OCP TLM for Architectural Modeling

Tim Kogel, CoWare, Inc.,

Anssi Haverinen, Nokia, Inc.,

James Aldis, Texas Instruments, Inc.

Abstract

Over the last years Transaction Level Modeling has established itself as a valuable strategy to solve system-level design problems prior to the implementation phase. The System-Level Design Working Group of the OCP-IP consortium has contributed SystemC TLM channel packages and methodology guidelines to help the proliferation of this modeling paradigm since 2002 [6]. Meanwhile, SystemC in general and TLM in particular have been improved and standardized by Open SystemC Initiative [1], [2], [3].

The predominant use models for SystemC based Transaction Level Modeling (TLM) are the highly abstract Programmers View (PV) [7], [8], [9] as well as fully cycle accurate modeling for verification and HDL co-simulation. However, so far no methodology for abstract performance modeling has been established addressing HW/SW partitioning and architectural exploration.

We propose an Architects View (AV) TLM use-model, which is based on the revised OCP Transaction Level 2 (TL2) and TL3 APIs. The goal is to allow SoC architects to identify and resolve bottlenecks in the organization of the SW and HW infrastructure. Questions to answer are the number of processors, how different pieces of SW and algorithms will access main memory and what interconnect approach can deliver the required communication bandwidth. The potential of the revised OCP TL2 and TL3 Channels goes clearly beyond modeling of systems, which will eventually lead to an OCP based implementation. Instead the new TL2 and TL3 APIs enable the unified modeling of arbitrary applications and architecture platforms above cycle accuracy.

We first give the overall picture of ESL design tasks and the corresponding TLM use-cases. The following chapters provide some technical background information on data-handling in TLM models and the OSCI TLM standard. The main body of this document is on the OCP based AV design methodology itself, introducing the revised OCP API, demonstrating compliance with OSCI TLM standard, and in particular explaining in detail the AV performance model. The last two chapters deal with the widely used Programmers View use-model and the interoperability of PV and AV.

This methodology document is accompanied by an extensive example package, which provides the source code of all mentioned components, transactors as well as numerous examples. Please refer to the html documentation in the package for more details on its contents and usage.

1 Introduction

We will first give an overview of the ESL design problems and the corresponding TLM use-cases. The second part of this chapter shows the overall picture of our methodology and explains the orthogonalization of modeling aspects as the enabling paradigm to achieve reuse of models across multiple TLM use-cases.

1.1 ESL Overview

TLM use-cases for ESL design tasks

Transaction level modeling should support the complete range of ESL design tasks. The key reason for this is specified in the requirements for TLM modeling: we're looking for a modeling environment to create a SystemC model for your design that is fast enough for SW development, architectural exploration and HW-SW verification. With that we cut the overall design time compared to doing these tasks in RTL, emulators, FPGA prototypes, etc. based on RTL implementation models you still cannot do without. Since TLM is not used for implementation the time spent in creating TLM models should be minimized. On the other hand, since TLM is not used for implementation it is possible to abstract certain details in order to ease the modeling effort and improve simulation speed. We need to be careful when removing detail from the model in order to make sure that it still can be used to solve the design problems. Thus the strength of TLM is also its Achilles' heel. TLM is not a straightforward set of modeling constructs and coding style, it is a flexible approach that allows creating the right model for each design task. In order for TLM to be efficient you need to *minimize effort to maximize effect*. Model only what you need for each design task; only include or enable the pieces of a model that are important for the current design task.

In order to characterize the requirements for the different design tasks we propose the definition of the following TLM use cases:

Functional View (FV): This is a TLM model, which is used for the *executable specification* design task; the focus is on algorithms. This implies that functionality is modeled accurately but still independent of any implementation. FV modeling can of course be done by using plain C++ or other sequential programming languages, however SystemC adds concurrency and separation of communication and behavior to the picture. Depending on the kind of application, there are different flavors of communication model for FV. Algorithm centric applications deploy data-flow semantics, i.e. the processes communicate using FIFO channels. On the other hand, control flow dominated embedded applications rather use a Remote-Procedure-Call (RPC) kind of mechanism.

Architects View (AV): This TLM use case targets *architectural exploration*, i.e. the definition of the SoC platform architecture as well as the partitioning and mapping of the application onto the platform. The application may be available as a FV SystemC model or may be represented in terms of non-functional workload models and traffic generators. On the other hand, the architectural elements like communication nodes, memories, peripherals should be available as performance models. For the purpose of architecture exploration there is usually no need for 100% timing accuracy since the impact of an architecture change is on a much bigger scope than a single clock cycle. Still an accuracy of 70-80% needs to be maintained to ensure the quality of the analysis results. However the architect's responsibility also includes defining the performance that the hardware is to achieve. In such cases the architect must define the cycle-by-cycle behavior for (parts of) the hardware he/she is specifying. Please refer to section 1 for more information on the Architects View.

Programmers View (PV): This is a TLM model created to enable *Embedded SW design*. Again this can be a further refinement of previous models. Here the functional correctness is important for the elements of the model, at least as far as there is impact to the Software. Other parts of the system may be excluded or abstracted to save modeling time and to improve simulation speed. For a SW designer it's important to have the memory map correctly modeled. Please refer to section 1 for more information on the Programmers View.

Verification View (VV): This is a TLM model built to enable *HW-SW and system verification*. In this case a model for the system is used to validate the implementation models of the different HW blocks in the context of the system, as well as to validate the SW with actual implementation models for the hardware components. For this purpose timing should be modeled fully accurately (100%) and it should be possible to interface with the HDL implementation models.

In this list, the Functional View stands somewhat apart from the other three TLM use cases, because it is used for model the application. Programmers, Architects and Verification View on the other hand are used to model the platform architecture.

Please note, that these use-models have not necessarily a one-to-one correspondence with a particular abstraction level. Instead, a use-model defines the requirements imposed on a transaction-level model to be adequate for a certain purpose.

General requirements

As we will see in the next section, transaction level modeling bears the potential to address the challenges of various ESL design tasks described above. However, a number of requirements need to be fulfilled to motivate implementing a SystemC based ESL flow in addition to a traditional RTL implementation flow. Next to some general requirements like standards compliance, simulation speed and modeling efficiency, a TLM based methodology guideline needs to foster model reusability. Given that the ESL tasks have different accuracy requirements, it is obvious that the modeling style needs reflect that.

Specific requirements

The general requirements for modeling efficiency, accuracy, simulation speed have already been mentioned. In Table 1, these requirements are refined to the specific constraints of the different TLM use-cases and rated for importance.

Requirement	Programmers View	Architects View	Verification View
Simulation Speed	++	+	0
Functional Accuracy	+	0/+	++
Timing Accuracy	0	+/++	++
Flexibility	-	++	0

Table 1: Modeling requirements of TLM use-cases

In this table, the flexibility refers to the turn-around time in the model development cycle, i.e. the effort it takes to change the platform model, re-run the simulation and evaluate the results. Hence the flexibility of a model covers different aspects like compilation time and configurability. The flexibility is of major importance for architects in order to efficiently explore architectural choices.In an ideal world, one would of course like to have highest simulation speed, accuracy and flexibility the same time. However, these requirements are contradictory to quite some extend, so we must cut back in areas not vital to the respective use-case. For example the simulation speed is of an utmost importance for the SW developer, whereas full cycle accuracy is often not essential.

As can be seen in the third column, the accuracy requirements for the Architects View use case are not entirely fixed. The grand decisions like the number of CPUs, the cache size, the amount of on-chip memory can be taken based on non-functional, cycle-approximate models, which provide highest flexibility for design space exploration. The fine-tuning of certain architectural parameters or the detailed definition of the performance however might require a fully cycle accurate model. In the latter case the accuracy requirements for the AV use-case become as stringent as for the Verification View use case.

Another notable observation in Table 1 is , that AV is not necessarily situated at a higher abstraction level than PV. In fact PV and AV are different abstractions of the some platform, which serve different purposes. Generally, PV puts more emphasis on functional accuracy, whereas AV requires more timing. Yet PV is also not completely un-timed, since the SW developer needs some timing information, e.g. to program a timer or to consider the synchronization in a system with multiple initiators.

The limited demand for flexibility in the VV use-case refers only to the TLM reference model itself being used for cycle-by-cycle equivalence checking against RTL and not to the surrounding testbench. Of course most bugs are fixed by changing the testbench, so the full flexibility of SystemC and C++ should be used to generate the stimuli.

The goal to achieve a reasonable degree of reuse gives certain directions for the modeling style: The initial AV model could do with an abstract performance model. On the other hand, PV and VV require memory-map true and functionally accurate peripherals. Therefore the register interface should be added to the peripheral model rather sooner than later to foster reuse. Indeed the register interface can be seen as a generic layer between on-chip communication and behavior. On the other hand, PV models should already be prepared for the annotation of some timing information.

TLM based ESL Design Flow

When designing an SoC, the selection of a design flow is dependent on the design tasks that will be needed. An example design flow is depicted in Figure 1. In this flow functional models are created for the different algorithms that will be used in the design. These functional models are used by architects to define the right HW/SW partitioning and to explore the system architecture. The result is a high-level, functionally correct model that can serve as an executable specification for the design. From this model a SW development model can be extracted. The goal here is to remove any detail that is not important for the embedded SW designer, this in order to achieve acceptable simulation speeds to allow for SW development and debugging. The same executable specification can be refined to a more accurate model that is used by verification engineers.

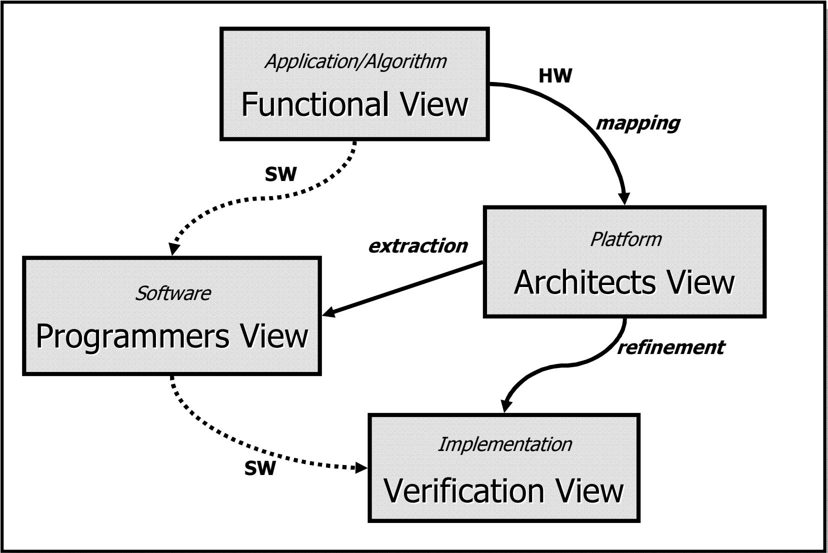

Figure 1: TLM based ESL Design Flow

1.2 Orthogonalization of Concerns

The goal of the modeling methodology described in this document is to create SystemC models of peripheral components in SoC platforms. The previous sections state the requirements as being modeling efficiency, reusability for all theses ESL design tasks and of course simulation speed.

So how can we hope to achieve our goal under consideration of all these demanding and conflicting requirements?

Our approach is based on two fundamental observations:

The first observation is that the abstraction level of the SoC platform model is mostly influenced by the abstraction level of the deployed bus model. More specifically, the accuracy and simulation speed inherent to the chosen model of the communication architecture determines the aptitude of a TLM simulation model for a specific use-case. This proposition assumes that the Instruction Set Simulators representing the programmable processing elements are sufficiently fast.

The second observation is that the interface based design principle of SystemC and TLM enable a mutual separation of communication, behavior, and timing. In other words we can decompose the problem of modeling a platform element into several orthogonal aspects.

These two general observations lead to the modeling strategy depicted in Figure 2. The three different bus models on the left hand side represent the fact that the accuracy and the resulting simulation speed of the communication model determine the use-model of platform simulation. Usually these different abstraction levels are not deployed in a single platform simulation. Instead, the three bus models represent the evolution of the platform model from a purely functional Programmers View system via a timing approximate TL2 or TL3 model and finally a fully cycle accurate TL1 model.

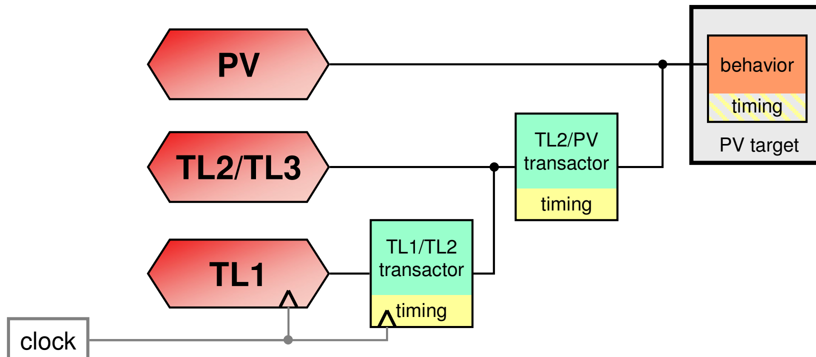

Figure 2: Methodology approach

We presume that the average user will be mostly concerned with the behavior and timing for his particular platform IP, because these parts are really specific for the current design. The bus-transactor together with the bus model itself is the responsibility of the provider of the bus IP.

The separation of different aspects of the design process is generally considered to be the key ingredient to tackle the complexity of SoC design in a divide-and-conquer kind of approach. In addition to the well-known concept of separation for *behavior* and *communication,* we propose to

further separate the *behavior* and *timing* of communication (in a TLM model). That way a TLM model can be hooked to a model of a different bus (or to a model of the same bus at a different abstraction level) by just replacing the bus transactor. Independently the accuracy of a model can be increased by adding information to the timing compartment of a model. Ultimately this leads to scalable accuracy, where the fine-granular separation of the different modeling aspects enables a seamless reuse of models across multiple levels of abstraction.

Separation of communication from behavior

From the modeling perspective the behavior represents the critical part of any peripheral platform component. Basically every user of a SystemC will have to create or will be exposed to the behavioral model of the platform component or application model. Therefore the efficient creation and reusability of the behavior is of utmost importance for the success of a SystemC based ESL design flow. This section describes our strategy to optimize the reusability of the behavioral model.

By keeping the communication aspect of a module separate from its behavior we can "plug" a given behavior to any platform communication requirements. Hence we are able to re-use the behavior part of the model at different abstraction levels and with different communication protocols. We advocate the Programmers View (PV) API introduced in the section 1 to be used as the generic behavioral interface. The PV API is indeed the simplest interface, as it consists only of one single interface method.

In general, a transactor is a device that converts TLM transactions from one type of interface to another. In the context of modeling platform peripherals, the bus transactor translates the bus protocol into the generic behavioral interface.

Very often a bus-transactor has to adapt two different levels of abstraction. The accuracy of the system simulation is mostly determined by the accuracy of the communication model. Therefore the same bus architecture might be modeled at different levels of abstraction, for example at the AV level for architectural exploration as well as at the VV level for verification and HDL co-simulation. On the other hand, the generic interface of the storage and synchronization layer is completely abstract, protocol agnostic, and (almost) without any timing information. This mismatch in the timing accuracy also needs to be mitigated by the bus transactor. The creation of bus transactors represents a more advanced topic and is exemplarily discussed in chapter 1.

Separation of timing from behavior

Again, our goal is to reuse the behavioral part of the model over multiple levels of abstraction. This is possible if we are careful to construct the model such that timing can be introduced into the behavior as and when it is required. In TLM the timing information is modeled either explicitly, implicitly, or based on clocks.

- *Explicit Timing Annotation* means that timing is modeled inside the behavior by using events and event synchronization or other usual SystemC timing modeling mechanisms. The advantage of this approach is that the created models more closely relate to the actual hardware, and that the internal timing of a block can be modeled more accurately.

- *Implicit timing Annotation* means that timing accuracy is achieved through timing annotation in the TLM API calls. Next to the improved simulation performance an additional benefit of this approach is that the timing annotation is orthogonal to the functionality in order to make sure that a system can be refined for timing information without breaking the previously validated functionality.

Clock based timing adopts the RTL modeling paradigm, where the behavior is modeled in terms of clocked processes. It is therefore intuitive for every SystemC user with a Hardware

background that this can achieve 100% cycle accuracy. On the downside, this modeling style is quite cumbersome, not at all orthogonal, and a single module of this kind can potentially ruin the simulation speed of an otherwise very fast platform model.

To achieve our goal to reuse the behavior at different levels of timing accuracy, implicit timing annotation is a preferable modeling style. Of course this requires specific support from the TLM API, but this is provided by the PV and AV APIs discussed later in this document.By using implicit timing annotation the modeling of the actual delays is deferred from the peripheral behavior into other parts of the system, like e.g. the bus-transactor of the bus model itself. Components actually performing the timing are for example the bus transactor, the communication channel or the bus model.

In practice it might prove difficult to reuse the model at multiple abstraction levels without any modifications. Therefore the timing modeling strategy should be considered and prepared right from the beginning of the creation of the TLM model. In general this timing information may not be used at some abstraction levels but later may be required at others.

In the model this can be done by including separate functions for the delay calculation, which by default just return zero. Later these timing functions can be refined by returning a configurable constant value, a statistic delay value using configurable constraint randomization or a delay calculated from the internal state and the actual data. By using implicit timing annotation the decision to deploy or ignore the timing information is still deferred to other components, so a refined model can always be used in a more abstract context.

As an example, take a memory model that has wait-states. The clock period and the number of wait states are supplied as parameters to the model. For a purely un-timed system we may choose to ignore the latency caused by the wait-states, such that transactions to the memory will occur in zero time. However, in a later version of the system, we may wish to account for these delays and start to make use of the additional timing parameters. By using implicit timing annotation the latency is passed as an attribute of the transaction in the generic behavior interface. Depending on the selected bus-transactor or connected bus model, this timing information can be ignored ore considered. Explicit timing annotation does not provide this kind of flexibility, except we obscure the behavior with conditional SystemC wait calls.

Limitations of implicit timing annotation

In some cases a single latency parameter might not be sufficiently accurate. For example, take the case of a pipelined model at PV level. The latency only models the overall delay for the total pipeline. However, if we wanted to go from un-timed to timed we also have to take into account that a new transaction can begin at the first stage once the previous transaction has been passed to the second stage, etc. In this case we would need a second implicit timing parameter to specify the initiation delay of the pipeline, which might not be available in the TLM API. Of course each stage of the pipeline could be modeled explicitly, but this would really intermingle timing and behavior. A better solution is to add the missing timing information by the bus transactor. The concept of a bus transactor with configurable timing annotation is discussed in section 1.

A second example might be a multi-port memory controller where arbitration is required on the incoming transactions. The problem here is that different communication protocols require different actions based on whether or not a particular port is granted immediately. For this type of models it is hardly feasible to separate the behavior and timing. In this case the behavior needs to be partitioned into a generic and a timing and protocol specific part. The generic part can still be reused, but the specific part of the model needs to be more or less re-implemented for all considered abstraction levels and communication protocols.

2 Data Representation in TLM models

There are two aspects related to the representation of data in TLM models, which are a major source of confusion and nasty bugs. The first aspect is given by the fact, that most TLM APIs provide the flexibility to template the data type and address type. The second aspect refers to the management of storage throughout the simulation.

2.1 Data-type Templates

Two different types of template arguments can often be found in most TLM APIs: Either the complete transaction data structure can be specified by the user, or the transaction data structure is given, but the user can still specify the type of the data and address attribute. The OSCI TLM API and the OCP TL3 API are representatives for the first group, whereas PV and OCP TL2 belong to the second group. The resulting flexibility comes at the expense of limited interoperability: Only TLM models with matching template arguments can be connected, so any bigger modeling project must establish rigid coding guidelines to ensure the model connectivity.

For the two different groups the reasoning for providing this kind of flexibility is completely different:

User defined transaction data structure

The undefined transaction data structure like in OSCI TLM greatly improves the applicability of the API for all kinds of use cases. The range of possible template arguments is quite comprehensive:

- Simple built-in data types like float are used in data-flow kind of FIFO interfaces.
- Abstract data types with user-defined members enable convenient modeling of applications.
- A pointer to the transaction data structure either optimizes the simulation speed and/or enables the use of polymorphic data structures.
- A shared pointer to the transaction data structure additionally avoids memory leaks.

Apart from the interoperability issue, the high degree of freedom provided here might also lead to further problems. For example using pointer types is dangerous in terms of thread safety and memory leakage, or using shared pointers requires a consistent coding style.

User defined type of the data attribute

The only purpose of a user defined type for the data attribute in the TLM transaction data structures is to pick the optimal data type depending on the required word-size. In any case the C++ built-in data types should be used where possible to optimize the simulation speed. The speed drops significantly when the TLM models deploy SystemC data types like sc_uint<24> for all their processing and communication. However, this convention fails if the required word-size is above 64 bits. In this case the unlimited SystemC containers like sc_bigunit<N> or sc_bv<N> have to be used. The choice between those should be taken based on the required operators.

Of course the interoperability problem is not that severe when only the data attribute is templatized. Still we can have annoying effects when connecting an unsigned to a signed data type, or when one 32 bit model uses unsigned int and the other sc_uint<32>. To improve the interoperability problem a strict type policy should be established. We propose to

stick to the built-in unsigned data types, which encompass the bit-width of the on-chip communication. The decision on which type exactly is used in the model should be based on the required data width.

required data width	recommended data type
0 < data-width ≤ 32	unsigned int
32 < data-width ≤ 64	unsigned long long
data-width > 64	no recommendation

Please note that this type policy refers to the data attribute of the transaction data structure, i.e. to the external communication of a TLM module. Inside the behavior the user is free to choose any type for local variables like e.g. signed integer or floating point kind of data types. It is then in the responsibility of the user to package the local data types into the unsigned integer types of the transaction data structure.

2.2 Storage Management

The organization of the memory is an important aspect of any modeling activity, because it has a major impact on the simulation speed and robustness of the models.

In the following we will discuss memory strategies, which are particular important for high-level TLM platform modeling. The typical problem in platform modeling is handling of the data that has to be exchanged between initiators and targets via an arbitrary number of bus nodes. In the cycle accurate Verification View this is a non-issue, since all data items are copied by value. However at higher abstraction levels whole chunks of data are moved along with a single transaction. Applying the same value-based communication would result in a lot of needless and time consuming copy operations. Instead using pointers to memories is much more efficient, but raises the problem of the ownership for the data storage.

Memory owned by initiator

As depicted in Figure 3 below in this memory strategy the storage for all transactions is owned by the initiator. It is responsible to allocate sufficient amount of data before sending out a read or write request and keep this storage valid until the transaction is completed. The transaction data structure contains a pointer to the storage in the initiator. Like this the target can immediately copy the data from its local memory to the initiator memory (in case of a read) or from the initiator memory to its local memory (in case of a write).

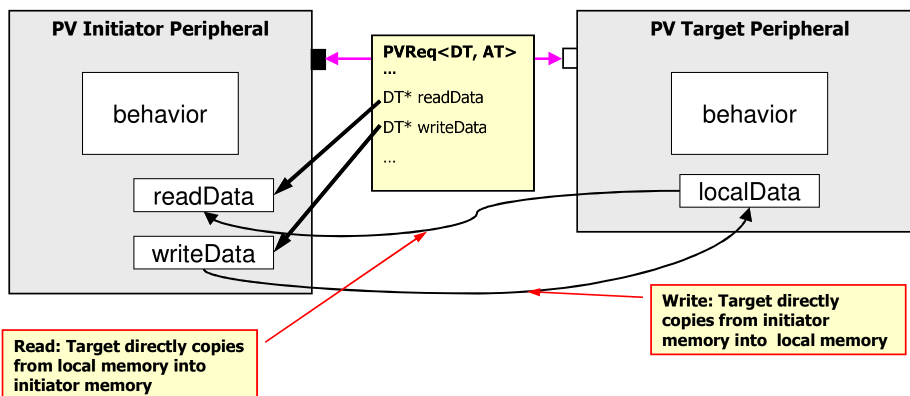

Figure 3: Memory owned by initiator

This concept is deployed in the Programmers View TLM use-case, because it yields the highest simulation performance. Even if multiple nodes are involved in the communication, there is only one copy operation per transaction. This performance advantage comes at the expense of the requirement that the initiator has to be carefully coded to keep the storage valid throughout the transaction. The pipelined sending of multiple requests would be particularly dangerous, but this pipelining is fortunately disabled by the bi-directional blocking semantics of the PV communication API (see section 1).

Memory owned by sender of data

This strategy depicted in Figure 4 is more suitable for split transaction schemes. Here the response is completely decoupled from the request, so the previous strategy would require quite some bookkeeping in the initiator to keep track of the storage of all outstanding responses. Instead the memory is always owned by the sender of the data, i.e. the write data is owned by the initiator and the read-data is owned by the target.

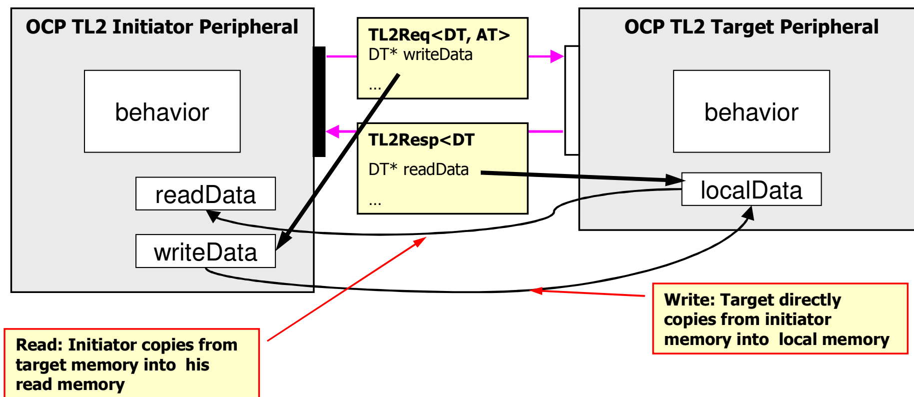

Figure 4: Memory owned by sender of data

Still this scheme requires some kind of acknowledge mechanism to control the lifetime of transactions. The respective sender needs to know how long the storage needs to be valid, i.e. by when the receiver has processed the data. For this purpose the OCP TL2 API (see section 1) limits the guaranteed lifetime of the data to a single point-to-point like between two modules. The downside of this safety precaution is the requirement to copy the data at each communication node, which slows down the simulation in complex multi-hop networks.

The decision between the strategies often depends on the selected TLM API. Particular care is required for the development of transactors, which might have to mediate between different memory strategies (see chapter 1).

Please refer to Appendix B of [2] for a discussion of storage management in the context of thread safety.

3 The OSCI TLM Standard

The foundation of TLM is defined by the OSCI TLM working group as *communication through function calls*. The goal is to abstract certain details of the implementation and work these out at a later stage in the design flow. Transaction-level modeling provides a way of minimizing the number of events and amount of information that has to be processed during simulation. Instead of driving the individual signals of a bus protocol the goal is to exchange only what is really necessary: the transaction attributes and the data payload. Since Transaction-level modeling also reduces the amount of detail the designer must handle, it makes modeling easier.

In RTL there is commonly a 'behavioral' abstraction style used. While this abstracts certain details of the computation below the clock cycle level, it does not abstract the communication interfaces. TLM is using function calls to encapsulate the details of the communication interface. This makes it easier to experiment and play with different communication architectures without having to recode all the models. This leads to a much more efficient coding style and faster simulation speed since unnecessary simulation events are suppressed. Using the SystemC interface methods mechanism complex transactions can be modeled by a few function calls.

For simple communication interfaces it still can be useful to apply signals, definitely in the cases where there is need for simple event communication between blocks and where there are no problems with the overwrite semantics of signals (a write never fails) and the fact that there are only events in case of a value change. In general however a TLM coding style will replace signals with function calls or event notification in order to eliminate the update cycle of signals and to be able to explicitly handle overwrite or blocking behavior in interfaces (to make sure the previous value is processed before a new write happens)

As depicted in Figure 5, TLM communication is defined as a set of layers. In its most primitive form the TLM standard API provides with the fundamental communication and synchronization constructs that can be used to create TLM models. Although these interfaces can be used in their primitive form, it is generally expected that they need to be complemented with a 'protocol-layer', which targets the basic interfaces to a specific on-chip communication protocol or design task. This protocol layer can provide with a set of convenience functions that create a TLM modeling abstraction that is much easier to use. The Programmers View API and the OCP APIs are exemplary representatives of such a protocol layer.

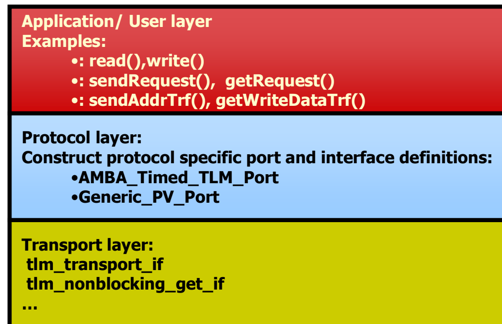

Figure 5: TLM communication layers

TLM Standard API

The basic communication mechanisms in the OSCI TLM standard are depicted in Table 2 [2].

The Bi-directional blocking interface, which we will refer to as the **Transport TLM** API, is a simple API call where all transaction information is communicated in one single call. It is allowed to use 'wait()' calls in the block implementing the transport interface, or to use signals and events. Basically this allows doing all communication immediately and then progress the simulation kernel. We suggest to use this API in a non-blocking fashion as much as possible. In this case the transport implementation does not use wait calls to synchronize with the rest of the SystemC simulation. This is useful when the behavior is simple register update behavior, and it allows further speed optimizations in the simulation kernel.

Uni-directional non-blocking interface, which we will refer to as the **Transfer TLM** API, is a bit more complex. It provides a function call for the data exchange as well as an event and a status call that allow synchronizing both ends of the communication. Usually two or more of these API calls are combined together to create a full transaction API. In its simplest form these are 'request' and 'response calls.

API Function	**Description**
Transport TLM: Bidirectional blocking TLM interface	
RSP transport(const REQ& req)	REQ data is provided by initiator and a RSP data contains attributes from target.
Unidirectional blocking TLM interface	
T get()	destructive read, blocks until data is available
T peek() const	non-destructive read, blocks until data is available
void put(const T &t)	append new data item, blocks until space is available
Transfer TLM: Unidirectional non-blocking TLM interface	
bool nb_get(T &t) const sc_event &ok_to_get() const bool nb_can_get() const	destructive read, return false if no data available notified when new data becomes available return false if no data available
bool nb_peek(T &t) const sc_event &ok_to_peek() const bool nb_can_peek() const	non-destructive read, return false if no data available notified when new data becomes available return false if no data available
bool nb_put(const T &t) const sc_event &ok_to_put() const bool nb_can_put() const	append new data item, return false if no space available notified when new space becomes available return false if no space available

Table 2: OSCI TLM standard API

Note that in the context of SystemC the terms 'blocking' and 'non-blocking' have a particular meaning. A blocking interface implies that this interface has to be called from within an SC_THREAD, as such the implementation of the interface is allowed to contain wait(.) statements. In contrast a non-blocking interface cannot contain a wait(.) statement since it is allowed to call such an interface from within an SC_METHOD which is not capable of performing the context switch that is required to implement the wait(.) call.

Please refer to [2] for an full introduction into the OSCI TLM standard and to [7] for more information about building protocol layers on top of the basic TLM primitives.

Standard Modeling beyond OSCI SystemC and TLM

SystemC is an open standard and enables all kind of the modeling styles. Transaction level modeling is designed into the language through the use of the Interfaces Method Call (IMC) mechanism. However there is a need for further standardization. The work of the OSCI TLM Working Group remains focused on standardizing the foundation layer of TLM communication APIs. On top of this, the System-Level-Design Working Group of OCP-IP defines user-level communication APIs.

4 Architects View

The key problems that an Architects View model needs to address is early architectural trade-off analysis for system design, and setting the right design constraints for HW and SW implementation teams. For SoC platforms the solution space for these problems is very large, and the problems get increasingly pressing in the network-on-chip and multi-processor era. As a consequence, designer experience is no longer sufficient to find the best solution. The goal for the AV modeling style is to enable the designer to evaluate different design decisions against each other in a quantitative way. The evaluation is done through simulation and analysis.

In order to define the HW/SW partitioning and the platform architecture according to the system requirements, a sufficiently accurate model of the processing, communication and memory requirements needs to be created. As a consequence the modeling style will focus on communication timing and resource sharing (point-to-point, shared bus, network-on-chip…) as well as on data size. The modeling style should promote the working with system requirements and not with the implementation details of these requirements. This leads to a representation of the systems, which deals mostly with high-level versions of the system functionality as well as with performance estimates of the anticipated architecture. On the other hand, many today's designs reuse legacy elements of previous designs. For this reason the modeling style should be open enough to allow insertion of more detailed implementation models like instruction set simulators and refined TLM models as well as functionally complete PV models.

4.1 OCP based Communication API

The Architects View use model requires a communication API, which supports timing approximate modeling of the platform architecture. On the other hand the API has to be agnostic to any particular bus protocol to enable the flexible and unbiased exploration of different communication architectures. In principle, these capabilities are delivered by the unidirectional non-blocking transfer API in the OSCI TLM standard. We propose to adopt the SystemC API of the Open-Core-Protocol (OCP) as a more convenient protocol layer for architectural modeling.

This choice is motivated by the following two reasons:

First, the Open-Core-Protocol defines a bus-independent interface to foster design reuse. Many popular bus architectures provide OCP bridges to integrate OCP compliant IP. This emphasizes the interoperability of OCP with all kinds of on-chip interconnect architectures, i.e. OCP can be considered a generic communication API. In that the OCP protocol meets the objective of the Architects View use model to be protocol agnostic.

Second, the System Level Design Working Group of OCP-IP has already defined a widely used SystemC based TLM API [5]. We only have to pick the most suitable of the three available levels of abstraction:

- The lowest level is called Transaction Layer 1 (TL1) and it enables fully cycle accurate description of the OCP protocol. This level is therefore not exactly protocol agnostic, but can be deployed in the Verification View use case.
- The next higher level is called TL2 and represents basically a cycle-approximate abstraction of the OCP protocol. The API is still pretty rich and contains a large number of OCP specific features like e.g. thread-busy, handshake-timing, or sideband signals.
- The Transaction Layer 3 represents the highest abstraction level of the OCP standard and can be considered as the protocol agnostic subset of TL2. The API is limited to a concise

set of primitives, which are essential to model timing approximate on-chip communication. All the functions and events in the TL3 API can be implemented on top of the non-blocking unidirectional OSCI TLM standard (see next section).

Obviously the TL3 API presents the best fit for the Architects View use case. It is compliant with the OSCI TLM standard (please refer to the next section). Additionally it is of reasonable complexity, and yet offers sufficient expressiveness to meet the accuracy requirements of the AV use case.

The TL3 API was first presented in the OCP-IP White Paper for SoC Communication Modeling [6] and implemented in the first release of the OCP-IP SystemC models [4]. The implementation was based on the underlying Generic Channel for OCP, which is not supported by the OCP-IP anymore, and is being replaced by the OSCI TLM. The latest OCP-IP TLM package (v2.1.1) does not include the TL3 for that reason. We are thus introducing a new, refined API for the TL3, which will be integrated with the OCP-IP TLM package in the later releases.

The TL3 Master API definition is listed and explained in Table 3.

API Function	Description
Regular Request Commands	
bool sendRequest(const REQ& req)	Puts a request on the channel. Returns true if the request was successfully placed on the channel. False otherwise.
bool sendRequestBlocking(const REQ& req);	Puts a request on the channel, waiting until the channel is free if necessary. Waits until the slave accepts the request and then returns. Blocking calls may only be called from SC_THREAD processes.
bool requestInProgress() const	True if there is currently an active request on the channel.
const sc_event RequestStartEvent()	Returns the event that is triggered when the master places a new request on the channel.
const sc_event RequestEndEvent()	Returns the event that is triggered when the slave accepts the current request.
Timed Request Commands	
bool sendRequest(const REQ&, const sc_time& t)	Delays the sending of the request for time t. Otherwise identical to sendRequest.
bool sendRequest(const REQ&, const int cycles)	Delays the sending of the request for cycles number of clock ticks. Otherwise identical to sendRequest.
Regular Response Commands	
bool getResponse(RESP& resp)	Gets a new response from the channel and returns true. Returns false if no new response transaction available.
bool getResponseBlocking(RESP& resp)	Waits for a new, unread OCP TL2 response to come on to the channel and then gets it. Can only be called from an SC_THREAD process.
bool acceptResponse()	Accepts the response immediately and returns true. Returns false if no response to accept.
bool responseInProgress() const	True if there is currently an active response on the channel.
const sc_event ResponseStartEvent()	Returns the event that is triggered when the slave places a new response on the channel.
const sc_event ResponseEndEvent()	Returns the event that is triggered when the master accepts the current response.
Timed Response Commands	
bool acceptResponse(const sc_time& t)	Delays accept by t SystemC time units from now. Returns false if no response to accept.
bool acceptResponse(const int cycles)	Delays accept by cycles OCP clock periods from now. If cycles=0 then the accept is immediate. Returns false if no response to accept.

Comment [AH1]: Are these absolutely necessary? This is a major departure from the TL2, and the original TL3 definition. Or should we introduce these to TL3 also?

Table 3: OCP TL3 Master Interface Definition

The conversion of integer cycles into SystemC time is based on the clock-period, which is a member of the channel. The slave interface is not depicted as it is perfectly symmetrical to the

master interface, with just Request and Response interchanged. See the full API in the methodology example package [11].

4.2 Mapping TL3 onto OSCI TLM

All the functions and events in the TL3 API can be implemented on top of the non-blocking unidirectional OSCI TLM standard. This section demonstrates the mapping of the TL3 primitives onto the OSCI TLM API. In a similar way, the complete TL2 API including thread-busy, handshake-timing, and reset, can be mapped onto the TLM API. The TL2-TLM mapping is also included in the methodology example package, but not discussed in detail in this document.

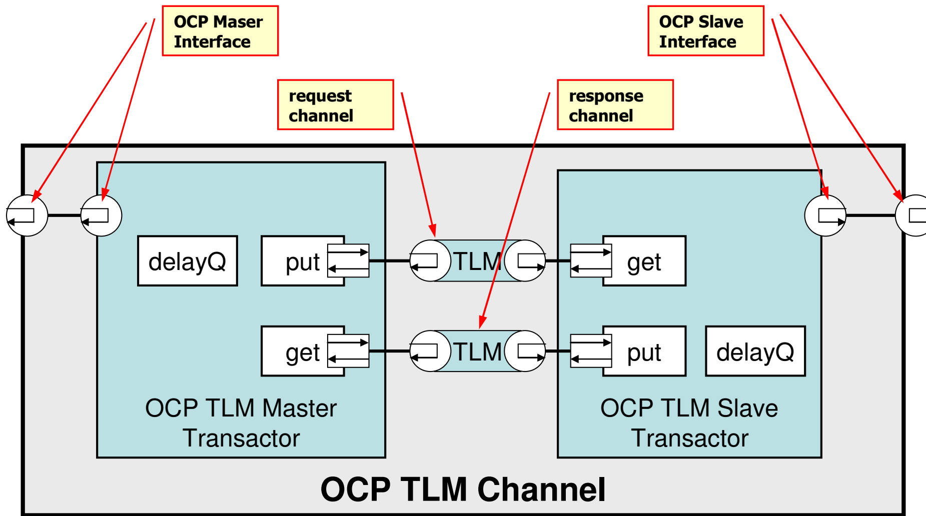

Figure 6: Mapping the TL3 API onto the OSCI TLM standard

Comment [AH2]: Could

The overall structure of the OSCI-TLM based OCP channel is depicted in Figure 6. From the outside perspective, the TLM based OCP channel looks just like any other OCP channel, i.e. it implements a master and a slave interface. As the only minor difference with the original OCP-channel, the OSCI-TLM based channel depicted above does not implement master and a slave interfaces but uses the sc_export feature of SystemC 2.1.

Inside the OCP TLM channel the interfaces are implemented by two separate modules, the master-transactor and the slave-transactor. Most importantly the master- and slave-transactor are completely separated, i.e. they communicate only via two OSCI TLM FIFOs. In that the mapping of the OCP API onto the TLM API is 100% complete. Each of these FIFOs is of course a size of 1, which corresponds to the *current* transaction in the OCP channel.

The master- and slave-transactor are composed of policy-classes, which each implement one aspect of the OCP protocol. The put-policy implements everything related to the transmitting of data and the get-policy implements everything related to the receiving of data. Both policies are templatized with the transaction data structure, so they can be used for both master and slave side. Apart from reusing the code in the transactor this structure nicely emphasizes the symmetry of the OCP protocol: the sending of request is handled in the exact same way as the sending of response.

The implicit timing annotation features of the TL3 API require additional functionality. The timed sendRequest (and sendResponse) methods are implemented using a special delay queue called ChronoQueue [11], which delays the sending of transactions by a specific amount of time. The

timed requestAccept (and responseAccept) methods are implemented by the get policy itself by means of delayed event notification.

The detailed mapping of OCP TL3 methods and events on the OSCI TLM standard is depicted in Table 4.

OCP TL3 Master API Function	OSCI TLM Standard API
Regular Request Commands	
bool sendRequest(const REQ& req)	bool nb_put(const T &t
bool sendRequestBlocking(const REQ& req);	derived from sendRequest and RequestEndEvent
bool requestInProgress() const	!(bool can_nb_put() const)
const sc_event RequestStartEvent()	local event
const sc_event RequestEndEvent()	const sc_event &ok_to_put() const
Timed Request Commands	
bool sendRequest(const REQ&, const sc_time& t)	derived from sendRequest and local delay queue
bool sendRequest(const REQ&, const int cycles)	
Regular Response Commands	
bool getResponse(RESP& resp)	bool nb_peek(T &t) and local flag
bool getResponseBlocking(RESP& resp)	derived from getResponse
bool acceptResponse()	bool nb_get(T &t)
bool responseInProgress() const	bool nb_can_peek() const
const sc_event ResponseStartEvent()	const sc_event &ok_to_peek() const
const sc_event ResponseEndEvent()	local event
Timed Response Commands	
bool acceptResponse(const sc_time& t)	derived from acceptResponse and delayed event notification
bool acceptResponse(const int cycles)	

Table 4: TLM mapping of OCP TL3 master interface

The regular request commands and events are implemented in the put-policy. These TL3 primitives have almost a one-to-one correspondence with the `tlm_nonblocking_put_if` of the TLM API. Only the RequestStartEvent is missing in the TLM API, but the occurrence of this event is of course known at the master side. Hence a local event in the put-policy is notified whenever a new request is put into the TLM FIFO.

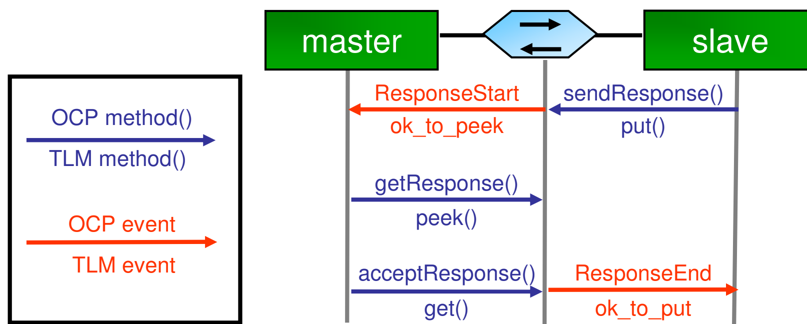

Figure 7: Sequence of methods and events

The get policy is slightly more sophisticated, because we need to mimic the get-accept mechanism of the OCP protocol. As illustrated in Figure 7 we cannot use the TLM-get method for the OCP-getResponse method, since the TLM-get is destructive and immediately notifies the ok_to_put event on the producer side. According to the OCP protocol, this event is not supposed to be notified until the master has released the response channel by calling the acceptResponse method. Hence we need the full expressiveness of the `tlm_nonblocking_get_peek_if`, which provides with the required non-destructive peek method.

In analogy to the put-policy, the get-policy provides a local ResponseEndEvent, which is notified upon the acceptance of the current response. Additionally, the get-policy needs a local `SC_METHOD` together with a separate event to implement the delayed accept methods.

Please refer to the online documentation of the methodology package [11] for detailed information on the implementation of the `tlm_tl3_transactor_channel` implementing the mapping.

4.3 AV Timing

The major purpose of the Architects View use-case is to investigate the performance of a given HW/SW partitioning and platform architecture based on an approximate timing model. Hence special care needs to be given to the modeling of timing. The basic concepts of implicit timing annotation for the separation of timing information from other modeling domains are discussed in section 1.2. Now we will focus on the timing annotation offered by the TL3 API primitives and their usage for communication and computation performance modeling.

In principle the implicit timing annotation primitives in the TL3 API operate at the *interval-level*, i.e. the granularity is limited to the boundaries of transactions. Two timing parameters characterize the performance of any activity in the system:

- The *accept-delay* Δt_{accept} specifies the minimum time between two consecutive start request events or two consecutive start response events. In essence the accept-delay constraints the bandwidth of a block, i.e. during this period a slave module is busy with the processing of a request or a master module is busy with the processing of a response.
- The *latency* $\Delta t_{latency}$ specifies the time between the process activation and the sending of the transaction. At the target side, this parameter is called *response-delay* $\Delta t_{response}$ and denotes the duration between request start event and the response start event.

In that way the timing requirements of arbitrary platform building blocks can be roughly specified. For example a pipelined ASIC block will exhibit an accept delay smaller than the response delay, whereas for a task executed on a programmable core it will be the other way around. Please note, that the accept-delay and latency are not specific for the performance modeling of a communication node or a processing element.

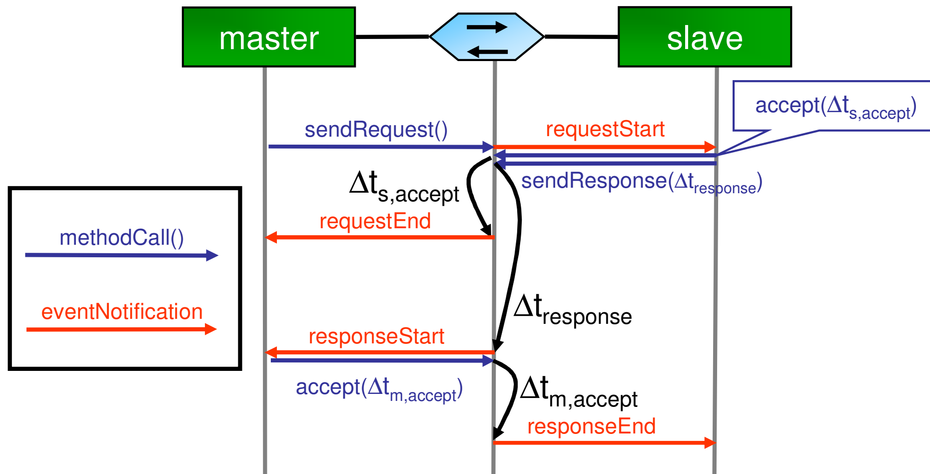

Figure 8: AV point-to-point timing annotation

The diagram depicted above shows the implicit timing annotation methodology of the TL3 channel applied to a typical sequence of method calls (blue) and events (red):

- The master initiates a transaction using the `sendRequest` API call
- The channel immediately triggers the `RequestStartEvent`
- The slave reads the data from the channel (not depicted) and at the same time uses the delayed `acceptRequest` method to specify the slave accept delay $\Delta t_{s,accept}$. The OCP TL3 slave interface also supports implicit timing annotation for the response delay in the `sendResponse` method, so even here the slave does not have to call `wait()`. (This feature is not implemented in the TL2 API, but is under consideration for the later versions.)
- After $\Delta t_{s,accept}$ the OCP channel self-acting releases the request path and notifies the `RequestEndEvent`
- After $\Delta t_{s,response}$ the OCP channel self-acting initiates the response phase by notifying the `ResponseStartEvent`. The remainder of the response phase is completely symmetric to the request phase.

Of course both delays could also be specified explicitly by calling `wait(Δt`$_{accept}$`)` and `wait(Δt`$_{response}$`)`. However the explicit way of modeling timing is unfavorable in terms of simulation speed and orthogonalization of timing and behavior. The latter limitation is not obvious and shall be further explained by means of a simplified example depicted in Figure 9.

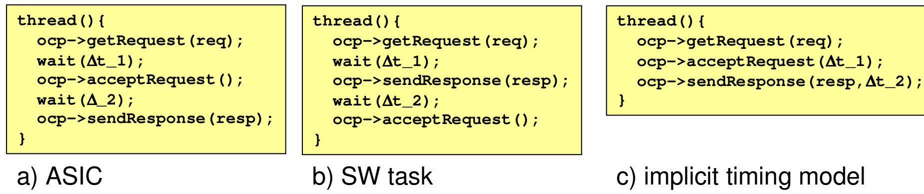

Figure 9: AV timing annotation

In case a) the anticipated implementation of a SystemC thread is an ASIC block, so the accept delay is smaller than the response delay. Modeled explicitly the request must be accepted before the response is sent. To change the implementation into a SW task, the source code of the SystemC model needs to be modified according to b). The implicit timing model depicted in c) is more flexible, in that only the values of `Δt_1` and `Δt_2` need to be modified to change the performance characteristic from ASIC to SW.

The actual value of the timing parameters can either be a (configurable) constant, a data-dependent variable, or can be drawn from a stochastic distribution function. The TL2 and TL3 APIs allow specifying the implicit delay in terms of cycles as well as in terms of time. In case cycles are used, the channel calculates the effective delay by multiplying the cycle count with its clock period parameter. In principle this cycle-based annotation is favorable, because it enables a more efficient exploration of the impact of clock frequency on performance. The frequency could even be modified during runtime to investigate e.g. the impact of dynamic voltage scaling on system performance. A delay specification based on cycles is also more re-usable than a timing value, as it does not tie a TLM model to a certain technology.

In principle we advocate the concept of *individual timing annotation*, i.e. the timing parameters should be maintained by the respective model. For example, a communication node should own the delay parameters related to communication latency and bandwidth. On the other hand, the computational element should own the delay parameters related to processing latency and bandwidth. It is absolutely discouraged to mix up the ownership of delay parameters, for example using the accept delay in a target processing element to annotate the communication latency.

The major advantage of individual timing annotation is that it is modular and compositional, i.e. components can be successively composed to systems without reworking the timing annotation. Figure 10 below shows the deployment of the individual interval-level timing annotation parameters in a typical request phase of a simple platform model. The platform comprises a shared bus connected to one or multiple initiators and one or multiple targets.

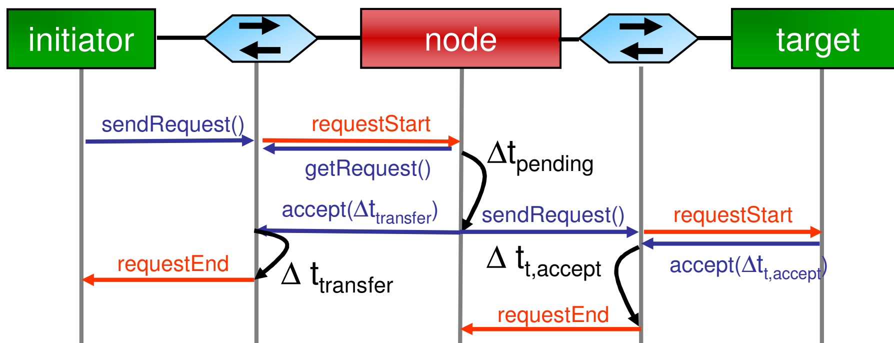

Figure 10: AV platform timing

In case of a computational element like the target module on the right hand side, the assignment of accept- and response-delays to processing delays is straight-forward. For the communication node however, the accept delay representing the bandwidth is split into a pending delay and a transfer delay. The former corresponds to the send-delay of the request and has to be modeled explicitly, because it depends on the current traffic situation. In contrast the transfer delay can be arithmetically derived from the performance parameters and is therefore applicable for implicit timing annotation.

The point of Figure 10 is that neither the initiator nor the targets need to change their timing annotation, when a bus node is plugged between them. Of course the system-level timing changes, but this merely turns out as a consequence of all the individual timing annotations.

In summary, this section has introduced the AV timing model by means of timing annotation:

- Implicit timing annotation refers to the modeling of timing by means of TLM communication API parameters in favor of using wait or delayed event notification.
- Interval level timing annotation refers to the transaction granularity of the AV timing model, in that the timing resolution is limited to the start and end of transactions.
- Individual timing annotation refers to a well-defined ownership policy of the timing annotation parameters in order to achieve a compositional performance model.

The OCP TL2 API also supports implicit timing annotation at the *word-level*, which specifies in more detail the timing of individual beats in a burst. This is done by a set of timing parameters, which are defined in the master and slave timing-groups. Word-level annotation may increase the timing accuracy in case the data-handshaking feature of the OCP protocol is used (see section 6.8 of the OCP channel documentation [5]). However it is rather specific for OCP and hence not part of the protocol agnostic TL3 API.

Scalable Accuracy

There is not necessarily a one-to-one relation between an OCP burst and a TL2 or TL3 request. The `LastOfBurst` attribute in the TL2 data structures allows the user defined segmentation of an OCP protocol burst into multiple *chunks*. This degree of freedom enables a scalable accuracy of the timing model:

The user can transfer an OCP burst as a single TL2 request, i.e. `LastOfBurst` is always true, and the `DataLength` equals the total OCP burst size. However the granularity of TL2 timing annotation is tied to the TL2 transfers, and no predication can be made the timing within the transfer. The parameters in the TL2 timing group specify the timing of individual beats in a burst, but this assumes these timing parameters are constant during the complete burst. In summary, this is for sure the fastest way to transfer the data, but on the other the least accurate.

On the other had, the user can decide to segment an OCP burst into multiple TL2 request, e.g. a burst of 8 word is split into 2 TL2 requests of `DataLength` 4 each. Now the timing information can be applied to each of the requests, so the performance model is more accurate. On the other hand, more events are required to transfer the data, so naturally the simulation speed degrades.

The decision on the granularity depends very much on the required accuracy of the performance model as well as on the dynamic in the system. As long as the state of a module does not change during an OCP burst, a finer granularity would not increase the accuracy.

5 Programmers View

The goal of the Programmers View use model is to contain enough detail to enable most of the embedded SW development for the programmable processing elements embedded into the SoC platform. Some requirements are straightforward: a PV model needs to be register accurate and bit true; it also needs sufficient timing information to be capable of interrupt handling. A notable exception on the scope of the Programmer View is the cycle accurate development of timing critical SW. This kind of SW needs to be developed on models that have more timing accuracy. The fact that communication over buses and on chip networks as well as target peripherals can be modeled with little or no timing information is very beneficial to simulation speed and model development time. It is clear that synchronization between different processing elements needs to be implemented in sufficient detail to ensure correct functionality. For this purpose a certain amount of timing information needs to modeled.

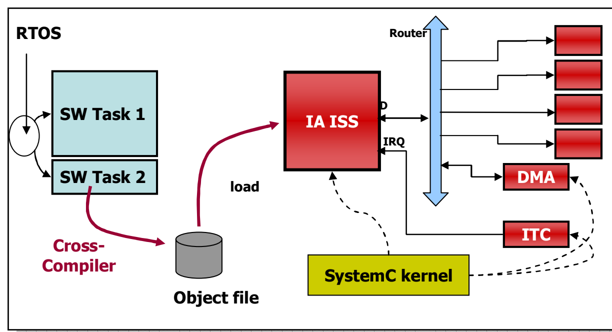

Figure 11: A typical PV model

A key aspect of the PV transactional modeling style is that it should link efficiently to an instruction set simulator. Embedded SW designers will use an ISS model for the processor they are using together with their usual debug environment. The goal of PV is to easily extend an ISS with a functional view of the platform architecture. A typical PV system will consist out of a processor model, a router to direct transactions to the right memory or peripheral and a functional description of every peripheral (see Figure 11). An embedded SW designer can develop the SW for the platform using the target RTOS API, and use the compiler for the target processor to create an object file for the SW. If the ISS model is linked to the debugging environment of the target processor the SW designer can continue to work in his usual environment and develop SW on a functionally accurate representation of the target system. Since the PV modeling style does not require modeling timing, it's possible to use the blocking bi-directional TLM interface for this modeling style.

5.1 Communication API

In essence the PV interface is identical with the bi-directional transport API of the OSCI TLM standard. Apart from the genuine transport function the PV API defines more precisely the transaction data structures along with the requirements of the Programmers View use model. The resulting PV interface exhibited in the table below is still templatized, but now the flexibility is limited to the types for the address and data attributes.

API Function	Description
PVResp<DT> transport(const PVReq<DT,AT> &)	Calls the transport implementation in the target

Table 5: Programmers View Interface Definition

In this PV interface definition *DT* represents the data type and *AT* represents the address type. This data types should be chosen according to the policy defined in section 2.2. The initiator has to be modeled using an `SC_THREAD` since the transport implementation is allowed to call wait.

For convenience the PV API also defines a PVInitiator_port and PVTarget_port, which inherit from sc_port and sc_export respectively. The target port extends the default capabilities of sc_export, in that multiple transport implementations (and hence multiple target ports) can co-exist in a single module.

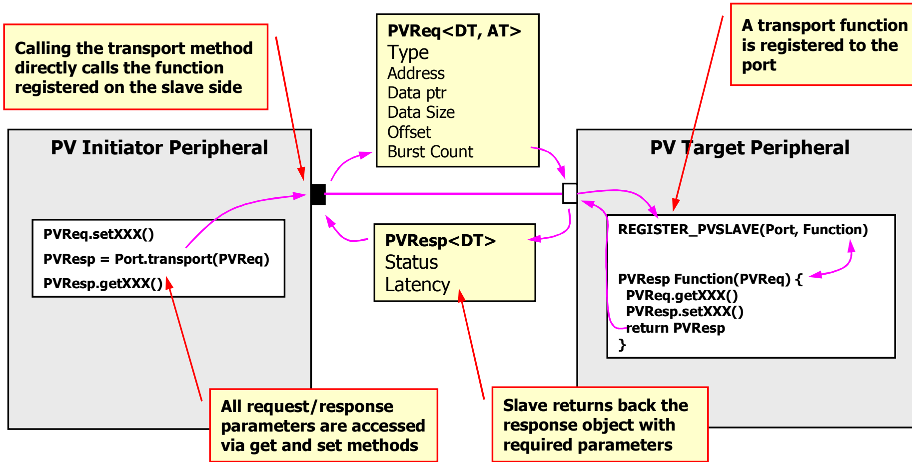

Figure 12: Functional Use of The PV API

The general sequence of a transport invocation is depicted in Figure 12 above. The initiator instantiates the request and response structure and specifies the request attributes. The transport call to the initiator port is directly forwarded to the corresponding implementation in the target. The target can serve the request by accessing the request attributes and filling out the response structure. Figure 12 also shows the REGISTER_PVSLAVE macro, which effectively registers a transport implementation function to a particular PV target port.

The transaction data structures for the request and response are introduced in the next section.

5.2 Transaction Data Structures

The following tables briefly introduce the attributes of the PV request and response structure. Please refer to [11] for a more complete documentation of the access methods to the respective attribute. Like any other API, the PV API requires that the usage of these attributes conform to certain conventions. These conventions are also documented in the table and are required to successfully establish communication between any two PV modules.

Attribute Name	Data Type	Description
Address	AT	The address to which the transport will be done. PV allows both byte addressing and byte aligned word addressing. In case of word addressing the offset parameter can be used to access subwords.

Attribute Name	Data Type	Description
WriteDataSource	DT*	To boost the simulation speed, the PVReq object only transports the pointer to the first word of write data for the request. The data is owned by the initiator (see section 2.2, paragraph "memory owned by initiator"). Additionally all data received at the target peripheral, regardless of access size and offset, is always shifted down to bit zero.
Type	enum	Possible values are pvWrite for write transfers, and pvRead for read transfers.
DataSize	unsigned int	The DataSize specifies the width of the transferred data in bits. Allowed values are one of 8, 16, 32, or 64. This attribute enables the sub-word access to data items, which have a DataSize smaller than the width available in data type DT. In case of a burst the DataSize refers to the size of each sub-word.
Offset	unsigned int	Used in case of word addressing together with DataSize to enable sub-word access, i.e. for data items which have a DataSize smaller than the width available in data type DT. The Offset specifies the position of the first data item in bits.
BurstCount	unsigned int	Specifies the transported number of data items.
ReadDataDestination	DT*	Specifies the memory location to which the target should write its data in case of a read transaction. The memory is owned by the initiator (see section 2.2, paragraph "memory owned by initiator") As such it must be allocated by the initiator before transport is called.
Response	PVResp*	When creating the PVReq object, a PVResp object is also created. A reference to this PVResp object is stored within the PVReq object. The target must use the PVResp object to return status information or read data.
CustomData	PVCustomReq &	When you want to transport custom information it is possible to pass along a reference to a user-defined child of the PVCustomReq class. In this user-defined class, extra custom data fields can be created.

Table 6: PV Request Data Structure

Attribute Name	Data Type	Description
ReadData	DT	The target copies the read data into the memory provided by the initiator.
Response	enum	Possible values are pvOk for successful transfers and pvError for failures.
Latency	unsigned int	This implicit timing attribute can be used by the target to communicate to the initiator how much time is consumed in the slave.

Table 7: PV Response Data Structure

The request object enables full control over all attributes related to a remote data access, whereas the response object provides mechanisms for feedback. Again, the PV API deploys the "memory

owned by initiator" storage policy (see section 2.2). Therefore the request structure carries the pointers to both read and write data.

5.3 Synchronization of PV models

Special care needs to be taken to synchronize PV models. Due to the Remote-Procedure-Call (RPC) semantic of PV, the modeler is responsible to yield the control to the SystemC kernel e.g. when a signal is written in a PV model. Signals are often used in PV models for the interrupts lines. However writing to a signal in a transport function has no immediate effect, because it affects only the projected value (see section 6.6.4, "Reading and writing signals" in [1]). The projected value is assigned to the current value only after the update function has been executed, i.e. after the SystemC simulation kernel has advanced one delta cycle. Calling the PV transport function does not cause the simulation engine to advance time or to introduce delta cycles. Hence multiple calls can be made to a transport function without the signals getting updated. Instead the every signal assignments overwrites the previous one, which may result in synchronization problems.

To solve this problem, a delta cycle needs to introduced by calling `wait(SC_ZERO_TIME)` after the signal has been written from a transport function, e.g.

```
sig_nIRQ.write( false);
wait ( SC_ZERO_TIME );
```

Note that this does not only apply to sc_signals, it applies to any SystemC construct that requires the two-cycle evaluation scheme, i.e. the request-update and update cycles.

The Programmers View API can be seen as the most simple communication interface for behavioral TLM platform models. For this reason it has been selected as the generic behavior interface.

6 AV-PV Transactor

Special attention should be given to these bus transactors, since they are highly reusable within a platform as well as across multiple design projects. However the creation of bus transactors is a more advanced topic, because it is rather specific to the respective bus protocol. So far no formalized modeling pattern for bus transactors has been identified. For this reason we have to confine ourselves to the discussion of examples in our transactor library. Further elements in this library are the PV-TL2-master transactor, as well as the TL3-TL2- and TL2-TL3-transactor channels.

The `ocp_tl2_pv_slave_transactor` is a good example for a highly reusable bus transactor, because the APIs on both sides are part of our strategy for modeling communication in a TLM platform. For this reason we will discuss the implementation of this transactor in quite some detail. The full source code and documentation can be found in the methodology package [11].

Apart from the bare transformation of one TLM API into another, the TL2-PV transactor can be considered as an *overlay performance model* for un-timed or implicitly timed PV target models. In other words this transactor enhances the missing or implicit timing information in the PV peripheral to the level of an AV performance model according to section 4.3.

Full TL2 compliance is maybe too big a claim for this transactor, mainly because most of the OCP specific features in the TL2 API like multi-threading, reset and sideband signals have no counterpart in the PV API. Hence the transactor basically considers the TL3 subset of the TL2 API and ignores the rest.

It should also be noted at this point that the timing aware handling of responses is a recurrent modeling task at the OCP TL2 and TL3 level. Therefore the mechanism used in the transactor for the processing of responses is applicable to any OCP TL2 target device. A major part of this functionality is encapsulated in special response queue class. Refer to documentation in ChronoQueue.h in the OCP methodology package.

6.1 Overview

As depicted in Figure 13, the transactor module contains one PV initiator port and one OCP-TL2 slave port. The timing is completely event driven using a combination of OCP-TL2 channel events and local sc_event objects.

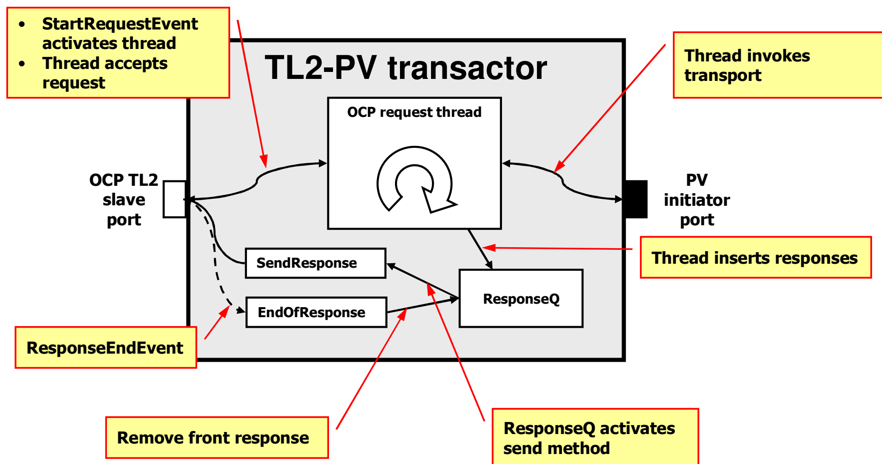

Figure 13: OCP-TL2 slave to PV initiator target bus-transactor

Figure 13 sketches the major part of the workflow in the transactor:

- The main functionality is enclosed in the OCP request thread, which is sensitive to the ResquestStartEvent of the OCP channel.
- The request thread converts the OCP request into the corresponding PV request structure and invokes the transport implementation in whatever slave is attached the PV initiator port.
- After the transport call returns the request thread creates the OCP response, and calculates the timing parameters. Based on this timing response is inserted into the ResponseQ and request is accepted.
- The ResponseQ notifies the SendResponse method when the front response is due for sending and the OCP channel is available to actually send a response. The SendResponse only puts the response on the OCP channel, but does not remove the response from the queue.
- After the ResponseEndEvent is notified by the OCP channel, the EndOfResponse method removes the front response from the queue. Like this the ResponseQ keeps track of the OCP channel availability and does not needlessly notify the SendResponse method.

The conversion between OCP TL2 and PV data structures as well as the conversion between the timing models is further investigated in the following sections.

6.2 Transformation of the timing model

At a cursory level, the operation of the transactor is quite simple: when a new OCP request arrives, the transaction parameters are passed on to a PV request. However, there are many timing considerations that need to be taken into account. Please refer to section 4.2 for more information on the TL2 timing model.

The transactor needs to take into account 4 types of delays:

- The accept-delay is an implicit timing parameter provided by the OCP-TL2 API, which models the processing bandwidth of the target. No further requests are delivered to the slave port during this lockout delay.
- In the OCP-TL2 API, the response-delay is an explicit timing parameter, which describes the processing latency in the target peripheral. OCP TL3 provides an implicit timing parameter for the response-delay.
- The *latency* is an implicit timing parameter provided by the PV API.
- By means of delayed event notification or clock objects the PV target could deploy any kind of *explicit timing*.

The connected slave peripheral may model explicit (actual) or implicit (annotated) timing delays, which needs to be taken into account when using the OCP-TL2 timing parameters. The transactor can be configured with the OCP accept-delay and OCP response-delay parameters through constructor arguments.

The tricky part is to determine the actual values of the accept-delay and response-delay. Please refer to Table 8 for the priorities in determining the actual timing parameters from the explicit and implicit annotation in the slave as well as from the configured timing in the constructor.

Priority	Configured OCP delay Parameter	Slave Timing	Result
Highest	Don't care	Explicit	Immediately accept request and send response, i.e. no additional delay.
Middle	>0	Implicit	Use constructor parameter * clock period
Lowest	=0	Implicit	Use PV latency parameter * clock period

Table 8: Calculation OCP timing parameters

As the table shows implicit timing in the PV target has the highest priority. That is simply because the transactor cannot do anything about it. In case the user has defined values for accept delay and/or the response delay, these settings override any implicit timing annotation in the slave. Like this the transactor becomes an overlay performance model for the attached slave. Only if no default timing is specified, the implicit timing from the PV target is considered. In this case the transactor transforms the implicit timing from the PV target into actual SystemC timing.

7 References

[1] "Draft Standard SystemC Language Reference Manual", April 2005 available at www.systemc.org

[2] Adam Rose, Stuart Swan, John Pierce, Jean-Michel Fernandez, "Transaction Level Modeling in SystemC", OSCI whitepaper

[3] SystemC Verification Working Group, "SystemC Verification Standard Specification", Version 1.0e, May 2003

[4] "A SystemC OCP Transaction Level Communication Channel", OCP-IP System-Level-Design Working Group, Version 1.0, January 2003

[5] "A SystemC OCP Transaction Level Communication Channel", OCP-IP System-Level-Design Working Group, Version 2.1.1, June 2005

[6] Anssi Haverinen, Maxime Leclerq, Norman Weyrich, Drew Wingard, "White Paper for SoC Communication Modeling", October 2002

[7] Bart Vanthournout, Serge Goossens, Tim Kogel, "Developing Transaction-level Models in SystemC", CoWare whitepaper, June 2004

[8] Adam Donlin, "Transaction Level Modeling: Flows and Use Models", ISSS/CODES September 2004

[9] Frank Ghennassia

[10] Thorsten Grotker, Stan Liao, Grant Martin, Stuart Swan, "System Design with SystemC", Kluwer Academic publishers.

[11] OCP Methodology Package documentation