# Automatic Synthesis of OSCI TLM-2.0 Models into RTL Bus-based IPs

Nicola Bombieri, Franco Fummi, and Valerio Guarnieri
Department of Computer Science
University of Verona
{firstname.lastname}@univr.it

***Abstract*—Transaction-level modeling (TLM) is the most promising technique to deal with the increasing complexity of modern embedded systems. TLM provides designers with high-level interfaces and communication protocols for abstract modeling and efficient simulation of system platforms. The Open SystemC Initiative (OSCI) has recently released the TLM-2.0 standard, to standardize the interface between component models for bus-based systems. The TLM standard aims at facilitating the interchange of models between suppliers and users, and thus encouraging the use of virtual platforms for fast simulation prior to the availability of register-transfer level (RTL) code. On the other hand, because a TLM IP description does not include the implementation details that must be added at the RTL, the process to synthesize TLM designs into RTL implementations is still manual, time spending and error prone. In this context, this paper presents a methodology for automating the TLM-to-RTL synthesis by applying the theory of high-level synthesis (HLS) to TLM, and proposes a protocol synthesis technique based on the extended finite state machine (EFSM) model for generating the RTL IP interface compliant with any RTL bus-based protocol.**

## I. Introduction

TLM is nowadays the reference modeling style for design and verification of modern system-on-chips (SoCs) at the electronic system-level (ESL) [1]. TLM greatly speeds up the design process by allowing designers to model and verify complex systems at different abstraction levels above RTL [2]. The design implementation at different abstraction levels is essential to quickly evaluate system-level exploration for architectural decisions, such as hardware and software design, verification, memory organization, and power management.

There has been a longstanding discussion in the ESL community concerning what is the most appropriate taxonomy of abstraction levels in TLM. Models have been categorized according to a range of criteria, including granularity of time, frequency of model evaluation, functional abstraction, communication abstraction, and use cases [2], [3], [4], [5], [6]. In this context, the Open SystemC Initiative (OSCI) [7] committee has been developing a reference standard for TLM in the last years for guaranteeing the maximum interoperability between suppliers and users. TLM-2.0 has become the final reference standard for SystemC TLM [8].

Nevertheless, even though a TLM-based design flow relies on such a standard, heavy manual intervention is required as soon as a TLM IP description has to be synthesized into an equivalent RTL implementation. In fact, a TLM-to-RTL refinement generally starts from a TLM IP model, where the system level (and mostly untimed) functionality is provided with a standard TLM interface (e.g., blocking, non-blocking, timed, untimed) (see Figure 1). In the refinement step, the TLM IP functionality is synthesized into a more accurate (i.e., cycle accurate) RTL implementation, while the TLM interface is replaced by a first cycle accurate and pin accurate interface (that will be called *basic interface* hereafter). The basic interface is generally composed of all the data I/O ports and some control ports for implementing basic mechanisms of handshaking (e.g., enabling input data, enabling result). Then, targeting to integrate the RTL IP model into a bus-based platform, designers extend the RTL IP implementation with an interface compliant with the target bus interface (that will be called *bus-based interface* hereafter), and with a *transducer* for translating the bus-based interface into the basic protocol interface and viceversa. Nowadays, the two synthesis steps, from the TLM IPs to the RTL implementations, are manual, time spending and error prone.

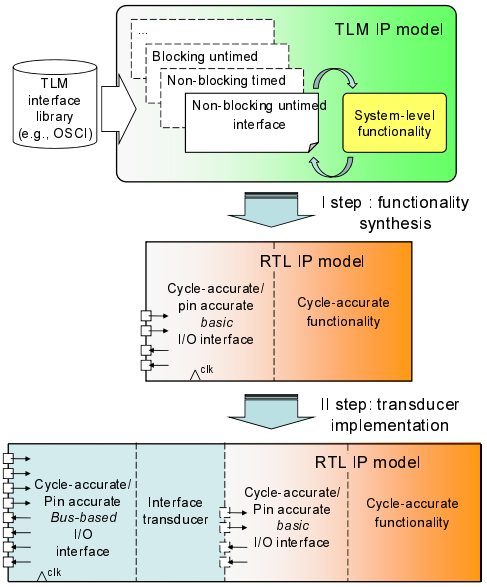


Fig. 1. TLM-to-RTL refinement flow.

On the other hand, high-level synthesis (HLS) is considered

This work has been partially supported by the European project COCONUT FP7-2007-IST-1-217069.

the reference paradigm for automatically generating RTL descriptions starting from high-level algorithmic models [9]. In a HLS-based design flow, designers begin the specification of an IP with a high-level description which captures the desired functionality, using an high-level language (HLL), such for example C. This first step thus involves writing a functional specification (an untimed description) in which a function consumes all its input data simultaneously, performs all computations without any delay, and provides all its output data simultaneously. At this abstraction level, variables (structure and array) and data types (typically floating point and integer) are related neither to the hardware design domain (bits, bit vectors) nor to the embedded software. A realistic hardware implementation thus requires conversion of floating-point and integer data types into bit-accurate data types of specific length with acceptable computation accuracy, while generating an optimized hardware architecture starting from this bit-accurate specification. HLS tools transform an untimed (or partially timed) high-level specification into a fully timed implementation [10]. They automatically or semiautomatically generate a custom architecture to efficiently implement the specification. In addition to the memory banks and the communication interfaces, the generated architecture is described at the RTL and contains a data path (registers, multiplexers, functional units, and buses) and a controller, as required by the given specification and the design constraints.

In this context, we present a methodology for automating the synthesis of TLM IPs into RTL implementations. In this work, we require the TLM IP interface to be compliant with the standard OSCI TLM-2.0 library as it is the most common and largely used in the TLM community. However, the methodology can be easily extended for supporting any different or user-defined TLM library. The methodology applies the theory of HLS to TLM and proposes a protocol synthesis technique based on the extended finite state machine (EFSM) model for generating the RTL IP interface compliant with any RTL bus-based protocol.

Our methodology relies on two main steps: (1) high-level synthesis and (2) TLM protocol synthesis into any standard bus-based protocols. Different academic/commercial tools (e.g., Forte [11]) do the first step but there is no work neither tool that performs correct-by-construction translation from TLM protocols into ”any” bus-based protocol. Our methodology generates an RTL interface that complies with such bus-based protocols and guarantees (by exploiting the EFSM formal model) that the TLM transactions are correctly mapped into RTL bus transfers.

A different solution is proposed in [12], in which designers start from a specific language (i.e., SHIM) and, then, perform the synthesis into HW or SW. Our work differs as we start from a TLM SystemC design and synthesize it into an RTL implementation

The rest of the paper is organized as follows. Section II summarizes the background for understanding the proposed methodology, including the key concepts of TLM and EFSM. Section III presents the TLM-to-RTL synthesis methodology as a whole. Section IV shows the experimental results, while conclusions are drawn in Section V.

## II. Background

In this section, we introduce the key concepts of TLM and EFSM, the formal model whereby we model the TLM and RTL communication protocols.

### A. OSCI TLM-2.0: use cases, interfaces, and coding styles

The OSCI committee explicitly recognizes the existence of a variety of *use cases* in TLM, such as SW development, SW performance analysis, architectural analysis, and HW verification. However, rather than defining an abstraction level around each use case, the TLM-2.0 standard describes a number of *coding styles* that are appropriate for, but not locked to, the various use cases. Two examples of TLM-2.0 coding styles proposed by OSCI are the following:

- *Loosely-timed.* The loosely-timed coding style is appropriate for software development, by using, for example, a virtual platform model of an MPSoC, where the software may include one or more operating systems. Models implemented with this coding style have a loose dependency between timing and data. They do not depend on the advancement of time to be able to produce a response and, normally, resource contention and arbitration are not considered.
- *Approximately-timed.* The approximately-timed coding style is appropriate for architectural exploration and performance analysis. Models implemented with this coding style have a much stronger dependency between timing and data. Since these models must synchronize the transactions before processing them, they are forced to trigger multiple context switches in the simulation, eventually resulting in performance penalties. On the other hand, they easily model resource contention and arbitration.

The best-suited coding style is applied depending on the target use case and each coding style is implemented by using a specific TLM interface. The TLM-2.0 standard defines the following interfaces:

- *Blocking interface.* It allows a simplified coding style for models that complete a transaction in a single function call, by exploiting the blocking primitive `b_transport(payload, time)`. Timing annotation is performed by exploiting the *time* parameter of the primitive. The blocking interface is suited to implement, for example, the *loosely timed* coding style.
- *Non-blocking interface.* It supports the association of multiple timing points with a single transaction.
  Ir relies on the use of non blocking primitives `nb_transport_fw(payload, time, phase)` and `nb_transport_bw(payload, time, phase)`. Timing annotation is still performed by exploiting the *time* parameter of the primitives while parameter *phase* is exploited for implementing more accurate communication protocols, such as the four phases *approximately timed* coding style.

- *Direct memory interface (DMI) and debug transport interface.* They are specialized interfaces distinct from the transport interface, providing direct access and debug access to an area of memory owned by a target. The DMI and debug transport interfaces each bypass the usual path through the interconnect components used by the transport interface. In particular, DMI is intended to accelerate regular memory transactions in a loosely-timed simulation, whereas the debug transport interface is for debug access without the delays or side-effects associated with regular transactions.

In TLM, communication is generally accomplished by exchanging packets containing data and control values (i.e., payloads), through a channel (e.g., a socket) between an initiator module (master) and a target module (slave).

### B. The EFSM model

An EFSM [13] is a transition system which allows a more compact and intuitive representation of the state space with respect to the traditional finite state machines (FSM). The EFSM model is widely used for modeling complex systems like reactive systems [14], communication protocols [15], buses [16] and controllers driving data-path [17].

*Definition 1:* An EFSM is defined as a 5-tuple $M = \langle S, I, O, D, T \rangle$ where: $S$ is a set of states, $I$ is a set of input symbols, $O$ is a set of output symbols, $D$ is a n-dimensional linear space $D_1 \times \ldots \times D_n$, $T$ is a transition relation such that T : $S \times D \times I \rightarrow S \times D \times O$. A generic point in $D$ is described by a $n$-tuple $x = (x_1, ..., x_n)$; it models the values of the variable (or registers) internal to the design.[1]

A pair $\langle s, x \rangle \in S \times D$ is called a *configuration* of $M$, while an operation on an EFSM $M = \langle S, I, O, D, T \rangle$ is defined as follows:

*Definition 2:* If $M$ is in a configuration $\langle s, x \rangle$ and it receives an input $i \in I$, it moves to the configuration $\langle t, y \rangle$ iff $((s, x, i), (t, y, o)) \in T$ for $o \in O$.

In an EFSM, each transition is associated with a couple of functions (i.e., an *enabling function* and an *update function*) acting on input, output and variable (or register) data. The enabling function expresses a set of conditions on data, while the update function consists of a set of statements performing operations on data.

*Definition 3:* Given an EFSM $M = \langle S, I, O, D, T \rangle$, $s \in S, t \in T, i \in I, o \in O$ and the sets $X = \{x | ((s, x, i), (t, y, o)) \in T \text{ for } y \in D\}$ and $Y = \{y | ((s, x, i), (t, y, o)) \in T \text{ for } x \in X\}$, the *enabling* and *update* functions are defined respectively as:

$$e(x, i) = \begin{cases} 1 & \text{if } x \in X; \\ 0 & \text{otherwise.} \end{cases}$$

$$u(x, i) = \begin{cases} (y, o) & \text{if } e(x, i) = 1 \text{ and} \\ & ((s, x, i), (t, y, o)) \in T; \\ undef. & \text{otherwise.} \end{cases}$$

[1]We consider data values internal to the design to be represented by variables and system-level data types in TLM models, while they are represented by registers and bit-accurate data types in RTL models.

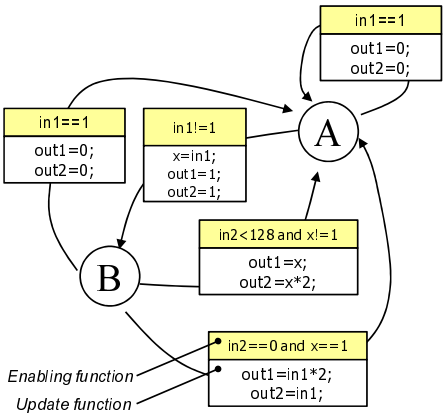


Fig. 2. Example of EFSM

Figure 2 gives an example of the state transition graph (STG) of an EFSM. A transition is fired only if all the conditions in the enabling function are satisfied, bringing the machine from the current state to the destination state and performing the operations included in the update function.

A deeper analysis and explanation on the EFSM model can be found in [13]. In the following sections we will adopt this model to formally represent the communication protocol of both TLM and RTL models.

## III. Methodology

Our approach starts from a TLM IP description and aims at producing a correct-by-construction RTL implementation of the IP. The result is a RTL IP where functionality is equivalent to the TLM functionality and the interface complies with a bus-based communication protocol. In order to achieve this result, our methodology needs to run through a number of different steps, as shown in Figure 3.

We firstly require functionality and communication implementation of the starting TLM IP description to be modularly separated, as described in Section III-A. If the TLM description satisfies such a prerequisite, the synthesis flow is completely automatic. In contrast, the methodology proposes a preliminary (manual) step that we call *normalization* (step 1).

Through normalization, we separate the code implementing the IP functionality from the code implementing the TLM interface. Then, we provide the former to an HLS tool (step 2), in order to obtain the equivalent IP functionality implemented at RTL (Section III-B).

The focus moves then on the interface (and the corresponding communication protocol) part, which is examined in Section III-C. In order to generate a RTL IP description equivalent by construction to the input TLM IP description, an analysis involving a formal model of the communication protocols at both abstraction levels (steps 3 and 4) is necessary. The result is a mapping between TLM and RTL protocols, in terms of transitions and data structures (step 5). Through this

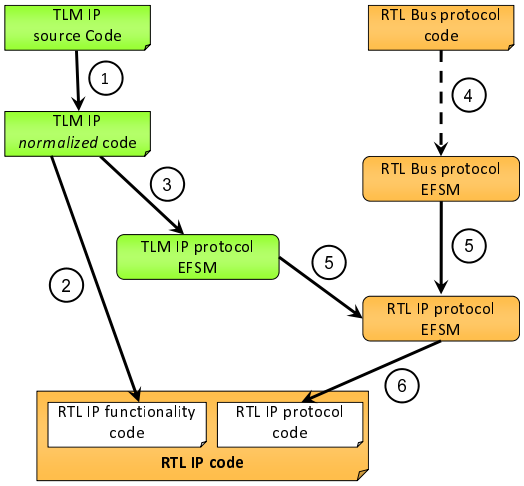


Fig. 3. Methodology overview.

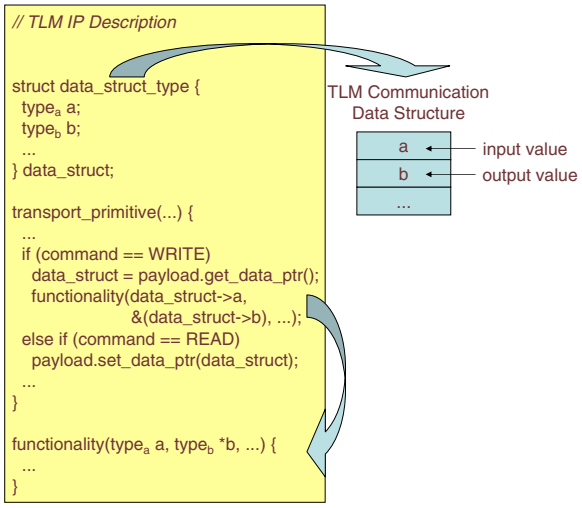


Fig. 4. A normalized TLM IP description.

information, we are able to generate the RTL code for the IP interface (step6). This is achieved by generating a transducer that translates the basic IP interface and the target bus-based interface.

### A. *Normalization (*step 1*)*

Figure 4 shows the outline of a normalized TLM IP description. Firstly, the definition of the data structure employed in the communication phase is provided. Fields of this structure may provide input data to the IP or be filled with output results, so that the IP will be able to access all the data it needs in order to process its functionality. This definition provides details about the size and data type for each field, which will correspond to a port in the RTL IP description.

Then the transport primitive is defined (e.g., `b_transport()` or `nb_transport_fw()` of the TLM-2.0 library), which branches according to the transaction type (i.e., write transaction or read transaction). In case of a write transaction, the data structure is extracted from the payload and a procedure is invoked. This procedure wraps all the IP functionality details, and executes the elaboration phase triggered after a transport primitive has been called. The signature of this procedure will contain a number of parameters, one for each field of the data structure employed in the communication phase. Otherwise, in case of a read transaction, the data structure containing the output data is stored into the payload, in order to provide results back to the initiator.

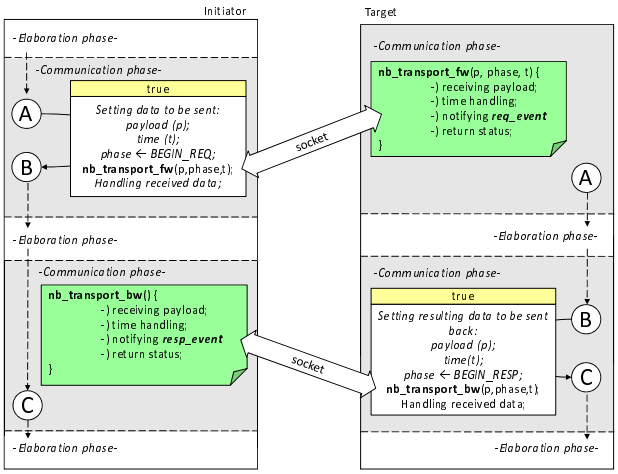


Fig. 5. EFSM of a TLM communication protocol based on the approximately-timed coding style.

As shown in Figure 4, the definition of communication procedures (e.g., `b_transport()` or `nb_transport_fw()`, etc.) are separated from the definition of the IP functionality procedure. In particular, the IP functionality procedure may be a C++ procedure or a procedure associated to a SystemC process (e.g., `SC_METHOD`, `SC_THREAD`).

### B. *Functionality Synthesis (*step 2*)*

As far as functionality is concerned, our approach relies on a HLS tool to generate RTL that implements the TLM IP functionality. This activity is greatly helped by the normalization process, which isolates the C++ code that has to be provided to the synthesis tool as input.

The RTL code produced as output will feature an interface having a port for each field of the data structure employed in the communication phase. This assumption allows us to successfully recombine functionality and communication in the RTL IP description, ensuring that these two parts properly interact together after being separated in our methodology flow.

### C. *Protocol Synthesis (steps 3 - 6)*

In order to produce a RTL communication protocol which is equivalent by construction to the TLM one, we need to describe these protocols with a formal model and find an association between them.

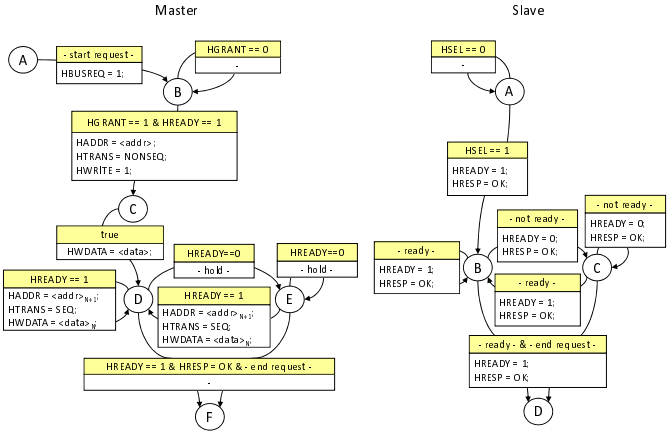


Fig. 6. EFSM of a AHB burst write transfer.

*1) EFSM Models of TLM Protocols (*step 3*):* All the TLM protocols can be modeled by means of EFSMs [18], [19], by composing the EFSM models representing each single TLM primitive. We show, for example, the EFSM representation of a TLM communication protocol based on the approximately-timed coding style (see Section II-A) as it is the most meaningful and complete example among all the TLM coding styles (see Figure 5).

The initiator starts the transaction by setting the payload fields. The protocol phase is set to `BEGIN_REQ` to indicate the beginning of a request. The transport primitive `nb_transport_fw()` is called through the initiator socket. The target implements this primitive by receiving payload, taking care of timing details and notifying an event. This event will activate a process which is responsible for the elaboration phase in the target, in order to perform the operation requested by the initiator. When this elaboration phase is completed, the target updates payload fields in order to provide results (if any) and enters the phase `BEGIN_RESP` to indicate the beginning of a response. It then calls the non-blocking transport primitive along the backward path (i.e., `nb_transport_bw()`). The initiator implements this primitive usually by receiving payload, handling timing details and notifying an event. Once again, this event will activate a process which allows the initiator to retrieve possible results sent by the target and to successfully complete the transaction.

In the next sections, we show how such EFSM models can be matched with corresponding EFSM model of RTL protocols. The matching result allows us to translate TLM transactions into equivalent RTL transactions.

*2) EFSM Models of RTL Bus Protocols (*step 4*):* The EFSM model can be automatically extracted from an HDL description, and all the RTL communication protocols can be modeled by means of EFSMs [13], [20]. As an example of EFSM model of a RTL bus-based protocol, we show the AMBA AHB [21] representation, as it is the most widely used and representative RTL bus-based protocol.

In the AHB protocol, a bus master starts a transfer (i.e., write or read) by firstly setting the `HBUSREQ` signal to 1 (see Figure 6). Once the bus arbiter grants his request, the master initiates the transfer by driving the address and control signals. These signals specify the address, the transfer type (i.e., write or read), and whether the transfer is part of a burst. Each transfer consists of an address and control cycle and one or more cycles for data to be written or read. The slave indicates whether it is ready to sample or provide data by driving the `HREADY` signal, which allows the introduction of wait states in the transfer. The status of the transfer being carried on is represented by the target through the `HRESP` signal. In conjunction with the `HREADY` signal, `HRESP` indicates the outcome of the transfer to the master.

Figure 6 shows the EFSM model of a complete burst write transfer.

*3) Mapping between TLM and RTL Transactions (*step 5*):* A TLM transaction is an abstraction of a data transfer between two design modules [8]. Write and read operations requested by an initiator and performed by a target are examples of transactions. These transactions are usually represented in the TLM description by a generic payload object, which is exchanged between modules through primitive calls and socket connections. A transaction consists of one or more transport primitive calls according to the adopted TLM interface. Since the TLM generic payload specifically aims at modeling memory-mapped buses, our approach relies on mapping a TLM transaction into RTL transactions.

In this context, a RTL transaction is a read or a write operation of a data object performed on the bus data signal. In general, a transaction may be part of a *burst*, which is a sequence of *bus transfers* to or from a contiguous region of address space. Thus, we propose a method to map a TLM transaction into one or more bus transfers.

Firstly, since data transferred in a TLM transaction is variable while RTL bus transfer rely on a fixed-length data bus, we distinguish two cases, according to the payload data length. If such value is not greater than the bus width, we map the TLM transaction into a single bus transfer. Otherwise, we decompose data into blocks having the same length as the bus width, therefore splitting a TLM transaction into a number of corresponding bus transfers. If the adopted bus allows burst transfers, the bus transfers corresponding to a TLM transaction will be enclosed in a burst.

Some mapping optimization is also possible to reduce the number of bus transfers. For example, the data structure included in the TLM payload generally contains fields for both input values (to provide to the target) and output values (to be retrieved from the target). In case of write transaction, only the input fields of the data structure are required, since no output will be sent back from the target. In contrast, in case of a read transaction, only the output fields of the data structure are required. As a consequence, we focus only on the length of the required subset of data structure fields instead of considering the total payload data length. This optimization ensures that bus transfers are not wasted, even if it introduces a degree of complexity since we have to deal with read and

| TLM | AHB | APB | STBus | Notes |
|---|---|---|---|---|
| command | HWRITE | PWRITE | OPC[3:0] | *TLM_IGNORE_COMMAND* has no bus counterparts |
| address | HADDR | PADDR | ADD | Only 32-bits supported |
| response | HRESP | - | R_OPC[0] | Only OK or generic error response |

TABLE I
MAPPING OF TLM GENERIC PAYLOAD BASIC CONTROL FIELDS.

write operations in a different way.

Then, since a transaction consists of control as well as data information, we consider also the payload control information to complete the mapping analysis. We associate the TLM generic payload control fields to the corresponding RTL buses signals. The attributes required for a basic bus transfer, in any standard bus-based RTL protocol are (i) command, (ii) address, and (iii) a response attribute to indicate the status of the ongoing transfer.

Table I shows some examples of mapping between the basic control fields of the TLM payload and the bus control signals of the most common RTL buses, such as, AMBA AHB and APB by ARM and STBus by STMicroelectronics. However, it is worth noting that the methodology is not dependent or limited to these buses.

*4) Transducer Generation (step 6):* In order to translate the basic interface of the RTL IP generated by the HLS tool and the more advanced bus-based interface generated in the earlier step, we extend the RTL IP with a transducer module. The transducer features a bus slave interface on one side and an IP basic interface on the other. Its main process will be responsible for receiving requests from an initiator through the bus, translating them to the IP and providing results from the IP to the initiator, once again through the bus.

In case of write transactions, the transducer will sample data from the write data bus and drive these values on the corresponding RTL IP input ports. In case of read transactions, the transducer will read the output ports of the RTL IP and drive these values on the read data bus. The behavior of the transducer in both cases is determined by the mapping between TLM transactions and RTL bus transfers we have proposed earlier.

Exploiting the EFSMs representing the RTL IP and TLM IP protocols, the methodology extracts (i) information concerning the RTL bus transfers, and (ii) the data structure employed in TLM communication, indicating whether a field is used as input or output (or both) value. Through these additional information, the transducer can be automatically generated.

## IV. EXPERIMENTAL RESULTS

The methodology presented in this paper has been implemented in *T2R*, a prototype tool built on the top of HIF-Suite [22]. It automatically synthesizes TLM IP descriptions into RTL implementations, where the TLM IP interfaces are compliant with the standard OSCI TLM-2.0 library. *T2R* relies on CatapultC by Mentor Graphics for the HLS synthesis step (see Section III-B).

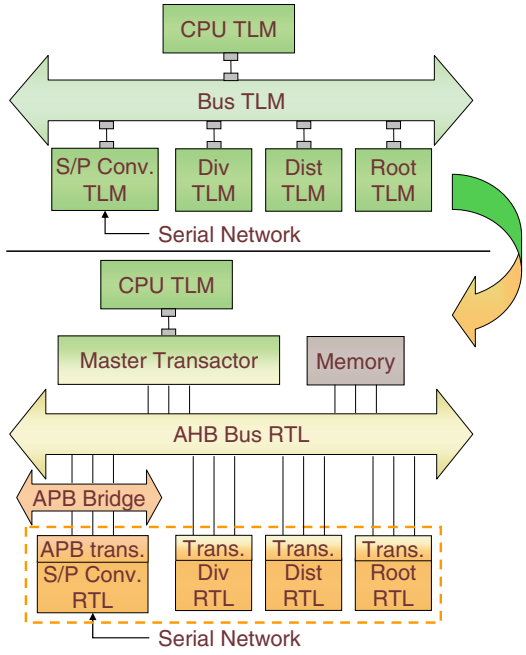


Fig. 7. Experimental results environment.

Experimental results have been conducted by applying *T2R* into the TLM-to-RTL design flow of the *Face Recognition System* platform [23] provided by STMicroelectronics. In particular, four modules have been considered for the synthesis: the first three modules (i.e., *Root*, *Dist* and *Div*) deal with pixel elaboration and analysis, while a forth module (*S/P conv.*) is a serial/parallel converter that receives data from the network and sends it to the system.

Figure 7 illustrates the experimental results environment. For the sake of clarity, only the primary initiator (i.e., the CPU) and the synthesized four modules of the face recognition system have been reported in Figure.

In a first system-level implementation of the platform, all these modules have been implemented at TLM. Then, they have been synthesized and connected to the target bus of the RTL implementation. The target bus at RTL is a hierarchical composition of AMBA AHB and AMBA APB.

Table II reports the experimental results. Column *TLM if* indicates the TLM-2.0 coding style for each input design considered. As we can see, the starting TLM platform is based on the approximately-timed coding style, except for the serial/parallel converter, which is modeled according to the loosely-timed coding style. Its interaction with the other

| IP | TLM if | TLM fun loc | TLM if loc | TLM tot loc | # data param | Bus if | Transaction mapping | RTL fun loc | RTL td loc | RTL tot loc |
|---|---|---|---|---|---|---|---|---|---|---|
| dist | AT4 | 38 | 83 | 121 | 3 (in)<br>1 (out) | AHB | 1 - 3 (W)<br>1 - 1 (R) | 110 | 91 | 201 |
| div | AT4 | 25 | 79 | 104 | 2 (in)<br>1 (out) | AHB | 1 - 2 (W)<br>1 - 1 (R) | 83 | 76 | 159 |
| root | AT4 | 5 | 94 | 99 | 1 (in)<br>1 (out) | AHB | 1 - 1 (W)<br>1 - 1 (R) | 307 | 77 | 384 |
| sp_conv | LT | 9 | 31 | 40 | 1 (in)<br>2 (out) | APB | 1 - 1 (W)<br>1 - 2 (R) | 52 | 68 | 120 |

TABLE II
EXPERIMENTAL RESULTS.

modules is achieved through a *simple_target_socket*, which takes care of converting calls to transport primitives from a transport interface to the other. Column *TLM fun* reports the size of the functionality part of the starting TLM descriptions in terms of line of code. Column *TLM if* refers to the interface side of the starting TLM descriptions, while *TLM tot* provides the total number of line of code. Column *# data param* shows the number of fields in the data structure employed in the TLM communication phase, separating between input and output values. Column *Bus if* indicates the adopted RTL bus interface for each module. Column *Transaction mapping* shows the mapping between a TLM transaction and the corresponding bus transfer(s), breaking down between write and read operations. Columns *RTL fun* and *RTL td* report respectively the line of code of the RTL description of the IP functionality and the automatically generated transducer. Column *RTL tot* reports the total number of line of code of the generated RTL implementations.

Translating each TLM module into the equivalent RTL implementation by using *T2R*, took up just a few minutes of work, thanks to the high degree of automation provided by the methodology.

On the other hand, a couple of work days have been spent for manually synthesizing each module, with an increasing chance of introducing design errors that may ultimately lead to the RTL design not being equivalent to the input TLM description.

## V. CONCLUDING REMARKS

The paper addressed the problem of synthesis of TLM IP models into RTL implementations. We proposed a methodology to automate the synthesis process, assuming that the TLM IP interface be compliant with the standard OSCI TLM-2.0 library. After a manual preliminary step, the methodology applies the theory of HLS to TLM and implements a protocol synthesis technique based on the extended finite state machine (EFSM) model for generating the RTL IP interface compliant with any RTL bus-based protocol. The methodology effectiveness and correctness have been shown by synthesizing different TLM IPs into equivalent bus-based RTL implementations of an industrial platform.

Several extensions of this work are part of our current and future work. These include handling of TLM event queues, pipelined transactions, and functional error handling.

## ACKNOWLEDGMENTS

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

## REFERENCES

[1] B. Bailey, G. Martin, and A. Piziali. *ESL Design and Verification: A Prescription for Electronic System Level Methodology*. Morgan Kaufman Publishers, 2007.

[2] L. Cai and D. Gajski. *Transaction Level Modeling: An Overview*. In *ACM/IEEE CODES+ISSS*, pp. 19–24. 2003.

[3] F. Ghenassia, A. Clouard, K. Jain, L. Maillet-Contoz, and J.-P. Strassen. *Using Transactional Level Models in a SoC Design Flow*. Kluwer Academic Publishers, 2003.

[4] A. Donlin. *Transaction Level Modeling: Flows and Use Models*. In *Proc. of ACM/IEEE CODES + ISSS*, pp. 75–80. 2004.

[5] T. Kogel, A. Haverinen, and J. Aldis. *OCP TLM for Architectural Modeling*, 2005. OCP methodology guideline, http://www.ocpip.org.

[6] N. Bombieri, F. Fummi, and G. Pravadelli. *Reuse and Optimization of Testbenches and Properties in a TLM-to-RTL Design Flow*. ACM TODAES, vol. 47(13), 2008.

[7] OSCI. 2009. Http://www.systemc.org.

[8] TLM-2.0. *OSCI TLM-2.0 Language Reference Manual*. Open SystemC Organization Initiative, 2009. http://www.systemc.org.

[9] P. Coussy, D. D. Gajski, M. Meredith, and A. Takach. *An Introduction to High-Level Synthesis*. IEEE Design and Test of Computer, vol. 13(24):pp. 8–17, 2009.

[10] P. Coussy and A. Morawiec. *High-Level Synthesis: From Algorithm to Digital Circuit*. Springer, 2008.

[11] Forte Design Systems. *Forte Cynthesizer*. http://www.forteds.com/products/index.asp.

[12] S. A. Edwards and O. Tardieu. *SHIM: a deterministic model for heterogeneous embedded systems*. In *Proc. of ACM EMSOFT*, pp. 264–272. 2005.

[13] K.-T. Cheng and A. Krishnakumar. *Automatic Generation of Functional Vectors Using the Extended Finite State Machine Model*. ACM TODAES, vol. 1(1):pp. 57–79, 1996.

[14] T. J. Koo, B. Sinopoli, A. Sangiovanni-Vincentelli, and S. Sastry. *A Formal Approach to Reactive System Design: Unmanned Aerial Vehicle Flight Management System Design Example*. In *Proc. of IEEE CACSD*, pp. 522–527. 1999.

[15] H. Katagiri, K. Yasumoto, A. Kitajima, T. Higashino, and K. Taniguchi. *Hardware implementation of communication protocols modeled by concurrent EFSMs with multi-way synchronization*. In *Proc. of ACM/IEEE DAC*, pp. 762–767. 2000.

[16] A. Zitouni, S. Badrouchi, and R. Tourki. *Communication Architecture Synthesis for Multi-bus SoC*. Journal of Computer Science, vol. 2(1):pp. 63–71, 2006.

[17] A. Guerrouat and H. Richter. *A component-based specification approach for embedded systems using FDTs*. ACM SIGSOFT Softw. Eng. Notes, vol. 31(2):pp. 14–18, 2006.

[18] N. Bombieri, F. Fummi, and G. Pravadelli. *A Mutation Model for the SystemC TLM 2.0 Communication Interfaces*. In *Proc. of ACM/IEEE DATE*, pp. 396–401. 2008.

[19] N. Bombieri, F. Fummi, and G. Pravadelli. *On the Mutation Analysis of SystemC TLM-2.0 Standard*. In *IEEE MTV*, pp. 1–6. 2010.

[20] G. Di Guglielmo, F. Fummi, C. Marconcini, and G. Pravadelli. *EFSM Manipulation to Increase High-Level ATPG Efficiency*. In *Proc. of ACM/IEEE ISQED*. 2006.
[21] ARM. *AMBA Specification 2.0*, 1999. http://www.arm.com.
[22] http://hifsuite.edalab.it.
[23] M. Borgatti, A. Capello, U. Rossi, G.L.Lambert, I. Moussa, F. Fummi, and G. Pravadelli. *An Integrated Design and Verification Methodology for Reconfigurable Multimedia System*. In *Proc. of ACM/IEEE DATE*, pp. 266–271. 2005.