

# Developing Transaction-level Models in SystemC

Bart Vanthournout, Serge Goossens, Tim Kogel, CoWare, Inc.

## Abstract

The purpose of this white paper is to give an overview of the support that transaction level modeling (TLM) in SystemC provides for System level modeling. More specifically we will address how TLM supports different SoC design and verification tasks. In this document we will first give an introduction of transaction level modeling describing what the purpose of TLM is, how it is standardized and give an overview of different design tasks it solves. In the other sections of this document we give an overview of TLM modeling styles for programmers view, architects view and verification view.

## 1 Introduction to Transaction level Modeling

Transaction level modeling has always been an important modeling technique within SystemC, for an introduction of how it is part of the design of SystemC see [4]. Since the introduction of SystemC 2.0 many modeling variants of transactional modeling have arisen, leading to the need for a standardized approach. For this purpose a standard is being proposed in OSCI, a description of this standard can be found in [1]. In this document we apply the standard to a set of design problems that originally lead to the need for TLM.

Today's increasingly complex system-on-chip (SoC) designs consist of embedded software running on multiple processor cores, connected to memory and peripherals. As complexity grows, an increasing proportion of the software and the hardware peripherals comprise of reused intellectual property (IP) blocks. There is a need for a design approach where a mix of application software and algorithms can be mapped to an architecture made up of reusable IP blocks. In this case, there is a need for a hardware-software co-design approach that allows quick analysis of the trade-offs between implementation in hardware and software. Verification and implementation work both become very important and an efficient methodology is required that involves the creation of a minimum number of models. In order to achieve this on complex designs with adequate simulation performance, high-level models are not just needed to simulate the software on a model of the hardware, but also to accelerate the process of modeling hardware IP in a bus independent, reuse friendly way. Models also need to allow easy and quick configuration of the on chip bus architecture or communication network. It is in this context that transaction level modeling is to be used.

The primary goal of transaction-level modeling is to achieve dramatically increased simulation speeds, while still offering enough accuracy for the design task at hand. 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 data payload. As a consequence Transaction level modeling applies best to a class of design problems where communication and system level integration is dominant.

Transaction-level modeling is also intended to reduce the amount of detail the designer must handle, therefore making modeling easier. Transaction-level models separate communication from behavior. This allows each to be modeled independently without affecting the other, and also allows alternate descriptions to be tried easily. The modeling technique can support different levels of abstraction, so that detail can be added or suppressed as needed for a given state of development.

### 1.1 Principles of TLM in SystemC

The transaction-level model is built as set of interfaces that define how models communicate. In its most primitive form the TLM basic interfaces provide 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 will need to be complemented with a 'protocol-layer', which will target 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 (see Figure 1). In this whitepaper we will pay most attention on this protocol layer and what consideration should be taken into account to design a protocol layer.

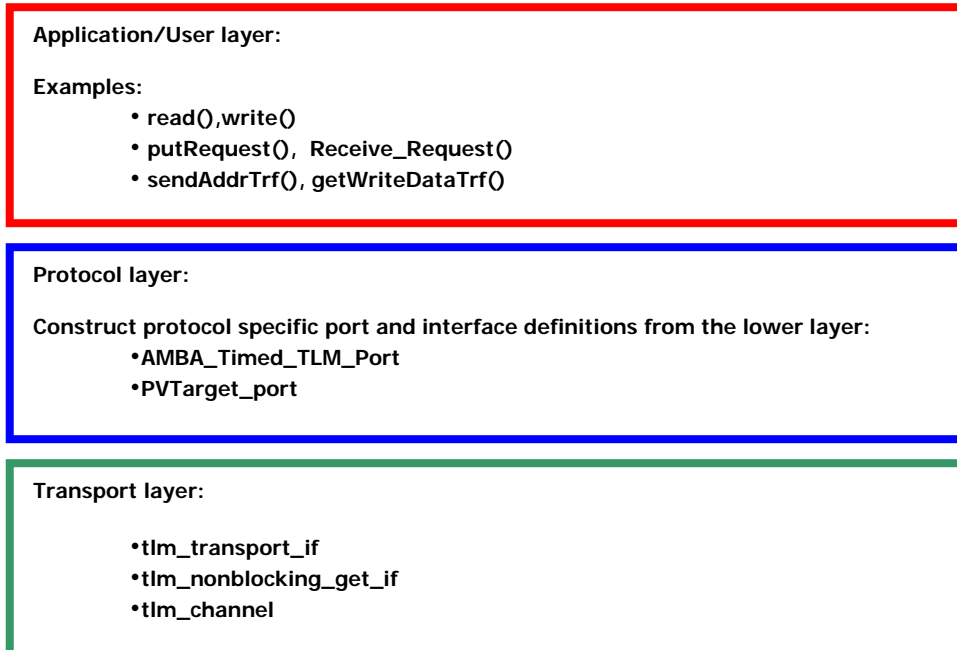


Figure 1: layered TLM API's

The TLM basic interfaces consist of bidirectional and unidirectional interfaces each with different possible synchronization mechanisms<sup>1</sup>.

### 1.1.1 Bidirectional blocking interface

The bidirectional interface provides with the most simple communication and synchronization mechanism. It uses a REQ data type for the information an initiator provides in the communication and a RSP data type for the information it receives from the target. The synchronization is such that the initiator expects the response packet to be available when the interface call returns. For this interface communication never fails and all communicated information is available with every call.

```
template<typename REQ, typename RSP>
class tlm_transport_if : public virtual sc_interface {
public:
    virtual RSP transport(const REQ&) = 0;
};
```

### 1.1.2 Unidirectional Interface

The unidirectional interfaces use ‘put’ and ‘get’ calls as communication interfaces. The synchronization of the blocking interface requires the interface method never to fail, while the non blocking interfaces return a bool value to indicate success or failure, for the same purpose they also have an event interface to indicate that data is available as well as an interface that indicates that data will be available.

#### 1.1.2.1 Blocking interface

```
template < typename T >
class tlm_blocking_get_if : public virtual sc_interface {
public:
    virtual T get( tlm_tag<T> *t = 0 ) = 0;
    virtual void get( T &t ) { t = get(); }
};

template < typename T >
class tlm_blocking_put_if : public virtual sc_interface {
public:
    virtual void put( const T &t ) = 0;
};
```

<sup>1</sup> In the context of SystemC the terminology ‘blocking’ and ‘non-blocking’ is used. 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.

### 1.1.2.2 Non blocking interface

```
template < typename T >
class tlm_nonblocking_get_if : public virtual sc_interface {
public:
    virtual bool nb_get( T &t ) = 0;
    virtual const sc_event &ok_to_get( tlm_tag<T> *t = 0 ) const = 0;
    virtual bool can_get( tlm_tag<T> *t = 0 ) const = 0;
};

template < typename T >
class tlm_nonblocking_put_if : public virtual sc_interface {
public:
    virtual bool nb_put( const T &t ) = 0;
    virtual const sc_event &ok_to_put( tlm_tag<T> *t = 0 ) const = 0;
    virtual bool can_put( tlm_tag<T> *t = 0 ) const = 0;
};
```

## 1.2 SoC Design Tasks and TLM

In a typical SoC design a number of DSP algorithms are combined with a programmable environment that allows user applications to be run. As a consequence there are a number of design disciplines or functions, each needing a level of modeling with a balance of performance and accuracy appropriate for their design task. In this paper, we introduce a consistent set of views on the SoC named after these design tasks.

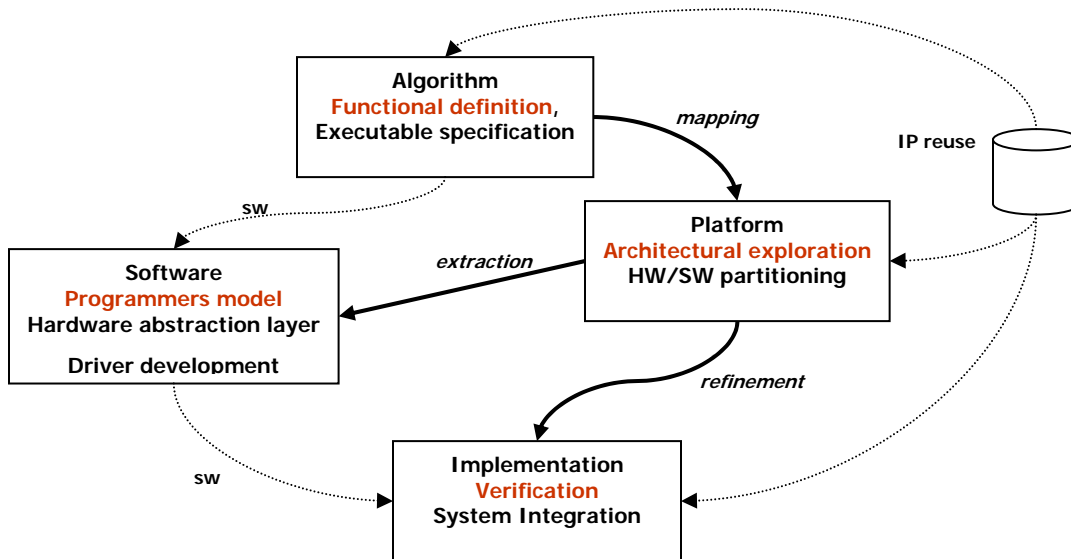
1. There is a need to create a functionally correct model of the SoC to enable embedded SW developers to develop the required firmware for the SoC, this is referred to as the **Programmers View (PV)**. An important aspect of such a model is that large amounts of SW need to be run for verification. On the other hand most of the hardware related aspects can be ignored. Timing should be considered on a different scale than typically used by hardware and verification engineers, or could be absent all together. For a SW designer its important to have the memory map correctly modeled, so a PV model will focus on that aspect.
2. There is the **Architects View (AV)**, a model that allows SoC architects to identify and resolve potential 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. For this purpose architects need to analyze the SoC in different operating conditions which puts high expectations on simulation speed. A model used by architects requires a functional model of all elements in the system and models that provide with sufficient timing information to analyze the performance of different architectures. In the OSCI TLM working group, this modeling level is known as PV+T, since it can be derived from the PV model by adding timing information.

3. The **Verification View (VV)** is for the verification engineers, who want to use the system level models as testbenches to validate implementation models of the different HW blocks in the system. For this purpose they need to be able to interface the system level models with detailed HDL models and verify correct behavior at cycle-accurate level. In the OSCI TLM working group, this modeling level is known by its implementation-oriented name of CC (cycle-callable).

Each of these views should be supported by a transactional modeling style, so that efficient SystemC models can be developed for these design tasks.

### 1.3 Design flows

When designing an SoC different design flows can be used, the selection of a design flow is dependent on the design tasks that will be needed. An example design flow is depicted in **Figure 2**. 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 a 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 2: design flow**

Of course it is possible that not all these steps are important for a certain type of SoC design, the architecture can be predefined, there may be no need for an embedded SW model since the design is derived from a standard platform for which SW development boards exist, etc.

Since we look at different design tasks, each with their own modeling style, it's obvious that we cannot expect that all models will be rewritten for each design task. For a number of key SoC

design elements it makes sense to create models for each abstraction (processor models and interconnect models come to mind) for others the goal is to reuse where appropriate and refine when necessary. For example when functional models are created to enable an architectural view these might be reused in a programmer's view as long as they do not reduce the memorymap accuracy or lower the simulation performance. In the same way PV and AV models can be reused for verification. To enable mixing modeling styles there is a need for adaptors and converters.

SystemC and transaction level modeling provide with modeling capabilities that allow to address all these design tasks and modeling considerations in a unified modeling environment where different modeling styles can be interfaced with each other. In the remainder of this document we will discuss how to use the SystemC transaction level modeling proposal to address these design tasks.

## **2 Programmer's View: PV**

### **2.1 Introduction**

The goal of a programmer's view is to contain enough detail to enable most of the embedded SW development for the different processing elements in a design. Some requirements are straightforward: a PV model needs to be register accurate and bit true; it also needs to be capable of interrupt handling. A notable exception on the requirement is the cycle accurate development of timing critical SW. This SW needs to be developed on models that have more timing accuracy (see below). The fact that communication over buses and on chip networks as well as target peripherals can be modeled untimed is very beneficial to simulation speed and development time. It is clear that synchronization between different processing elements needs to be implemented in sufficient detail to ensure correct functionality. It is equally important that this synchronization cannot depend on time.

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 system. 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 3). 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 to model timing it's possible to use the blocking bidirectional TLM interface for this modeling style.

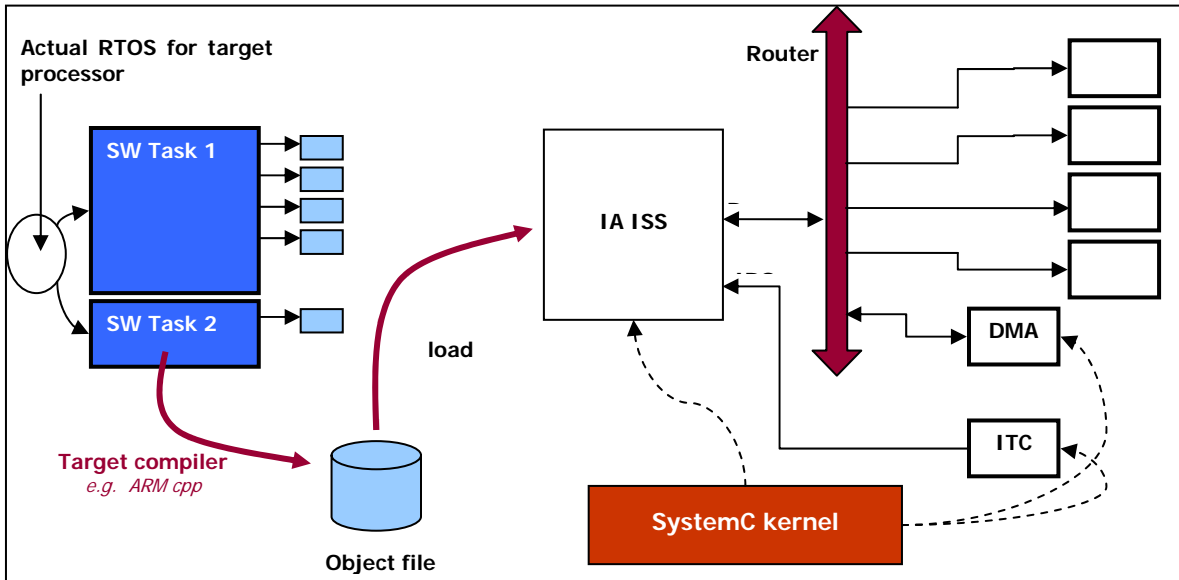


Figure 3: typical PV model

## 2.2 Protocol Layer

The bidirectional blocking interface is used as the lowest layer in the protocol stack for PV. The protocol layer builds a set of convenience functions on the transport layer and offers these to the user layer.

- Transport will forward the request to the interface method transport in the transport layer.
- Read takes an address, builds a request structure, calls transport and returns the data.
- Write puts the address and data item in a request structure and calls transport.

These convenience functions can obviously be inlined. The key point is that the user code does not call the transport layer directly. Note that the initiator has to be modeled using a SC\_THREAD since the transport layer is blocking. The PV modeling style can also be used for functional descriptions as in the figure below.

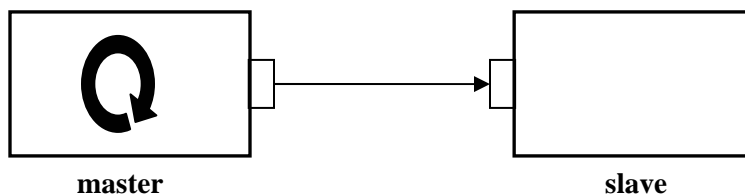


Figure 4: functional use of PV interfaces

## 2.3 Data Structures

The transport function is bidirectional, but the data which is transported is split out in 2 structures. The data which transported from the initiator to the target is put in the request structure and the response structure contains the data which is sent back from the target to the initiator. The request structure typically has the following data members:

- Address
- Write data
- Type of transaction: read or write.
- Attributes, such as access size, byte enable, ...

The response structure typically has the following data members.

- Read data
- Response status: OK, error, ...

Of course there are many different solutions to build Request and Response datatypes for PV modeling, each targeted to a different type of processor interface. It is also possible to go for a more abstract data type that does not directly relate to the processor but models the information packets as they will be transported over the interconnect of the SoC.

## 2.4 Implementation in SystemC

The TLM transport interface is instantiated with the custom data structures.

```
template< typename AT, typename DT >
class PV_if :
    virtual public tlm_transport_if< PVReq< AT, DT>, PVResp< DT> > {
public :
    virtual PVResp< DT> transport( const PVReq< AT, DT> & ) = 0;
};
```

The initiator has PVInitiator ports which are built using sc\_port instantiated with the interface defined above. In this class one adds the convenience function that make up the protocol layer.

```
template< typename AT, typename DT >
class PVInitiator_port : public sc_port< PV_if< AT, DT > > {
public:
    explicit PVInitiator_port( const char * name) ;

    // PROTOCOL layer: convenience functions
    PVResp< DT > transport( const PVReq< AT, DT > & arg_Req) {
        return (*this)->transport( arg_Req);
    }
    void write( const AT & a, const DT & d );
    void read( const AT & a, DT & d );
    DT read( const AT & a );
};
```

The target has PVTarg ports which are built using `sc_export` instantiated with the interface defined above. `Sc_export` is a System C 2.1 feature, which maintains a pointer to an interface. In this case a process which implements the transport function is attached to the target port. This is done to alleviate the constraint of a single transport function in the target. A class can implement an interface only once, `sc_export` allows multiple instantiations of the same interface (multiple target ports).

To further ease the coding style there is target process attached to the target port, it implements the transport function by calling a 'transport like' method in the target module (i.e. a method with the same signature as `transport`) that was registered for this target port. The binding between target ports and target member function is done in the target's constructor by means of the `REGISTER_PVSLAVE` call.

```
PVSlave::PVSlave(sc_module_name name) : sc_module(name) {
    ...
    REGISTER_PVSLAVE( p_slave, theTransportfunc);
};
```

An alternative approach is to define target base classes which implement `transport`. Multiple objects of these classes can be instantiated in the target and bound directly to the corresponding `sc_export` based target ports. This corresponds to building a hierarchical target. The problem with this approach is that the target data members cannot be accessed directly by this `transport` implementation classes. Hence the data or a reference to the data or the target's this pointer needs to be passed down into these classes.

Another alternative approach is to bind the same `transport` function to multiple ports. This requires to pass information in the `REQ` structure that allows to decode which target port was actually triggered. This coding style also conflicts with the principle of separating communication and behavior.

## 2.5 How to introduce an ISS in PV

In order to link an ISS to a PV router, the implementation of the `PVReq` and `PVResp` datatypes is driven by the typical requirements of an ISS, an example definition for Request and Response is as follows:

```
template <class AT, class DT>
class PVReq {
private :
    AT          m_address;
    DT*         m_writeData;
    PVType      m_type;
    unsigned int m_dataSize;
    unsigned int m_offset;
    unsigned int m_burstCount;
    PVCustomReq* m_customData;
    ...
};
```

The request data structure contains logically address and data parameters. The write data member is a pointer, basically this allows a reference to a array of data members to be passed so that burst transactions can be modeled with a single transport API call. The burst count parameter indicates how many data elements are transported in a single burst. With datasize it is possible to indicate that a data element with size smaller than normal is to be transported (e.g. 8 bits or byte), in this case the offset indicates where exactly this data element fits in the data type (e.g. 32 bits or word).

There is a custom data field to store any other information you care to transport, e.g. whether to increase or decrease the address on burst, or protection information to indicate in what mode the processor is operating.

```
template <class DT>
class PVResp{
private :
    PVResponse    m_response;
    unsigned int  m_latency;
    DT*           m_readData;
    PVCustomResp* m_customData;
    ...
};
```

The most important aspect of the response data structure is that it contains, again, a data parameter. Also here we want to be able to respond to burst requests. The m\_response member indicates whether the transaction was successful or failed. The latency parameter is added to enable cycle count estimates in the processor model, with this value extra delay can be modeled, e.g. wait cycles in a memory.

Also here there is a custom data field added for any extra information. The definition of the PVType and PVResponse is listed below:

```
enum PVType { pvWrite = 0, pvRead = 1};
enum PVResponse { pvOk = 0, pvError = 1};
```

An ISS ideally is integrated as an object that has a method which fetches, decodes and executes a single instruction. It should have a memory access API that is called every time a memory access is required, from this memory API the transport function in a router/decoder can be called. It should also have methods that allow setting interrupt flags so that the processor model can switch to interrupt mode when necessary. In the same way other important interfaces can be added to the processor model. Such an object can be easily integrated as a sc\_module in SystemC.

**Note I. Performance:** One of the most important goals of PV modeling is to enable SW developers to use this model to design embedded SW. In order to provide with a sufficiently effective user experience, it is critical that a PV model delivers sufficient simulation speed. For this reason there are a number of considerations to take into account when creating PV SoC models.

- In the context of SystemC it is important to look at how often the model will switch between different threads. Issuing a `wait()` call requires the simulation kernel to select another thread to run, which in its turn implies to store the current program context and load a new one. Although this can be very efficiently implemented, it does infer a significant simulation overhead. For this reason it is always good to look for synchronization mechanisms that do not rely on SystemC events. Worst example of this is the `sc_clock` object which uses events to schedule itself and on its own already puts an upper bound on simulation speed that is too low for a PV user. The easiest way to minimize the number of context switches is to minimize the number of threads in the model. In the extreme a complete PV model can be run from a single thread, most likely however this will require all models to be developed specifically for this platform and as such prevent reuse of the models of the different components of a platform. A simple rule to define whether a platform component requires a thread is to question whether this component is an ISS model, whether it leads independently to an interrupt for the ISS, or whether it models some synchronization behavior between different processor independent of the SW running on them. In most cases it is possible to model a platform component as a simple peripheral that implements a transport call. For example a DMA can be modeled as a component that is reactive to transport calls from the ISS, usually the ISS or another peripheral will initiate the DMA at which point the DMA component can immediately issue all data transfers to perform the actual DMA (see paragraph 2.6.2 below).
- Next to minimizing the number of threads, context switches can be minimized by optimizing the use of the `wait()` calls. This can be done by looking at the rate at which the different threads synchronize with each other. As example, when a system contains a processor and a timer, where the timer will send an interrupt every 10ms, and the processor runs at a instruction rate of about 200MIPS then it's clear that ideally the processor can execute 2 million instructions before it needs to synchronize with the timer. So instead of calling `wait()` after every instruction it is possible to run a large number of instructions and then wait for the total number of cycles that are estimated by the processor model. Similarly a timer should not count down and call `wait()` at every count, but immediately wait for the total time it is set to. Of course these optimizations reduce the accuracy since reprogramming the timer will only have effect after the next timer tick, and the processor will not see any other events as between consecutive `wait()` calls.
- Another aspect to look at is how to deal with the data that is transferred over the transport function. In certain cases the overhead of creating and deleting the response data structure in the target module may be too big. In this case it may be better to manage the response data structure from within the initiator, which is anyway better when considering safety (see Note III)

## 2.6 Example PV blocks

### 2.6.1 Simple peripheral

The functionality of a PV peripheral module is triggered by the transport mechanism, the simpler the implementation is, the faster it will simulate. Hence a pure function call implementation is preferred. Peripheral modules usually can be modeled without external synchronization but sometimes to impact the system timing, and although we use a blocking bidirectional interface which implies it is allowed to call `wait()`, it is better to use the latency parameter in the response structure. Below is a simple example of a PV peripheral model.

```
class PVSlave : public sc_module{
public:
    PVTarget_port p_slave;
    PVSlave(sc_module_name name);
    SC_HAS_PROCESS(PVSlave);
    myPVResp theTransportfunc(const myPVReq& arg_Req);
};

PVSlave::PVSlave(sc_module_name name) : sc_module(name),
                                         p_slave(){
    REGISTER_PVSLAVE( p_slave, theTransportfunc);
};

myPVResp PVSlave::theTransportfunc(const myPVReq& arg_Req){
    cout << sc_time_stamp() << name()
         << " transport function is called with address "
         << hex << arg_Req.getAddress() << endl << endl;

    myPVResp theResp;
    theResp.setResponse(pvOk);
    return theResp;
};
```

### 2.6.2 DMA model

A DMA can be modeled at PV as a target module. When the processor programs the DMA registers the DMA functionality can be triggered/called at the same time and executed in the context of the calling initiator. In that case the DMA will access the bus and will execute the requested transfer (in no time), see Figure 5. The execution sequence for a DMA starts with an initiator (1), usually a processor model. The SW will setup the DMA by programming the start and end addresses and then enable the DMA through some other memory mapped access(2). When this last access happens the DMA function can be immediately called (3), in this function the DMA can be performed by moving data from one target(4) to the other (5), after all of this the function returns to the initiator (6).

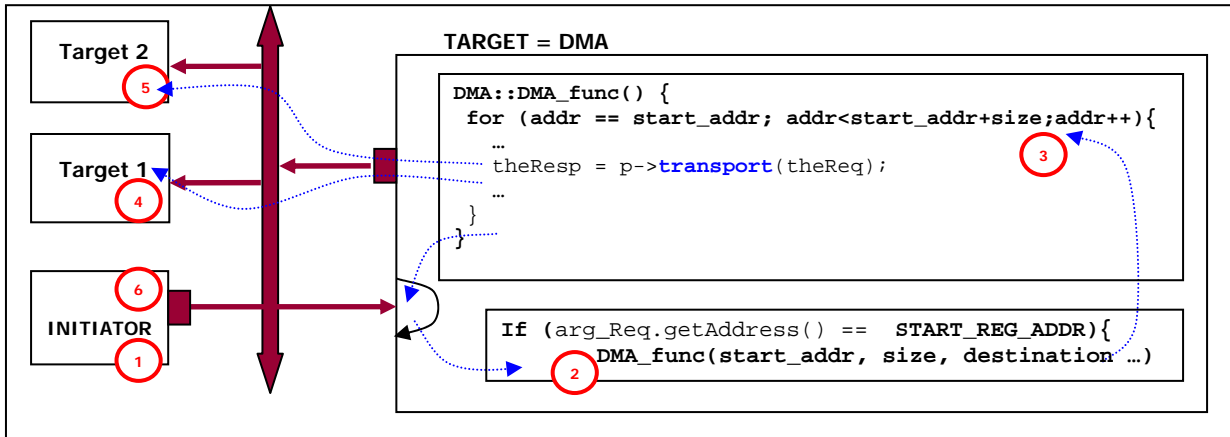


Figure 5: DMA model

It is possible to use latency in the response structure to indicate how many cycles were ‘stolen’ from the processor cycles, and we could use the custom data field in the request to the interrupt pin of the processor to indicate how many more cycles the processor should process before it should see the interrupt, however that’s typically not required.

### 2.6.3 Simple initiator

A PV initiator contains an `sc_thread` in order to be scheduled by the SystemC kernel, a `wait(n, SC_NS)` call is used to allow the SystemC scheduler to switch to another master. The value of ‘n’ can be calculated from the clock period or expected instruction rate and the number of instructions or clock ticks that are estimated between different synchronization points. The example below shows the basics of a processor model that has a memory port (initiator) and a interrupt port (target).

```
class PVMaster : public sc_module{
private :
    unsigned int i,j;
    unsigned int interrupt;
public:
    PVMaster(sc_module_name name);
    void theInitThread();

    PVInitiator_port<uint, int> p1;
    PVTARGET_port<uint, int> p2;

};
```

Since we do not expect an interrupt every cycle an `ATOMIC_BURST` parameter is defined which indicates the number of memory accesses or instructions that need to be performed before the model checks whether an interrupt occurred. This does introduce some inaccuracy since the exact timing of the interrupt or the exact instruction at which the software execution is interrupted is not

known, but most embedded SW does not rely on this. As such this still allows for a functional verification of the embedded SW.

```
PVMaster::PVMaster(sc_module_name name): sc_module(name),p("p"){
    SC_HAS_PROCESS(PVMaster);
    SC_THREAD(theInitThread);
    REGISTER_PVSLAVE( p2, receive_Int);
    i = 0; interrupt = 0;
};

myPVResp PVSlave::receive_Int(const myPVReq& arg_Req){
    intterrupt = arg_Req.getData();
}

void PVMaster::theInitThread(){
    while (1){
        if (interrupt == 1) {
            interrupt_handler();
        } else {
            for (j=0;j< ATOMIC_BURST;j++) {
                cout << sc_time_stamp() << "Initiator " << name();
                PVReq theReq;
                PVResp theResp;
                theReq.setAddress(i+ (0x10 * (i%2)));
                theResp = p.transport(theReq);
            }
            wait((SysClkPeriod * ATOMIC_BURST),SC_NS);
        }
    }
};
```

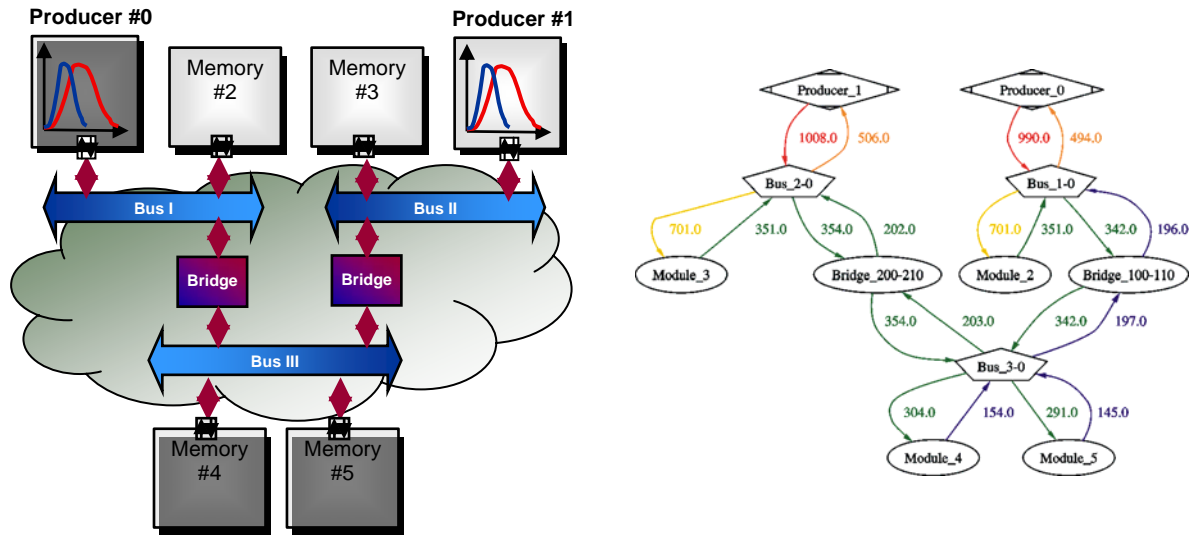
### 3 Architect's View: AV

#### 3.1 Introduction

The key problem that an architect's view model needs to address is to enhance the early architecture trade-off for system design, and to enable to set the right design constraints for HW and SW implementation teams. The solution space for this problem is very large, as a consequence designer experience is required to find the best solution. The goal of the modeling style is to enable the designer to evaluate different design decisions against each other. The evaluation is done through simulation and analysis.

In order to have good design decisions for the architectures a sufficiently accurate model of the communication and/or memory requirements needs to be derived from the system requirements. As a consequence the modeling style will focus on communication timing and resource sharing (point2point, shared bus, ...) as well as on data size. The modelling style should work well when

working with system requirements and not with an implementation of these requirements in which case it's dealing mostly with high level versions of the system functionality and estimates. On the other hand a lot of today's designs reuse elements of previous designs, so the modelling style should be open enough to allow to insert more detailed implementation models (like ISS's and refined TLM models) as well as functionally complete models.



**Figure 6: Architectural exploration**

The Architect's View (AV) has sufficient timing to quantify overall performance and to identify the potential bottlenecks in the system. Timing can be either explicit or implicit.

- Explicit timing means that timing is modeled in the initiators and targets, by using events and event synchronization or the 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 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. For performance profiling purposes, the basic timing characteristics of the target architecture can be expressed by the temporal relationship of transactions. This relationship is completely independent from the pure functionality, it only reflects different implementation strategies. Different block implementations are explored by annotating a delay time that is calculated from different timing models.

### 3.2 Protocol Stack built on a Channel implementing 4 Unidirectional Non Blocking interfaces

An Architectural modeling style can use an interface that is very similar to the moded extensions to the unidirectional non-blocking interface as described in [1] paragraph 4.3.1. This interface provides with the necessary flexibility to control the synchronization between models. In the architectural modeling view a channel is used to connect the different models in the system.

This channel has an initiator interface that combines a `t1m_extended_put_if` for the request, and a `t1m_extended_get_if` for the response. At the target side, the opposite is used (`t1m_extended_get_if` for request, and `t1m_extended_put_if` for response). An example of such a channel is the `t1m_req_rsp_channel` in the TLM proposal.

The basic get method in the interface performs three tasks:

1. read the data (this is what the “peek” function does)
2. consumes the data (free the buffer used to store the data)
3. notify the event (this is what the “unshrink” function does)

Besides “peek” and “unshrink” the extended proposal also provides “shrink” which combines the first two actions and “pop” which combines the last two actions.

In this section we consider a channel that uses these extended interfaces but has a set of convenience functions that allow it to support both implicit and explicit timing modeling.

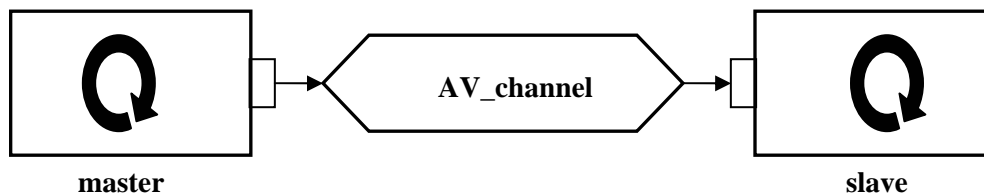


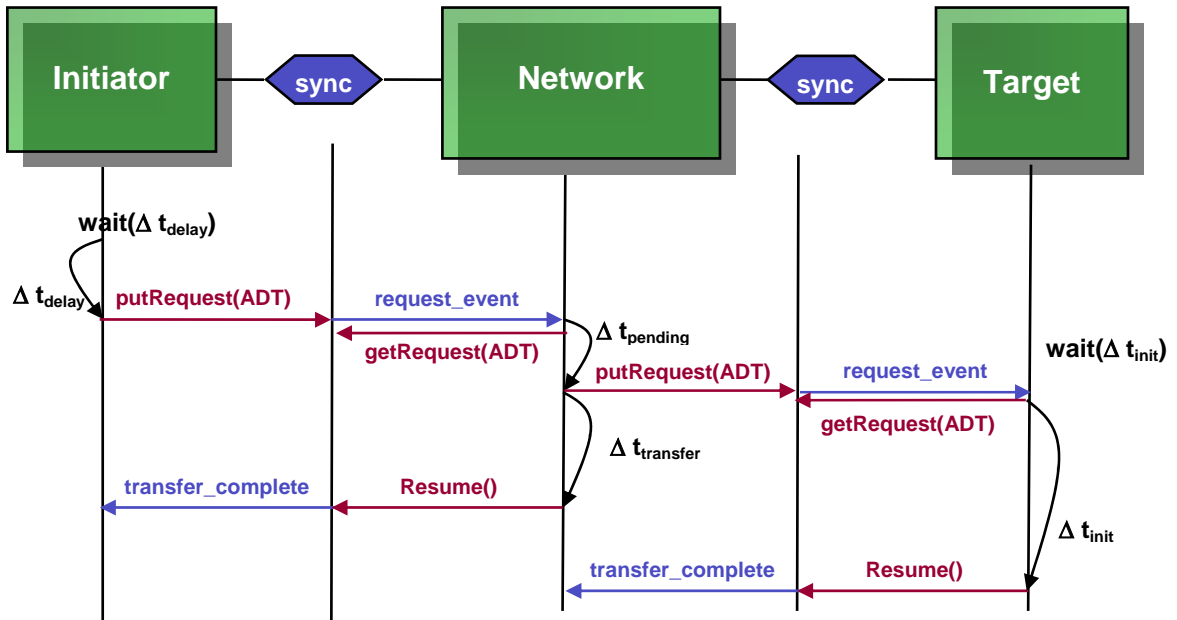
Figure 7: using the AV\_channel for functional descriptions

### 3.3 Data Structures

The data accuracy is typically at the packet level, i.e. the considered data granularity are sets of functionally associated data, which are combined to Abstract Data Types (ADTs). The REQ and RESP abstract data types are application dependent, ranging from the data structures outlined for use with the programmers view, augmented with some data members to do implicit timing, to structures carrying large amounts of data, e.g. a complete IP packet.

### 3.4 Implementation in System C: Explicit Timing

A request is put in the channel by the initiator (I), transported by the network (N) to the target (T). The initiators and targets are connected to the network by means of a synchronization channel (C). The sequence of events for a typical request interaction is as follows (see also Figure 8):



**Figure 8: architecture modeling with explicit timing**

1. (I) calls `wait( sc_time )` to model its initial delay time
2. (I) `putRequest( ADT )`: calls `nb_put()` to put the abstract data type token in the channel
3. (C) notify `ok_to_get()` event, here labeled `request_event`; this corresponds to start of request transfer event
4. (N) issue a `getRequest()` to obtain the token. In this case there is no delay time argument in the `getRequest` call, so the request completion is stalled. The initiator to network channel can be configured to release immediately, i.e. to free the buffer in C instantaneously.
  - a. Normal mode (immediate release): `shrink()` is called to obtain the token and to free the buffer in the channel.
  - b. Delayed release mode: `peek()` is called to obtain the token and the buffer in the channel is not freed.
5. (N) The time pending is the time spent in queuing up for the network resources and the addressed target.
6. (N) issues a `putRequest()` to put the token on the channel for the target.
7. (N) computes transfer timing, this is the timing it takes to complete the request transfer, the network will schedule when to complete the transfer (step 11)
8. (C) notifies `ok_to_get()`, here labeled `request_event`; this corresponds to start of request transfer event.
9. (T) issues a `getRequest()` to obtain the token. In this case there is no delay time argument in the `getRequest` call, so the request completion is stalled. The target does not release immediately, i.e. does not free the buffer in C instantaneously. So `peek()` is called to obtain the token and the buffer in C is not freed.
10. (T) calls `wait( sc_time )`;

11. (N) calls `resume()` to release the channel to I after the transfer time has elapsed. Depending on how this channel has been configured (see step 4)
  - a. Normal mode (immediate release): `unshrink()` is called to notify the `ok_to_put()` event.
  - b. Delayed release mode: `pop()` is called to free the buffer in the channel to notify the `ok_to_put()` event.
12. (T) calls `resume()` after its explicit wait time is over. Since the target does not release immediately, this calls `pop()` to free the buffer in the channel and to notify the `ok_to_put()` event.

### 3.5 Implementation in System C: Implicit Timing

The differences with explicit timing are outlined here. The `putRequest` in the initiator and the `getRequest` in the target are different. They take an `sc_time` argument to annotate the delay time. Based on this time value the actions that need to be done following this call are scheduled.

The sequence for the initiator's `putRequest` with implicit timing is as follows.

- (I) `putRequest( ADT, sc_time )` calls `nb_put()` to put the abstract data type token in the channel and to inform the channel of the delay time
- (C) queues the request for further processing
- (C) the `ok_to_get()` event gets notified after the delay time

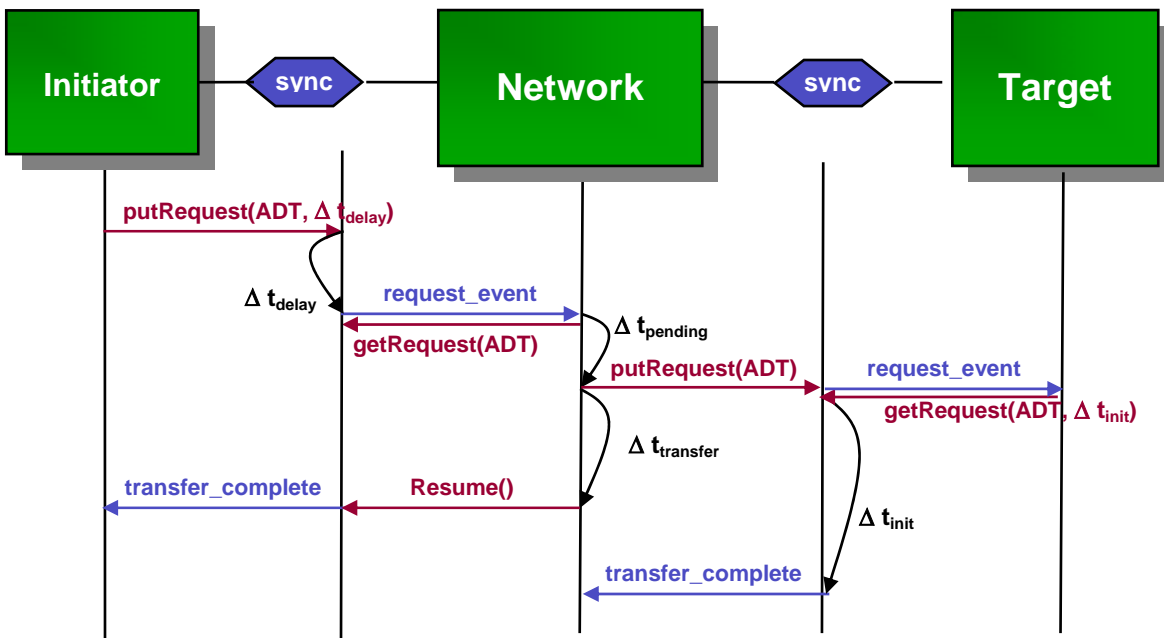


Figure 9: architectural modeling with implicit timing

The sequence for the target's `getRequest` with implicit timing is as follows.

- (T) `getRequest( ADT &, sc_time )` calls `peek()` to get the abstract data type token from the channel and to inform the channel of the delay time
- (C) queues the `pop` action corresponding to this `getReques` for further processing
- (C) calls `pop()` after the implicit wait time is over to free the buffer in the channel and to notify the `ok_to_put()` event.

### 3.6 Convenience functions in Protocol Layer

The protocol layer for such a architectural modeling channel can build a set of convenience functions on the transport layer and offer these to the user layer, as an example the table below lists a number of functions. The protocol layer will also need to provide with the data type definition to deal with timing annotation etc.

<b>putRequest</b>	forwards the request to the interface method in the transport layer
<b>getRequest</b>	obtains the request from the interface method in the transport layer
<b>putResponse</b>	forwards the response to the interface method in the transport layer
<b>getResponse</b>	obtains the request from the interface method in the transport layer
<b>next</b>	is an implicit timing call to schedule the release of the channel
<b>stall</b>	is an explicit timing call to hold the channel (and implies resume will be called later)
<b>resume</b>	is an explicit timing call to release the channel
<b>Read or Receive</b>	takes an address, builds a request structure, calls transport and returns the data
<b>Write or Send</b>	puts the address and data item in a request structure and calls transport
<b>Fetch</b>	for a burst read for a cache line fill

### 3.7 Mixing modeling styles

When creating architectural models it would be good to be able to reuse models from the programmers view. In this case adaptors need to be created that convert the ‘transport’ protocol of PV into the architectural modeling interface. These need to take care of the synchronization as well as the data type conversions. In pseudo code an adaptor that connects a PV initiator to a AV channel looks like below:

```
RESP Transport(const & REQ){
    putRequest (REQ);
    wait (ok_to_get_response);
    getResponse (RESP);
    return RESP;
}
```

It is important to know that re-entrant code is required when multiple threads can call this transport function (see also Note III). When connecting an AV channel to a PV target an adaptor is needed that contains a thread since the bidirectional transport API is blocking and thus needs to be called from a thread. Pseudo code for such a thread is listed below:

```
My_thread {
  wait(ok_to_get_request);
  get_request();
  RESP transport(REQ);
  PutResponse(RESP);
}
```

**Note II. SW debugging and PV-AV runtime switching.** Also for architectural models a processor model can be used to run embedded SW to analyze system performance. With a simple PV-AV adaptor an instruction accurate ISS can be connected to an architectural view. However we need to ensure that SW debugging activities do not influence the system model. Typically it is required to view the content of the registers in the peripherals to functionally verify a model. When a debugger wants to access these values it would do a transaction over the bus. Also when loading the program content in memory the processor model would do transactions over the bus. To ensure this does not influence analysis results or changes the state of the interconnect model there is a need for a specialized debugging interface.

This can be done by adding a PV interface to the AV interfaces which allows to create an 'invisible channel' connection between initiator and target. The advantage of this approach is that all address and data translations that happen along the way can be performed in the same way as for a regular AV transaction. It requires to separate all behavioral code from the transport and AV threads/methods, so that it can be reused without implications on other port accesses.

This approach can also be used to enable 'run-time' switching of modeling style. By adding a simple switch in all 'PV' initiators in the system you can select between running in 'PV' mode or in 'AV' mode. This is useful in cases where an architecture needs to be evaluated based on applications running in an OS. Here it is possible to boot the OS in PV mode and then run the application in AV mode. This will give an accurate analysis of the impact of the application on the architecture.

### 3.8 A simple memory model

The memory model is implemented as a SC\_METHOD sensitive to the event which is notified when a request arrives.

```
SC_METHOD( request_activate);
dont_initialize();
sensitive << req_port.request_event();
```

The convenience function used here is

```
void put_request( const Token & token, const sc_time & delay).
```

Its implementation is straightforward: it calls the non blocking put (“nb\_put”) interface method and performs the timing delay annotation.

```
void SimpleMemory<Token>::request_activate() {
    Token req = P.get_request();
    int address = req.GetDestId();
    int src_id = req.GetSourceId();
    bool is_read = (req.GetRequestType() == READ_REQ);
    if (is_read) {
        Token data = m_memory[address];
        data.SetDestId(src_id);
        data.SetSourceId(m_resp_port_id);
        sc_time processingDelay =
            m_processingDelay * sc_get_default_time_unit();
        P.put_request(data, processingDelay);
    } else {
        m_memory[address]=req;
    }
    // release input port after initiation interval
    sc_time iterationInterval =
        m_iterationInterval * sc_get_default_time_unit();
    P.next(iterationInterval);
}
```

## 4 Verification View: VV

### 4.1 Introduction

In the previous section we have seen how to create a transactional modeling interface that is easy to use, generic and provides with good simulation speed. But the API is not capable of modeling all timing intricacies of every conceivable protocol. As design refinement goes down in abstraction level there is a point where all cycle timing details of a protocol are important. For a certain class of interconnect strategies the cycle by cycle interaction of design modules with the interconnect have a large impact on the overall system performance. Also for detailed HW design more cycle timing information is required. The detailed HW design can happen in SystemC but also using hardware description languages as VHDL and Verilog. In all these cases there is a need to reuse the models we discussed before. There are 2 approaches to this problem one is to use transactors (TLM modeling style adaptors) to connect the refined models to a TLM system description. This allows to verify whether a refined block still implements the original functionality. To verify the overall system timing all models need to be refined. It does not allow to verify the impact of the cycle by cycle behavior of the interconnect.

It is however possible to take transactional modeling further down into the refinement process, and to create a transactional modeling style that has all the required cycle timing detail for hardware design. This is a style to create the verification view described in this section. The verification view contains enough detail to enable cycle accurate HW development and verification. Hence cycle accurate bus models are required as well as transactors to enable co-simulation with RTL, where signals are used rather than a TLM API.

## 4.2 Protocol Stack built on many Transfers

The verification view builds on the same unidirectional nonblocking interfaces as used in the architecture view. The architecture view uses 2 basic interfaces or transfers to create a protocol. A protocol implementation for the verification view uses more transfers to add more detail to the transactional modeling protocol. A transaction is a single object that encompasses a sequence of signals and handshakes required for system components to exchange data. A transfer is an atomic operation, such as the transfer of an address or data. It represents a breakdown of a transaction into finer detail. This detail may be necessary for modeling, e.g. to allow to start a transaction when some of the data is still unavailable. With every transfer a set of attributes are associated which represent the information of the transaction, examples of such information are address, type, size, etc. Using the nonblocking interfaces of the TLM proposal every transfer has a method and an event to implement communication and synchronization.

The unidirectional nonblocking interface, provided by OSCI, is used as the lowest layer, in the protocol stack for VV. The protocol layer builds a set of convenience functions on the transport layer and offers these to the user layer. Let's consider a few examples of protocol layer convenience functions that will typically be implemented in the ports.

- Send a transfer, which will call `nb_put`.
- Test whether a transfer can be sent, by calling `nb_can_put`
- Receive a transfer, using the `nb_get` call.
- Test whether a transfer can be received, by calling `nb_can_get`.
- Get access to the attributes of a transfer or transaction, either through the parameters of the API or through some other safe access method, we'll use an example of the latter in this document ('obtain').

These convenience functions can obviously be inlined and provide all the services required so the user code does not have to call the transport layer directly. Note that the peripherals can be modeled using a `SC_METHOD` since the transport layer is non blocking in the OSCI sense, i.e. it is method safe, meaning implementations are not allowed to call the `wait()` family of functions.

The convenience functions outlined above are transfer based. Since we multiple transfers are used to cycle accurately model a transaction, several sets of these convenience functions are required. An initiator port will typically have the obtain and the put interfaces implemented for the address and write data transfers and will have the get interface implemented for the read data and end of transaction transfers.

The non blocking interfaces also provide events, which are notified when it is ok to get or put, hence the name. At the time the event is notified, the corresponding `can_get` or `can_put` observer function will return true and the call the corresponding get or put will succeed. These events can be used for

- dynamic sensitivity, by calling `wait()` in an `SC_THREAD` and,
- static sensitivity, by declaring an `SC_METHOD` or `SC_THREAD` sensitive to an event finder for this event.

### 4.3 Data Structures

The get and put functions are unidirectional. The transaction data which is transported is split out in multiple structures, called transfers. A transfer typically contains attributes which are logically grouped together in the protocol and have the same timing. Typical examples of transfers sent from the initiator to the target are:

- Address Transfer containing address, type of transaction, access size, burst information, etcetera.
- Write Data Transfer containing the actual write data.

Typical examples of transfers sent from the target to the initiator are:

- Read Data Transfer containing the read data (the actual data and not a pointer or reference).
- End of Transaction transfer containing the status response (typically OK, error, etcetera).

Transactions are built by combining several transfers, e.g. a simple pipelined protocol could be built by defining a write transaction as the sequence of address transfer, write data transfer and EOT transfer, and making sure the write data transfer gets sent a clock cycle later than the address transfer.

It is advantageous to store the data structures containing these transfers in a centrally allocated circular buffer, which is controlled by the bus functional model. This is motivated by the fact that it is easier to ensure the required thread safety and lifetime properties for the transfers when the data storage and control are centralized rather than distributed. Pointers or references to these data structures can be obtained from the bus model (using the “obtain” call in an extended interface). Consequently there is no need to copy these transaction data structures ever, which is clearly advantageous for simulation speed and memory requirements.

It is important that it is understood how the channel manages these data structures. In this modeling style it is clear that only the channel is allowed to create and delete these structures. Initiator and target modules are not allowed to delete these structures. This implies the channel should decide how long it keeps these structures alive, this can either be the time a transaction is valid, or it's possible to use an ‘endOfTransaction’ method that indicates when the target finished processing the transaction and allows the channel to delete the data structure.

### 4.4 Implementation in SystemC

The nonblocking interfaces are used in the base layer. In order to implement access to the centralized circular buffer for transaction data storage, the obtain interface is provided.

```

template< typename T >
class tlm_obtain_if : virtual public sc_interface {
public:
    virtual bool obtain( T & ) = 0;
};

typedef BusTypeDefs::pAddrTrf PAT ;
typedef BusTypeDefs::pEotTrf PET ;
typedef BusTypeDefs::pReadDataTrf PRT ;
typedef BusTypeDefs::pWriteDataTrf PWT ;

```

The initiator port is composed of multiple `sc_port` objects, hence we have an `sc_port` for every transfer. In the code examples used below, `AddrTrfPort` is used for the address transfer, `WriteDataTrfPort` for the write data transfer, `ReadDataTrfPort` for the read data transfer and `EOTTrfPort` for the EOT transfer

In the code below the definition of a number of convenience functions on the compositionport are

```

class InitiatorPort : public sc_port<VV_TLM_Initiator_if> {
public:
    // portlist
    sc_port<AddrTrf_if> AddrTrfPort;
    ...
    // convenience data member
    PAT AddrTrf ;
    PET EotTrf ;
    PRT ReadDataTrf ;
    PWT WriteDataTrf ;
    ...
    // convenience functions
    PAT getAddrTrf();
    bool sendAddrTrf( PAT t );
    bool canSendAddrTrf();
    sc_event_finder & getSendAddrTrfEventFinder();
    ...
};

```

shown.

1. The `getAddrTrf` convenience function obtains an address transfer from the centralized circular buffer.
2. Sending an address transfer boils down to putting address transfer through.
3. A query function to test whether a transfer can be sent calls `nb_can_put`. Note that the write data transfer is processed over another `sc_port`.
4. Receiving a read data transfer boils down to getting the pointer.
5. A query function to test whether a transaction can be received is easily built on top of the `nb_can_get`.
6. For static sensitivity an event finder is required.

```

inline PAT InitiatorPort::getAddrTrf() {
    if( ! AddrTrfPort->obtain( AddrTrf)) { AddrTrf = 0; }
    return AddrTrf ;
}

inline bool InitiatorPort::sendAddrTrf( PAT t ) {
    return AddrTrfPort->nb_put( t);
}

inline bool InitiatorPort::canSendWriteDataTrf() {
    return WriteDataTrfPort->nb_can_put();
}

inline PRT InitiatorPort::receiveReadDataTrf() {
    if( ! ReadDataTrfPort->nb_get( ReadDataTrf )) { ReadDataTrf = 0; }
    return ReadDataTrf ;
}

inline bool InitiatorPort::canReceiveEotTrf() {
    return EOTTrfPort->nb_can_get();
}

inline sc_event_finder & InitiatorPort::getReceiveEotTrfEventFinder() {
    return( * new sc_event_finder_t< EOTTrf_if >
            (EOTTrfPort, &EOTTrf_if::findGetEvent));
}

```

#### 4.5 Relations between Transfers of same Transaction

Below are some code snippets of the cycle accurate TLM implementation of a slave in System C. Transfer sensitive threads are being used.

```

SC_THREAD (wDataTr);
sensitive << p.getReceiveWriteDataTrfEventFinder ();
SC_THREAD (rDataTr);
sensitive << p.getSendReadDataTrfEventFinder ();

```

The write data transfer sensitive thread is triggered when the slave has to sample the data from the WDATA pins. The read transfer sensitive thread is triggered when the slave has to drive the data on the RDATA pins. In case a back to back write and read transaction are performed using a pipelined protocol, it is possible that the slave samples the write data (for the first transaction) from the WDATA pins and drives read data on the RDATA pins (for the second transaction) at the same point in time.

Given a (read or write) data transfer, the so-called cross references allow access to the address corresponding to this transaction. So for the write and read transfers the address is obtained as

```

P2.WriteDataTrf->getAddrTrf()->getAddress();
...
p3.ReadDataTrf->getAddrTrf()->getAddress();

```

These 2 pieces of code are executed at the same point in time and result in a different address. The address for the read transaction is sampled of the ADDR pins; while the address for the write transaction is obtained from a register that contains the address sampled earlier of the ADDR pins.

Hence we have a coding style which is independent of relative timing of address and corresponding data. A typical write transaction can be coded as follows.

```
void Slave::wDataTr () {
    for (;;) {
        wait ();
        if (p.getWriteDataTrf ()) {
            const unsigned int data = p.WriteDataTrf->getWriteData();
            const unsigned int address =
                p.WriteDataTrf->getAddrTrf()->getAddress();

            // more code
        }
    }
}
```

Handling a read transaction would typically look like this.

```
void Slave::rDataTr () {
    for (;;) {
        wait ();
        if (p.getReadDataTrf ()) {
            const unsigned int address =
                p.ReadDataTrf->getAddrTrf()->getAddress();
            const unsigned int data = address + (address << 16);
            p.ReadDataTrf->setReadData (data );
            p.sendReadDataTrf ();
        }
    }
}
```

It is clear that cross references can be implemented efficiently when a centralized memory allocation is used for the transaction data structures. A transfer is a partial view on a transaction. Hence if the transactions are centrally allocated, it is easy to provide access from one view to another view, i.e. another transfer. If on the other the individual transfers are passed by value (which is the opposite of the transactions being centrally allocated), implementing these cross references becomes a tedious and error prone task. In this case one transfer needs to store a pointer to the other transfer. Now if an address transfer is being copied (e.g. because it is used as an argument in a pass by value function call), the copy constructor of the address transfer needs to update the pointers in all read and write transfer that refer to this address transfer. Hence a two way reference scheme would be necessary and a lot of pointer fiddling to keep the cross references consistent would be required. These considerations lead us to believe that it is advantageous to make it not possible to copy transfers to allocate the transactions in a centralized buffer.

**Note III. Thread Safety:** In transactional modeling, just as with all SystemC modeling, data is transferred between modules. Often this information is passed between different threads, whenever that happens special care is required to ensure that the right data ends up in the right place, as already discussed in Appendix B of [1]. Special care is required to ensure that when different threads can access the same data, that this data is not deleted too soon, or not at all, or that the data is overwritten by another thread before the first finished. As SystemC is a modeling library within C++ it is important that SystemC users understand the issues of cooperative multithreading in C++, as much as they need to understand the basic aspects of C++ modeling. This implies that the code of a blocking interface method needs to be re-entrant and manage the data it uses accordingly.

- As mentioned in [1] there are 3 rules for safety: first of all it is important to try to have the master to manage the data, this means to have the data managed inside the object that contains the SystemC thread. Secondly it is possible to use shared data pointers or to have some other mechanism to manage the data, this does not protect against editing the data and requires attention to coding style. A third option is to copy data whenever there is a `wait()` call in a target module, or not to call `wait()` at all. This is actually the safest method since it ensures that there is a unique copy per communication. However this could have a severe impact on simulation speed. In the examples discussed above the second method to maintain safety is chosen, since it allows an easy TLM coding style, and provides with the required safety and speed. In the PV modeling style all data is managed from the initiator as for the AV modeling style
- Another aspect of safety is to consider the local data of the modules, as several threads can access the same peripheral module they can overwrite local variables of the module. E.g. a peripheral can contain some variables that store the state of the module between interface method calls, whenever another thread accesses the same peripheral it is possible this thread updates these variables before the first one finished, causing the first to continue with the wrong data values. In more general terms: whenever 2 different threads have access to the same data there is some 'protocol' that needs to be agreed to ensure data coherency, e.g. if I have a state variable in an `sc_module`, I need to make sure I can predict its value before and after a 'wait()' call in a blocking bidirectional transport call.
- As a general rule it is advised to avoid `wait()` calls in the modules that implement a TLM interface hence the latency parameter in the PV protocol implementation and the use of non blocking unidirectional interfaces with implicit timing style in AV.

#### 4.6 Examples of Verification view modeling

In the first example the block is modeled with a clocked thread where on every clock cycle there is a check for the transfers that are arriving or can be sent by this peripheral. With this coding style there is still a lot of unnecessary simulation events modeled but when needed it would allow to model the timing of the peripheral very accurately. In the constructor of the module we create a process sensitive to the clock.

```
SC_METHOD(Commprocess);
sensitive << bus_clock.pos();
dont_inititalize();
```

The implementation of the clocked thread looks like below.

```
void CommProcess() {
    if(P1.getWriteDataTrf()) {
        myvar = P1.WriteDataTrf->getWriteData();
    }
    ...
    if(P1.getEotTrf()) {
        P1.sendEotTrf();
    }
    ...
}
```

As an alternative way to model a peripheral we can look at a second example where we model the same peripheral using transfer sensitive events. In this case there are 2 threads, but each will only be triggered when there is an actual transfer available for that peripheral or when it is possible for that peripheral to send a transaction. This coding style leads to an enormous speed improvement.

```
SC_METHOD(receiveWriteData);
sensitive << P1.getReceiveWriteDataTrfEvenFinder();
dont_inititalize();

SC_METHOD(sendEot);
sensitive << P1.getSendEotTrfEventHandler();
dont_initialize();
```

The implementation of the 2 transfer sensitive threads looks as follows:

```
void receiveWriteData() {
    P1.getWriteDataTrf();
    myvar = P1.WriteDataTrf->getWriteData();
}

void sendEotT() {
    P1.sendEotTrf();
}
```

## 5 Conclusions

In this document an overview was given of the different TLM modeling styles and how they can be applied to the different SoC design tasks. The API's and their usage was presented in some detail, but by no means is this a complete guide to TLM modeling. The intention is to show how to apply TLM modeling techniques and how to create a TLM modeling style for a specific purpose.

## 6 References

- [1] Adam Rose, Stuart Swan, et.al., “Transaction level modeling in SystemC”, work in progress
- [2] Burton, M and Donlin, A, “Transaction Level Modeling : Above RTL design and methodology”, Feb 2004, internal OSCI TLM WG document.
- [3] “Functional Specification for SystemC 2.0”, Version 2.0-Q, April 5<sup>th</sup> 2002 available at [www.systemc.org](http://www.systemc.org)
- [4] Thorsten Grotker, Stan Liao, Grant Martin, Stuart Swan, “System Design with SystemC”, Kluwer Academic publishers.