Automatic SystemC TLM Generation for Custom Communication Platforms

Lucky Lo Chi Yu Lo

Center for Embedded Computer Systems
UC Irvine, CA 92697
lochi.yu@uci.edu

Samar Abdi

Center for Embedded Computer Systems
UC Irvine, CA 92697
sabdi@cecs.uci.edu

Abstract

This paper presents a tool for automatic generation of transaction level models (TLMs) in SystemC for MPSoC designs with custom communication platforms. The MPSoC platform is captured as a graphical net-list of components, busses and bridge elements. The application is captured as C processes mapped to the platform components. Once the platform is decided, a set of transaction level communication APIs is automatically generated for each application C process. After the C code is input, an executable SystemC TLM of the design is automatically generated using our tool. This TLM can be executed using standard SystemC simulators for early functional verification of the design. Although, several TLM styles and standards have been proposed in the past, our approach differs in the fact that the designers do not need to understand the underlying SystemC code or TLM modeling style to verify that their application executes on the selected platform. Another key advantage of our tool is that the platform can be easily customized for the application and a new TLM for that platform can be automatically generated. The TLM can be used to program the custom platform early in the design cycle before the components are available. Our experimental results demonstrate that for large industrial applications such as MP3 decoder and H.264, high-speed TLMs can be generated for several platforms in a few seconds.

1 Introduction

The rise in complexity, size and heterogeneity of modern embedded system designs has pushed modeling to new abstraction levels above RTL. Transaction level modeling using SystemC is emerging as a new paradigm for system modeling. On the other hand, platform based design [11] of multi processor SoCs (MPSoC) is being adapted to combine the best features of top down and bottom up system design. Although several SystemC modeling styles for MPSoC have been proposed, no clear semantics for model-

ing objects and composition rules have emerged yet. This makes automatic TLM generation difficult. Most surveys point to usage of transaction level models for early system verification and embedded SW development. Therefore, SW developers who use TLM have to understand TL modeling and SystemC semantics. In this paper, we propose a system development framework and TLM generation tool that removes the need for SW developers to understand either the platform communication architecture or to learn new modeling languages like SystemC.
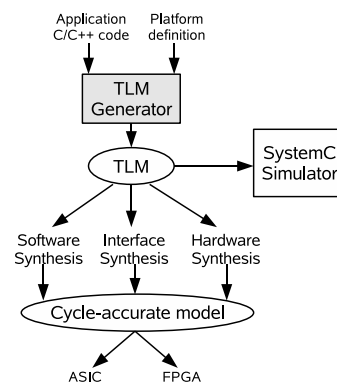

Figure 1. Design Flow

The complete design flow for our tool is shown in Figure 1. The inputs for our tool are the application C code and the platform definition. The output is a TLM from which the software, hardware and interfaces will be synthesized to construct a Pin Cycle Accurate Model to implement in a FPGA or ASIC. This paper will be centered around the TLM generator.

The input platform to the generator is a high level net-list of the system consisting of processing elements (PEs), busses and bridges. The bridges interface between busses to allow multi-hop communication. Each PE consists of 1 or more processes that can be accessed on the bus. The processes themselves are described using a set of C files that contain the functions implemented for that process. The

output is a TL Model of the custom communication platform. The design is the top level module consisting of submodules for each PE, bridge and memory. Processes executing on the PE are modeled as threads inside PE modules. A generic bus channel (GBC) is used to model system busses. A generic bridge module (GBM) models a bridge between two busses. GBM allows for communication between processes/memories that are not connected to a common bus. Since, the semantics for GBC and GBM are well defined, the TLM can be automatically derived from a set of platform parameters.

In order to develop the C code, a standard set of APIs is automatically generated for each process in each PE. These APIs provide communication services for rendezvous communication with other processes in the platform. Software developers need only to use these APIs to construct the TLM, so therefore, the C code developers do not have to understand the communication architecture or write any SystemC code to verify that their code executes on the platform.

2 Related work

TLM has gained a lot of attention recently ever since it was introduced [7] as part of high level SystemC [10] modeling initiative. Several use models and design flows [3, 6] have been presented centering around TLM. In [2, 13], the authors present semantics of different TL models based on timing granularity. Similarly, design optimization and evaluation has also been proposed using practical TLMs [9]. These approaches do not clearly separate computation from communication. A generic bus architecture was defined in [8], however, none of the above approaches address automatic TLM generation or the designer burden in learning new TLM styles and languages. There have been several approaches to automatically generate executable SystemC code from abstract descriptions. Modeling languages as UML [1] and behavioral descriptions of systems in SystemC [12] have been proposed. These approaches do not address transaction level platform modeling without the need to use another language. The closest work is in SpecC TLM generation for design space exploration [14], which still requires designers to understand complex channel modeling in a non-standard SpecC language. One important difference is that their modeling abstraction requires implementation decisions for synchronization to be already made. Moreover, there is no discussion of modeling communication processes such as bridges and routers. The novelty and utility of our approach lies in that we require only application C code and provide a programming model that is agnostic of communication architecture.
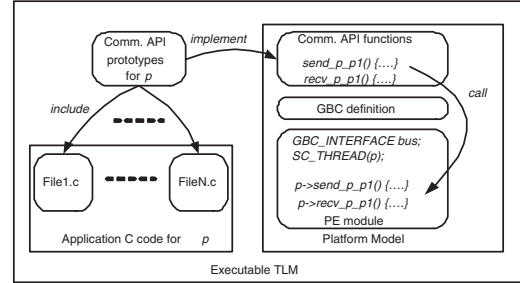

Figure 2. Code organization for TLM

3 Platform modeling

Each object in the platform is modeled according to a well defined SystemC template. Busses use the Generic Bus Channel (GBC) template, bridges use Generic Bridge Module (GBM) template, processes are *sc_threads* and PEs are *sc_modules*. Figure 2 shows the code organization for the executable TLM.

The design is modeled as a top level *sc_module* which instantiates all the GBCs, GBMs and PEs as captured in the GUI. We focus here on one process called *p*. This process will access its assigned PE's port by using the communication APIs. Each PE is declared also as a *sc_module* which contains one or more *sc_threads* representing the C processes of the PE. The communication APIs exported to the application C code are global functions that call the GBC access methods inside the corresponding process' *sc_thread*. For any process to send or receive data to any other process, a clear semantic is defined, which is independent of the platform defined by the user in the graphical capture. These should be used in the original C code. No further modification to these files is needed after that. The only limitation in the C code is that even if different functions are mapped into different processes in different PEs, they should have different names.

The communication API generated depends on the inter-process communication specification in the platform. A GBC send and receive call will be generated for each pair of communicating processes. These functions will access the corresponding GBC send/receive functions, even if the destination process is not located in the same GBC as the source process. In this case, the function will route the data through the necessary GBCs and GBMs to arrive at the destination process. All these steps are generated automatically, without the user needing to do anything else except calling the communication APIs.

3.1 Generic bus channel

GBC is a channel model that abstracts the system bus as a single unit of communication. GBC provides the

basic communication services of synchronization, arbitration and data transfer that are part of a transaction. At the transaction level, we do not distinguish between different bus protocols. GBC provides 5 bus communication functions namely: *Send/Recv* for synchronized communication, *Read/Write* for memory access and *MemoryService* for memory control.

Synchronization is required for two processes to exchange data reliably. A sender process must wait until the receiver process is ready, and vice versa. Synchronization between two processes takes place by one process setting the flag and the other process checking and resetting the flag. We will refer to the process setting the flag as the *initiator* and the process resetting the flag as *resetter*.

Since a bus is a shared resource, multiple transactions attempted at the same time must be ordered sequentially. This arbitration is modeled in the GBC using the SystemC *sc_mutex* class. to reflect such a sequential ordering of transactions. After arbitration, the *resetter* process sets *Address* which is read by the initiator process.

3.2 Generic bridge module

The GBM models the bridge connected to two busses. Its purpose is to facilitate multi-hop transactions, where one process sends data to another process that is not connected directly to the sender via a GBC. The basic functionality of the GBM is to simply receive data from the sender process, store it locally and send it to the receiver process once the latter becomes ready. The receiver can be a processing element or another GBM in the case of multi-hop transactions. There are three types of objects used to model the GBC as described in this section.
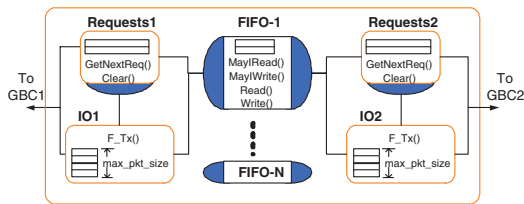

Figure 3. TLM for bridge module

3.2.1 FIFO channels

The data in transit via the GBM is stored locally in FIFO channels. The number of such channels is equal to the total number of communication paths through the GBM. The number of such paths can be easily derived from the platform specification. The size of each FIFO can be defined in GUI while parameterizing the bridge. Each FIFO supports four functions as follows: *MayIWrite* checks if space

is available in FIFO; *MayIRead* checks if data is present in FIFO; *BufferWrite* copies the incoming data to the FIFO buffer and updates the tail pointer; *BufferRead* copies data from the FIFO buffer to the output and updates the head pointer.

3.2.2 Request buffers

In general, before any data is sent/received to/from the GBM, a request must be made such that the GBM interface may check if the internal FIFOs can accomodate the data or supply it. Such a request may be included in the packet itself, but if the packet cannot fit, additional logic is needed in the bridge to reject the packet and in the process to check for rejection and resend it. For simplicity, we will only consider the scenario where the PE writes the request, followed by synchronization and data transfer. In case of multiple competing processes, the requests from different processes are arbitrated by the GBM and the communication with the successful process is initiated.

There are two request buffers in the GBM, one for each bus interface. The number of words per request buffer is equal to the number of communication paths through the bridge. The request buffer is modeled as any other memory module in a PE and thus has an address range on the bus. Each word in the request buffer has a unique bus address. The requesting process writes the number of bytes it expects to read/write into the communication path's corresponding request buffer. The request buffer is a module that supports four functions: *GetNextReady* checks the request words in the buffer in a round-robin fashion. For the chosen, request, it checks if the corresponding FIFO has enough data/space to complete the transaction of requested size. If yes, it returns the request ID and path, else it checks the next pending request. *Clear* removes the request from the buffer. *WriteMem* writes to the other request buffer in the same GBM in the case of multi-hop transactions. *Write* this function performs the write to the request buffer itself. It is exposed to the other Request Buffer in the GBM (namely to its *WriteMem* function). In the case of multi-hop transactions the GetNextReady function will call *WriteMem* in order to write to the next Request Buffer (if the data route continues in the same GBM).

3.2.3 IO module

The IO module is the interface function of the GBM that talks to other processes and GBMs on the bus. It consists of a local buffer of the size of maximum data packet and *F_Tx*. It starts by calling the *GetNextReady* function in the request buffer. Then, for the selected sender or receiver process, it calls the GBC receive or send function respectively. The data received from sender is written to the corresponding FIFO. The data to be sent to the receiver is first read

from the corresponding FIFO before calling the GBM send function. Once the requested transaction is completed, the request removed by calling the *Clear* function in the request buffer module. In the case of multi-hop transactions, the IO module will write the send request to the next GBM in the packet route, and proceed to send it.

4 Automatic TLM generation

In this section, we present the algorithms for generating the TLM described in Section 3 from the platform specification. For brevity, we will describe generation of GBC, GBM and top level module only.

Algorithm 1 Generate GBC

1: //Generate GBC flags and events
2: gen:"sc_mutex arbiter;"
3: gen:"int BusAddr; sc_event AddSet;"
4: P_{bus} = Set of proc.s on GBC
5: **for all** p1, p2 $\in P_{bus}$ **do**
6:     Declare synchronization flags and sc_events
7:     //Code gen. for synchronization
8:     gen:"if (sender==p1 && receiver==p2){"
9:     **if** p1.type = INITIATOR,p2.type = RESETTER **then**
10:        Set synchronization flag and notify sc_event
11:        Wait for AddrSet event if BusAddress not defined
12:     **else**
13:        Wait for event if synchronization flag is not set
14:        Arbitration and Bus Address set
15:     **end if**
16: **end for**

4.1   GBC code generation

The GBC is modeled as a SystemC channel class as described in Section 3.2. For each bus in the platform, a unique GBC channel implementation is generated. Algorithm 4 shows the method for creation of the GBC internal structure. Here, we present only the pseudo-code for send function generation due to lack of space. The receive function is similar to the send function, the read and write functions do not carry the code for synchronization and the memory service function simply executes and endless loop checking for bus address.

We start by creating the arbiter, which is an instantiation of the sc_mutex module, and create the variable and event for addressing (lines 2-4). Then, for all the communicating processes defined, we select processes that are directly connected to this GBC and include them in the set P_{bus}. The interface processes of the GBMs connected to this bus are also included in P_{bus}. The synchronization and addressing

code is generated for all pairs of processes in P_{bus} (Line 5). We create the synchronization flags and events as described in Section 3.1 for all pairs of processes in P_{bus} (Line 6). If the sender is an initiator, then code is generated to set the flag and notify the synchronization event (Line 10). Otherwise, we generate code to wait until the flag is set (Lines 11). The resetter is eventually responsible for acquiring the bus and setting the address. The corresponding code for locking the arbiter mutex and setting the bus address is generated if the sender is resetter (Line 13). If the sender is initiator, code is generated to snoop for the right address for this pair of communicating processes (Line 14). Finally, after the addressing, data transfer is performed by setting the local channel data pointer (*DataPtr*) to the pointer (*data_ptr* passed in the send function call, then code is generated to check if the sender is the resetter, and release the bus by unlocking the arbiter mutex.

Algorithm 2 Generate GBM: Address labels and GetNextReady()

1: //Bridge addresses generation
2: **for** each $\{p1, p2, address\} \in path$ **do**
3:     **if** $bridge \in path$ **then**
4:         //Bus Addresses generation using p1 and p2
5:     **end if**
6: **end for**
7: //Request:GetNextReady()
8: $count = 0$
9: **for** each $\{src, dest\} \in path$ **do**
10:     **if** $bridge \in path$ **then**
11:         gen:"if (RequestBuffer[$count$]) {"
12:            Set source,destination,size and transfer type
13:         gen:" if($fifo \rightarrow$MayIWrite(*src,*dest,*size)==Yes)"
14:         **if** $dest$ is not local to this bus **then**
15:             gen:"WriteMem(*src,*dest,*size,*TransferType);"
16:         **end if**
17:         gen:" return true; }"
18:     **end if**
19:     $count + +$
20: **end for**

4.2   GBM code generation

The Generic Bridge Module generated by our tool consists of two sets of modules, one set for each GBC. Each set consists of a FIFO channel, a Request buffer and an IO module as described in Section 3.2. Shown in the algorithm 4.1 is the generation of the GetNextReady function of the Request buffer.

The bridge addresses are uniquely named using the source, destination and address (*p1,p2*) in each *path* (Lines 2-6). In the request buffer, the function *GetNextReady()*

checks if any of the request addresses has been modified, and sets the variables {*src, dest, size, transfer type*} accordingly for the transfer (Lines 9-20). The tool checks each *path*, and if the current *bridge* is part of it (Line 10), uses its source and destination (*src,dest*) to generate the proper process IDs for the pointer assignment (Line 12) , and the buffer permission to read or write(Line 13). In order to determine which fifo to read and write, each IO module connection to the busses is checked along with the busses information in each route. This determines which process is assigned to the pointers *src* and *dest*. In the same iteration loop, the function *Clear* described in Section 3.2.2 can be generated. The other modules in the bridge are generated iterating through the FIFOs, and generating the functions *MayIWrite, MayIRead, BufferWrite, BufferRead* for each of them.

Algorithm 3 Generate Top Module

1: **for** each $\{PE, bus, bridge \in design\}$ **do**
2:     Instantiate process in PE
3: **end for**
4: //Connections inside the constructor
5: **for** each $\{PE \in design\}$ **do**
6:     **for** each $\{proc \in PE\}$ **do**
7:         **for** each $\{port \in proc\}$ **do**
8:             **for** each $\{conn \in design\}$ **do**
9:                 **if** $conn == PEname$ **then**
10:                   gen:"$proc\_inst \rightarrow port(*bus\_inst)$;"
11:                 **end if**
12:             **end for**
13:         **end for**
14:     **end for**
15: **end for**

4.3 Top module code generation

After the GBM and GBC algorithms generate the busses and the bridges, Algorithm 4.2 shows the final class generation, where the process, PE, bus and bridge instantiations and connections are made. Instantiations are made for each *PE*, *bus* and *bridge*. Once inside the constructor (Lines 5-15) , a connection is made between its port and the corresponding bus (Line 10), by checking connections (*conn*) for each *PE* (Line 9).

5 Experimental Results

The algorithms shown in Section 4 were implemented in a C++ tool for automatic generation of TLMs from application C code and platform specification. The input to the tool was a high level net-list of the system, with pointers to

C code. The output is a complete set of executable SystemC files (PEs, GBCs and GBMs).

We selected two large industrial applications namely a MP3 decoder and a H.264 fixed point decoder to test our automatic TLM generation tool. All tests were performed on an Pentium 4, 3 GHz, 1 GB RAM machine running Linux kernel 2.6. The MP3 decoder reference C code[5] consisted of 9463 lines of C code. The simulation testbench for MP3 was a input file of 138 KB size. The original reference C code for H.264 decoder[4] consisted of simplified decoder with 3419 lines of C code and its simulation was performed using a 27 KB clip of frame size 352 by 288 pixels. Several platforms with different communication architectures, as shown in Figures 4 and 5, were used to generate the TLMs.
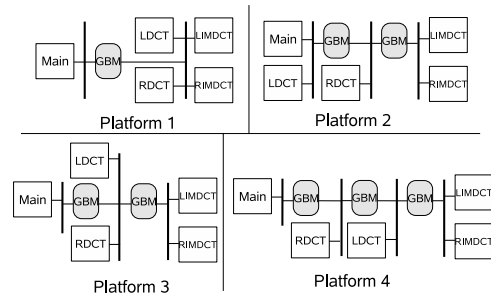
Figure 4. Platforms used for MP3 decoder
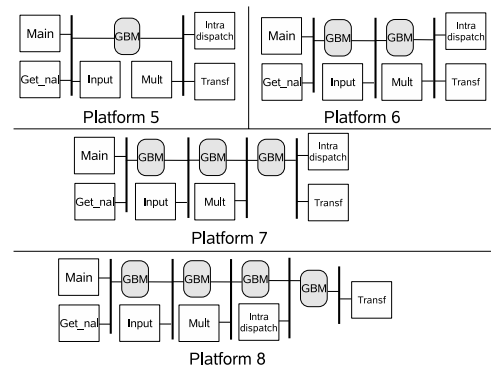
Figure 5. Platforms used for H264 decoder

Table 1 shows the automatic generation results for different platforms for MP3 and H.264 TLMs. The first column indicates the platform configuration. The second column shows the lines of SystemC code that were generated by the tool. The third column demonstrates the productivity gain of our tool by showing the estimated time in person-hrs for doing this SystemC coding manually. We use an optimistic figure of 20 lines of code per person-hr. The fourth column shows the generation time in seconds for TLM to be created for each platform. The fifth column shows the model

Table 1. TLM generation time and quality for different MPSoC designs.

Application	Platform Configuration	Generated SC LOC	Manual est. time	Generation time	Simulation time
MP3	reference C	-	-	-	1.29s
	Platform 1	2095	104 hrs	0.633s	3.268s
	Platform 2	2894	144 hrs	0.661s	5.519s
	Platform 3	3148	157 hrs	0.645s	5.764s
	Platform 4	3653	183 hrs	0.741s	7.424s
H.264	reference C	-	-	-	2.027s
	Platform 5	1722	86 hrs	0.245s	7.542s
	Platform 6	2796	140 hrs	0.244s	9.935s
	Platform 7	3853	192 hrs	0.267s	13.326s
	Platform 8	4910	245 hrs	0.260s	15.415s

simulation time on the testbench described above for each application. We can see from these results that the tool generates thousands of lines of code in a fraction of a second. The manual coding time would cost numerous hours of precious designer time that we can save using automatic TLM generation. The results shows that the quality of our TLMs is very high since simulation time is of the same order as the reference C simulation.

6   Conclusions and Future Work

In this paper, we presented a methodology and tool for automatically generating TLMs from graphical capture of platform and application C code. The key differentiation is in the separation of not only computation and communication, but also between application and platform. In other words, with our tool, there is no need for designers to understand SystemC modeling, event semantics etc. which makes it very attractive for SW developers. Currently we are developing synthesis semantics for GBC and GBM to provide system synthesis from automatically generated TLMs. In the future, we plan automatic generation of TLMs for application specific NoCs.

References

[1] F. Bruschi, E. Di Nitto, and D. Sciuto. Systemc code generation from uml model. In *Proc. Int. Forum on Specification and Design Languages. FDL'04*, Frankfurt, September 2003.

[2] L. Cai and D. Gajski. Transaction level modeling: an overview. In A. Press, editor, *CODES+ISSS '03: Proceedings of the 1st IEEE/ACM/IFIP International Conference on Hardware/software Codesign and System Synthesis*, pages 19–24, New York, NY, 2003.

[3] A. Donlin. Transaction level modeling: Flows and use models. In A. Press, editor, *CODES+ISSS '04: Proceedings of the 2nd IEEE/ACM/IFIP International Conference on Hardware/software Codesign and System Synthesis*, pages 75–80, New York, NY, 2004.

[4] M. Fiedler. Implementation of a basic h.264/avc decoder. Seminar Paper, June 2004. Chemnitz University of Technology.

[5] MAD fix point mp3 algorithm implementation. http://sourceforge.net/projects/mad/.

[6] F. Ghenassia. *Transaction-Level Modeling with SystemC: TLM Concepts and Applications for Embedded Systems*. Springer, November 2005.

[7] T. Grotker. *System Design with SystemC*. Kluwer Academic Publishers, Norwell, MA, 2002.

[8] W. Klingauf, R. Gunzel, O. Bringmann, P. Parfuntseu, and M. Burton. Greenbus - a generic interconnect fabric for transaction level modeling. In *Proceedings of the 43rd annual conference on Design Automation*, pages 905–910, San Francisco, CA, July 2006.

[9] O. Ogawa. A practical approach for bus architecture optimization at transaction level. In I. C. Society, editor, *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe*, page 20176, Washington, DC, 2003.

[10] OSCI. Systemc. www.systemc.org.

[11] A. Sangiovanni-Vicentelli. System level design: Orthogonalization of concerns and platform-based design. *IEEE Transactions on Computer-Aided Design of Circuits and Systems*, 19(12):1523–1543, December 2000.

[12] A. Sarmento, W. Cesario, and A. Jerraya. Automatic building of executable models from abstract soc architectures made of heterogeneous subsystems. In *Proceedings of the 15th IEEE International Workshop on Rapid System Prototyping*, June 2004.

[13] G. Schirner and R. Doemer. Quantitative analysis of transaction level models for the amba bus. In *Proceedings of the Design Automation and Test Conference in Europe*, March 2006.

[14] D. Shin, A. Gerstlauer, J. Peng, R. Doemer, and D. Gajski. Automatic generation of transaction-level models for rapid design space exploration. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis*, Seoul, Korea, October 2006.