In Proceedings of *Forum on Specification & Design Languages 2007* (FDL 2007)
Barcelona, Spain, September 18-20, 2007, ©ECSI

Mapping Actor-Oriented Models to TLM Architectures

Jens Gladigau[1], Bernhard Niemann[2], Christian Haubelt[1], and Jürgen Teich[1]

[1]University Erlangen-Nuremberg, Erlangen, Germany
{jens.gladigau,haubelt,teich}@cs.fau.de

[2]Fraunhofer Institute for Integrated Circuits, Erlangen, Germany
nmn@iis.fraunhofer.de

ABSTRACT

Actor-oriented modeling approaches are convenient for implementing functional models of embedded systems. Architectural models for heterogeneous system-on-chip architectures, however, are usually implemented using *transaction level modeling* (TLM). Even though both modeling paradigms, actor-oriented design and TLM, can conveniently be implemented using a common language such as SystemC, a methodology to smoothly integrate both approaches is still missing. In order to benefit from the best of both worlds, i.e., analyzability of actor-oriented models as well as fast simulation and synthesis from TLM, a combination of these two methodologies could be the next step towards electronic system level design. In this paper, we propose a systematic approach for mapping actor-oriented models onto complex TLM architectures. In particular, we will show how to map synchronization and data transport between actors to TLM 2.0 compliant bus protocols. The key contribution of this paper lies in our proposed methodology, which requires only a few additional lines of source code to map actor communication in actor-oriented SystemC models to TLM architectures and does not demand any reimplementation of the actors.

1. INTRODUCTION

Actor-oriented design [17, 16] is a widely accepted methodology for co-design. In a modular manner, concurrency and communication between components, called actors, is emphasized. Actors execute concurrently and communicate with each other over abstract channels, e.g., FIFO channels. Many important models of computation such as Petri nets [19], finite state machines (e.g., [9]) as well as many data flow models (e.g., [15]) may be considered and described as actor-oriented. Actor-oriented models allow for rapid development and, by means of their modularity, reuse is easy. They simplify analysis [11], verification [6], and automatic design space exploration and synthesis [10]. The well known Ptolemy project [3] supports modeling and simulation of different and heterogeneous actor-oriented models. SystemC [8] with its concept of modules and channels is a design language that easily allows to implement actor-oriented models.

Transaction level modeling (TLM) with SystemC [20, 5] has emerged as the de-facto industry standard for architecture modeling and virtual prototyping of system-on-chip platforms. Models at the transaction level are characterized by a separation of communication and behavior and their level of abstraction, which is above register transfer level. They achieve high simulation speed by replacing the details of bus protocol signaling with transactions, which can be modeled at different abstraction levels. The two most commonly used levels of abstraction are *programmers view* (PV) and *programmers view with timing* (PVT) [20]. At the PV, communication is modeled without any notion of time; using a generic bus, this view reflects system architecture and bus topology including the memory map. Using PVT, approximate timing is added to the transactions, making the communication bus specific and allowing a first evaluation of system performance. In an attempt to create a standard way of using TLM and to enforce interoperability of models, the Open SystemC Initiative (OSCI) has recently released a draft version of the OSCI TLM 2.0 standard [18].

Mapping a purely functional model to a specific architecture is an important task in system design. As stated above, functional modeling using an actor-oriented approach has great advantages, while architectural models can be described in a very suitable manner using TLM. Combining both modeling approaches by mapping, and therewith refining, actor-oriented models to transaction level models enables advanced and efficient system design, e.g., by fast exploration. Thereby, mapping abstract communication channels (e.g., FIFO channels or even more complex channels) to the transaction level with its bus-based communication is the most challenging task. The semantics of the abstract model must not be altered, e.g., FIFO channel sizes must be retained, so analyses and simulation results from the abstract model hold for the transaction level model. Finally, a communication mapping/refinement should be possible with little effort, and changing the implementation of actors should be avoided. Therewith, the development process and finding suitable solutions is sped up.

In the following, we propose a methodology where designers can map their SystemC-based actor-oriented models to TLM architectures by simply adding a small amount of additional code to their top level designs. Most important, actors are neither required to be reimplemented nor modified. With abstract communication mapped to the transaction level, existing TLM interconnect structures (e.g., GreenBus [14] or TLM conform IPs) can easily be used and explored in system architectures. In our examples, we use the PVT bus example shipped with *OSCI SystemC TLM 2.0 Draft 1* [18]. The used SystemC framework for actor-oriented modeling elaborated here is called SysteMoC [4], but our methodology is not limited to this framework. We use extended FIFO channels from the SysteMoC library for communication between actors. These channels enable transport of complex data types and allow for random access to a specified amount of data inside channels.

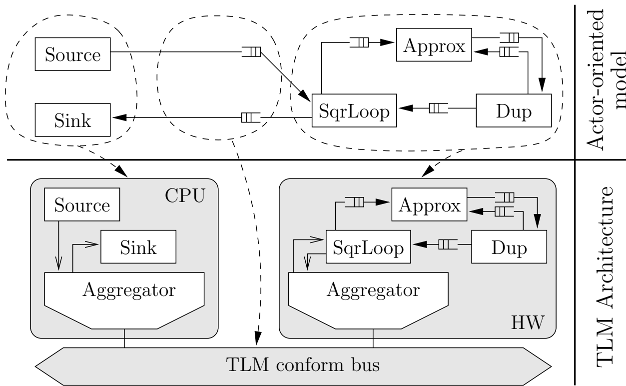

Figure 1: Mapping of an actor-oriented model (actors and FIFO channels) to TLM architecture components (shaded). This architecture consists of a bus and two components, *CPU* and *HW*. The communication transaction takes place in *Aggregators*.

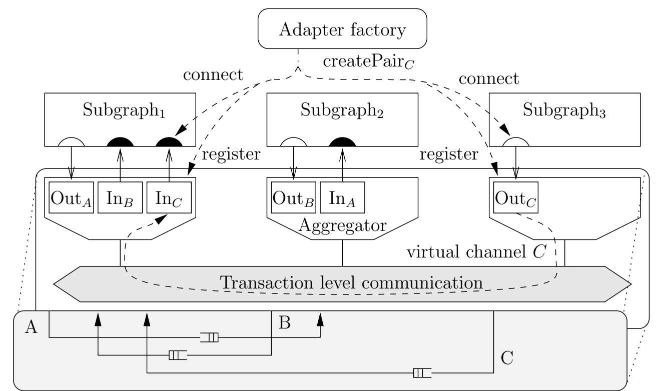

Figure 2: A SysteMoC model partitioned into *subgraphs*, which are connected via three channels among each other in actor-oriented modeling (*A*, *B*, *C*—grey, foreground). For TLM, adapter pair creation for *C* and connecting/registering is indicated.

The rest of this paper is structured as follows: Section 2 reflects related work. Section 3 gives an overview about our mapping approach. In Section 4, a brief introduction to the actor-oriented model of computation used in this paper is given. Section 5 describes the mapping in detail. First experimental results are presented in Section 6.

2. RELATED WORK

In [22], an approach for automatic generation of transaction level models from abstract models using message-passing, shared memory, and event semantics was presented. This approach differs from our proposal in the model of computation used as input; while we use actors communicating via buffered FIFO channels, communication over unbuffered abstract channels is used in [22]. Besides, we use SystemC for describing the (executable) models and for the resulting TLM models. Within the resulting transaction level models, TLM 2.0 compliant ports are used; we are able to use buses written for TLM, and, in contrast to [22], we do not rely on a database, which includes buses and communication elements. This allows for rapid development and exploration of very different available communication models without the need to modify them. Communication refinement from abstract data transfer to bus functional level using a protocol library is described [1], and in [21], a refinement to networks was shown. Both use models and a methodology similar to [22]. Other approaches use more restricted models (e.g., direct acyclic graphs in [7]). Hence, all cited approaches cannot be applied to the FIFO channels usually used in actor-oriented models.

In general, our approach is to lower the level of abstract communication of actor-oriented models to the transaction level, without implying any assumptions on the underlying communication architecture. SystemC TLM allows fast simulation and exploration. Afterwards, approaches such as [13, 12] can be applied to further lower the level of abstraction towards hardware implementations.

3. MAPPING OVERVIEW

In system synthesis from actor-oriented models, actors and channels of the model are clustered and mapped to components of a system architecture [2, 23]. In Figure 1, this is shown for the small but illustrating actor-oriented model of a square root calculator algorithm, which implements Newton's approximation algorithm. Actor *Source* produces numerical values for which their square root is to be approximated. This is done in a loop by the actors *SqrLoop*, *Approx* and *Dup*. Eventually, upon reaching a certain accuracy of the approximation, the result is sent to actor *Sink*.

If adjacent actors are implemented on the same resource, their communication is considered to be internal and can be implemented using, e.g., shared memory. In Figure 1, communication between actors *SqrLoop*, *Approx*, and *Dup* is internal communication, as all three actors are mapped to the same component (*HW*). Those channels are irrelevant in the communication refinement process described here. Moreover, the refinement of actors and their internal communication to hardware or software is out of scope of this paper. Other channels, mapped to communication resources (like buses), must be refined to the transaction level—without changing their semantics. This task is the focus of our paper. In the example, the two channels between *Source*/*Sink* and *SqrLoop* have to be refined to the transaction level, as they cross component boundaries (from *CPU* to *HW*). This is done by *adapters* (omitted in Figure 1) and *aggregators*. Adapters act as links between actor ports and the transaction level. Because many actors with external communication can be mapped to a single transaction level component, an aggregator is needed which encapsulates an arbitrary number of adapters and is connected to a transaction level communication component. Aggregators contain TLM ports and perform transaction level communication. So, adapters and aggregators substitute designated abstract channels of the actor-oriented model.

Substitution of a channel mapped to the transaction level is illustrated for a second actor-oriented model in Figure 2. This model was hierarchically partitioned into three subgraphs. Communication among these subgraphs occurs via abstract FIFO channels in actor-oriented modeling (shown in the shaded box with rounded corners in the foreground). This communication has to be mapped to the transaction level using adapters and aggregators (box with rounded corners in the background). The figure emphasizes the refinement procedure for FIFO channel *C*. Adapters and aggre-

gators are described in detail in Section 5. Basically, for each abstract channel mapped to a TLM communication resource, a pair of adapters is generated: one input and one output adapter. These adapters offer interfaces to the actors which match the interfaces of the abstract channel, and thus allow to replace it. The adapters are connected to the actors instead of the channel. Each computation resource (*CPU* and *HW* in Figure 1) to which parts of the partitioned actor-oriented model are mapped, contains one aggregator. Each adapter is *registered* in the aggregator of the computation resource its actor is mapped to. This can be seen as a two-layered communication. On the upper layer, two adapters communicate with each other. Low level communication (transaction level specific) is done by aggregators. As an example, for the channel between *Source* and *SqrLoop*, a pair of adapters is generated. The output port adapter for *Source* is connected to *Source* and registered in the aggregator included in *CPU*, the input port adapter for *SqrLoop* is connected to *SqrLoop* and registered in *HW's* aggregator. Issues that have to be clarified when refining FIFO-based communication to bus-based communication include:

- FIFO channels contain internal buffers, buses do not. How to handle these buffers? How to support different handling strategies for the buffers without adding complexity to the methodology?
- Actors may exchange arbitrarily complex data. How to handle serialization and other data format?

The mapping step should not be done all by hand, because rewriting the model is an error prone and time consuming process. We will propose a methodology for communication refinement which enables designers to flexibly map systems to heterogeneous architectures with little effort—only by adding a few lines of code to their top-level designs. Before we describe our mapping procedure in detail, a more in depth explanation of actor-oriented models is given in the following section.

4. MODEL OF COMPUTATION

For the mapping of the communication from actor-oriented models to the transaction level, the channel interface and communication semantics between actor and channel are the important aspects to consider. Synchronization between actors must not be altered, even if the FIFO buffer is distributed. To achieve this, understanding of the used model of computation is a prerequisite. This section gives a brief summary to actor-oriented models with focus on channel interface and semantics of the used SysteMoC library.

In actor-oriented *models*, *actors* are concurrently executed and communicating entities. Communication is restricted to dedicated *channels*. *Tokens* are produced and consumed by actors and transmitted via those channels. Actors contain *ports*, to which the channels are connected. Here, we focus on port to port communication media with FIFO semantics that connect exactly one *output port* with exactly one *input port*. In general, FIFO channels allow for great flexibility in high level design, but are very complex and hinder system synthesis, because of their internal storage and complex access semantics.

Due to its well defined semantics, we will use SysteMoC [4] in the following approach. SysteMoC is a SystemC library for actor-oriented modeling. However, our proposed methodology is not limited to this framework. In SysteMoC,

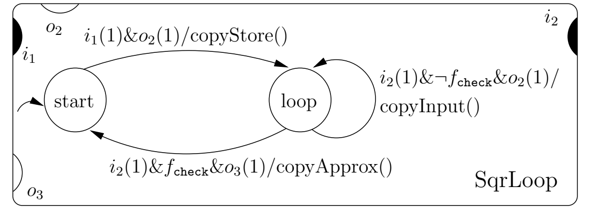

Figure 3: *SqrLoop* actor from Figure 1 with input ports i_1 and i_2, output ports o_2 and o_3, and state machine. The state machine consists of two states and three transitions. Each transition is a pair of *activation pattern / action.*

actors are divided into three parts: (1) the communication ports, (2) the actor *functionality*, and (3) the actor *communication behavior*. See Figure 3 for the *SqrLoop* SysteMoC actor from the square root calculator. Functionality is a collection of functions, *actions* and *guards*, that can access data on channels via ports. Actions are allowed to manipulate some internal state (internal variables) of the actor, while guards can depend on the internal state, but are prohibited to alter it. These functions are only executed during transitions of a finite state machine which implements the communication behavior of the actor. Transitions in this state machine include an *activation pattern* and an *action*, which is executed, if the transition is taken. The activation pattern determines under which conditions a transition may be taken, e.g., depending on a certain minimal amount of input tokens, values of tokens on input ports, or guards. For an example look at the transition from state *loop* to state *start* in Figure 3. This transition is activated, if its activation pattern evaluates to true, i.e., one token is available at port i_2, guard $f_{\texttt{check}}$ returns *true*, and space for one token is available at port o_3. While executing an action, access to all tokens (as stated in the activation pattern) in a random manner must be possible. Additionally, guards may depend on token data. This requires the possibility for non-consuming (non-destructive) read of tokens from channels. When the execution of the action function is finished, this is committed to the relevant channels. Thereby the transition of the finite state machine is completed and tokens are consumed and produced finally and atomic; free space and new tokens become visible to connected actors. Transition-based execution of actors can be summarized: (1) activation patterns are evaluated, (2) one of the activated transitions is chosen for taking, (3) the associated action is executed, and (4) completion of the action is committed, which triggers atomic token consumption/production. A more detailed discussion on SysteMoC can be found in [4].

For the mapping of FIFO channels to transaction level models, the interface through ports is important. As described, tokens can be accessed in a random manner, and only a commit signal triggers token production and/or token consumption, which is visible to the connected ports—the underlying channel (or our proposed substitution) will handle this. Additionally, peeking tokens for guard evaluation must be possible. In SysteMoC, channels perform an additional task, which a substitution has to mimic. For simulation speed, the channels use events to notify changes in token numbers to the actors. Even if for the used SysteMoC library some more effort is needed, in general our methodology relies only on small restrictions on ports/interfaces of

the used framework: (1) the framework has to maintain restrictions on token access and (2) actors have to signal completed transitions for token consumption/production. Besides SysteMoC, other frameworks for actor-oriented design can easily be adapted or created, e.g., designs based on pure FIFO channel communication as provided by SystemC.

5. MAPPING CONCEPTS

In mapping of actor-oriented models to TLM architectures, adapters and aggregators have a central role (see Figure 2). The main idea of our methodology was already described in Section 3. More details of the concept of adapters and aggregators follow in the next subsections. At first, we show some source code of the proposed mapping framework applied to map the square root example.

As stated before, for communication refinement of abstract channels, adapters are created in pairs. For convenience, creation is done in a *factory* (see Figure 2), which automatically associates input port and output port adapters with each other. These adapters must be connected to ports of subgraphs and registered to aggregators of computation resources, in which they are eventually implemented. This is shown in Listing 1 for the top-level design of the square root calculator. The essential tasks for mapping—*adapter pair creation*, *connection*, and *registration*—are underlined. For easy registering, adapters need to be data type-independent; for connecting to ports, a data dependent part of the adapter (called *plug*) is used. Function *createPair* creates an adapter pair with internal storage of given size, takes two pointers which eventually point to the created adapters, and returns a pair of data type-dependent plugs. These plugs are connected to ports and the adapters are registered to aggregators. More sophisticated functions for pair creation exist.

Port Adapters

A port adapter is connected to a SysteMoC port and adapts between the actor and the aggregator it is registered to. Therefore, the adapter implements two interfaces. The adapter's interface towards the port is equal to the abstract channel which should be replaced. Via this interface, the port accesses tokens, possibly more than one in a random manner, and commits completed actor-oriented transitions with a signal. Besides, output ports allow only for writing, input ports allow only for reading, tokens must not be transmitted or removed before a transition is completed, etc. Adapters must respect all of this. Towards the aggregator, data types of tokens have to be converted (e.g., serialized and deserialized) by the adapter. Additionally, meta information (e.g., available tokens, total FIFO size) must be provided for aggregators, a commit signal for each transition is forwarded from the actor to the aggregator, and event handling for the actor is done. In SysteMoC, actors need to be informed when specified amounts of data/space are available; this is done inside the SysteMoC channel, so adapters need to generate equal events for actors.

To preserve semantics of the unmapped actor-oriented model, the internal storage of the FIFO channels being replaced has to be transferred to the mapped model. Additionally, the transition-driven synchronization of actors must be maintained. In our solution, we assume that each adapter can buffer at least one token locally to allow for efficient (de)serialization. Different solutions for FIFO storage allocation are possible: (1) allocate the whole storage of the FIFO channel in one of the adapters (and hereby in its computation resource), (2) split the storage and distribute its allocation in two adapters/computation resources, or (3) allocate the storage in some other resource, e.g., a memory attached to the bus. Which alternative to choose must be resolved in a problem dependent manner. For example, if an actor produces tokens, one at a time, and its peer actor consumes tokens at a higher rate, splitting the storage and exploiting burst capabilities of the underlying bus may lead to better results than keeping the storage altogether at the receiver's side. Another favorable strategy is to choose local storages in the adapters as big as the maximum consume/produce rate of the connected actor to avoid data transfers when executing a single transition. This may result in more total memory allocation than the original buffer size of the FIFO channel. Exceeding this limit—dictated by the actor-oriented model's original channel—has to be prevented by the communication protocol. The communication protocol is part of the aggregator. It determines how tokens, respectively communicated data, is transmitted between adapters and guarantees synchronization between actors. Adapters are *passive* elements, i.e., they offer only interfaces and are triggered from the outside.

To support many different requirements, adapters are parametrized by (1) *token data type*, (2) *storage class*, and (3) *commit policy*. The data type may be of any arbitrary type which can be serialized. Storage class is either a local storage of specific size which is implemented in the adapter, or a storage proxy which redirects token read/write to other resources via the transaction level, or a mixture of

Listing 1: Complete top-level design for the square root calculator.

```
1  class SqrRootTop : public smoc_graph {
     SqrSrSi sqrSrSi; // Source & Sink
3    SqrCalc sqrCalc; // Calculation
     AdapterFactory factory;
5  public:
     SqrRootTop(sc_module_name name,
7               Aggregation &aggSrSi,
                Aggregation &aggCalc) :
9      smoc_graph(name), sqrSrSi("SqrSrSi"),
       sqrCalc("SqrCalc")
11   {
       InPortAdapter  *inAdapA,  *inAdapB;
13     OutPortAdapter *outAdapA, *outAdapB;

15     pair<InPlug<double>*,
            OutPlug<double>*> plugsA, plugsB;
17
       plugsA = factory.createPair<double>(
19       1, 1, inAdapA, outAdapB);
       connect(*(plugsA.1st), sqrCalc.in);
21     connect(*(plugsA.2nd), sqrSrSi.out);
       aggCalc.registerIn(inAdapA);
23     aggSrSi.registerOut(outAdapA);

25     plugsB = factory.createPair<double>(
         1, 1, inAdapB, outAdapB);
27     connect(*(plugsB.1st), sqrSrSi.in);
       connect(*(plugsB.2nd), sqrCalc.out);
29     aggCalc.registerIn(inAdapB);
       aggSrSi.registerOut(outAdapB);
31   } };
```

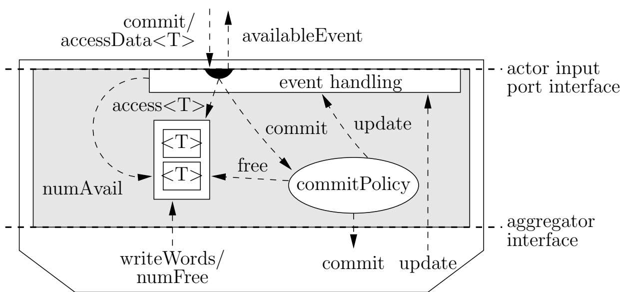

Figure 4: An input port adapter (shaded) and its interfaces to the actor port (top) and to the aggregator, it is registered to (bottom). It is parametrized with data type *T*, a local storage of size two, and a commit policy.

both. The commit policy specifies how the adapter handles a transition completion (commit) signal from its connected actor and mostly depends on the storage class. E.g., when committing a transition reading from the local storage, data in this storages must be freed, fill size information must be updated, and events for the actor must be notified. This all is determined by the commit policy.

An example adapter for a SysteMoC input port is shown in Figure 4. For the replaced abstract channel, a FIFO size of two is assumed which is all allocated in the local storage included in the adapter. In this case, only writing words into the storage is sufficient for the aggregator—there is no need to read words. Additionally, a commit signal from the adapter is forwarded to the aggregator, when the connected actor commits a actor-oriented transition, and the aggregator is able to trigger events for the actor by calling *update* and can determine free space by calling *numFree*.

As mentioned above, adapters are created in pairs within a factory; each adapter pair substitutes a single abstract channel. The factory mechanism is needed to ascertain the association of the adapters—they include references to each other which are only needed before *end of elaboration*. In SystemC, a static module hierarchy is build up in the *elaboration phase* (module instantiation and port binding) and before simulation starts. In this elaboration phase, also adapters are created and registered at aggregators. At the end of this phase, static maps are created by the aggregators, to address specific adapters. Details follow in the next subsection.

Aggregators

Aggregators implement a communication protocol for adapters at the transaction level. While different protocols for data transmission/access are possible, we only introduce a *push protocol* as an example. Aggregators are, in contrast to adapters, *active* components and contain a thread. Via their TLM port, they communicate among each other over arbitrary TLM communication resources. Therefore, each aggregator is assigned a dedicated address-range, timing parameters, etc. When registering an adapter in an aggregator, addresses are assigned to this adapter. These addresses allow other aggregators to transmit data to this adapter and access meta information from this adapter. The registering process is done in the elaboration phase. At the end of elaboration, each aggregator builds two static maps. One map includes addresses of meta information and addresses of data from peer adapters associated with registered adapters. This map is used by the aggregator when accessing meta information or transferring data to remote adapters. The other map contains addresses associated with data or meta information for registered adapters. This map is used to respond to requests from other aggregators. Whether or not meta information and data access is needed at all depends on the communication protocol. In Figure 5, two aggregators and their internal maps required for the push protocol are shown. In this example, a single map is sufficient for each aggregator. The left hand side aggregator pushes data from its adapters to the right hand side aggregator. For the pushing aggregator, only a map to addresses is needed. For the other aggregator, a map from addresses to its adapters is sufficient. In this particular example, one master (MA) and one slave (SL) port are needed. In general, aggregators may need to be as well master as slave.

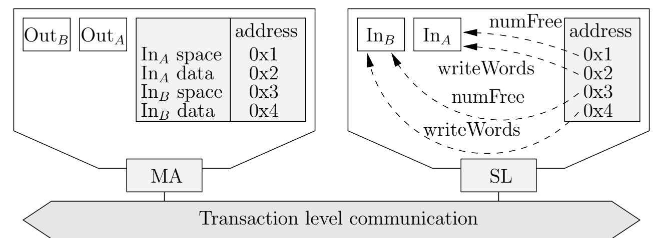

Figure 5: Two aggregators and their maps. For simple data pushing, a map to peer information and data access addresses is needed (left). The other aggregator needs to map from addresses to adapters.

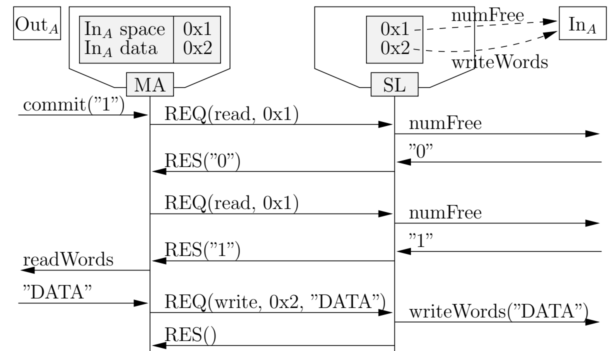

Figure 6: Message sequence chart for pushing data from adapter Out_A to adapter In_A.

We now describe the simple push data communication protocol in more detail. For a FIFO channel, its storage is split and distributed into both adapters which replace the channel. Therefore, both adapters are parametrized with local storages. As soon as data is available from the output port adapter, its aggregator checks the available space of the peer input port adapter's storage. This is done by looking up the address for this meta information in the map and sending a request to this address. If space is available, the data is pushed immediately, i.e., the needed amount of communication data for the token is sent to the address noted in the map. In case of insufficient space, available space is checked regularly and data is sent, as soon as space is available. The message sequence chart of a data transmission is shown in Figure 6. After the left hand side aggregator receives a commit signal from adapter Out_A, it checks available space in In_A by sending an according request. In the

example, two queries are sent, because there was insufficient available space at the first time. Then the data is pushed to adapter In_A.

With more than one output port adapter (or with a different protocol) conflicting bus accesses are possible. This demands a scheduler in the aggregator which resolves those conflicts, e.g., in a round robin manner.

6. EXPERIMENTAL RESULTS

In this section, we present first results on mapping an actor-oriented SystemC square root model to a SystemC transaction level architecture using the proposed framework. Source code of the transaction level model was shown already in Listing 1. Table 1 shows the number of additional lines of source code needed for partitioning and mapping the example to a TLM architecture with one bus (Figure 1). Rewriting the code to get the mapped model took only a few minutes when using our library. Mapping the model without our framework is hardly possible in such a short time. Note that about half of the additional code lines in the mapped example are needed for TLM setup (typedefs, TLM channels and bus). Mapping one additional FIFO channel would be possible by adding five new lines of source code. For the sake of completeness, we added the run times needed to simulate 10^5 square root approximations for each model. Measurements were done on a standard workstation with 3 GHz Pentium 4 processor.

Table 1: Additional lines of code (loc) needed for partitioning and mapping the SqrRoot model and average simulation run times (in seconds) to approximate 10^5 square root values.

SqrRoot model	additional loc	runtime
flat	-	8,79
partitioned	32	8,94
part. + mapped to TLM	32 + 48	16,10

7. CONCLUSIONS AND FUTURE WORK

In this paper, we have proposed a systematic approach of mapping actor oriented SystemC designs to TLM architectures. Our main goals were easy usage and rapid refining of models: our approach requires only a few additional lines of source code to map actor communication to TLM architectures and does not demand any reimplementation of the actors—designers can explore and simulate TLM architectures in a fraction of time needed for all by hand refinement without our library.

In future work, we will use our library to optimize the buffer distribution in the implementation, i.e., we will determine the best mapping of memories of the abstract channels from the actor-oriented modeling to memories in the TLM system. Here, different options could be explored: Place buffers inside the sender, inside the receiver, or into some external memory, or even distribute the memories over these components. Also, we will provide more protocols for communication a client can choose from when creating adapter pairs. Long-term objective is to automate the refinement process and to integrate it into design space exploration tools. This enables exploration of different communication media, e.g. buses, due to respecting their characteristics. Also the exploration becomes more accurate by better estimating communication overhead, conflicts, etc.

8. REFERENCES

[1] S. Abdi, D. Shin, and D. Gajski. Automatic Communication Refinement for System Level Design. In *Proc. 40th DAC*, pages 300–305, 2003.

[2] R. P. Dick and N. K. Jha. MOGAC: A Multiobjective Genetic Algorithm for Hardware-Software Co-Synthesis of Distributed Embedded Systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(10):920–935, Oct. 1998.

[3] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming Heterogeneity – the Ptolemy Approach. *Proceedings of the IEEE*, 91(1):127–144, Jan. 2003.

[4] J. Falk, C. Haubelt, and J. Teich. Efficient Representation and Simulation of Model-Based Designs in SystemC. In *Proc. FDL'06*, pages 129 – 134, Darmstadt, Germany, Sept. 2006.

[5] F. Ghenassia, editor. *Transaction-Level Modeling with SystemC*. Springer, Dordrecht, 2005.

[6] C. Girault and R. Valk. *Petri Nets for Systems Engineering – A Guide to Modeling, Verification, and Application*. Springer, Berlin, Heidelberg, New York, 2003.

[7] G. Gogniat, M. Auguin, L. Bianco, and A. Pegatoquet. Communication Synthesis and HW/SW Integration for Embedded System Design. In *Proc. 6th CODES*, pages 49–53, 1998.

[8] T. Grötker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Kluwer Academic Publishers, 2002.

[9] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[10] C. Haubelt, J. Falk, J. Keinert, T. Schlichter, M. Streubühr, A. Deyhle, A. Hadert, and J. Teich. A SystemC-based Design Methodology for Digital Signal Processing Systems. *EURASIP Journal on Embedded Systems, Special Issue on Embedded Digital Signal Processing Systems*, 2007:Article ID 47580, 22 pages, 2007. doi:10.1155/2007/47580.

[11] M. Jersak, K. Richter, D. Ziegenbein, R. Ernst, C. Haubelt, F. Slomka, and J. Teich. SPI - Workbench for the Analysis of Embedded Systems. In *Modelle, Werkzeuge und Infrastrukturen zur Unterstützung von Entwicklungsprozessen*, pages 247–261, Aachen, Germany, Mar. 2002.

[12] Kim Grüttner, Cornelia Grabbe, Thorsten Schubert, Claus Brunzema, Frank Oppenheimer. OSSS Channels: Modeling and Synthesis of Communication with SystemC. In *Proc. FDL'06*, pages 327–343, 2006.

[13] W. Klingauf, H. Gädke, and R. Günzel. TRAIN: A Virtual Transaction Layer Architecture for TLM-based HW/SW Codesign of Synthesizable MPSoC. In *Proc. DATE'06*, pages 1318–1323, 2006.

[14] W. Klingauf, R. Günzel, O. Bringmann, P. Parfuntseu, and M. Burton. GreenBus: A Generic Interconnect Fabric for Transaction Level Modelling. In *Proc. 43rd DAC*, pages 905–910, 2006.

[15] E. A. Lee and D. G. Messerschmitt. Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing. *IEEE Transactions on Computers*, C-36(1):24–35, Jan. 1987.

[16] E. A. Lee and S. Neuendorffer. Actor-oriented Models for Codesign: Balancing Re-Use and Performance. In *Formal Methods and Models for System Design*, pages 33–56. Kluwer Academic, Norwell, MA, USA, 2004.

[17] E. A. Lee, S. Neuendorffer, and M. J. Wirthlin. Actor-Oriented Design of Embedded Hardware and Software Systems. *Journal of Circuits, Systems, and Computers*, 12(3):231–260, 2003.

[18] Open SystemC Initiative (OSCI). OSCI SystemC TLM 2.0 Draft 1 for Public Review. http://www.systemc.org/web/sitedocs/TLM_2_0.html.

[19] C. A. Petri. Interpretations of a Net Theory. Technical Report 75–07, GMD, Bonn, Germany, 1975.

[20] A. Rose, S. Swan, J. Pierce, and J.-M. Fernandez. Transaction Level Modeling in SystemC. OSCI TLM Working Group, 2005. http://www.systemc.org.

[21] D. Shin, A. Gerstlauer, R. Dömer, and D. D. Gajski. Automatic Network Generation for System-on-Chip Communication Design. In *Proc. 3rd CODES+ISSS*, pages 255–260, 2005.

[22] D. Shin, A. Gerstlauer, J. Peng, R. Dömer, and D. D. Gajski. Automatic Generation of Transaction-Level Models for Rapid Design Space Exploration. In *Proc. 4th CODES+ISSS*, pages 64–69, 2006.

[23] F. Vahid, T. D. Le, and Y.-C. Hsu. Functional Partitioning Improvements Over Structural Partitioning for Packaging Constraints and Synthesis: Tool Performance. *ACM Transactions on Design Automation of Electronic Systems*, 3(2):181–208, Apr. 1998.