

Automatic TLM Generation for C-Based MPSoC Design

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) for MPSoC designs using only C-code and graphical capture. 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 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. Moreover, the platform can be easily modified and a new TLM for that platform can be automatically generated. Our experimental results demonstrate that for large industrial applications such as MP3 decoder and H.264, high-speed TLMs can be generated for a wide variety of 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 (PBD) [14] 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 modeling objects and composition rules have emerged yet. Most surveys point to usage of transaction level models for early system verification and embedded SW development. This

Table 1. Levels of Abstraction

Model	Functional Verification	Performance Verification
Reference C	Application	None
Concurrent Point-to-point	Computation Mapping	Processing Elements
Transaction Level Model	Communication Mapping	Communication Architecture
Pin Cycle Accurate Model	Cycle Accurate Behavior	Microarchitecture

requires application developers 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 application developers to understand either the platform communication architecture or to learn new modeling languages like SystemC.

The key idea behind PBD is the orthogonalization of functional modeling and architectural modeling. In practice, this turns into the orthogonalization of concerns for application developers and architecture designers. Table 1 shows the different abstraction levels of design modeling. The most abstract model is the reference C model, followed by concurrent processes communicating with point-to-point channels, followed by our communication architecture accurate TLM and finally the pin-cycle accurate model. While the architecture designer is concerned about accuracy of the different models (performance verification), the application developer requires a fast model with the right amount of detail to provide sufficient confidence that if the application executes correctly on the model, it would also execute correctly on the actual platform. Although accuracy is an easy metric to define quantitatively, confidence is hard to define. However, we can say that an application developer would have relatively higher confidence in a TLM with the communication modeled at the granularity of busses and bridges, rather than a model with concurrent processes and point-to-point abstract communication. This is because the former has more observable platform specific events than

the latter.

The performance model is simply the functional TLM annotated with *waitfors* at the observable *checkpoints* in the code. The quality of the performance model depends on the accuracy of the *waitfors* and the granularity of the *checkpoints*. This topic, although extremely relevant, is orthogonal to functional verification and beyond the purview of this paper. Instead, we will focus exclusively on how to automatically generate high confidence functional TLMs.

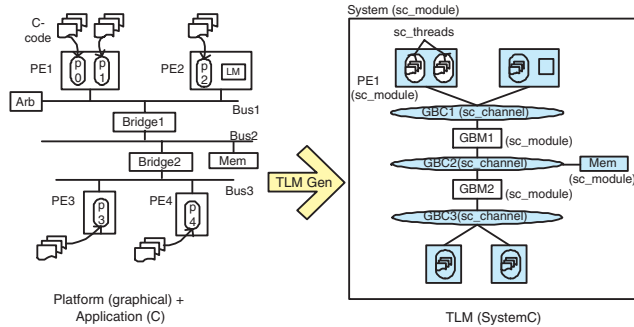


Figure 1. TLM generation for MPSoC platform

The input platform for our tool shown in the LHS of Figure 1 is a high level net-list of the system consisting of processing elements (PEs), memories, busses and bridges. The bridges are used to interface between busses to allow multi-hop communication. Each PE consists of 1 or more processes and possibly memory 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 product of our tool (RHS) is a transaction level model for the design in SystemC notation. The design is the top level module consisting of sub-modules 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 templates for the basic platform modeling objects are well defined, GBC and GBM 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 or to access memory anywhere in the platform. Therefore, the C code developers do not have to understand the communication architecture or write any SystemC code to verify that their code runs correctly 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 [6] have been presented centering around TLM. In [2], the authors present semantics of different TL models. Verification with TLM has been proposed using both simulation [4] and formal methods [8]. Similarly, design optimization and evaluation has also been proposed using practical TLMs [13]. There have been several approaches to automatically generate executable SystemC code. Modeling languages as UML [1] and behavioural descriptions of systems in SystemC [15] have been tried before. Other tools require the user to specify abstract models using SARTS [9] or Task Graphs [11].

Several graphical tools and design environments exist for system design. In [12], several models of computation can be used, but no automatic generation of executable models, and any complex communication elements must be designed manually. The same drawback is present in [16], where no clear template exists for these elements, and manual coding is needed to implement translators and wrappers. In [5], a graphical tool is also used to drag and drop components and generate SystemC code, but all models should already be in the predefined libraries.

Other system design tools [3] generate SystemC TLMs, but are abstract representations of the system, and do not contain platform specifications (no communication elements such as busses or bridges).

These options do not provide a way to quickly experiment a platform and simulate a TLM model without the need to use another language. This paper describes a new approach that requires only C files and use of a graphical capture (to generate xml data). Functions prototypes are given for the user to interface the C code with the SystemC model.

3 Platform and application capture

The platform is captured graphically along with the application C code. An xml based internal representation is used to represent the net-list of PEs and the mapping of processes to PEs. The platform provides two types of communication services to the application. The first is rendezvous communication which assumes that the communicating processes must have an end to end synchronization before data transfer. The second is an unsynchronized memory access from processes. The user does not need to model the memory explicitly, since the tool creates the transaction level memory controller and array. The services are sufficient to model both deterministic systems (with synchronized communication) and real time systems (with un-

synchronized communication). The Abstract connectivity matrix (ACM) is simply a table of all the processes in the MPSoC. It determines the restricted set of communication API functions that are generated during TLM creation. If process P1 needs to send data to process P2, then the entry with column P1 and row P2 is checked else it is crossed. Similarly, the access of memories by processes may be restricted. By default, functions are generated for communication between all pairs of processes and memories.

For each bus in the design, an address map must be defined. This corresponds to addresses that are assigned to memories on the bus as well as special addresses used for inter-process communication. For processes and memories that may communicate as specified in ACM, the table entries indicate the range of addresses used for that communication.

4 Platform modeling

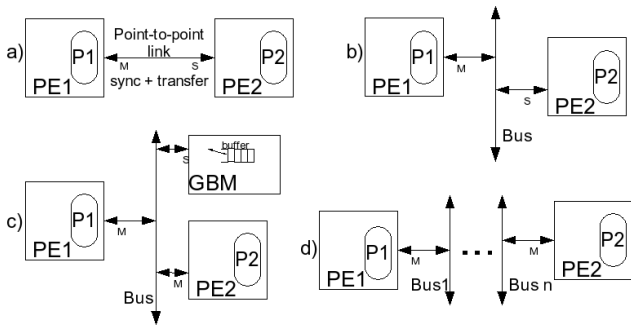


Figure 2. Communication Architectures

Figure 2 shows the different communication architectures supported by our tool. Processing elements can be connected in a point-to-point fashion as shown in a). The channels linking the elements offer synchronization and data transfer functions only. In b), objects are connected to a shared bus, which offers synchronization, arbitration and data transfer functions. A processing element with a *master* role can communicate with another *master* (shown in c)) by using a one-port GBM which functions as a shared memory. This setup allows multi-PE communication without the need of using interrupts and interrupt controllers. Finally, shown in d), a PE can communicate with another PE connected to different bus. These busses are linked by any number of GBM.

Each object in the platform is modeled according to a well defined SystemC template. Busses use the GBC, bridges use GBM, processes are *sc_threads* and PEs are *sc_modules*. Figure 3 shows the code organization for the executable TLM. We focus here on process *p*. The design is modeled as a top level *sc_module* that instantiates all the

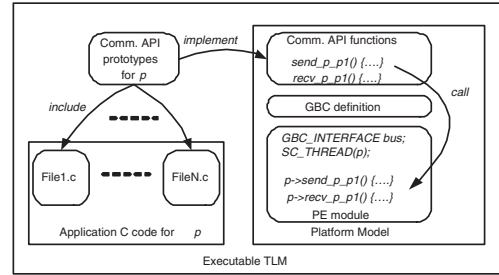


Figure 3. Code organization for TLM

GBCs, GBMs and PEs as captured in the GUI. 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 each process a header file is generated that includes the prototypes for its communication API. The header is included by the application C code. The communication API generated depends on the needs specification in the ACM. For example, if process *p* must send data to *p1*, then a function *send_p_p1* is generated for that uses the GBC send call. Also, a corresponding function *recv_p1_p* is generated for *p1*. The application developer just needs to call this API from the C files to interface with the generated SystemC code.

In this section, we will discuss the details of GBC and GBM to explain how the communication functions are implemented.

4.1 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. In TLM, we do not distinguish between different bus protocols since at this level of abstraction, the primary concern is the transfer of data through a bus, not the protocol used by the bus to achieve it. The GBC provides 5 bus communication functions namely: *Send/Recv* for synchronized communication, *Read/Write* for memory access, and *MemoryService* for memory control.

4.1.1 Synchronization

Synchronization is required for two processes to exchange data reliably. A sender process must wait until the receiver process is ready, and vice versa. A Synchronization Table in the GBC keeps flags and events (indexed by process ids) that are used by a process to notify its transaction partner process that it is ready. Synchronization between two processes takes place in the following way: one process first sets the flag, while the other process is checking it. After

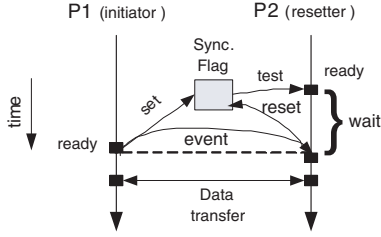


Figure 4. Flag based synchronization between processes

it is set, the second process will reset it and the transacting processes are said to be synchronized. We will refer to the process setting the flag as the *initiator* and the process resetting the flag as *resetter*. In Figure 4, assume P1 is the initiator process and P2 as the resetter process. If P2 is ready before P1, it must wait for the event in which P1 sets flag to complete the synchronization. Once the flag has been set and reset, both processes can continue to the data transfer stage.

4.1.2 Arbitration

Since a bus is a shared resource, multiple transactions attempted at the same time must be ordered sequentially. Arbitration is modeled in the GBC to reflect such a sequential ordering of transactions. After synchronization, the resetter process attempts to reserve the bus for data transfer. This is achieved by an arbitration *request* by the resetter process. Since the GBC model is exclusive for functional verification, we simply model the arbiter as a mutex. An arbitration request corresponds to a mutex lock operation and once the transaction is complete, the arbiter is released with a mutex unlock operation.

4.1.3 Addressing and data transfer

In order to do addressing and data transfer, the GBC uses several variables and events. The variable *TxAddress* stores the starting address of the active transaction, variable *DataPtr* keeps the pointer to the transacted data and variable *RdWr* identifies if a transaction is read or write. The event *AddrSet* is notified when *TxAddress* is set.

For synchronized communication, the *resetter* process sets *TxAddress* to the appropriate value from the bus address table. For memory transactions, the reader or write process sets *TxAddress*. This is followed by the notification of event *AddrSet* that wakes up the other process or memory controller that is snooping the address bus. In case of memory transaction, the memory controller reads the address *TxAddress* to check if the address falls in its range and

computes the offset. It then sets *DataPtr* to the right address in the local memory according to computed offset.

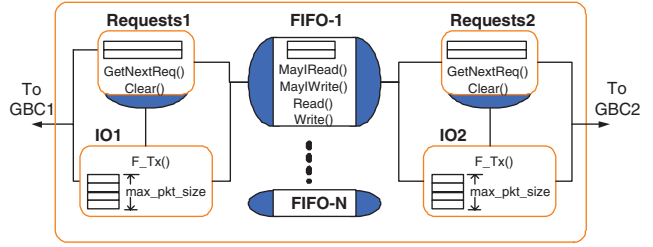


Figure 5. TLM for bridge module

4.2 Generic bridge module

The GBM can model a shared memory used for inter-process communication, or a bridge connected to two busses. Its purpose in this case 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. There are three types of objects used to model the GBM as described in this section and are shown in Figure 5. These are: Request buffers, Input/Output modules and FIFO channels.

4.2.1 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 ACM. The size of each FIFO can be defined in GUI while parameterizing the bridge. Each FIFO supports four functions as follows:

1. *MayWrite (MayRead)* returns true if the requested space (data) is available in the FIFO to write (read) else returns false;
2. *BufferWrite (BufferRead)* writes (reads) the incoming (outgoing) data to (from) the FIFO buffer and updates the tail (head) pointer;

4.2.2 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 accommodate 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 two functions:

1. *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.
2. *Clear* removes the request from the buffer.

4.2.3 IO module

The IO module is the interface function of the GBM that communicates to other processes 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.

5 Automatic TLM generation

In this section, we present the algorithms for generating the TLM TLM described in Section 4 from the platform specification. For brevity, we will describe generation of GBC, GBM and top level module only.

5.1 GBC code generation

The GBC is modeled as a SystemC channel class as described in Section 4.2. For each bus in the platform, a

Algorithm 1 Generate GBC

```

1: //Generate GBC flags and events
2: gen:“sc_mutex arbiter;”
3: gen:“int BusAddr; sc_event AddrSet;”
4:  $P_{bus}$  = Set of proc.s on GBC
5: for all p1, p2  $\in P_{bus}$  do
6:   gen:“bool p1_p2_flag;”
7:   gen:“sc_event p1_p2_event;”
8:   //Code gen. for synchronization
9:   gen:“if (sender==p1 && receiver==p2){”
10:  if p1.type = INITIATOR,p2.type = RESETTER
    then
11:    gen:“p1_p2_flag = 1;”
12:    gen:“p1_p2_event.notify();”
13:    //Addressing
14:    gen:“while(BusAddr! = p1_p2_addr){”
15:    gen:“wait(AddrSet);}”
16:  else
17:    gen:“while(p1_p2_flag! = 1){”
18:    gen:“wait(p1_p2_event);}”
19:    //Arbitration and Addressing
20:    gen:“arbiter.lock(); //get bus”
21:    gen:“BusAddr = p1_p2_addr;”
22:    gen:“AddrSet.notify();}”
23:  end if
24: end for
25: gen:“DataPtr=data_ptr;”
26: gen:“wait (BusDelay);”
27: gen:“if (sender.type=RESETTER){”
28: gen:“Arbiter.unlock();}”

```

unique GBC channel implementation is generated. Algorithm 5 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 in the ACM, 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 4.1.1 for all pairs of processes in P_{bus} (Lines 6-7). If the sender is an initiator, then code is generated to set the flag and notify the synchronization event (Lines 11-12). Otherwise, we generate code to wait until the flag is set (Lines 17-18). The resetter is eventually responsible for acquiring the bus and setting the address as explained in Section 4.1.3. The corresponding code for locking the arbiter mutex and setting the bus address is generated if the sender is resetter (Lines 19-22). If the sender is initiator, code is generated to snoop for the right address for this pair of communicating processes (Lines 14-15). 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 (Line 25). Finally code is generated to check if the sender is the resetter, and release the bus by unlocking the arbiter mutex (Lines 27-28).

5.2 GBM code generation

The Generic Bridge Module generated by our tool consists of several sets of modules, one set for each connected GBC. Each set consists of a FIFO channel, a Request buffer and an IO module as described in Section 4.2. Shown in the algorithm 5.2 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 9-22). The tool checks each *path*, and if the current *bridge* is part of it (Line 11), uses its source and destination ($src, dest$) to generate the proper process IDs for the pointer assignment (Lines 12-13), and the buffer permission to read or write (Line 16). In order to

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 if a request is to read or write, and which process is assigned to the pointers *src* and *dest*. In the same iteration loop, the function `Clear` described in Section 4.2.2 can be generated. The other modules in the bridge are generated iterating through the partitions in each FIFO (*partition*), and generating the functions `MayIWrite`, `MayIRead`, `BufferWrite`, `BufferRead`.

Algorithm 2 Generate GBM: Address labels and GetNextReady()

```

1: //Bridge addresses generation
2: for each {p1, p2, address} ∈ path do do
3:   if bridge ∈ 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} ∈ route do
10:  if bridge ∈ route then
11:    gen:“if (RequestBuffer[count]) {”
12:    gen:“ *src=ID_src;”
13:    gen:“ *dest=ID_dest;”
14:    gen:“ *size=RequestBuffer[count];”
15:    gen:“ *TransferType=SEND;”
16:    gen:“                                     if(fifo-
17:    gen:“   return true;”
18:  end if
19:  count ++
20: end for

```

5.3 Top module code generation

After the GBM and GBC algorithms generate the busses and the bridges, Algorithm 5.3 shows the final class generation, where the process, PE, bus and bridge instantiations and connections are made. The pointer definitions are made for each *proc* in each *PE* (Lines 1-5), and for each *bus* (Lines 6-8) and *bridge* (Lines 9-11) in the design. The pointers are named appending the word “instance” and the process, bus or bridge name. Once inside the constructor (Lines 13-23), for each *proc* of each *PE*, a connection is made between its port and the corresponding bus (Line 18), by checking all connections (*conn*) for each *PE* (Line 17). We go through all connections in the design until all PEs are connected (Lines 16-20). The bridge connections (not shown) are done checking each bridge interface for its corresponding bus.

Table 2. TLM generation time and quality for different MPSoC designs.

Application	Platform Configuration	Generated LOC	Manual time	Generation time	Simulation time
MP3	reference C	-	-	-	1.29s
	2 PEs + 1 Bus	1248	60 hrs	0.164s	1.440s
	2 PEs + 1 Bus + 1 Br	1964	98 hrs	0.259s	3.980s
	3 PEs + 1 Bus + 1 Br	2081	104 hrs	0.259s	4.130s
	4 PEs + 1 Bus + 1 Br	2164	108 hrs	0.285s	5.200s
H.264	reference C	-	-	-	2.027
	2 PEs + 1 Bus	1014	51 hrs	0.120s	2.275s
	2 PEs + 2 Bus + 1 Br	1326	67 hrs	0.223s	2.850s
	3 PEs + 2 Bus + 1 Br	1556	78 hrs	0.227s	4.024s
	4 PEs + 2 Bus + 1 Br	2778	139 hrs	0.231s	6.245s

6 Experimental Results

Algorithm 3 Generate Top Module

```

1: for each {PE ∈ design} do
2:   for each {leaf ∈ PE} do
3:     gen:“leafname * leafname_inst”
4:   end for
5: end for
6: for each {bus ∈ design} do
7:   gen:“busname * busname_inst”
8: end for
9: for each {bridge ∈ design} do
10:  gen:“bridgename * bridgename_inst”
11: end for
12: //Connections inside the constructor
13: for each {PE ∈ design} do
14:   for each {leaf ∈ PE} do
15:     for each {port ∈ leaf} do
16:       for each {conn ∈ design} do
17:         if conn == PEname then
18:           gen:“leaf_inst → port(*bus_inst);”
19:         end if
20:       end for
21:     end for
22:   end for
23: end for

```

The algorithms shown in Section 5 were implemented in a C++ tool for automatic generation of TLMs from application C code and platform specification. The input to the tools was an xml description of the platform, with pointers to C code. We selected two large industrial applications, namely a MP3 decoder and H.264 fixed point decoder to test our automatic TLM generation tool. All tests were performed on an Intel Pentium 4, 3 GHz, 1 MB RAM machine running Linux 2.6.9. The MP3 decoder reference C code consisted of 3251 lines of C code. The simulation testbench for MP3 was an input file of 138 KB size and an output PCM file of 1.7 MB size. The original reference C code for H.264 decoder consisted of 2588 lines of C code. The H.264 simulation was performed using a 27KB clip of frame size 352 by 288 pixels. Table 2 shows the automatic generation results for different platforms for MP3 and H.264 TLMs. The first column shows the different platform configurations by pointing out the number of PEs, busses and bridges in the design. 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 compared to manual SystemC coding. The fourth column shows the generation time in seconds for TLM to be created for each platform. The final column shows the model simulation time on the testbench described above for each application. We can see from these results that the tool is able to generate thousands of lines of SystemC TLM code in a fraction of a second that would otherwise cost up to hundreds of hours of precious designer time. The simulation speed shows that the quality of our TLMs is very high because simulation time is comparable to reference C simulation even for complex platforms.

7 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 anything about systemC modeling, event semantics etc. which makes it very attractive for SW developers. Currently we are targeting synthesis from our automatically generated TLMs and developing synthesis semantics for objects such as GBC and GBM. Also, we are adding new functionality to the tool to expand its domain to MPSoCs that have potentially several routes for communication between two PEs. In the future, we would like to automatically generate 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] J. Cockx, K. Denolf, B. Vanhoff, and R. Stahl. Sprint: A tool to generate concurrent transaction-level models from sequential code. *EURASIP Journal on Advances in Signal Processing*, 2007.
- [4] CoWare. N2c. www.coware.com/cowareN2C.html.
- [5] CoWare. Platform architect. <http://www.coware.com>.
- [6] 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.
- [7] T. Grotker. *System Design with SystemC*. Kluwer Academic Publishers, Norwell, MA, 2002.
- [8] A. Habibi and S. Tahar. Design for verification of systemc transaction level models. In I. C. Society, editor, *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 560–565, Washington, DC, 2005.
- [9] P. Hastono and S. Huss. Automatic generation of executable models from structured approach real-time specifications. In *Proceedings of the 25th IEEE International Real-Time Systems Symposium (RTSS)*, Lisbon, Portugal, December 2004.
- [10] O. S. Initiative. Systemc. www.systemc.org.
- [11] S. Klaus, S. Huss, and T. Trautmann. Automatic generation of scheduled systemc models of embedded systems from extended task graphs. In *Proc. Int. Forum on Design Languages*, Marseille, France, September 2002.
- [12] U. of California-Berkeley. The ptolemy project.
- [13] 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.
- [14] 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.
- [15] A. Sarmiento, 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.
- [16] Vanderbilt-University. Generic modeling environment.