Automatic Generation of SystemC Transactors from Graphical FSM

Tareq Hasan Khan, Sofiène Tahar, and Otmane Ait Mohamed

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

Ali Habibi

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

***Abstract*— To specify, design, and implement complex system-on-chip (SoC), a new modeling method, transaction level modeling (TLM), has been proposed recently. TLM allows designers to focus on functionality while abstracting implementation details. At the register transfer level (RTL), however, different modules communicate through detailed 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 abstraction levels. This module, called transactor, can be modeled using a finite state machine (FSM) providing a functional specification of the protocol's behavior. In this paper, we propose a methodology to specify transactors using graphical finite state machine (FSM). This technique enables an automatic generation of SystemC TLM-RTL transactors via an intermediate translation of the user-defined FSM to the Abstract State Machines Language (AsmL). The UTOPIA standard protocol is provided as an illustration of this approach.**

***Index Terms*— SystemC, Transactor, Finite State Machine, AsmL**

I. Introduction

Transaction level modeling (TLM) is becoming a popular practice for system-level design and architecture exploration. It allows the designers to focus on the functionality of the design, while abstracting away implementation details that will be added at lower abstraction levels [5]. The intellectual property (IP) blocks of a system-on-chip may be composed of different blocks which are modeled at various levels of abstraction. Transaction level models (TLM) 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. In order to be able to link modules modeled at different levels of abstraction, the notion of transactor has been recently introduced [3]. A TLM-RTL transactor connects with TLM and RTL modules using two explicit interfaces, namely, the TLM interface, which is the declarations of the TLM functions and the RTL interface, which is the declaration of the RTL ports. Each TLM function is implemented 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, an FSM can be implemented inside the transactor [10]. Inside a TLM-RTL transactor, one or more RTL hardware protocols need to be implemented to accomplish a particular task on the RTL module. The protocol designers generally specify these protocols in natural languages such as in English texts. But natural languages are often imprecise and incomplete. Also, verification of natural language specification is difficult because there is no mathematical mean to prove its precision. Moreover, these specifications are not executable and thus cannot be validated by simulation for different scenarios. These problems might cause more bugs and faults in the product, delays for time to market, etc. In this paper, we propose to generate a formal model of the transactor protocol by drawing FSM based on the natural language text specification. Hardware designers are well familiar with graphical FSM and thus our approach reduces the overhead to learn new specification languages. Furthermore, a visual representation of FSM simplifies the access of protocol description. After the protocol is drawn, we translate the FSM description to Abstract State Machine Language (AsmL) [7] using an AsmL code generation algorithm.

AsmL is an executable modeling language which is fully integrated in the .NET framework and Microsoft development tools. AsmL is based on the theory of Abstract State Machines (ASMs) [4]. An ASM is a state machine which computes in each step a set of *updates* of the machines variables. After a step is completed, all *updates* are fired simultaneously. AsmL models are precise, concise and readable to a wide range of

people due its simple and intuitive language constructs [4]. Syntax and semantics of AsmL is formalized and thus it gives us the opportunity to verify formally the transactor protocol at early stages of the SoC design. This verification will enhance the confidence in the correctness of the finally generated transactor. Once the AsmL model is completed and verified, it can be used to automatically generate the transactors in other languages.

We have developed a technique to automatically generate SystemC transactors from the generated AsmL description [6]. The AsmL specification is translated to SystemC based on a set of syntax and semantics translation rules. To test the efficiency of our approach, we have applied it on several case studies including an UTOPIA [2] standard protocol which is presented here.

The rest of the paper is organized as follows. Section II discusses related work. Section III describes the proposed methodology to generate SystemC transactor from graphical FSM and an algorithm for AsmL code generation. In Section IV, the UTOPIA case study and experimental results are discussed. Finally, Section V concludes the paper.

II. Related Work

To formally specify an interface, regular expressions and temporal logic [9] have been used. They can be expressed with finite-state automata. Some standard languages like Property Specification Language (PSL) and the System Verilog Assertions (SVA) have been proposed recently to specify system properties. These languages are based on temporal logic, but both of them also have an ability to specify regular expressions. In PSL, such an extension is called Sequential Extended Regular Expressions (SEREs). Balarin *et al.* [3] proposed to specify TLM-RTL transactors using PSL. They took advantage from the SEREs aiming at generating transactors which must be synthesizable. Hence, it presents a limitation of the use of transactors in the SystemC design flow only at RTL. Several commercial tools include features to generate transactors in SystemC such as SystemC Transactor Generation Wizard from Aldec's Active HDL [1], Catapult C from Mentor Graphics, TransactorWizard from Structured Design Verification [12], and Cohesive from Spiratech [11]. For instance, 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 user has to write the transactor code in SystemC manually. In [6], a method and tool for generating SystemC TLM-RTL transactors from AsmL specifications has been introduced. The work presented in this paper is different from [6] and [3] as it allows users to specify the transactor as graphical FSM.

III. Proposed Methodology

From the natural language text specification, a formal model of the transactor protocol is created by drawing the protocol as a finite state machine (FSM). FSM drawing is common in hardware design environment, thus it reduces the overhead to learn new specification language. Also, a graphical FSM is intuitive, easy to follow, and understand.

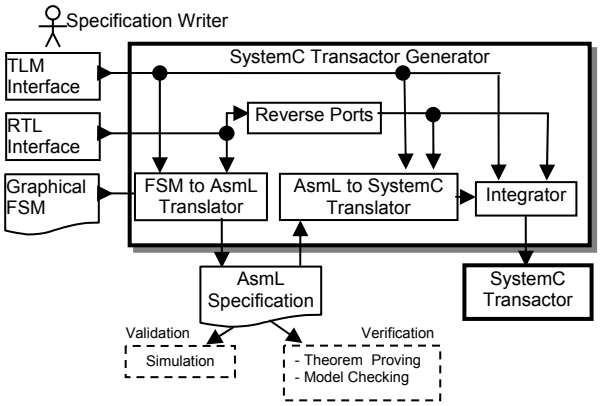

Figure 1: Transactor Generation from FSM

To generate a SystemC transactor, the *TLM Interface*, the *RTL interface* and the *Graphical FSM* description of the protocol are given as input as shown in Figure 1. The *FSM to AsmL Translator* generates AsmL code from the FSM description. The generated AsmL specification is executable, thus it gives the opportunity to do validation by simulation. Also, AsmL specification can be formally verified by model checking tools like SMV and theorem proving tools like PVS, Isabelle, etc. Once the AsmL code is verified, it is then passed to the *AsmL to SystemC Translator* to generate SystemC code. The *Reverse Port* block reverses the ports direction of the transactor w.r.t. the RTL unit. The *Integrator* adds other necessary codes to generate the complete SystemC transactor.

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

- The action statements in a state must be written in the syntax of AsmL. They are executed simultaneously according to the *update* semantics.
- State transitions occur after one clock cycle and *updates* of the variables and ports are fired.
- 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.
- FSM inside a transactor must be terminated when the operation on the RTL side is completed. So, to indicate the state at which the FSM will be terminated, we set that state as *trap state*.

The graphical FSM is compiled to a lexical format *Active HDL State Machine Format (ASF)* from Active HDL [1]. The *FSM to AsmL Translator* reads the FSM objects from the ASF file to the data structures as shown in Figure 2 and then

automatically generates AsmL code according to the algorithm shown in Figure 3.

State	Action
ID: Integer Label: String isDefState: Boolean isTrapState: Boolean	ID: Integer StateID: Integer Statement: String
Condition	**TransLine**
ID: Integer TransLineID: Integer Expression: String	ID: Integer SrcStateID: Integer DstStateID: Integer Priority: Integer

Figure 2: FSM Objects and their properties

A graphical FSM is a discrete structure consisting of vertices and edges like directional graph. The algorithm shown in Figure 3 is developed with the flavor of directional graph traversing.

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

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: AsmL Code Generation Algorithm from FSM

An enumerated type state variable `CurrentState` is used to hold the present state. A `step while` block [7] 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 [7] which is used to switch to different states depending on the `CurrentState`. For a *State*, the code generator writes its *Label* followed by a 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. Like AsmL, SystemC also has the notion of *update* in its simulation semantics. We map the AsmL syntax and semantics to SystemC so that the behavior of the AsmL code is preserved in the translated SystemC code. The detailed mapping rules from AsmL to SystemC are described in [6].

IV. Case Study: UTOPIA Transactor

UTOPIA [2] is a standard protocol used to connect devices that implement ATM (Asynchronous Transfer Mode) and PHY (PHYsical) 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 4.

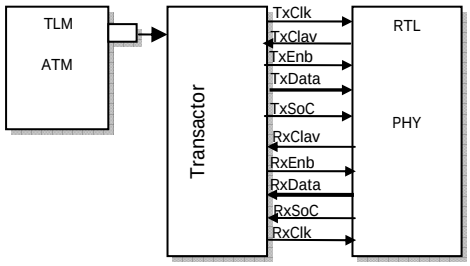

Figure 4: UTOPIA Transactor

From the ATM module, when a TLM function, e.g., `SendCell()` is called, the transmit protocol must be followed by the transactor to complete the task. We draw the FSM of the protocol for sending cells which consists of four states namely *S_CheckCellAvailable*, *S_SendCell*, *S_CloseTxWindow*, and *S_End* as shown in Figure 5. At first, the state machine enters the initial state *S_CheckCellAvailable*. If *TxClav* is asserted then it sets the next state as *S_SendCell*. At the state *S_SendCell*, the transactor opens the *transmit window* [2] by asserting *TxEnb*. *TxSoC* is asserted when transmitting the first byte of the cell. It also drives *TxData* with the corresponding byte from the source cell 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 cells (if required) can be transmitted. If PHY is unable to accept more cells, then it sets

the next state as *S_CloseTxWindow*. At the state *S_CloseTxWindow*, *TxEnb* is de-asserted and thus the *transmit window* is closed. If all cells are transferred, then the state machine enters state *S_End* and the `SendCell` function ends. Otherwise it sets the next state as *S_CheckCellAvailable*.

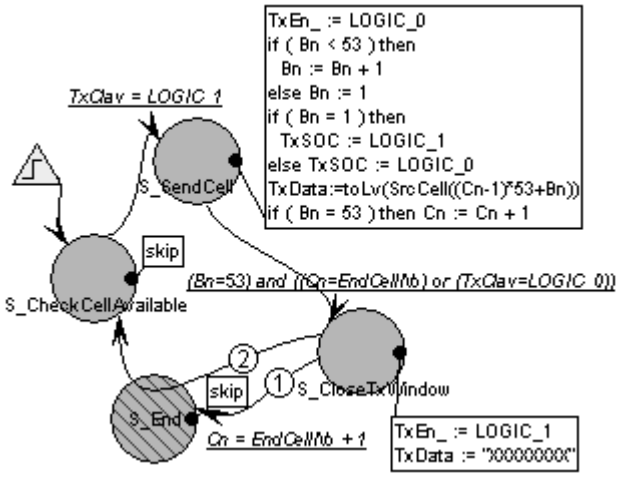

Figure 5: Graphical FSM for the Function SendCell()

The FSM description is saved in ASF format [1] and then AsmL code is generated from the FSM. A snapshot of the generated AsmL code is shown in Figure 6.

```
enum typeState
  S_CheckCellAvailable
  S_SendCell
  S_CloseTxWindow
  S_End

public SendCell (  StartCellNo as Integer, EndCellNo
as Integer, SrcCell as Seq of Integer )
  var CurrentState as typeState=S_CheckCellAvailable

  step while ( CurrentState <> S_End )
    match ( CurrentState )
      S_CheckCellAvailable :
        skip // Actions
        // Next State
        if ( TxClav = LOGIC_1 ) then
          CurrentState := S_SendCell
      S_CloseTxWindow :
        TxEn_ := LOGIC_1 // Actions
        TxData := "XXXXXXXX"
        // Next State
        if ( Cn = EndCellNo + 1 ) then
          CurrentState := S_End
        else
          CurrentState := S_CheckCellAvailable
        …
       // Trap State
       otherwise
        // Next State
        CurrentState := S_End
```

Figure 6: Generated AsmL Code from Graphical FSM

The AsmL code is then translated to SystemC and other necessary codes are added to generate the complete SystemC transactor. We draw the FSM specification of the transactor functions `SendCell` and `GetCell` for both blocking and non-blocking [10] cases and generated the SystemC transactor based on the proposed methodology. 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. The number of AsmL lines is linearly proportional to the number of states in the FSM drawing, number of action and condition statements in a state. Table 1 shows that the number of SystemC lines of code grows linearly with the AsmL code.

TABLE I
EXPERIMENTAL RESULTS

Transactor Function	No. of States	No. of Lines		Time/Cell in SystemC	
		AsmL	SystemC	Sim (µs)	CPU (ms)
`SendCell`	4	41	82	2.2	148
`nb_SendCell`	4	42	83		156.5
`GetCell`	4	32	66	2.2	70
`nb_GetCell`	4	38	78		78

This linear relationship promises expected CPU or machine execution time. Table 1 also shows the required simulation time for sending and receiving a cell in SystemC, which depends on the UTOPIA model clock signals such as *TxClk* and *RxClk*. The experiments were conducted on a Pentium Mobile processor (1.8 GHz) with 512 MB of memory.

V. CONCLUSION

We proposed an approach for the automatic generation of SystemC transactors from graphical FSM description. Visual representation of transactor protocol is easy and intuitive to understand. We developed an algorithm to translate the FSM to AsmL. AsmL specifications are executable and verifiable as they adhere to formal syntax and semantics. We have conducted a case study with the UTOPIA Interface. Our future work includes providing a library for standard protocols so they can be used in generating transactors that implement standard protocol interfaces.

REFERENCES

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

[2] ATM Forum Technical Committee. Utopia Level 2, Ver. 1.0, June 1995.

[3] 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.

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

[5] 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.

[6] T.H. Khan, A. Habibi, S. Tahar and O. Ait Mohamed. Automatic Generation of SystemC Transactors from AsmL Specifications; In Proc. *Forum on Specification & Design Languages*, pages 104-109, Barcelona, Spain, September 2007.

[7] Microsoft Corporation. AsmL: Abstract state machines Language, 2007. http://research.microsoft.com/fse/asml/.

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

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

[10] A. Rose, S. Swan, J. Pierce, J.M. Fernandez. Transaction Level Modeling in SystemC; Open SystemC Initiative, 2006.

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

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