Why we need Standards for Transaction-Level Modeling

Wolfgang Klingauf, Ulrich Golze
TU Braunschweig, E.I.S.
{ klingauf, golze } @ eis.cs.tu-bs.de

Mark Burton
GreenSocs Ltd.
mark.burton @ greensocs.com

1. INTRODUCTION

Transaction-level modeling has been touted to considerably improve productivity in System-on-Chip design. Recently many popular SoC development environments have been flavored with the spirit of TLM, typically based on the favorite design language for TLM, which seems to be SystemC. Indeed TLM fabrics for SystemC spring up like mushrooms in the EDA community. The promise is that with transaction-level prototypes, busses and networks-on-chip (NoC) can be simulated magnitudes faster than with RTL models, while achieving almost the same accuracy of simulation results, and all this early in the design cycle where decisions made on these results have the biggest impact. But, does TLM live up to the hype?

To stand a chance, TLM engineers need to hit the ground running – they need their models to work fast. The common goal for the fabrics that are being offered is to ease the construction of model-to-model communication. The communication mechanism is seen as being relatively complex, common to many models, and above all, if it were common, then re-use of models might be possible. The key word is "interoperability".

TLM designers are spoilt for choice. Reviews of TLM fabrics reveal a diversity of opinions about what a TLM fabric should look like, and each TLM fabric boasts another set of features and limitations. In an attempt to get a general idea of the *status quo* in transaction-level modeling, we consider three main aspects of TLM fabrics:

- The API ("User View")
- Supported levels of communication abstraction ("TLM View")
- The techniques used for the implementation ("Technical View")

In the following, these three "views" on TLM fabrics are discussed and major differences in transaction-level modeling are pointed out. From this list of key issues we think standards in TLM should emerge. We will consider the proposed OSCI standard (TLM 2.0) and examine ST Microelectronics TAC framework (available from www.greensocs.com) , IBM's CoreConnect (available from www.ibm.com) OCP-IP (available for research from www.ocpip.org), OSSS (available from www.offis.de), OCCN (available from occn.sourceforge.net), and finally GreenBus, a collaborative open-source project that attempts to take the best of all of these ideas, and available from www.greensocs.com.

2. THE USER VIEW

From the user's point of view, the API (presumably) is the most important part of a TLM fabric. Its basic task is to provide a convenient interface with which to perform transactions over the TLM interconnects. Reviews show that there is industry wide cohesion, with transactions being either read or write transfers initiated by a master and processed by a slave. However, the API calls required to set up and perform transactions differ significantly.

ST's TAC package for example, which is based on OSCI's TLM framework for SystemC, provides simple read and write methods that transport arbitrary datatypes over point-to-point channels and abstract bus models. TAC operations are blocking and they are not designed to support advanced control of communication behavior such as specification of bus access priorities or byte-enables. Thus, TAC is a good choice for high-level models aiming at early embedded software development, but it does not support communication refinement towards lower levels of abstraction, thus architecture exploration with TAC is limited.

IBM's CoreConnect models for SystemC, on the other hand, provide a precise simulation environment for CoreConnect bus architectures. To this end, a tailor-made CoreConnect API with a comprehensive set of communication methods has been developed. This API perfectly assists designers who deal with CoreConnect bus interfaces everyday, but is inconvenient for others. Thus, IBM's TLM fabric is an excellent tool to create simulation models of already existent CoreConnect chips, but is less appropriate for top-down SoC design.

The two examples show that ideally the API is completely decoupled from the underlying TLM fabric. If, e.g., the IBM models were equipped with a TAC API, that would allow TAC-based SW models to communicate with CoreConnect IP over a precisely simulated bus architecture. Put another way, ideally, a designer should be at liberty to choose which interface suits his needs best, independently of the technology and details of the communication mechanism. A designer used to "speaking" CoreConnect would rather use the CoreConnect API, while a different designer may prefer TAC.

This sounds idyllic, and if the underlying technology is insufficient to support the API it is unrealistic, but it remains one of the key areas in which TLM fabrics can help increase engineers productivity, and get models out quicker. If TLM is to live up to the hype, then one key is providing the TLM engineer with a choice of environments that they can choose from in order to better support their specific style and their specific needs, in which they can operate effectively and write

their models quickly. This notion also sits tantalizing on the edge of the industry buzzword of the moment – interoperability. If the API is independent from the communication fabric, then theoretically, swapping out one communication fabric for another should be possible, models should become interoperable. Again, an idyllic notion, and not practical unless there is common agreement about the data being passed across the transaction interfaces. These issues will be revisited below.

3. THE TLM VIEW

The accuracy of simulation results achievable with transaction-level models depends on the implemented level of abstraction. In a SoC model, different aspects of abstraction can be considered:

- Time abstraction
- Data abstraction
- Structure abstraction
- Computation abstraction

Communication abstraction is closely related to both time and data abstraction. In figure 1, two fictitious bus transactions are shown. A master sends data to a slave. The transactions are initiated by a clocked request signal. After the bus arbiter grants access, the master sets the target address, which is acknowledged by the slave. Then data transmission is performed. The first transaction is a single-beat transfer, whereas the second transaction is a burst transfer.

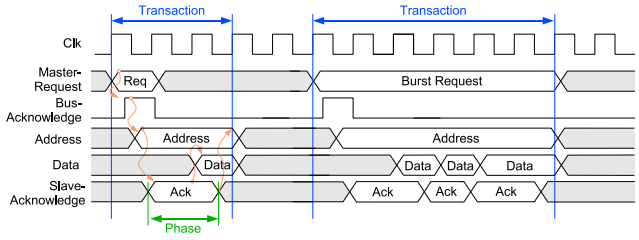

Figure 1: Bus transactions

Now we can model this communication at different levels of time abstraction:

None: No attempt is made to record any of the time points whatsoever. Complete data records can be transmitted by single method calls and simulation speed is very high, which is convenient to software developers. There is a problem with ensuring that multiple masters (i.e. concurrent processes) in a simulation progress at realistic rates and one is not blocked waiting for another. This is typically termed system synchronization. It can be implemented in two ways, either by keeping a very rough estimate of time and ensuring that each master "yields" after some time to the others. Or specific "synchronization points" can be coded into the model. These are points at which synchronization is required, and hence it's only here that other master need to be given the chance to execute. In real-time operating systems (RTOS), this is supported by message queues, semaphores, and special events. The former method has the advantage of being simple to code, but it is always sub-optimal, requires a notion of time (which is otherwise meaningless in an un-timed model), and removes the need for the system designer to think about where synchronization happens in the system. However, it is a good idea to categorical avoid such a modeling style, because it often will result in models that boast non-deterministic behavior.

Transaction accurate: TLM fabrics allow for modeling of synchronized communication between both hardware and software processes. At the transaction accurate level of abstraction, a lightweight TLM API is used to describe module to module communication, such as provided by ST's TAC package or the FIFO channels that are part of OSCI's TLM framework. Here, only transactions as a whole are of interest. That is, only the transaction boundaries (start and end times) are considered. Once started, an atomic transaction cannot be interrupted. Transaction phases such as request and address are not taken into account. This is often used to implement a rough time calculation. Transaction accurate models are proposed as an appropriate entry point for systematic embedded software development. Although the TLM APIs provided at this level of abstraction are quite easy to handle (TAC: read, write, OSCI: put, get), they anyhow are rather unusual for software developers who would prefer object-oriented modeling based on application-specific method interfaces. However, user experience shows that software modeling at the transaction level is convenient and thus this abstraction level often is referred to as *Programmer's View* (PV).

Bus accurate: Here the simulation accuracy is determined by the different phases a transaction is comprised of. Typical phases in bus communication are request, address, data, and acknowledge. For most busses and also NoCs, this level of abstraction is sufficient to produce precise simulation results, since arbitration of request phases is considered independently of the other communication phases, so that, e.g., a high priority master can cut of a lower priority transaction. While sufficient for communication simulation, the *Bus Accurate* (BA) level of abstraction does not support cycle-timed simulation of a whole SoC model, because masters and slaves are not aware of the individual time points when data chunks are sent and received. Thus, BA models are the ideal means for communication architecture exploration, but are not precise enough for the final verification of a fully integrated SoC model. The BA abstraction level sometimes is referred to as *Programmer's View Timed* (PVT), which often causes confusion, since the term PVT has also been used for PV models which boast basic delay approximation features.

Cycle-count accurate: At the Cycle-Count (CC) level of abstraction each phase is further chunked into its individual data transfers. Thus, the number of clock-cycles necessary to complete a phase can be accurately calculated, and precise simulations of any SoC interconnect are rendered possible. Nevertheless CC models perform considerably faster than native RTL models, mostly because the data and functionality used is typically "abstracted" to being more suitable to execute on a host platform, rather than being an accurate reflection of the hardware implementation.

3.1 Ambiguity and incompatibility

Interpretation of communication abstraction is ambiguous in the TLM fabrics we have reviewed. For example, OSSS channels in principle allow for cycle-count simulation of bus architectures, but the provided bus models do not support combinatorial arbitration of concurrent request. ST's TAC package provides a PVT mode of operation delivering an accuracy somewhere between PV and BA. OSCI's new TLM framework 2.0 (which currently is available for public beta

review) basically supports all abstraction levels, but it is unclear how they should be implemented. Some PV level request and response data structures are defined, while those for timing are left to the user to determine (though the expectation seems to be that the PV level request and response structures would be re-used). IBM's CoreConnect models only support a CC mode of operation, but from the documentation it does not become clear how accurate these models are. The OCP channel library provides the most comprehensive coverage of abstraction levels. However, the adapters by which OCP channels of different abstraction levels are connected are not made public, and different OCP configurations may result in incompatible behavior.

The plurality of approaches results in the obvious problem that transaction-level models of different designers are not compatible. Moreover, even models created with the same TLM fabric may not fit together, because the designers used incompatible configurations or different levels of abstraction.

The whole extent of the problem becomes apparent when we take data abstraction into account. There is no standard for the representation of data in transactions. Many TLM fabrics allow for individual configuration of the transported data types by means of template parameters. Other frameworks use fixed data types in their interfaces. As a result, hoping for interoperability of transaction-level models designed by different companies today unfortunately seems futile.

4. THE TECHNICAL VIEW

Many techniques for the implementation of transaction-level communication have been proposed. In general, all TLM fabrics we have reviewed try to make the most of the features of the underlying system-level design language, that is SystemC or SpecC. The interface between modules and the TLM interconnect is realized by ports. Ports sit at the boundary of modules and can be bound to channels, thus providing a means for user processes to invoke channel functions by interface method calls on the port. That is, the channel implements the TLM API, and the port acts as intermediary in channel access. While first communication architecture models based on the model-port-channel approach (such as the Simplebus example that is provided with the open-source SystemC kernel) were of quite simple nature, today's sophisticated bus and NoC simulation fabrics are comprised of dozens of interacting classes that are often built of complex and interwoven class hierarchies. Features include comprehensive transaction monitoring and debugging, global memory and register models, system-wide address space management, as well as extensive configuration capabilities. Thus, the original basic ports and interfaces are snowed under with extensions and modifications, of which makes clear why compatibility of different TLM fabrics is so hard to achieve.

5. GREENBUS – TOWARDS A STANDARD FOR TLM FABRICS

We believe that significant productivity gains in TLM-based SoC design can be achieved if there would be industry-wide agreement on the fundamental techniques of TLM fabrics. Our research shows that, the part of the TLM fabric implementations that in fact needs to be standardized is surprisingly small. Experiments revealed that heterogeneous IP cores at different levels of abstraction and with different convenience APIs can work seamlessly together over various TLM interconnects if the following technical fundamentals are carefully considered:

- Data representation
- Transport mechanism

The outcome of this research is the open-source GreenBus framework for SystemC, which is based on the results of intensive and ongoing collaboration of numerous TLM designers in both industry and academia. In GreenBus, all pieces of information that are related to a transaction are stored in a *transaction container* (fig. 2).

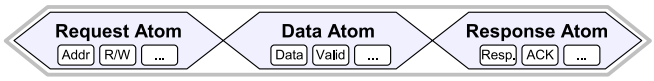

Figure 2: GreenBus Transaction Container

This container is a generic data structure comprised of the following parts:

Atoms: We believe that every transaction can be composed from a number of so called "atoms". An atom is the smallest uninterruptible part of a transaction that once started will complete its lifecycle. We have chosen "atom" as a neutral term, others have used different names, for instance OCP refers to these as transfer phases. All the bus and IO structures we have seen can be represented as containing transactions with up to just three different atoms: init atoms, data handshake atoms and finalize atoms. The init atom carries all the transfer qualifiers, after its completion both master and slave are ready to exchange data. The data handshake atom is used to transfer write or read data and all accompanying qualifiers like byte enables or error flags. The finalize atom finishes the transfer and can carry final responses or information needed to release the connection properly. It is possible that a transaction does not use all atoms, e.g. there are busses that won't need a finalize atom, a simple IO interface may only use one atom.

Quarks: We refer to the payload carried by the atom as "quarks". A quark is nothing more than a basic data type. A fundamental principle of GreenBus is that quarks are predefined. Again, other bus fabrics have similar notions. We simply suggest that for all features of a bus there should be a one-to-one mapping of feature and underlying transport type. For instance, any bus capable of transporting exactly 64 bits of data should always use the same data structure to do so. This fundamental principle is the key to providing model inter-working with the minimum cost. The "quark" data types need not be exhaustive, as bus and IO features which are really unique will always require some interpretation between IP not designed to the same interface. In this case, inter-working will always come with some cost, hence standardizing on the types for unique features does not help. As an initial set, we are persuaded that the set of types defined by OCP is relatively comprehensive, with some minor additions.

To illustrate how the transaction container is used in GreenBus, fig. 3 shows the atoms and quarks of a transaction via the Processor Local Bus (PLB), which is part of IBM's CoreConnect.

Depending on the level of abstraction chosen for the bus simulation, the information carried by the transaction container may be sampled in different resolution. This is pointed

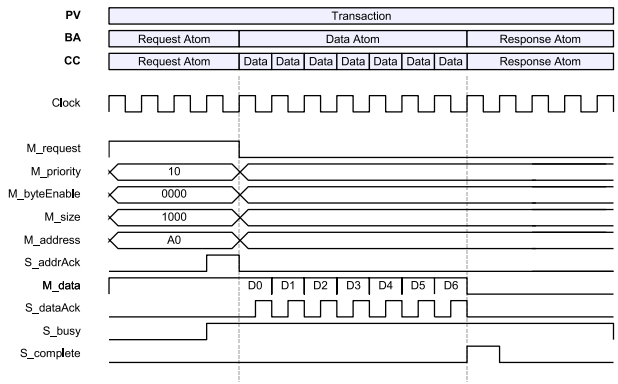

Figure 3: Representation of a PLB transaction with atoms and quarks

out at the top of fig. 3. For a PV model it is sufficient to consider the atomic transaction. At the BA level of abstraction, also the start and end times of the atoms are of importance. Finally, in a cycle-count accurate simulation (CC), each quark transfer will be considered individually.

A transport mechanism for transaction containers must support all three types of operation: atomic transactions, atom-based communication, and individual quark transmissions. By doing so, it should be fast (in terms of simulation performance), safe (in terms of data handling), easy to use (in terms of APIs built on top of it), and adhere to existing standards, i.e. OSCI's TLM proposal for SystemC.

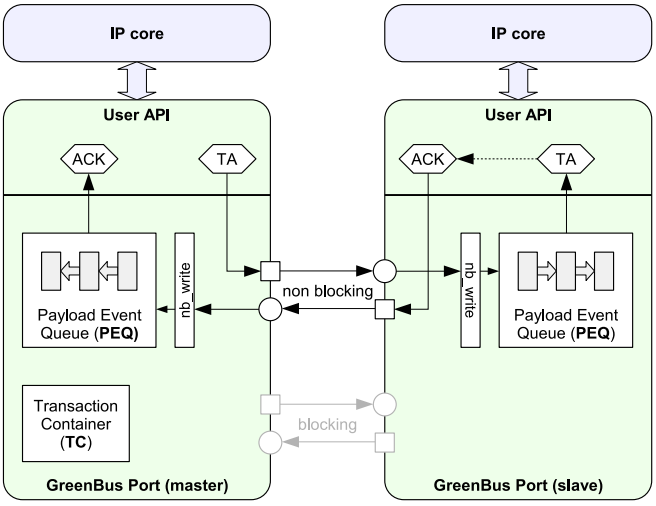

Figure 4: GreenBus basic ports

The transport mechanism we have found sufficient is based on *payload events*. A payload event provides the same functionality as normal events. That is, it can be fired by calling a notify method which will result in an activation of all processes that have been made sensitive to the event prior to its notification. The beauty of events in system-level design languages is that they can carry timing information. Their notification can be scheduled for the future. Thus, events are a perfect means for modeling latencies in module-to-module communication. Our experiments show that with event-based communication in a real system, a simulation performance close to method calls can be achieved. With the payload events implemented in GreenBus, in addition to the notification of the event itself a payload is transported from the initiator to the target. To this end, the basic communication channel in GreenBus is a *payload event queue* (PEQ). Using two PEQs, one in the initiator port and one in the target port, transaction containers can be transported back and forth between a master and a slave module. This approach is outlined in fig. 4.

Each payload event marks the start or completion of one atom. Thus, for a BA simulation of the PLB transaction shown in fig. 3, six payload events are required. To perform a CC simulation, however, a considerably higher number of events is required. But in this case it is not necessary to transfer the atom for each quark again. Instead, only simple events are used to indicate a quark's arrival. The payload itself is simply a reference to the transaction to which this event is related. Thus, optimal simulation performance is achieved while providing very high modeling flexibility, which is vital to support the creation of virtually any communication architecture based on this concept.

It is important to point out that the transaction container itself is never moved during a transaction. During its entire lifecycle, it resides at the master port. Payload events only carry a reference to the transaction container. By using *smart pointers*, safety is maintained. They do not release a transaction container until it is made sure that no process uses it anymore. Thus, communication in GreenBus always uses the concept of *pass-by-smart-reference*, which is considerably faster than *pass-by-value* (since no data copy is required) and significantly more secure than simple *pass-by-pointer* techniques as used in TAC and the OSCI-TLM 2.0 draft (of course, we are working with OSCI to improve that!).

Access to the information in the transaction containers is granted by getters and setters. Different access sets have been defined. While a master port may read and write all quarks of the request atom, it is not allowed to write the quarks inside a response atom, which is under the slave's responsibility. This provides a basic safety guard to detect inaccurate usage of the TLM fabric.

Communication with payload events is non-blocking. For PV modeling, GreenBus also provides a blocking "bypass" interface (indicated by the gray arrows in fig. 4). The PV interface provides a `transport` method call similar to that proposed by OSCI-TLM. It can be used for high-performance simulation of un-timed transactions. Both non-blocking and blocking interfaces are based on SystemC basic ports (`sc_port` and `sc_export`).

This simple architecture allows for building highly complex bus architectures that enable seamless interoperability of IP that

- reside at different levels of abstraction,
- boast different interfaces, and
- use different APIs to access the TLM interconnect.

Fig. 5 shows a typical use case of GreenBus where a PLB IP is connected to a TAC IP over an OPB bus. The OPB is simulated by a bus protocol class. To develop such bus simulation models, the GreenBus kit provides a *Generic Router* and a set of *Schedulers*, with which various bus protocol simulators can be set up quickly. Access to the interconnect layer is provided by so called *User APIs*, which translate the GreenBus API into the IP's APIs. To ease the creation of application-specific User APIs, we have developed a *Generic Protocol* on top of the basic PEQ and PV interfaces. This

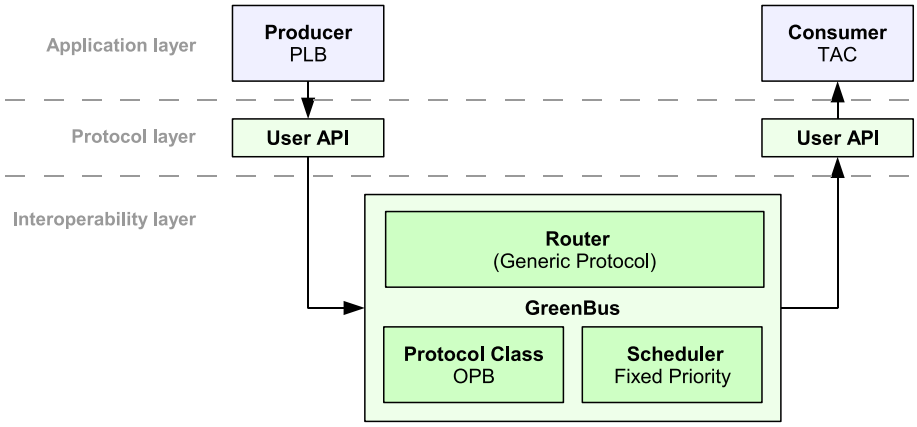

Figure 5: GreenBus use case with a PLB and a TAC core talking via OPB

protocol provides an implementation of elementary communication phases and has been shown to reduce the complexity of User APIs significantly. For example, a TAC User API for GreenBus requires 48 lines of code and one of our OCP-tl1 User APIs requires 56 lines of code (we chose a single-request multiple databurst OCP-tl1 API as an example).

6. CONCLUSION

So, does TLM live up to its hype. For some, maybe. For many, caught in the headlights of the dazzling array of TLM fabrics, and wondering which way to turn ? probably not. Within this, GreenBus is yet-another-TLM-fabric, but it does capture some of the best features of other fabrics, it is openly available, and it is a live project. It is therefore of more than academic interest. Its aim is to isolate the user from the underlying ?standard? interoperability interface, and to this degree, it may be of some help to people today. Our findings about that interoperability interface itself belong in the OSCI standard and GreenBus must adhere to that standard once it is formed.

So, where to next?

A casual bystander would be forgiven for concluding that this article presents a mess of different fabrics and options. It concludes that today interoperability in any meaningful sense of the word is a pipe-dream. However, we have also tried to give some hope. We conclude that the amount that is required to be standardized, in order to allow diverse IP to be reused, is small. We need a trivial API and a number of data types. We have also concluded that this will never be sufficient for a TLM engineer who will always wish to have an API to work with that is both convenient for their specific IP needs, and familiar to them. Hence our principle conclusion is that we need a SMALL standard (which may expose an unfriendly, but efficient interface), and a LOT MORE fabrics that use the standard (and provide the engineer with tailored APIs)!