Integrating RTL IPs into TLM Designs Through Automatic Transactor Generation*

Nicola Bombieri Nicola Deganello Franco Fummi
Dipartimento di Informatica
Università di Verona, Italy
{bombieri, deganello, fummi}@sci.univr.it

Abstract

Transaction Level Modeling (TLM) is an emerging design practice for overcoming increasing design complexity. It aims at simplifying the design flow of embedded systems by designing and verifying a system at different abstraction levels. In this context, transactors play a fundamental role since they allow communication between the system components, implemented at different abstraction levels. Reuse of RTL IPs into TLM systems is a meaningful example of key advantage guaranteed by exploiting transactors. Nevertheless, transactors implementation is still manual, tedious and error-prone, and the effort spent to verify their correctness often overcomes the benefits of the TLM-based design flow. In this paper we present a methodology to automatically generate transactors for RTL IPs. We show how the transactor code can be automatically generated by exploiting the testbench of any RTL IP.

1. Introduction

TLM is nowadays the reference modeling style for HW/SW design and verification of digital systems [9]. TLM greatly speeds up the verification process by providing designers with different abstraction levels whereby digital systems are modeled and verified. Thus, the complexity of the modern systems can be handled by designing and verifying them through successive refinement steps [12].

Top-down and bottom-up approaches are often mixed in a TLM-based design flow (see Figure 1). The system is firstly modeled at high-level in order to check the pure functionality, disregarding details related to the target architecture. Due to the lack of implementation details, the simulation speed is some orders of magnitude faster than at RTL [8]. Then, step by step, designers refine and verify the system description more accurately, towards the final implementation.

*This work has been partially supported by the European project VERTIGO FP6-2005-IST-5-033709.

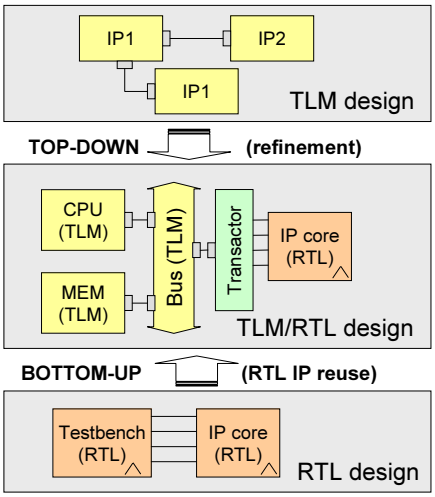

Figure 1. Top-down and bottom-up approaches in TLM design flows.

On the other hand, reuse of previously-developed Intellectual Properties (IP) modules (i.e., RTL computational cores, RTL buses, etc.) is another key strategy that guarantees considerable saving of time in TLM [5]. In fact, modeling a complex system entirely at transaction level could be inconvenient when IP cores are already available on the market, usually modeled at RTL.

In this context, EDA companies and academic researchers have proposed modeling and verification methodologies based on transactors [7, 16, 14]. Despite technical differences, all of them exploit the concept of a *transactor* to allow the mixed TLM-RTL co-verification based on simulation. A transactor works as a translator from TLM function calls to sequences of RTL statements, that is, it provides the mapping between transaction-level requests, made by TLM components, and detailed signal-level protocols on the interface of RTL IPs.

Even if transaction-based verification (TBV) is increasingly used, the problem of transaction generation for TLM-RTL co-simulation has been only partially explored being a

978-3-9810801-3-1/DATE08 © 2008 EDAA

new challenge for designers and verification engineers. Designers actually implement transactors by interpreting either the communication protocol specifications or the RTL code of the related IP. As a consequence, the effort spent to implement transactors and to verify their correctness often overcomes the benefits of the TLM-based design flow.

Considering the state of the art, the transactor generation in TLM can be seen as a special case of automatic generation of adapters for incompatible protocols in the RTL context [15, 18, 3, 17]. A technique is presented in [15] for interfacing standard components that have incompatible protocols. Given an RTL HDL description of the two protocols, an interface process is generated to allow the two protocols to communicate with each other. In [18, 3, 17], different approaches based on finite automata are presented. In all these works, designers have to manually specify the protocols by using some formalism such as regular expressions [13] or temporal logic [19]. Starting from the accurate description of control and data lines, and the sequence of data transfers over those lines, a synthesis process generates the protocol converter.

The generation of transactors specific for TLM-RTL communication has been more recently analyzed in [4] and [6]. Both works present an approach based on finite automata, in which the protocol specifications have to be described in Property Specification Language (PSL) in the first paper while Extended Finite State Machines (EFSMs) are exploited in the second paper. Nevertheless, generation time and correctness of the result fully depend on the designer accuracy and capability which manually describes the formal model of the communication protocols.

In this paper, we present a methodology that automates the transactor generation for RTL IP components to be reused into TLM systems. Assuming that an RTL testbench is released with the RTL IP component, the methodology automates each step of the generation process. Protocol information is extracted from the testbench and represented by the Extended Finite State Machine (EFSM) model [10]. Finally, relying on a TLM APIs library, the SystemC code of the transactor is automatically generated.

The paper is organized as follows. Section 2 presents an overview of the transactor object and the main categories of transactors. Section 3 describes the proposed methodology and each step of the generation process. Implementation details and experimental results are presented in Section 4, while concluding remarks in Section 5.

2. Transactor Overview

A transactor is associated with two API's, one for the side at the higher level of abstraction (i.e., TLM) and one for the lower level side (i.e., RTL). It works as a translator from function calls to sequences of statements implemented at a lower abstraction level and viceversa. The two main functionalities of transactors are the following:

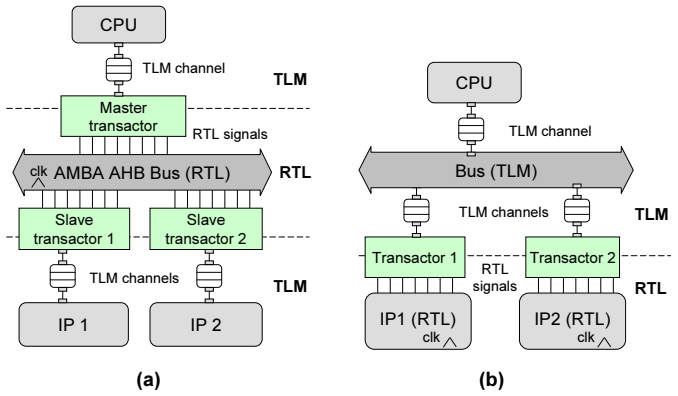

Figure 2. Examples of transactor applications.

1. Mapping of TLM API to RTL API. On one side, depending on the abstraction level of the TLM component, different function calls and different data structures can be used to define the TLM API. On the other side, RTL APIs can differ in data and control ports depending on the adopted communication protocol.

2. Translation of TLM function calls to sequences of RTL signals (i.e., RTL handshaking sequences) and viceversa. Since TLM components usually do not implement any "low level" communication protocol, handshaking and accurate temporization towards the RTL component must be carried out by the transactor.

Let us classify transactors in two main categories depending on their RTL side.

2.1 Transactors for standard RTL communication protocols

Transactors for buses are meaningful examples of this category. Different buses are available in the commerce, which present accurate communication protocols, that are well defined by formal specifications (i.e., AMBA AHB, STBus, OCP-IP, etc.). Figure 2(a) shows an example, where three TLM components (one master and two slaves) communicate through a RTL bus synchronized by using transactors. Because of the TLM compositional approach, the three modules (master and slaves) and the bus are implemented at different abstraction levels. This model represents the majority of cases in which an existent RTL bus is reused and transactors are exploited to bridge the abstraction gap between the TLM and RTL modules. Thus, a *Master-transactor*, translates TLM master calls into sequences of RTL signals for the bus and viceversa, while a *Slave-transactor* translates RTL signals of the bus into TLM function calls for the slave and viceversa. This configuration, for example, is adopted when performance analysis related to the communication protocol of the system is taken by monitoring the clock accurate RTL bus.

2.2 Transactors for non-standard RTL communication protocols

Figure 2(b) shows an application example in which the master (CPU) and the bus compose the TLM side of the system, while the slaves (IP1, IP2) are implemented at RTL. In this case, the configuration is suitable for two different purposes:

1. The RTL IP is reused and connected to the system to analyze the whole system integrity and correctness.
2. The RTL IP is under development and its refinement process strictly depends on the environment, which is implemented at TLM.

In this case, the bus is actually implemented at a level of abstraction in which the very simple communication mechanism relies on TLM function calls (i.e., write(), read(), put(), get()). On the other side, the API of the RTL components have been embedded depending on the component functionality rather than following any standard or well known communication protocols.

In the transactor generation, unlike the TLM side in which APIs are quite simple, controlled by a reference standard (i.e., OSCI TLM) and in any case limited in number, the main difficulties arise in the implementation of the RTL side. As explained in Section 1, even the methodologies that aim at automating the transactor generation needs a manual intervention to accurately describes the RTL protocol. Differently to transactors for standard protocols for which a library of formal specification could be created and reused, the manual step of RTL protocol description is needed to generate transactors implementing non-standard protocols. The methodology presented in Section 3 aims at avoiding this manual step, by extracting the corresponding information from the RTL testbench released with the RTL IP.

3. Generation Flow

Figure 3 shows the overview of the methodology for transactors generation that is detailed in the following subsections. We assume that an RTL testbench is available together with the RTL IP. The RTL testbench actually sends testvectors to and receives results from the IP core by performing an ordered sequence of *write* and *read* operations in compliance with the IP communication protocol. We call *RTL driver* that sequence of write and read operations on the PIs and POs of the IP interface.

The proposed methodology exploits the RTL driver information to implement the RTL side of transactors while the TLM side is settled by exploiting any standard TLM API (e.g., the OSCI TLM core interfaces).

A preliminary manual effort is required to identify the EFSMs actually implementing the RTL communication protocol and to provide mapping information between TLM

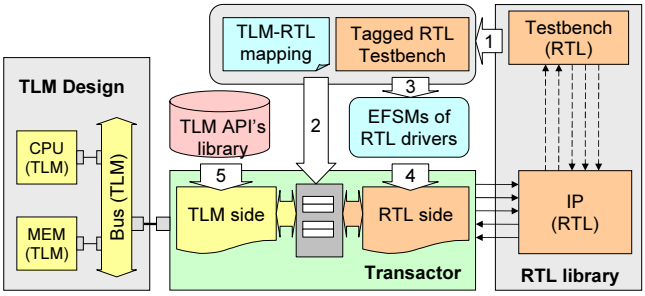

Figure 3. Testbench-centric methodology

I/O values and RTL I/O ports (step 1). Once the preliminary step has been settled, the EFSM representing the RTL driver and the RTL interface are extracted from the tagged RTL testbench (step 2). A formal model is needed to represent the communication protocol for both TLM and RTL sides of transactors. The EFSM model has been chosen for three main reasons:

1. The Finite State Machine (FSM) model has been extensively used to model and verify protocols [11]. EFSMs are an extension of FSMs.
2. The EFSM model maintains controlled the state explosion problem by adding expressivity to the transitions [21].
3. Thanks to the expressiveness of transitions, transactor code can be automatically generated starting from the EFSM representation.

Then, the SystemC code representing the RTL and TLM sides is extracted respectively from the EFSM of RTL driver (step 3) and from the TLM API library (step 4). Finally, the infrastructure supporting data-exchange between the TLM and RTL side is generated by exploiting the previously defined mapping function (step 5).

3.1 Preliminary step (step 1)

A preprocessing stage is required to provide those information which cannot be automatically determined, and it represents the only necessary manual task. Once the preliminary information is settled, a fully functional transactor is generated without needing any additional manual modification.

Two types of information are needed:

- *EFSM borders.* EFSMs of the RTL drivers representing *WRITE* and *READ* operations are identified in the testbench by tagging the initial and final states visited during an access for sending data to or receiving data from the RTL IP (see Figure 4). This provides the necessary support to extract information of the RTL protocol encapsulated in the testbench. For the sake

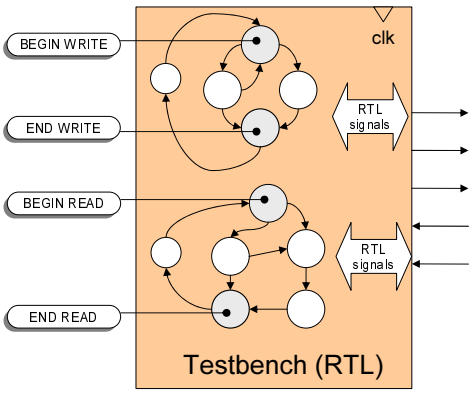

Figure 4. Tagging of EFSMs

of clarity, it is assumed that *WRITE* and *READ* operations are performed through the RTL interface by two distinct EFSMs. In fact, if the testbench implements a single EFSM performing both operations, the EFSMs performing *WRITE* and *READ* will be treated as being distinct but equal. Figure 4 shows an example of the tagging process. The initial and finals state of EFSM performing *WRITE* operations are tagged with `BEGIN WRITE` and `END WRITE`. Similarly, `BEGIN READ` and `END READ` tags are used to identify the EFSM performing *READ* operations.

- *Mapping between TLM values and RTL ports.* A set of "relevant" I/O objects is settled for representing data shared between the TLM and RTL sides. Any object of this set corresponds to a PI or a PO that is present in both the TLM interface (i.e., as function call parameter) and the RTL interface (i.e., as input or output port). For example, data ports (i.e., input ports, result ports) of the RTL IP core can be considered relevant rather than control ports specific to the RTL protocol (i.e., ports for enabling flags, ports for acknowledgment, etc.). This provides the support to generate the *data-exchange structures* which ensure a proper communication between the TLM and RTL sides.

3.2 Data-exchange structure generation (step 2)

Data structures providing support for exchanging data between TLM and RTL side are generated by exploiting the mapping functions defined in the preliminary step. It is composed of the following parts:

- A *request extension* record whose field names correspond to the names of RTL ports involved in sending data operations.
- A *response extension* record whose field names correspond to the names of RTL ports involved in receiving data operations.

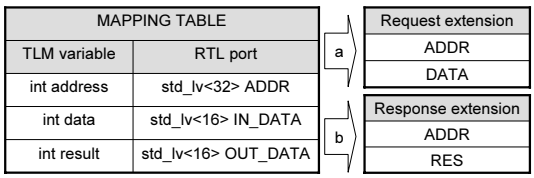

Figure 5. Generation examples of request extension (a) and response extension (b)

Figure 5 shows an example in which three shared I/O objects compose the TLM-RTL mapping table. Variables `address`, `data` and `result` of the TLM request are respectively mapped into RTL ports `ADDR`, `IN_DATA` and `OUT_DATA`. The *Request extension* record with fields `ADDR` and `DATA`, and the *Response extension* records with fields `ADDR` and `RES` are thus generated.

It is important to note that these data structures compose the actual border layer between TLM and RTL. Thus, even if this step adds a degree of redundancy concerning the exchanged data, it ensures modularity in the generation process of transactors. In fact, different TLM interfaces can be chosen for composing the TLM side of the transactors, as explained in Section 3.4.

3.3 RTL side generation (steps 3, 4)

The RTL side is composed of the following parts, that are automatically generated from the tagged RTL testbench:

- The set of input and output ports composing the *RTL interface*. They correspond to the testbench ports that are directly linked to the RTL IP core.
- The EFSMs implementing *write* and *read* operations through the RTL interface. They correspond to the EFSM sub-graphs included into the EFSMs of the testbench, which perform the corresponding write and read operations.

The automatically extracted EFSMs are elaborated to support communication with the TLM side, by exploiting the data-exchange structures generated at the previous step. Thus, request values are received by the TLM side of the transactor and they are available to the RTL side through the *request extension* record. Similarly, the values of RTL ports are available to the TLM side through the *response extension* record.

In this context, only transaction-specific values (e.g., address, data, result, etc.) are considered in the data-exchanged structures, as they represent the only information which flows between the TLM and RTL sides through the communication layer.

On the other hand, protocol details specific to the RTL interface (i.e., handshaking sequences, pipelining, burst cy-

cles, etc.) are extracted from the RTL testbench and preserved in the RTL side. Thus, from the TLM point of view, data exchanging is performed disregarding these details, since they are inherited from the testbench and transparently handled by the transactor.

3.4 TLM side generation (step 5)

TLM communication is based on API function calls (e.g., `read()`, `write()`, `put()`, `get()`) and it is usually controlled by a standard (i.e., OSCI TLM library). In this context, the TLM interfaces do not present the same variety of their RTL counterparts and they are limited in number.

Three properties characterize the TLM communication protocols:

- type of *communication primitives* (i.e., blocking function calls, non-blocking function calls);
- *temporization* of communication primitives (i.e., untimed function calls, timed function calls);
- type of *communication channels* (i.e., bidirectional interface, unidirectional interface).

A taxonomy of TLM communication protocols can be drawn up by combining these properties each other. The most relevant are the following:

1. *Bidirectional Blocking Untimed.* An initiator writes data to and reads data from a target, for example, by sending a request packet (i.e., `REQ`) and receiving the response (i.e., `RSP`) through an atomic function call, as in the follows:
   `void transport(const REQ&, RSP&);`

2. *Bidirectional Blocking Timed.* In this context, timing annotation is added as function parameter to the atomic function call:
   `void transport(const REQ&, RSP&, sc_time&);`

3. *Unidirectional Blocking Untimed.* A write or read transaction is split into a sequence of blocking function calls. For example, an initiator firstly requests a transaction by means of `put(const REQ&)`. Once the request has succeeded, the initiator peeks the channel for a response, by means of `peek(RSP&)`, waiting in a suspended state in case the channel is empty. Finally, it gets the response from the channel by means of `get(RSP&)`.

4. *Unidirectional Non-blocking Untimed.* In the non-blocking interface, function calls are not allowed to wait in a suspended state. Thus, every call returns a boolean value to indicate whether the non-blocking access succeeded. Thus, an initiator initiates a transaction by calling `nb_put(const REQ&)` in a polling-based mechanism. Once the correct conditions for the call succeed, `nb_peek(RSP&)` and `nb_get(RSP&)` are called in the same way to peek and get the response.

5. *Unidirectional Non-blocking Timed.*. The semantic of this communication protocol preserves the sequence of unidirectional non-blocking function calls. Nevertheless, the function calls are provided with timing annotations. Thus, put, peek and get functions are still called with the polling mechanism, in which a delay or a latency is explicitly expressed for each call (i.e., `nb_put(const REQ&, sc_time&)`, `nb_peek(RSP&, sc_time&)`, `nb_get(RSP&, sc_time&)`).

4. Experimental Results

The presented methodology has been implemented in TGEN, a tool built on the top of HIFSuite [1]. The effectiveness of the methodologies has been evaluated by generating transactors of two different types: for standard and non-standard RTL communication protocols (see Section 2). In particular, transactors have been generated for the the following RTL designs:

- AMBA AHB Bus.
- STBus type 2.
- Fast Fourier Transform (FFT).
- FIR Filter.

The first two designs have been provided by STMicroelectronics while FFT and FIR are RTL designs provided with the example set of SystemC 2.2 [2]. The transactor codes have been generated in SystemC, considering the OSCI TLM library [20] as reference library. For the specific architectural choices, the communication mechanism has been implemented in the Unidirectional Blocking Untimed form (see Section 3.4).

Table 1 shows the obtained results. Columns *Testbench* shows the number of lines of code of the testbenches which have been analyzed by the TGEN parser for extracting the RTL drivers. Column *RTL ports* reports the number of I/O ports of the RTL design interface. The number of relevant objets manually settled for representing data shared between the TLM and RTL sides is reported in Column *Relevant I/O objects*. Columns *READ RTL driver* and *WRITE RTL driver* shows the number of states and transitions of the EFSMs extracted from the RTL testbench, which model the read and write operations towards the design. Columns *RTL side* and *TLM side* report respectively the number of code lines of the RTL and TLM sides of the transactors. Finally, column *Transactor* shows the total number of code lines of the transactor implementations.

For each design, few minutes of manual work have been spent for the preliminary step. Then, the automatic transactor generation has been instantaneously accomplished by

Design	Testbench (loc)	RTL ports (#)	Relevant I/O objects (#)	READ RTL driver		WRITE RTL driver		RTL side (loc)	TLM side (loc)	Transactor (loc)
				#states	#trans	#states	#trans			
AMBA AHB	79	15	4	3	3	3	3	110	26	237
STBus t2	89	9	4	3	3	3	3	56	26	187
FFT	244	10	4	2	2	2	2	75	26	208
FIR	280	8	2	1	1	1	1	24	26	139

Table 1. Experimental results

the TGEN tool. On the other hand, 3 days/man have been spent for manually implementing the four transactors. Correctness of the obtained results has been proven by using testbenches provided by STMIcroelectronics.

5. Concluding Remarks

The paper addressed the problem of automatic transactor generation for reusing RTL IPs in TLM designs. We proposed a methodology that automates the generation process, assuming that an RTL testbench is released with the RTL IP component. This aims at obtaining their correct-by-construction implementation. After a manual preliminary step, the methodology extracts the protocol information from the testbench by exploiting the Extended Finite State Machines (EFSMs) model. Finally, relying on a TLM APIs library, the SystemC code of the transactor is automatically generated. Even if the preliminary manual step can be considered a limitation, it is much easier and quicker than completely coding transactors from scratch. The methodology effectiveness and correctness have been shown by generating transactors of different RTL designs which implement both standard and non-standard protocols.

Acknowledgments

The authors would like to thank Andrea Fedeli of STMicroelectronics for providing the case study and supporting the analysis of the application results.

References

[1] http://esd.sci.univr.it.

[2] http://www.systemc.org.

[3] J. Akella and K. McMillan. Synthesizing converters between finite state protocols. In *Proc. of IEEE ICCD*, pages 410–413, 1991.

[4] F. Balarin and R. Passerone. Specification, synthesis and simulation of transactor processes. *IEEE Transactions on Computer Aided Design of Integrated Circuits*, 26(10):1749–1762, 2007.

[5] N. Bombieri, A. Fedeli, F. Fummi, and G. Pravadelli. Hybrid incremental ABV for functional validation in TLM design flows. *IEEE Design and Test of Computer*, 24(2):140–152, March-April 2007.

[6] N. Bombieri and F. Fummi. On the automatic transactor generation in tlm-based design flows. In *Proc. of IEEE HLDVT*, pages 85–92, 2006.

[7] D. Brahme, S. Cox, J. Gallo, M. Glasser, W. Grundmann, C. N. Ip, W. Paulsen, J. Pierce, J. Rose, D. Shea, and K. Whiting. The transaction-based verification methodology. Technical Report CDNL-TR-2000-0825, Cadence Berkeley Labs, 2000.

[8] A. Bruce, A. Nightingale, N. Romdhane, M. M. K. Hashmi, S. Beavis, and C. Lennard. Maintaining consistency between systemC and RTL system designs. In *Proc. of ACM/IEEE DAC*, pages 85–89, 2006.

[9] L. Cai and D. Gajski. Transaction level modeling: An overview. In *ACM/IEEE CODES+ISSS*, pages 19–24, 2003.

[10] K.-T. Cheng and A. Krishnakumar. Automatic generation of functional vectors using the extended finite state machine model. In *ACM TODAES*, volume 1, pages 57–79, 1996.

[11] P. Chu and M.T.Liu. Synthesizing protocol specifications from service specifications in FSM model. In *IEEE CNS*, pages 173–182, 1988.

[12] T. Grotker, S. Liao, and G. M. ans Si Swan. *System Design with SystemC*. Kluwer Academic Publishers, Norwell Massachusetts, 2002.

[13] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, reading, MA, 1986.

[14] R. Jindal and K. Jain. Verification of transaction-level systemC models using RTL testbenches. In *ACM/IEEE MEMOCODE*, pages 199–203, 2003.

[15] S. Narayan and D. D. Gajski. Interfacing incompatible protocols using interface process generation. In *Proc. of ACM/IEEE DAC*, pages 468–473, 1995.

[16] C. Norris Ip and S. Swan. A tutorial introduction on the new SystemC verification standard. White paper. www.systemc.org, 2003.

[17] R. Passerone, L. De-Alfaro, T. A. Henzinger, and A. Sangiovanni-Vincentelli. Convertibility verification and converter synthesis: two faces of the same coin. In *Proc. of ACM/IEEE ICCAD*, pages 132–139, 2002.

[18] R. Passerone, J. A. Rowson, and A. Sangiovanni-Vincentelli. Automatic synthesis of interfaces between incompatible protocols. In *Proc. of ACM/IEEE DAC*, pages 8–13, 1998.

[19] A. Pnueli. The temporal logic of programs. In *Proc. of IEEE SFCS*, pages 46–57, 1977.

[20] A. Rose, S. Swan, J. Pierce, and J.-M. Fernandez. Transaction level modeling in SystemC. White paper. www.systemc.org, 2004.

[21] F. Slomka, M. Dorfel, and R. Munzenberger. Generating mixed hardware-software systems from SDL specifications. In *ACM/IEEE CODES+ISSS*, pages 116–121, 2001.