Automatic Generation of SystemC Transactors from AsmL Specifications

Tareq Hasan Khan, Ali Habibi[§], Sofiène Tahar, Otmane Ait Mohamed

Dept. of Electrical & Computer Engineering, Concordia University
Montreal, Quebec, Canada
Email: {tare_kha, tahar, ait}@ece.concordia.ca

[§]MIPS Technologies
Mountain View, California, USA
Email: habibi@mips.com

Abstract

The SoC design flow consists of different levels of abstraction. At Transaction Level Modeling (TLM) different modules communicate with each other through function calls. In contrast, at the Register Transfer Level (RTL), different modules communicate through pin level signaling. The notion of transactors has been introduced recently in order to link modules (IP blocks) written at different levels of abstraction. A transactor can be modeled using a finite state machines (FSM) describing the functional protocol behaviors. In this paper, we propose to specify transactor behavior using the Abstract State Machine Language (AsmL). We also define a methodology and a tool that automatically generates SystemC TLM-RTL transactors from these AsmL specifications. The proposed approach has been implemented and applied on several case studies including an UTOPIA standard protocol.

1. Introduction

System-on-Chip (SoC) design methodologies involve the integration of intellectual property (IP) blocks modeled at different levels of abstraction. The ultimate goal in developing 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 crucial 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. At the Transaction Level Modeling (TLM), different modules communicate with each other through functional calls. At the Register Transfer Level (RTL), however, different modules communicate through pin level signaling [3]. In order to be able to link these two common levels of abstraction, the notion of transactor has been recently introduced [21]. A *transactor* is a module which is used between blocks that are modeled at different levels of abstraction so that they can communicate with each other. A TLM-RTL transactor 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. To accomplish the task of a TLM function on the RTL side, there can be a finite state machine implemented inside the transactor [21].

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 problem 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 [13] or even SystemC [12][17], we will not have the feasibility to use high level abstract constructs to specify the protocol early. In this paper, 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) [16] as a formal means for specification and communication among the members of the SoC design team. AsmL models are precise, concise and readable to a wide range of people who have different areas of expertise due its simple and intuitive language constructs [5]. 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 the work presented here, we have developed a methodology to automatically generate SystemC transactors from AsmL specifications. We have defined a set of syntax and semantics translation rules and implemented them in the transactor generator tool. To test the efficiency of our method, we have applied it on several case studies including an UTOPIA standard protocol.

The rest of the paper is organized as follows. The next section introduces AsmL and SystemC. Section 3 discusses related work. Section 4 presents the proposed methodology for AsmL to SystemC transactor generation. Section 5 describes the SystemC Transactor Generator Tool. Section 6 discusses the UTOPIA case study and experimental results. Finally, Section 7 concludes the paper.

2. Preliminaries

2.1. AsmL

The Abstract State Machine Language (AsmL) is an executable modeling language which is fully integrated in the .NET framework and Microsoft development tools. The most unique feature of AsmL is its foundation on Abstract State Machines (ASMs) [4][7]. An ASM is a state machine which in each step computes a set of *updates* of the machines variables. Upon the completion of a step, all updates are "fired" (committed) simultaneously. The update semantics of AsmL is based on the theory of partial updates [8][9].

ASM languages are used to specify both software and hardware. A TLM-RTL transactor deals with Transaction Level Model where the model is described from 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 [16]. ASM languages like AsmL (Abstract State Machine Language) provide powerful constructs and language features to model finite state machines. For instance, inside a TLM-RTL transactor, the hardware protocol can be modeled as a finite state machine using language features such as step, *update* semantics, etc.

In summary, an ideal AsmL specification presents following advantages [5]:

- *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 mechanized or manual proofs.
- *Validation* can be done for different scenarios due to the machine executability of AsmL models.

2.2. SystemC

SystemC, one of the proposals of the electronic design automation (EDA) community has become the IEEE standard (IEEE1666-2005) library [12] for system level design [14]. 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 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.

3. Related Work

Regular expressions and temporal logic [18] are the two main formalisms that have been used for formal interface specifications. Both formalisms can be expressed with finite-state automata [11]. More recently, standard languages have been proposed to specify system properties (in particular, the Property Specification Language (PSL) [1] and the System Verilog Assertions (SVA)[13]. 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.* [3] proposed to specify TLM-to-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. Several commercial tools include features to generate SystemC transactors, for example: SystemC Transactor Generation Wizard from Aldec's Active HDL [2], Catapult C from Mentor Graphics [15], TransactorWizard from Structured Design Verification [20], and Cohesive from Spiratech [19]. For example, the Cohesive tool uses the CY language as transactor specification. In Active HDL v7.1, 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 transactors.

4. Methodology for AsmL to SystemC Transactor Generation

The proposed methodology is depicted in Figure 1. We create a formal model of the transactor protocol in AsmL based on the natural language text. AsmL models are precise, concise and unambiguous. This model also provides an efficient way of verification and validation. In fact, AsmL specifications are executable, thus can be validated by simulation. Verification can be done by theorem proving (e.g., HOL) and model checking (e.g., SMV). 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.

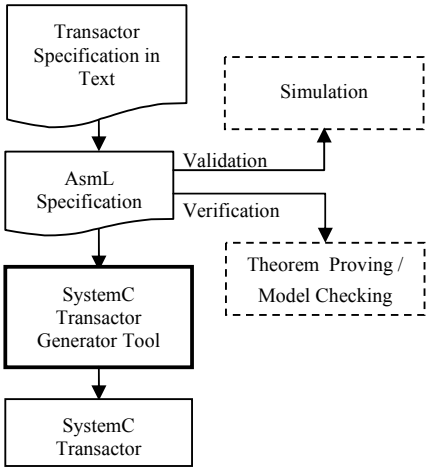

Figure 1. Methodology for generating SystemC Transactor from AsmL

4.1. Specifying Transactors in AsmL

4.1.1. AsmL Subset

We chose a subset of AsmL for transactor specification. The subset contains constructs and symbols that can be used for RTL hardware protocol 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.

4.1.2. 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 distinct steps to perform the task is 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*.

4.1.3. Hardware Data Types in AsmL

To represent hardware data types for RTL ports, we have used strings composed of '1', '0', 'X', 'Z'. We also put length constraints on the string types according to the port bus-width. We have also developed a dynamic library which contains functions to convert binary strings to their equivalent decimal value and vice versa. These functions are frequently used inside a transactor where we have to deal with both binary strings for the RTL ports and decimal values for the TLM function parameters and for other user defined variables.

4.2. Translation from AsmL to SystemC

The translation from AsmL to SystemC is done based on several rules we defined so that the original behavior of the AsmL code is preserved in the translated SystemC code. In [10], some rules for AsmL to SystemC translation have been proposed. We have expanded and in some cases modified some of these rules according to our definitions.

4.2.1. Data Type Mapping

AsmL basic data types are translated to their equivalent SystemC data types, e.g., `Boolean` to `bool`, `Byte` to `unsigned char`, `Short` to `short`, `Integer` to `int`.

4.2.2. Semantics

The AsmL variables behave different from other sequential programming languages like C/C++. If we assign a value with an AsmL variable and 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. In SystemC, the signals declared as `sc_signal <type>` also behave similar like AsmL variable. If we write a value to a SystemC signal, it is not *updated* at that simulation cycle. If we read that signal at the same simulation cycle, we will get its old value, not the newly written value. For SystemC THREAD process, the signals are *updated* with newly written values whenever the program reaches a `wait ()` statement [6]. So we have found that there is a semantical similarity between AsmL variables and SystemC signals. For instance,

- AsmL variable declaration is translated in SystemC like `Var a as Integer` to `sc_signal<int> a ;`
- `step` statement in AsmL is translated to SystemC as `wait(clk->posedge_event())`, where `clk` is the clock signal name. The `posedge_event()` method makes the `wait` statement sensitive to the positive edge event of the clock signal. This translation satisfies the *update* semantics of AsmL and also respects *the step rule* as this `wait` statement will cause the SystemC scheduler to increment its simulation time by one clock cycle [6].

For transactors that communicate with cycle accurate RTL models through request-grant handshaking 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 the time, $t_{bs} = T - t_{su}$ [T=Clock period, t_{su} = setup time]

4.2.3. Syntactical Translation

The mapping of AsmL syntax to SystemC syntax for different keywords and symbols are shown in Table 1.

Table 1. Syntax Translation

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

Iteration Statement	
AsmL	*SystemC*

`step while (`*exp*`)`
`  statement_1`
`  …`
`  statement_n`
	`while (`*exp*`)`
	`  {`
	`  statement_1 ;`
	`  ...`
	`  statement_n ;`
	`  Update () ;`
	`  }`

Conditional Statement	
AsmL	*SystemC*

`if (`*exp1*`) then`
`  statement_1`
`elseif (exp2) then`
`  statement_2`
`else statement_3`
	`if (`*exp1*`)`
	`{  statement_1 ;}`
	`else if (exp2)`
	`{  statement_2 ; }`
	`else`
	`{  statement_3 ; }`

`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;`
	`}`

Assertion Statement	
AsmL	*SystemC*
`require (exp)`	`assert (exp);`

Symbols and Operators			
AsmL	*SystemC*	*AsmL*	*SystemC*
`=`	`==`	`-`	`-`
`<>`	`!=`	`*`	`*`
`>=`	`>=`	`/`	`/`
`<=`	`<=`	`mod`	`%`
`(`	`(`	`and`	`&&`
`)`	`)`	`or`	`||`
`+`	`+`	`not`	`!`

Assignment Statement	
AsmL	*SystemC*
`a := b + c`	`a.write(b.read()+c.read());` *where a, b, c are user defined variables or RTL ports.*

4.2.4. Generating Block

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. We have developed a stack based algorithm to generate blocks in SystemC.

5. SystemC Transactor Generator Tool

Figure 2 describes the general structure of the SystemC transactor generator tool. 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. Then the tool generates an AsmL Template in DOC format so that can be edited and executed in MS Word environment.

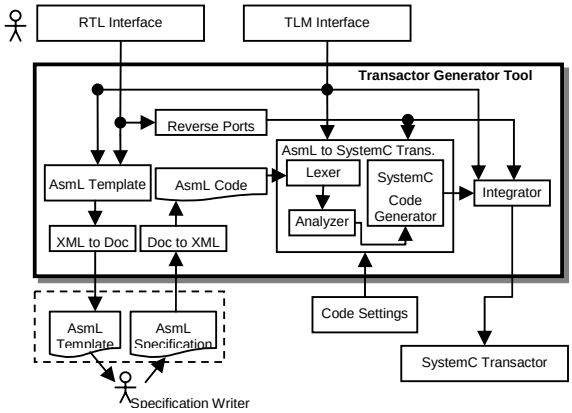

Figure 2. SystemC Transactor Generator Tool

The specification writer provides the transactor specification in the AsmL template. This specification can be executed and used for validation and verification purposes. Also, the specification can be used to generate transactors in languages other than SystemC. Then the specification is given as input to the tool. The tool then extracts ASCII AsmL code text from it and passes it to the *AsmL to SystemC Translator*. The *Lexer* splits an AsmL line to tokens. It uses white spaces, single character symbols and double character symbols as punctuators between words. Here, the grammar checking is omitted because it is done once when the AsmL specification is executed by the *asmlc* compiler [16]. After tokenizing, the *Analyzer* is used to recognize the tokens as keywords, identifiers, constants, symbols etc. Then the *SystemC Code Generator* translates the analyzed AsmL tokens to SystemC according to the rules discussed in Section 4. The *Reverse Port* block reverses the ports direction of the transactor w.r.t. the RTL unit. The *integrator* integrates the translated SystemC code for all TLM functions and adds other necessary SystemC codes to generate the complete transactor.

6. Case Study: UTOPIA Transactor[1]

We tested our tool on several samples provided in the SystemC library. In this section, we discuss our experiments on the generation of the transactor protocol used in the UTOPIA standard [22] interface. 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. These two models are connected through a TLM-RTL transactor as shown in Figure 3.

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 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 clock cycles. If the PHY module becomes unable to accept more cells, it deassert TxClav at least 4 cycle before the end of a cell. The ATM module ends its transmission by deasserting TxEnb.

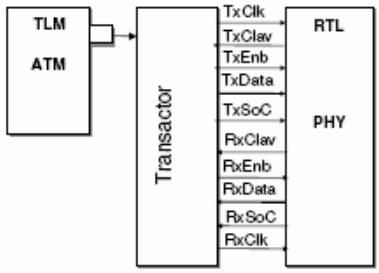

Figure 3. UTOPIA Transactor

From the ATM module, when a TLM function like `SendCell ()` is called, the above 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.* The AsmL specification of the state *TransamitCell* for the function `SendCell()` is shown in Figure 4.

```
public SendCell ( StartCellNo as Integer, EndCellNo as
                  Integer,SrcCell as Seq of Integer)

  var C_State as typeState = WaitForCellAvailable
  var Cn as Integer = StartCellNo
  var Bn as Integer = 1

  step while ( C_State <> S_End )
    match ( C_State )

      TransamitCell :

        TxEnb := LOGIC_0 //open tx window

        if ( Bn < 53 ) then Bn := Bn + 1
        else Bn := 1

        if ( Bn = 1 ) then TxSoC := LOGIC_1
        else TxSoC := LOGIC_0

        TxData := toLv ( SrcCell ((Cn-1) * 53 + Bn ))

        if ( Bn = 53 ) then
         Cn := Cn + 1
         //close tx window
         if ((Cn=EndCellNo) or (TxClav=LOGIC_0)) then
           C_State := CloseTxWindow
```

Figure 4. AsmL Spec. of state *TransmitCell*

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 by setting the next state as *S_End* and the `SendCell` function ends. Otherwise it sets the next state as *WaitForCellAvilable* and so on.

We wrote AsmL specification of the transactor functions `SendCell` and `GetCell` for both blocking and non-blocking [21] cases and executed them. Then the specification was given as input to our SystemC Transactor Generator Tool. The tool generated the SystemC transactor and it was then simulated with the ATM and PHY model in SystemC. The transactor gave expected simulation result. The timing diagram of the simulation matched with the UTOPIA specification which verified the correct behavior of the generated transactor.

Table 2. Experimental Results

Transactor Function	No. of Lines		Time / Cell SystemC	
	AsmL	SystemC	Sim (μs)	CPU (ms)
SendCell	37	74	2.2	140
nb_SendCell	38	75		148
GetCell	27	56	2.2	78
nb_GetCell	31	62		78.5

The number of lines metric provided in Table 2 shows that AsmL specifications is more concise (approximately 50%) than SystemC code yet preserving the accurate transactor behavior. The number of SystemC line grows linearly with AsmL line. This linear relationship

[1] The experiments were conducted on a PC having Pentium Mobile processor (1.8 GHz) with 512 MB of memory.

promises expected CPU execution time. Table 2 also shows the time required for sending and receiving one cell (53 bytes) in SystemC simulation. The simulation time depends on the frequency of the UTOPIA model clock signals. For our simulation, we set the frequency of TxClk and RxClk as 25MHz. The CPU time validates the very light overhead of the translation process from AsmL to SystemC. For instance, it could be always argued that such a process may introduce longer execution time. However, the values displayed in Table 2 show a pretty fast execution of each of the transactor's functions. However, due to small size of the application, a deeper investigation through more complex case studies should be used to support this aspect of the proposed approach.

7. Conclusion

In this paper, we proposed a methodology to use AsmL specifications for specifying transactors and automatically generating semantically equivalent SystemC designs. We illustrated our approach on the UTOPIA interface case study. For instance, our tool was able to generate automatically the equivalent SystemC code for the transactor originally specified in AsmL. Along with the AsmL approach discussed in this paper, the tool also provides other approach where the transactor behavior is described by drawing graphical finite state machines. The tool can also generate templates for writing SystemC code by hand. The future work includes providing a library for standard protocols so they can be used in generating transactors that implement standard protocol interfaces. Furthermore, it is possible to define a monitor between the transactor and the RTL unit where assertions can be easily plugged and checked by simulation or verified formally using model checking.

Reference

[1] Accellera Organization. Accellera Property Specification Language Reference Manual, Version 1.01, 2006.

[2] Aldec Inc. Acctive-HDL Tool, 2007. Website: http://www.aldec.com/.

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

[4] M. Barnett, E. Börger, Y. Gurevich, W. Schulte, and M. Veanes. Using Abstract State Machines at Microsoft: A Case Study. In *Abstract State Machines, Theory and Applications, LNCS* 1912, pages 367–379. Springer, 2000.

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

[6] T. Grotker, S. Liao, G. Martin, and S. Swan. System design with SystemC. Kluwer Academic Publishers, 2002.

[7] Y. Gurevich: Evolving Algebra 1993: Lipari Guide, *in Specification and Validation Methods*, Ed. E. Börger, Oxford University Press, 1995.

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

[9] Y. Gurevich and N. Tillmann. Partial Updates Exploration II. In *Proc. Abstract State Machines 2003*, *LNCS*, 2589, pages 57-86, Springer, 2003.

[10] 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.

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

[12] IEEE Standards Association. *IEEE Std 1666TM Open SystemC Language Reference Manual*. 2005. http://standards.ieee.org/.

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

[14] 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), 2000.

[15] Mentor Graphics Corp. Catapult C Synthesis, 2006. http://www.mentor.com/.

[16] Microsoft Corp. AsmL: Abstract state machines Language, 2007 . research.microsoft.com/fse/asml/.

[17] Open SystemC Initiative. The SystemC Library, 2007. http://www.systemc.org/.

[18] A. Pnueli. The Temporal Logic of Programs. In I. C. S. Press, editor, *In Proc. Symposium on the Foundations of Computer Science*, pages 46–57, Providence, Rhode Island, USA, 1977.

[19] SpiraTech Ltd. Cohesive, 2007. Website: http://www.spiratech.com/.

[20] Structured Design Verification Inc. TransactorWizard, 2006. http://www.sdvinc.com.

[21] A. Rose, S. Swan, J. Pierce, J.M. Fernandez: Transaction Level Modeling in SystemC; Available at Open SystemC Initiative Website, 2006. http://www.systemc.org

[22] The ATM Forum Technical Committee. Utopia Level 2, Version 1.0, June 1995.