full, the *nb_SendCell()* became unable to send the remaining cell. Due to its non-blocking nature, it returns to the caller indicating that it did not complete its task.

After 12000ns, the *GetCell()* function got activated and it started to receive a cell from the PHY module. After the PHY module sent a complete cell, an empty space in the FIFO buffer became available. But although there is now an empty space, we see that the *nb_SendCell()* function did not send the remaining cell.

We also did simulations with *blocking* and *non-blocking GetCell()* functions and got expected simulation results which verified the correctness of the automatically generated transactor.

5.1.7 Experimental Results

Table 5.3 shows the number of AsmL lines and the number of lines in the automatically generated SystemC code for different functions of the UTOPIA transactor. It shows that AsmL specifications can be more concise (about 50%) than SystemC code yet preserving the accurate transactor behavior.

Table 5.3: Experimental Results

Transactor Generation Method	Transactor Function	No. of Lines		Time for 1 Cell in SystemC	
		AsmL	SystemC	Simul. µs	CPU ms
AsmL Specification	`SendCell`	37	74	2.2	140
	`nb_SendCell`	38	75		148
	`GetCell`	27	56	2.2	78
	`nb_GetCell`	31	62		78.5
Graphical FSM	`SendCell`	41	82	2.2	148
	`nb_SendCell`	42	83		156.5
	`GetCell`	32	66	2.2	70
	`nb_GetCell`	38	78		78

The number of SystemC lines grows linearly with AsmL lines as shown in Figure 5.7. This linear relationship promises expected CPU execution time.

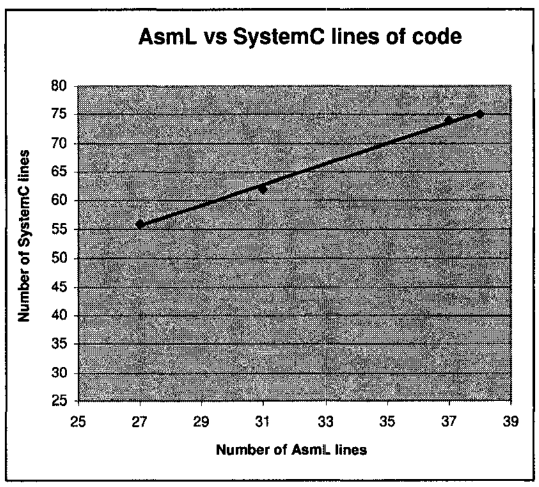

Figure 5.7: Relationship between AsmL and SystemC lines of code

Table 5.3 also shows the simulation time for sending and receiving 1 cell (53 bytes) from ATM to PHY. This simulation time depends on the UTOPIA models clock frequency and the protocol. The clock frequency for *TxClk* and *RxClk* for was made 25 MHz. So, the clock period becomes 40ns. To send 53 bytes in each clock cycle, it takes 40 * 53 = 2120ns. Additional two more clock cycles (40 * 2 = 80ns) are required to accomplish the request-grant handshaking. So, in total, it takes 2.2μs to send a cell.

The CPU time is the time required for a particular PC (Personal Computer) or workstation to execute the transactor functions. It depends on the processor speed and the available memory of the PC. The higher the processor speed and also the higher the memory, the lesser the CPU execution time. We conducted the experiments on a Pentium Mobile processor (1.8 GHz) with 512 MB of memory.

5.2 Memory Interface Transactor and Library Generation

In this section, we discuss another case study with a memory access protocol transactor [11] as shown in Figure 5.8, where we also made the transactor as a library.

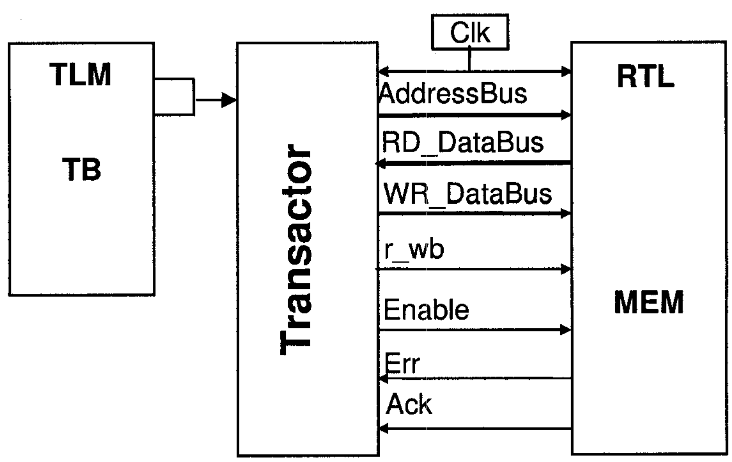

Figure 5.8: Memory Access Transactor

The TLM module is a test bench which calls the functions *mem_read()* and *mem_write()* to read data from and write data to the RTL memory. The RTL module is a memory block having an *AddressBus*, a separate *DataBus* for read and write, read and write control signal, *Enable* signal, *Acknowledge* signal and *Error* signal. We modeled the TLM test bench and the RTL memory block in SystemC.

We then wrote the transactor protocol both in AsmL and in graphical FSM. The AsmL specification of the *mem_read()* function is shown in Figure A.4 and the graphical FSM of the function protocol *mem_write()* in shown in Figure 5.9

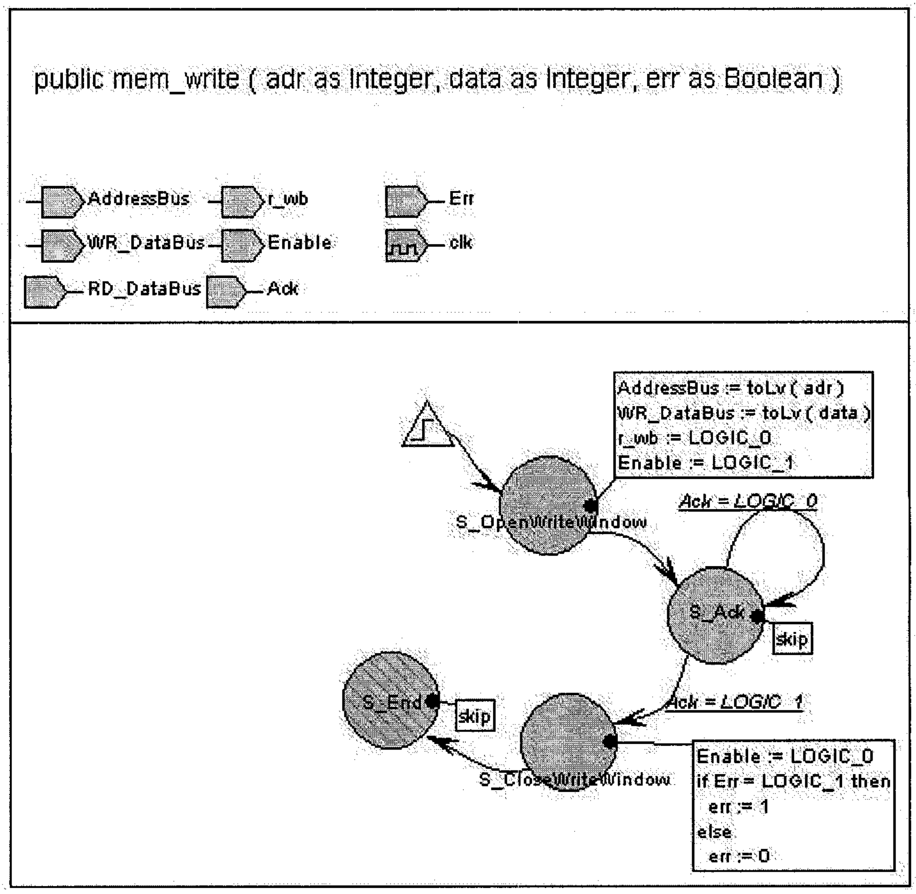

Figure 5.9: Graphical FSM for the function mem_write()

The memory write protocol is as follows. At state *S_OpenWriteWindow*, the address is placed on the *AddressBus*, data is placed on the *WR_DataBus*. In addition, the control signals *R_Wb* is de-asserted and *Enable* is asserted. At the positive edge of the *Enable* signal, the RTL memory starts its operation. Then at the next state *S_Ack*, the transactor waits for the *Ack* signal from the memory which will indicate the end of the memory operation. Once the *Ack* signal is received, the state machine enters the state *S_CloseWriteWindow.* Then *Enable* is de-asserted and the *Err* signal is read.

We used the *SystemC Transactor Generator Tool* to generate the transactor from the specifications. In the *Code Settings* of the tool, we choose to generate a library file for the transactor. The tool then generated the transactor library for the memory access protocol.

After the library has been generated, we opened the UTOPIA transactor project again and then added the memory access transactor library to it. Then the tool generated the transactor which can be used to access IP blocks which use both UTOPIA and memory access protocol as shown in Figure 5.10. In this way, we can generate transactor libraries and once the library is made, we can add them in projects without re-writing the protocol again.

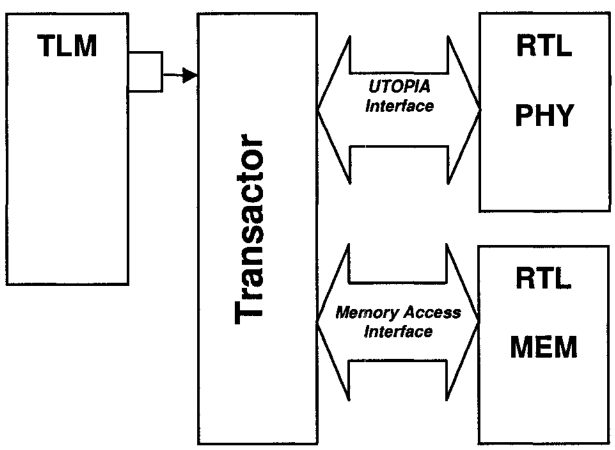

Figure 5.10: UTOPIA Transactor after Adding Memory Access Protocol Library

Chapter 6

Conclusion

6.1 Summary

In this work, we have developed a methodology to generate SystemC transactors from specifications given in AsmL and also from graphical representation of FSMs. We have defined a subset, rules and guidelines to specify transactors in AsmL. Also, we have defined hardware data types and constants in AsmL to declare RTL ports and to represent hardware oriented information. A set of semantic and syntax translation rules were developed to translate the AsmL specification to SystemC. To specify transactors by graphical FSMs, we have defined a set of rules and also developed an algorithm to generate AsmL code from graphical FSM description. A *SystemC Transactor Generator Tool* for automatic generation of SystemC transactors both from AsmL specification and from graphical FSM description has been developed. The tool consists of GUI, FSM to AsmL Code Generator, AsmL to SystemC Compiler and other necessary modules. The tool also provides features to generate transactor libraries. We conducted case studies with UTOPIA and memory interface transactors. We wrote AsmL specifications and also drew graphical FSMs to specify the transactors. Then SystemC transactors were automatically generated using our developed *SystemC Transactor Generator Tool*. We have also modeled TLM ATM module and an RTL PHY module in SystemC and simulated them using the generated transactor. We have made library for the memory interface transactor and added it with the UTOPIA transactor. From the experimental results, we found that AsmL specifications are more concise (approximately 50%) than automatically generated SystemC code. Also, the number of automatically generated SystemC lines of code grows linearly with that of AsmL code.

6.2 Discussion and Future Work

Some of the limitations of this work are as follows:

- Synthesis of SystemC designs are difficult. The transactor codes generated by the *SystemC Transactor Generator Tool* is not restricted to the synthesizable subset of SystemC. So the generated code is not directly synthesizable, it can only be used for simulation.
- AsmL is mostly used for software specification and it needs more language support to specify hardware oriented systems.

Our future work includes the followings:

- Formal verification of AsmL models. It can be done by model checking or theorem proving.
- Generating synthesizable transactor code.
- Generating transactor code in languages other than SystemC, such as SystemVerilog [30].
- SystemC transactors can be interfaced with RTL modules that are written in VHDL, inside a SystemC-VHDL co-simulation environment.

Appendix A

A.1 AsmL Code for UTOPIA

The AsmL specification for the UTOPIA transactor for the TLM function *Send-Cell()* is shown in Figure A.1

```
namespace nsSendCell
import nsHardwareDatatype
import nsRTL

// Declare Enumeration here
enum typeState
  S_CheckCellAvailable
  S_SendCell
  S_CloseTxWindow
  S_End

// Function Declaration
public SendCell ( StartCellNo as Integer, EndCellNo as Integer,SrcCell as Seq
of Integer)

  // Declare Local AsmL Variables here
  var C_State as typeState = S_CheckCellAvailable
  var Cn as Integer = StartCellNo
  var Bn as Integer = 1

  // Specify the RTL Clock Signal here
  SetClockSignal ( TxClk , true )

  // Start writing State Machine from here
  step while ( C_State <> S_End )

    match ( C_State )

      S_CheckCellAvailable:
        //next state
        if ( TxClav = LOGIC_1 ) then
          C_State := S_SendCell
```

```
S_SendCell:
  //open tx window
  TxEn_ := LOGIC_0
  //increament byte no
  if ( Bn < 53 ) then
    Bn := Bn + 1
  else
    Bn := 1
  //TxSOC
  if ( Bn = 1 ) then
    TxSOC := LOGIC_1
  else
    TxSOC := LOGIC_0
  //TxData
  TxData := toLv ( SrcCell ( (Cn - 1) * 53 + Bn ) )
  //end of sending a cell
  if ( Bn = 53 ) then
    Cn := Cn + 1
    //close tx window if all cell sent or no space in phy fifo
    if ( ( Cn = EndCellNo ) or ( TxClav = LOGIC_0 ) ) then
      C_State := S_CloseTxWindow

S_CloseTxWindow:
  //close tx window
  TxEn_ := LOGIC_1
  TxData := "XXXXXXXX"

  //next state
  if ( Cn = EndCellNo + 1 ) then
    //all cell sent
    C_State := S_End
  else
    //all cell not sent, wait for TxClav
    C_State := S_CheckCellAvailable
```

Figure A.1: AsmL Code for function SendCell()

A.2 SystemC Transactor Code

The automatically generated SystemC code from the AsmL specification for the UTOPIA transactor is shown in Figure A.2. Among the four TLM functions *SendCell()*, *GetCell()*, *nb_SendCell()*, and *nb_GetCell()*, only *SendCell()* function implementation is shown.

```
/***************************************************

--------------- Transactor Header File ---------------

Title:   UTOPIA Transactor
Author:  Tareq Hasan Khan
Company: HVG
Date:    Tuesday, Jun 12, 2007 @ 12:59:07 PM

----------------------------------------------------
      This file is automatically generated by
        SystemC Transactor Generator v1.0
----------------------------------------------------

***************************************************/

#ifndef _TRAN_H_
#define _TRAN_H_

#include "tran_aic.h"


//Transactor Class
class clsTran : public sc_module, public clsTranAIC
{
public:

        //TLM Function Declaration
        void SendCell ( const unsigned int  StartCellNo, const unsigned int  EndCellNo,
const unsigned char  SrcCell [] ) ;
        void GetCell ( unsigned char  DstCell [] ) ;
        void nb_SendCell ( const unsigned int  StartCellNo, const unsigned int
EndCellNo, const unsigned char  SrcCell [], bool & isCompleted ) ;
        void nb_GetCell ( unsigned char  DstCell [], bool & isCompleted ) ;


        //RTL Port Declaration
        sc_out <sc_lv <8> > TxData ;
        sc_out <sc_logic> TxSOC ;
        sc_out <sc_logic> TxEn_ ;
        sc_in <sc_logic> TxClav ;
        sc_in <bool> TxClk ; //Clock Signal
        sc_in <sc_lv <8> > RxData ;
        sc_in <sc_logic> RxSOC ;
        sc_out <sc_logic> RxEn_ ;
        sc_in <sc_logic> RxClav ;
        sc_in <bool> RxClk ; //Clock Signal


        //Constructor
        clsTran ( sc_module_name name ) : sc_module (name)
        {
                //Initialize RTL Output Ports
                TxData.initialize ("XXXXXXXX") ;
                TxSOC.initialize (SC_LOGIC_0) ;
                TxEn_.initialize (SC_LOGIC_1) ;
                RxEn_.initialize (SC_LOGIC_1) ;
        }

private:

        //TLM-RTL Type Conversion Functions
```

```
        unsigned int toInt ( sc_lv <64> temp ) //64 is the MAX bus width
        {
            assert ( temp.is_01 () == true ) ;
            return (temp.to_uint()) ;
        }

        char * toLv ( unsigned int tn )
        {
            static char s [65] ;
            _itoa (tn, s, 2) ;
            return (s) ;
        }

        // Update Function Declaration
        void SendCell_Update ( void ) ;
        void GetCell_Update ( void ) ;
        void nb_SendCell_Update ( void ) ;
        void nb_GetCell_Update ( void ) ;

        // Declare Enumeration here
        enum SendCell_typeState
          {
          SendCell_S_CheckCellAvailable, SendCell_S_SendCell, SendCell_S_CloseTxWindow,
SendCell_S_End
          };

          // Declare Local AsmL Variables here
          sc_signal <SendCell_typeState> SendCell_C_State ;
          sc_signal <unsigned int> SendCell_Cn ;
          sc_signal <unsigned int> SendCell_Bn ;

        // Declare Enumeration here
        enum GetCell_typeState
          {
          GetCell_S_CheckCellAvailable, GetCell_S_DoNothing, GetCell_S_GetCell,
GetCell_S_End
          };

          // Declare Constants here

          // Declare Local AsmL Variables here
          sc_signal <GetCell_typeState> GetCell_C_State ;
          sc_signal <unsigned int> GetCell_Bn ;

        // Declare Enumeration here
        enum nb_SendCell_typeState
          {
          nb_SendCell_S_CheckCellAvailable, nb_SendCell_S_SendCell,
nb_SendCell_S_CloseTxWindow, nb_SendCell_S_End
          };

          // Declare Local AsmL Variables here
          sc_signal <nb_SendCell_typeState> nb_SendCell_C_State ;
          sc_signal <unsigned int> nb_SendCell_Cn ;
          sc_signal <unsigned int> nb_SendCell_Bn ;

        // Declare Enumeration here
        enum nb_GetCell_typeState
          {
          nb_GetCell_S_CheckCellAvailable, nb_GetCell_S_DoNothing, nb_GetCell_S_GetCell,
nb_GetCell_S_End
```

```
        };

        // Declare Local AsmL Variables here
        sc_signal <nb_GetCell_typeState> nb_GetCell_C_State ;
        sc_signal <unsigned int> nb_GetCell_Bn ;
} ;

#endif

/****************************************************

---------------- Transactor C++ File -----------------

Title:   UTOPIA Transactor
Author:  Tareq Hasan Khan
Company: HVG
Date:    Tuesday, Jun 12, 2007 @ 12:59:07 PM

------------------------------------------------------
      This file is automatically generated by
        SystemC Transactor Generator v1.0
------------------------------------------------------

*****************************************************/

#include <systemc.h>
#include "tran.h"

/****************************************************
       Tranasactor TLM Function Implementation
*****************************************************/

void clsTran :: SendCell_Update ( void )
{
  wait ( 30, SC_NS ) ;
  wait ( TxClk->posedge_event () ) ;
}

void clsTran :: SendCell ( const unsigned int  StartCellNo, const unsigned int
EndCellNo, const unsigned char  SrcCell [] )
{
  SendCell_C_State.write ( SendCell_S_CheckCellAvailable  ) ;
  SendCell_Cn.write ( StartCellNo  ) ;
  SendCell_Bn.write ( 1  ) ;

  wait ( SC_ZERO_TIME ) ;

  // Start writing State Machine from here
  while ( ( SendCell_C_State.read () != SendCell_S_End )  )
    {

    switch ( ( SendCell_C_State.read () )  )
      {
      case SendCell_S_CheckCellAvailable  :
        {
        //next state
        if ( ( TxClav.read () == SC_LOGIC_1 )  )
          {
          SendCell_C_State.write( SendCell_S_SendCell  ) ;
          }
        } break ;

      case SendCell_S_SendCell  :
        {
```

```
        //open tx window
        TxEn_.write( SC_LOGIC_0  ) ;
        //increament byte no
        if ( ( SendCell_Bn.read () < 53 )  )
          {
          SendCell_Bn.write( SendCell_Bn.read () + 1  ) ;
          }
        else
          {
          SendCell_Bn.write( 1  ) ;
          }
         //TxSOC
        if ( ( SendCell_Bn.read () == 1 )  )
          {
          TxSOC.write( SC_LOGIC_1  ) ;
          }
        else
          {
          TxSOC.write( SC_LOGIC_0  ) ;
          }
        //TxData
        TxData.write(toLv(SrcCell [(SendCell_Cn.read() - 1 )*53+SendCell_Bn.read()]) ) ;
         //end of sending a cell
        if ( ( SendCell_Bn.read () == 53 )  )
          {
           //increment cell no
          SendCell_Cn.write( SendCell_Cn.read () + 1  ) ;
          //close tx window if all cell sent or no space in phy fifo
          if (((SendCell_Cn.read () == EndCellNo)||(TxClav.read () == SC_LOGIC_0)))
            {
            SendCell_C_State.write( SendCell_S_CloseTxWindow  ) ;
            }
          }
        } break ;

      case SendCell_S_CloseTxWindow  :
        {
        //close tx window
        TxEn_.write( SC_LOGIC_1  ) ;
        TxData.write( "XXXXXXXX"  ) ;
         //next state
        if ( ( SendCell_Cn.read () == EndCellNo + 1 )  )
          {
          //all cell sent
          SendCell_C_State.write( SendCell_S_End  ) ;
          }
        else
          {
          //all cell not sent, wait for TxClav
          SendCell_C_State.write( SendCell_S_CheckCellAvailable  ) ;
          }
        } break ;
      }
    SendCell_Update () ;
    }
}
```

Figure A.2: Automatically Generated SystemC Code

A.3 SystemC Transactor Template

The automatically generated SystemC transactor template for the UTOPIA transactor is shown in Figure A.3

```
/****************************************************

---------------- Transactor C++ File -----------------

Title:   UTOPIA Transactor
Author:  Tareq Hasan Khan
Company: HVG
Date:    Tuesday, Jun 12, 2007 @ 01:37:46 PM


------------------------------------------------------
      This file is automatically generated by
        SystemC Transactor Generator v1.0
------------------------------------------------------

****************************************************/

#include <systemc.h>
#include "tran.h"

/****************************************************
       Tranasactor TLM Function Implementation
****************************************************/

void clsTran :: SendCell ( const unsigned int  StartCellNo, const unsigned int
EndCellNo, const unsigned char  SrcCell [] )
{


}


void clsTran :: GetCell  ( unsigned char  DstCell [] )
{


}


void clsTran :: nb_SendCell ( const unsigned int  StartCellNo, const unsigned int
EndCellNo, const unsigned char  SrcCell [], bool & isCompleted )
{


}


void clsTran :: nb_GetCell  ( unsigned char  DstCell [], bool & isCompleted )
{


}
```

Figure A.3: Automatically Generated SystemC Code Template

A.4 AsmL Code for Memory Access

The AsmL specification for the memory access transactor for the TLM function *mem_read()* is shown in Figure A.4

```
namespace nsmem_read
import nsHardwareDatatype
import nsRTL

// Declare Enumeration here
enum typeState
  S_OpenReadWindow
  S_Ack
  S_CloseReadWindow
  S_End

// Function Declaration
public mem_read ( adr as Integer, data as Integer, err as Boolean )

  // Declare Local AsmL Variables here
  var C_State as typeState = S_OpenReadWindow

  // Specify the RTL Clock Signal here
  SetClockSignal ( clk , true )

  // Start writing State Machine from here
  step while ( C_State <> S_End )

    match ( C_State )

      S_OpenReadWindow :
        AddressBus := toLv ( adr )
        r_wb := LOGIC_1
        Enable := LOGIC_1
        // next state
        C_State := S_Ack

      S_Ack :
        if Ack = LOGIC_1 then
          C_State := S_CloseReadWindow
        else
          C_State := S_Ack

      S_CloseReadWindow :
        Enable := LOGIC_0
        data := toInt ( RD_DataBus )
        if Err = LOGIC_1 then
          err := true
        else
          err := false
        // next state
        C_State := S_End
```

Figure A.4: AsmL Code for function mem_read()

Bibliography

[1] A. T. Abdel-Hamid, M. Zaki, and S. Tahar. A Tool Converting Finite State Machine to VHDL. In *Proc. Canadian Conference on Electrical and Computer Engineering*, volume 4, pages 1907–1910, Niagara Falls, Canada, May 2004.

[2] Absint Inc. http://www.absint.com/aisee.html, 2007.

[3] Accellera Organization. Accellera Property Specification Language Reference Manual, version 1.01, http://www.accellera.org, 2007.

[4] Aldec Inc. Active-HDL Tool, version 7.1, http://www.aldec.com, 2007.

[5] ARM Limited. AMBA AHB Specification, rev 2.0, http://polimage.polito.it/lavagno/esd/ihi0011a_amba_spec.pdf, May 1999.

[6] ARM Limited. AMBA Specification, rev 2.0, http://polimage.polito.it/lavagno/esd/ihi0011a_amba_spec.pdf, May 1999.

[7] ATM Forum Technical Committee. Utopia Level 2, version 1.0, http://www.mfaforum.org/ftp/pub/approved-specs/af-phy-0039.000.pdf, June 1995.

[8] F. Balarin and R. Passerone. Functional Verification Methodology based on Formal Interface Specification and Transactor Generation. In *Proc. Design Automation and Test in Europe*, pages 1013–1018, Munich, Germany, 2006.

[9] F. Balena. *Programming Microsoft Visual Basic 6.0.* Microsoft Press, 1999.

[10] J. Basker. *A SystemC Primer.* Start Galaxy Publishing, 2002.

[11] D. C. Black and J. Donovan. *SystemC: From the Ground Up.* Kluwer Academic Publishers, 2004.

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

[13] N. Bombieri, F. Fummi, and G. Pravadelli. On the Evaluation of Transactor-based Verification for Reusing TLM assertions and Testbenches at RTL. In *Proc. Design Automation and Test in Europe*, pages 1007–1012, Munich, Germany, 2006.

[14] G. D. Castillo and K. Winter. Model Checking support for the ASM high-level language. In *Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science*, volume 1785, pages 331–346. Springer Verlag, 2000.

[15] M. Frohlich and M. Werner. Demonstration of the interactive Graph Visualization System daVinci. In *Graph Drawing, Lecture Notes in Computer Science*, volume 894, pages 15–22. Springer Verlag, 1995.

[16] E. R. Gansner, E. Koutsofios, S. C. North, and K. P. Vo. A Technique for Drawing Directed Graph. *IEEE Transactions on Software Engineering*, 19(3):214–230, 1993.

[17] A. Gargantini and E. Riccobene. Encoding Abstract State Machines in PVS. In *Abstract State Machines: Theory and Applications, Lecture Notes in Computer Science*, volume 1912, pages 303–322. Springer Varlag, 2000.

[18] A. Gawanmeh, S. Tahar, and K. Winter. Interfacing ASMs with the MDG tool. In *Abstract State Machines - Advances in Theory and Applications, Lecture Notes in Computer Science*, volume 2589, pages 278–292. Springer Varlag, 2003.

[19] T. Grotker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC.* Kluwer Academic Publishers, 2002.

[20] GtkWave. VCD Waveform Viewer, http://www.cs.man.ac.uk/apt/tools/gtkwave, 2007.

[21] Y. Gurevich. Evolving Algebra 1993: Lipari Guide. In *Proc. Specification and Validation Methods.* Oxford University Press, 1995.

[22] Y. Gurevich and N. Tillmann. Partial Updates: Exploration. *Journal of Universal Computer Science*, 11(7):917–951, 2001.

[23] Y. Gurevich and N. Tillmann. Partial Updates Exploration ii. In *Abstract State Machines, Lecture Notes in Computer Science*, volume 2589, pages 57–86. Springer Verlag, 2003.

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

[25] Ali Habibi. *A Framework for System Level Verification: The SystemC Case.* Ph.D. thesis, Concordia University, Montreal, Canada, September 2005.

[26] M. Himsolt. Graphed: A Graphical Platform for the Implementation of Graph Algorithms. In *Graph Drawing, Lecture Notes in Computer Science*, volume 894, pages 182–193. Springer Varlag, 1995.

[27] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison Wesley, 1979.

[28] J. Hubbard. *Programming with C++.* Schaum's Outline Series, 1996.

[29] IEEE Standards Association. IEEE Std 1666TM Open Systemc Language Reference Manual, http://standards.ieee.org/, 2005.

[30] IEEE Standards Association. IEEE Std 1800TM, SystemVerilog: Unified Hardware Description and Verification Language (HDVL) Standard, http://standards.ieee.org/, 2005.

[31] Isabelle. http://isabelle.in.tum.de, 2007.

[32] K. Keutzer, S. Malik, R. Newton, J. Rabaey, and A. Sangiovanni-Vincentelli. System Level Design: Orthogonalization of Concerns and Platform-Based Design. *IEEE Transactions on Computer-Aided Design*, 19(12):1523–1543, 2000.

[33] Mentor Graphics Corp. Catapult C Synthesis, http://www.mentor.com/.

[34] Microsoft Corp. Introducing AsmL: A Tutorial for the Abstract State Machine Language, available at http://research.microsoft.com/fse/asml/, 2006.

[35] Microsoft Corp. Parameter Generation, available at http://research.microsoft.com/fse/asml/, 2006.

[36] Microsoft Corp. Spec Explorer, available at http://research.microsoft.com/projects/specexplorer/, 2007.

[37] Microsoft Corporation. AsmL: Abstract State Machines Language, http://research.microsoft.com/fse/asml/, 2007.

[38] Philips Semiconductors. The I2C Bus Specification, version 2.1, http://www.nxp.com/acrobat_download/literature/9398/39340011.pdf, January 2000.

[39] A. Pnueli. The Temporal Logic of Programs. In *Proc. Symposium on the Foundations of Computer Science*, pages 46–57, Rhode Island, USA, 1977.

[40] C. Pruteanu. Kiss to Verilog FSM Converter, http://codrin.freeshell.org, 2007.

[41] PVS. http://pvs.csl.sri.com, 2007.

[42] A. Rose, S. Swan, J. Pierce, and J.M. Fernandez. Transaction Level Modeling in SystemC, available at open systemc initiative website: http://www.systemc.org, 2006.

[43] K.H. Rosen. *Discrete Mathematics and Its Applications.* Tata McGraw-Hill, 2001.

[44] SpiraTech Ltd. Cohesive, http://www.spiratech.com/, 2006.

[45] Structured Design Verification Inc. TransactorWizard, http://www.sdvinc.com/, 2007.

[46] S. Swan. SystemC Transaction Level Models and RTL Verification. In *Proc. Design Automation Conference*, pages 90–92, San Francisco, California, USA, 2006.

[47] F. van Ham, H. van deWetering, and J. J. vanWijk. Interactive Visualization of State Transition Systems. *IEEE Transactions on Visualization and Computer Graphics*, 8(4):319–329, 2002.

[48] VCG tool. http://rw4.cs.uni-sb.de/ sander/html/gsvcg1.html, 2007.

[49] Xilinx ISE Tools. http://www.xilinx.com/ise/design tools/, 2007.