Synthesizing TLM-2.0 Communication Interfaces

Synthesizing TLM-2.0 Communication Interfaces

Nadereh HATAMI Paolo PRINETTO

Politecnico di Torino - Dipartimento di Automatica e Informatica, Torino, Italy

{Nadereh.Hatami, Paolo.Prinetto}@polito.it

Abstract

TLM-2.0 consists of a set of core interfaces containing blocking and non-blocking transport interfaces and the direct memory interface. These interfaces facilitate communication between initiators, targets and interconnect modules. Although TLM models simulate faster, there is a long way from a high abstraction TLM model to a synthesizable RTL description. This paper proposes algorithms to facilitate synthesizing TLM-2.0 interfaces. We introduce an intermediate layer in abstraction level that is lower than TLM-2.0 and higher than RTL. To describe this intermediate layer, we take advantage of TLM-1.0 constructs. Algorithms for mapping high level TLM-2.0 to the intermediate level are proposed to add necessary details for RTL synthesis to the synthesized designs in this level. We have focused our attention on Direct Memory Interface and left the rest as future work. The experimental results show that automation caused by the proposed rules improves simulation speed and reduces modeling effort.

1. Introduction

Transaction Level Modeling (TLM) approaches have been proposed to describe Systems-On-Chips (SoCs) at an abstraction level higher than RTL. [2, 3, 4] describe TLM as a high level transaction language based on SystemC. TLM simulates faster than RTL, even for complex systems, and allows embedded software validation and integration testing to be done earlier in the design cycle, i.e., before RTL being complete.

TLM modules consist of two distinct parts: *TLM core modules* that provide us with the functionality of the individual components of the SoC, and *TLM interfaces* that are the means of communication between IP cores. Interfaces transfer data in the form of transactions through module ports.

As design proceeds in the design flow, the abstract model should be replaced by a more detailed architecture of what is really going to be done. This detailed architecture can then be synthesized to RTL.

Some works have been done on TLM synthesis to extract RTL model of the design from the transaction level description. [4] is a high-level solution that integrates electronic system level designs with block-level implementation. [5] also uses a library of synthesizable TLM protocols to synthesize transaction level descriptions into SystemC RTL code. Both of these tools use TLM-1.0 as input code and have some limitations in their synthesis approach.

In this paper, we focus on interfaces as communication tools which transfer transactions through sockets. We start from TLM-2.0 high level models which provide us with four types of interfaces and concentrate on TLM Direct Memory Interface (DMI). We propose a novel approach to purify this interface to lower levels of abstraction using channels. We introduce an intermediate level based on TLM-1.0 standards to map high level TLM-2.0 designs to a lower level of abstraction to take advantage of available tools.

In the sequel of the paper, after an overview of TLM-2.0 as the recent standard of TL modeling in SystemC, we describe our methodology for TLM synthesis in Section 3. In Section 4, we explain the experimental results by considering the coding styles of high level and synthesizable designs. Section 5 draws some conclusions.

2. An overview of TLM-2.0

As the abstraction level goes higher, the current OSCI TLM-1.0 standard becomes less applicable and not fast enough for describing high level designs [9]. In addition, with the lack of model interoperability in the TLM-1.0 standard, producing IP cores to be used by a common customer becomes another bottleneck.

TLM-2.0 is the new OSCI standard, enabling SystemC model interoperability and reuse, and providing an essential framework for architecture analysis, software development, performance analysis, and hardware verification [8].

TLM-2.0 takes the approach of distinguishing between interfaces (APIs) on the one hand, and coding styles on the other one. The TLM-2.0 standard defines a set of interfaces which should be thought of as low-level software programming mechanisms for implementing transaction level models. This standard also describes a number of coding styles that can be used for various cases.

TLM-2.0 consists of core interfaces from TLM-1.0 together with the blocking and non-blocking transport interfaces, the direct memory interface (DMI), the debug transaction interface, the write interface, and the analysis interface [1]. Each interface is suitable to work with one of the three TLM-2.0 coding styles each providing a design with different level of details. A fast, loosely-timed model is typically expected to use the blocking transport interface and the direct memory interface. A more accurate, approximately-timed model is

typically expected to use the non-blocking transport interface and the payload event queues [1].

The transport interfaces are the primary interfaces used to transport transactions between initiators, targets and their interconnection components [1]. They are clustered into blocking and non-blocking categories, respectively.

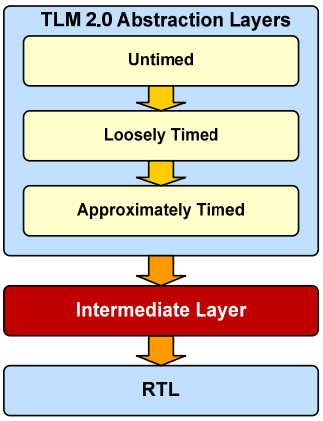

Figure 1.TLM-2.0 to RTL synthesis steps

The application of the non-blocking transport interface is in cases when one needs modeling the detailed sequence of interactions between initiator and target during the course of each transaction [1].

The direct memory interface is able to bypass the normal path of multiple transport interface calls from initiator through interconnect components to target; it thus offers a large potential increase in simulation speed for memory access between initiator and target.

All together, TLM-2.0 provides a very high abstraction level description of hardware components.

3. TLM-2.0 Interface Synthesis approach

TLM-2.0 describes hardware architectures in a very abstract manner, but to be synthesized to silicon, each hardware description should be described in RTL. TLM-2.0 provides four distinct interfaces to grant communication for any possible architecture while preserving interoperability. Interfaces contain virtual functions to support data transaction between IP cores without concerning about how this communication is going to be implemented in lower levels. Coming down the abstraction levels, we have to get involved to implementations rather than declarations. Any TLM-2.0 interface should be mapped to a well defined channel which implements the communication details of the transaction. In other words, as channels include the interface's function implementation and provide us with more accuracy at the expense of an increase in complexity and simulation time, we can see them as the next step of refining TLM-2.0 designs. This approach is shown in Figure 1.

The main idea of this paper is to provide an *Intermediate Layer* which contains the necessary details needed to map a design to RTL. [1] presents TLM-1.0 as an abstraction level lower than TLM-2.0. It benefits from channels to transfer data between and in main cores through ports and exports. Tools are available to map TLM-1.0 descriptions to lower level HDLs such as VHDL or synthesizable SystemC [4, 5]. So it can be seen as an intermediate level between RTL and TLM-2.0 that can be used to synthesize TLM-2.0 designs.

In this paper, we focus on DMI, and introduce algorithms to map this interface to lower level TLM-1.0 structures.

3.1. Direct memory interface

Using Direct Memory Interface, or DMI, an initiator can get direct access to an area of memory owned by a target. In this case, accessing that memory can be done using a direct pointer rather than through the transport interface.

DMI uses both forward and backward paths in which the first is from initiator to target, used to request a read, write or both accesses to a given address specified by an initiator, and returns a reference to a DMI descriptor of type *tlm_dmi*, which contains the bounds of the DMI region. The target would use the backward path to invalidate the DMI pointer.

Getting a step down to the intermediate level, we can see DMI as a DMA channel which requests a bus and transfers data through the granted bus. Figure 6 shows this mapping process. A channel with the functionality of the classic DMA would be inserted between the initiator and interconnect or the target module.

The custom DMA channel has the responsibility of direct reading and writing from and to the target, while bypassing all interconnects. It also has a port to communicate with interconnect to get the bus access. As other modules may read or write to the target through transport interface, an arbiter is also required to control which device is granted the bus and has access to target. We prove that the designed channel is faster in simulation in comparison with the request-response channel and can suite in all the cases that DMI interface can be used.

3.1.1. DMA channel implementation

To implement the DMA channel, we take advantage of TLM-1.0 hierarchical channels which are inherited from *sc_module*. The described DMA custom hierarchical channel has two exports to send and receive data from device connected to it. It also has an embedded arbiter to arbitrate between the DMA channel and the module immediately connected to the target component. In addition, it has another two exports to communicate with the interconnect modules or the processor. The exports of the DMA channel are demonstrated in Table 1.

Table 1 DMA Channel input and output ports and exports

Type	Name
sc_export	input_port
sc_export	output_port
sc_port	arbiter_input
sc_port	arbiter_output
sc_export	dma_to_interconnect_fw
sc_export	dma_to_interconnect_bw

The arbiter_output port connects the embedded arbiter in DMA channel to the target component. The arbiter_input port, connects interconnect immediately connected to the target component to the embedded arbiter.

The DMA channel receives data from the device connected to it. When the device requests a “read” or “write” from memory, the DMA sends a bus request to the arbiter.

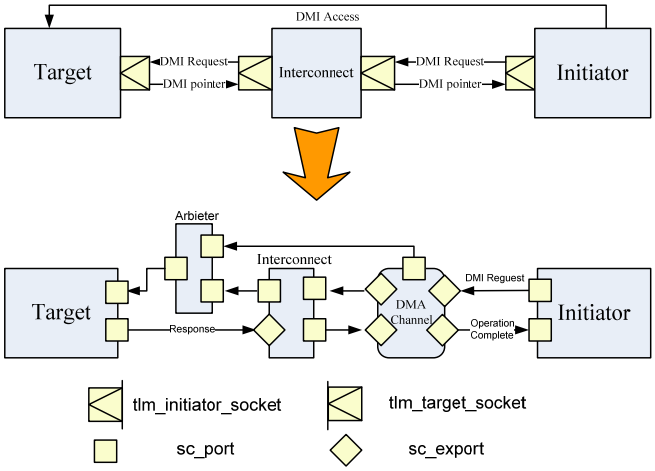

Figure 2. Direct Memory Interface mapping approach

3.1.2. DMI mapping algorithm

The DMI mapping algorithm should be able to recognize three different components: initiator, target, and the component immediately connected to the target. The initial step of the mapping algorithm that implements the interconnections of Figure 2 is shown in Figure 3.

```
1  int k = 0;   int l = 0;
2  for (int i = 0; i < LUT.number_of_rows; i++)
3  if (LUT[i].DMI) {
4    RLUT[K].port = Concat(LUT[i].Socket, "_fw");
5    RLUT[k+1].port = Concat(LUT[i].Socket, "_bw");
6    if (!LUT[l].type) { //initiator socket
7      RLUT[k].interface = "tlm_blocking_put_if";
8      RLUT[k+1].interface = "tlm_blocking_get_if";
9      if (component is an initiator component)    { Step 1 }
10     else if (component is connected to the target) { Step 2 }
11     else { Step 3 }
12     }
13     else  { //target port
14        if (component is a target component)  { Step 4 }
15        else  { Step 5 }
16        k += 2;
17     }
18 }
```

Figure 3. Initial Step in mapping DMI interface to DMA channel

An initiator with an initiator socket (Step 1): As shown in Figure 4, the initiator should be bound to the DMA channel to request a direct memory access.

```
1  if (component is an initiator component )   {
2    define a DMA channel (dma_ch);
3    for (int j=0; j<binding.length(); j++)
4     if (binding[j].initiator == LUT[i].socket)   {
5       Rbinding[l].port0 = dma_ch.input;
6       Rbinding[l].port1 = RLUT[k].port;
7       Rbinding[l+1].port0 = dma_ch. dma_to_interconnect_fw;
8       Rbinding[l+1].port1 = Concat(binding[j].target, "_fw");
9       Rbinding[l+2].port0 = dma_ch. dma_to_interconnect_bw;
10      Rbinding[l+2].port1 = Concat(binding[j].target, "_bw");
11      Rbinding[l+3].port0 = dma_ch.output_port;
12      Rbinding[l+3].port1 = RLUT[k+1].port;
13      l+= 4;
14    }
15 }
```

Figure 4. Step 1

The component is directly connected to target component and the socket is an initiator socket (Step 2): This component is usually a processor which has direct access to the memory. In this case, in lower levels of abstraction, the component should be connected to the arbiter rather than the target to be able to grant the bus to the DMA when needed. The implementation of this process is shown in Figure 5.

```
1  if (component is connected to the target component) {
2    define a tlm_req_rsp_channels (req_rsp_ch);
3    for (int j=0; j<binding.length(); j++)
4      if (binding[j].initiator == LUT[i].socket) {
5          Rbinding[l].port0 = dma_ch.arbiter_input_port;
6          Rbinding[l].port1 = RLUT[k].port;
7          Rbinding[l+1].port0 = tlm_req_rsp_cha.put_rsp_export;
8          Rbinding[l+1].port1 = Concat(binding[j].target, "_bw");
9          Rbinding[l+2].port0 = tlm_req_rsp_ch.get_rsp_export;
10         Rbinding[l+2].port1 = RLUT[k+1].port;
11         l += 3;
12       }
13 }
```

Figure 5. Step 2

Any other component with an initiator socket (Step 3): In this case, the path is the normal transport path which uses request-response channels to communicate. The algorithm is shown in Figure 6.

```
1  else { // components rather than the initiator and those
           //directly connected to the target
2    define a tlm_req_rsp_channels (req_rsp_ch);
3    for (int j=0; j<binding.length(); j++)
4      if (binding[j].initiator == LUT[i].socket) {
5          Rbinding[l].port0 = tlm_req_rsp_ch.put_req_export;
6          Rbinding[l].port1 = RLUT[k].port;
7          Rbinding[l+1].port0 = tlm_req_rsp_ch.get_req_export;
8          Rbinding[l+1].port1 = Concat(binding[j].target, "_fw");
9          Rbinding[l+2].port0 = tlm_req_rsp_ch.put_rsp_export;
10         Rbinding[l+2].port1 = Concat(binding[j].target, "_bw");
11         Rbinding[l+3].port0 = tlm_req_rsp_ch.get_rsp_export;
12         Rbinding[l+3].port1 = RLUT[k+1].port;
13         l+= 4;
14     }
15 }
```

Figure 6. Step 3

The component is a target component and the socket is a target socket (Step 4): In this case, the component is the end target responsible for the reads or writes. So, it should be bound to the arbiter output to get the required information for the requested action from the processor or the DMA. This is implemented in Figure 7.

```
1  if (component is a target component)
2  {
3      Rbinding[l].port0 = dma_channel.arbiter_output;
4      Rbinding[l].port1 = concat (LUT[i].Socket, "_fw");
5      l++;
```

```
6 }
```

Figure 7. Step 4

Any other component with target socket (Step 5): In this case, the components are interconnects which communicate with their adjacent components via transport channels. This is shown in Figure 8.

```
1 else { //Components except the target component
2 RLUT[k].interface = "tlm_get_if";
3 RLUT[k+1].interface = "tlm_put_if";
4 find the socket's functionality in nb_transport_fw function
5 define a function with the name conat("run_",LUT[i].Socket,
    "_fw") and put the gained functionality there
6 find the socket's functionality in nb_transport_bw function
7 define a function with the name conat("run_",LUT[i].Socket,
    "_bw") and put the gained functionality there
8 }
```

Figure 8. Step 5

4. Case Study

To examining our translations of DMI, test case shown in Figure 9 is used which uses a DMI between an I/O device and a memory connected to a processor. The DMI shown is translated to TLM-2.0 channels using our proposed procedures.

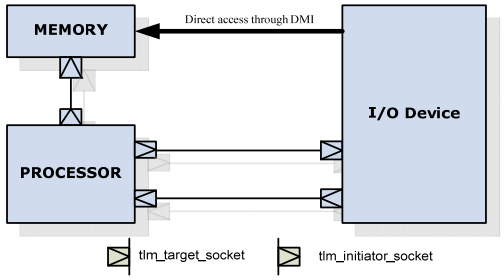

Figure 9. Using DMI interface to model hardware designs

5. Experimental Results

We evaluated our proposed approach for two criteria: modeling effort and simulation speed. To do the evaluation, we first implemented the architecture described in Section 4 in TLM-2.0 loosely timed coding style and then synthesized the design's interfaces using the algorithm proposed in Section 3. The modeling effort evaluated by the number of lines of code is shown in Table 2.

Table 2. Measuring modeling effort with lines of code

	Original Model(lines)		Synthesized Model (lines)		Modeling Effort (%)	
	.h	.cpp	.h	.cpp	.h	.cpp
Dma	*49*	*94*	*53*	*131*	*8*	*28*
Read processor	*29*	*91*	*36*	*95*	*19*	*4*
Write Processor	*32*	*48*	*34*	*55*	*6*	*13*
Memory	*41*	*55*	*46*	*118*	*11*	*53*
CPU	*42*	*101*	*45*	*119*	*7*	*15*

As shown in Table 2, with the increase in the number of initiator and target sockets and also in the complexity of functionality, the proposed method reasonably reduces the modeling effort. As it can easily be calculated, the overall modeling effort reduction is 21% for the header files and 38% for source files, which will decrease the time to market of the product for large designs. Therefore, automating the mapping process using the proposed algorithm will provide us with gain of 30%.

Using the DMI will speed up the design simulation. Figure 10 shows that with the increase in the number of routed packets, the DMI is much faster than the DMA channel. This implies that we can use our DMA channel synthesis rules to save simulation time without having to manually perform the translation process.

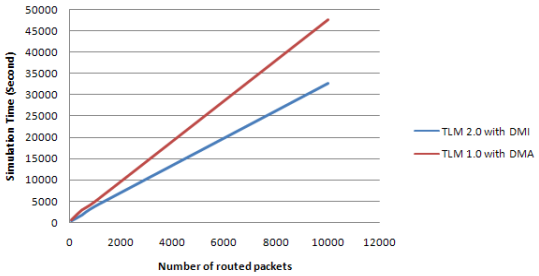

Figure 10. Simulation time comparison of DMI versus DMA channel

6. Conclusion and Future Works

In this paper, we have presented a method to synthesize the loosely-timed interfaces of the TLM-2.0 standard by defining an intermediate layer which contains necessary details for RTL synthesis. TLM-1.0 components were chosen to describe the intermediate layer. We proposed algorithms to map the interfaces in TLM-2.0 standard to TLM-1.0 channels and experimentally verify the correctness of the algorithms. In particular, the mapped TLM-2.0 interfaces on TLM-1.0 channels were functionally equivalent.

The proposed method simplifies the transition from transaction level to lower level implementations and speeds up the whole process. Further research activities are aimed at introducing methods to synthesize the intermediate level to RTL. Applying the proposed approach to other parts of the TLM-2.0 components rather than interfaces is an additional goal of our future works.

7. References

[1] M. Montoreano. "Transaction Level Modeling using OSCI TLM 2.0", Technical report, Open SystemC Initiative, 31 May 2007.

[2] F. Ghenassia, **Transaction Level Modeling with SystemC**. Springer, Dordrecht, Netherlands, 2005.

[3] J. Cornet, F. Maraninchi, L. M. Contoz, "A Method for the Efficient Development of Timed and Untimed Transaction-Level Models of Systems-on-Chip", IEEE Design & Test in Europe 2008 (DATE 2008), pp. 9-14.

[4] http://www.forteds.com/products/tlmsynthesis.asp, October 26, 2008.

[5] http://utcadlab.net/Projects.aspx, October 26, 2008.

[6] Z. Navabi, **VHDL: Analysis and Modeling of Digital Systems**, Second Edition, McGraw-Hill, 1998.

[7] SystemC 2.0.1 Language Reference Manual Revision 1.0, Open SystemC Initiative, San Jose, California, 2003.

[8] Open SystemC Initiative, http://www.systemC.org, November 16, 2008.

[9] M. Montoreano, "Transaction Level Modeling using OSCI TLM 2.0", Synopsys, Inc. May 31, 2007.

[10] A. Rose, S. Swan, J. Pierce, and J. Fernandez, "Transaction Level Modeling in SystemC", OSCI TLM Working Group, 2004.

[11] D. C. Black, J. Donovan, B. Bunton, **SystemC from the Ground Up**, Kluwer Academic Publishers, 2004.