Transaction Level Modeling in SystemC

Adam Rose, Stuart Swan, John Pierce, Jean-Michel Fernandez

Cadence Design Systems, Inc

ABSTRACT

In the introduction, we describe the motivation for proposing a Transaction Level Modeling standard, focusing on the main use cases and the increase in productivity such a standard will bring. In Section 2, we describe the core tlm proposal in detail. Section 3 shows refinement of a single master / single slave from a programmers view model down through various levels of abstraction to an rtl only implementation. Section 4 shows how to code commonly occurring System Level design patterns such as centralized routers, arbiters, pipelines, and decentralized decoding schemes using the proposed tlm standard. In order to do this we briefly describe and use some extensions to the core proposals. Section 5 shows how to combine and recombine the generic components in section 4 to explore different switch architectures.

In the first Appendix, we outline the uses of sc_export, which relied on in many of the examples. In the second Appendix, we briefly discuss some guidelines for using the TLM proposal in a concurrent SystemC environment in an efficient and safe way. The final appendix shows the TLM interface inheritance hierarchy.

Code for all the examples contained in this paper is available in the OSCI TLM kit available at www.systemc.org.

1. Introduction

Transaction Level Modeling (TLM) is motivated by a number of practical problems. These include :

- Providing an early platform for software development
- System Level Design Exploration and Verification
- The need to use System Level Models in Block Level Verification.

A commonly accepted industry standard for TLM would help to increase the productivity of software engineers, architects, implementation and verification engineers. However, the improvement in productivity promised by such a standard can only be achieved if the standard meets a number of criteria :

- It must be easy, efficient and safe to use in a concurrent environment.
- It must enable reuse between projects and between abstraction levels within the same project.
- It must easily model hardware, software and designs which cross the hardware / software boundary.
- It must enable the design of generic components such as routers and arbiters.

Since the release of version 2.0, it has been possible to do TLM using SystemC. However, the lack of established standards and methodologies has meant that each TLM effort has had to invent its own methodologies and APIs to do TLM. In addition to the cost of reinventing the wheel, these methodologies all differed slightly, making IP exchange difficult.

This paper will describe how the proposed OSCI TLM standard meets the requirements above, and show how to use it to solve various common modeling problems. We believe that widespread adoption of this proposal will lead to the productivity improvements promised by TLM.

2. The TLM Proposal

2.1 Key Concepts

There are three key concepts required to understand this proposal.

- Interfaces
- Blocking vs Non Blocking
- Bidirectional vs Uni Directional

2.1.1 Interfaces

The emphasis on interfaces rather than implementation flows from the fact that SystemC is a C++ class library, and that C++ (when used properly) is an object orientated language. First we need to rigorously define the key interfaces, and then we can go on to discuss the various ways these may be implemented in a TLM design. It is crucial for the reader to understand that the TLM interface classes form the heart of the TLM standard, and that the implementations of those interfaces (e.g. tlm_fifo) are not as central. In SystemC, all interfaces should inherit from the class sc_interface.

2.1.2 Blocking and Non Blocking

In SystemC, there are two basic kinds of processes: SC_THREAD and SC_METHOD. The key difference between the two is that it is possible to suspend an SC_THREAD by calling wait(.). SC_METHODs on the other hand can only be synchronized by making them sensitive to an externally defined sc_event. Calling wait(.) inside an SC_METHOD leads to a runtime error. Using SC_THREAD is in many ways more natural, but it is slower because wait(.) induces a context switch in the SystemC scheduler. Using SC_METHOD is more constrained but more efficient, because it avoids the context switching [2].

Because there will be a runtime error if we call wait from inside an SC_METHOD, every method in every interface needs to clearly tell the user whether it *may* contain a wait(.) and therefore *must* be called from an SC_THREAD, or if it is *guaranteed* not to contain a wait(.) and therefore *can* be called from an SC_METHOD. OSCI uses the terms blocking for the former and non blocking for the latter.

The OSCI TLM standard strictly adheres to the OSCI use of the terms “blocking” and “non-blocking”. For example, if a TLM interface is labeled “non-blocking”, then its methods can NEVER call wait().

OSCI Terminology	Contains wait(.)	Can be called from
Blocking	Possibly	SC_THREAD only
Non Blocking	No	SC_METHOD or SC_THREAD

2.1.3 Bidirectional and Unidirectional Transfers

Some common transactions are clearly bidirectional, for example a read across a bus. Other transactions are clearly unidirectional, as is the case for most packet based communication mechanisms. Where there is a more complicated protocol, it is always possible to break it down into a sequence of bidirectional or unidirectional transfers. For example, a complex bus with address, control and data phases may look like a simple bidirectional read/write bus at a high level of abstraction, but more like a sequence of pipelined unidirectional transfers at a more detailed level. Any TLM standard must have both bidirectional and unidirectional interfaces. The standard should have a common look and feel for bidirectional and unidirectional interfaces, and it should be clearly shown how the two relate.

2.2 The Core TLM Interfaces

2.2.1 The Unidirectional Interfaces

The unidirectional interfaces are based on the sc_fifo interfaces as standardized in the SystemC 2.1 release. Sc_fifo has been used for many years in many types of system level model, since the critical variable in many system level designs is the size of the fifos. As a result, the fifo interfaces are well understood and we know that they are reliable in the context of concurrent systems. A further advantage of using interfaces based on sc_fifo is that future simulators may be able to perform well known static scheduling optimizations on models which use them. In addition to this, the interface classes are split into blocking and non blocking classes and non blocking access methods are distinguished from blocking methods by the prefix “nb_”.

However, for TLM we have two new requirements

- We need some value free terminology, since “read” and “write” in the current sc_fifo interfaces are very loaded terms in the context of TLM
- These interfaces may be implemented in a fifo, some other channel, or directly in the target using sc_export.

To address the first of these concerns, when we move a transaction from initiator to target we call this a “put” and when we move the transaction from target to initiator we call this a “get”.

A consequence of the second requirement is that we need to add tlm_tag<T> to some of the interfaces. This is a C++ trick which allows us to implement more than one version of an interface in a single target, provided the template parameters of the interfaces are different.

2.2.2 The Unidirectional Blocking Interfaces

```
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;
};
```

Since we are allowed to call wait in the blocking functions, they never fail. For convenience, we supply two forms of get, although since we provide a default implementation for the pass-by-reference form , an implementer of the interface need only supply one.

2.2.3 The Unidirectional Non Blocking Interfaces

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

The non blocking interfaces may fail, since they are not allowed to wait for the correct conditions for these calls to succeed. Hence nb_put and nb_get must return a bool to indicate whether the nonblocking access succeeded. We also supply nb_can_put and nb_can_get to enquire whether a transfer will be successful without actually moving any data.

These methods are sufficient to do polling puts and gets. We also supply event functions which enable an SC_THREAD to wait until it is likely that the access succeeds or a SC_METHOD to be woken up because the event has been notified. These event

functions enable an interrupt driven approach to using the non blocking access functions. However, in the general case even if the relevant event has been notified, we still need to check the return value of the access function – for example, a number of threads may have been notified that a fifo is no longer full but only the first to wake up is guaranteed to have room before it is full again.

2.2.4 Bidirectional Blocking Interface

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

The bidirectional blocking interface is used to model transactions where there is a tight one to one, non pipelined binding between the request going in and the response coming out. This is typically true when modeling from a software programmers point of view, when for example a read can be described as an address going in and the read data coming back.

The signature of the transport function can be seen as a merger between the blocking get and put functions. This is by design, since then we can produce implementations of tlm_transport_if which simply call the put(.) and get(.) of two unidirectional interfaces.

2.3 TLM Channels

One or more of the interfaces described above can be implemented in any channel that a user cares to design, or directly in the target using sc_export. However, two channels seem to be useful in a large number of modeling contexts, so they are included as part of the core proposal.

2.3.1 tlm_fifo<T>

The tlm_fifo<T> templated class implements all the unidirectional interfaces described above. The implementation of the fifo is based on the implementation of sc_fifo. In particular, it addresses many (but not all) of the issues related to non determinism by using the request_update / update mechanism. Externally, the effect of this is that a transaction put into the tlm_fifo is not available for getting until the next delta cycle. In addition to the functionality provided by sc_fifo, tlm_fifo can be zero or infinite sized, and implements the fifo interface extensions discussed in 4.3.1 below.

2.3.2 tlm_req_rsp_channel<REQ,RSP>

The tlm_req_rsp_channel<REQ,RSP> class consists of two fifos, one for the request going from initiator to target and the other for the response being moved from target to initiator. To provide direct access to these fifos, it exports the put request and get response interfaces to the initiator and the get request and put response interfaces to the target.

As well as directly exporting these four fifo interfaces, tlm_req_rsp_channel<REQ,RSP> implements three interfaces. The first two combine the unidirectional requests and responses into convenient master and slave interfaces :

```
template < typename REQ , typename RSP >
class tlm_master_if :
  public virtual tlm_extended_put_if< REQ > ,
  public virtual tlm_extended_get_if< RSP > {};

template < typename REQ , typename RSP >
class tlm_slave_if :
  public virtual tlm_extended_put_if< RSP > ,
  public virtual tlm_extended_get_if< REQ > {};
};
```

In addition to this, it implements tlm_transport_if<REQ,RSP> as follows :

```
RSP transport( const REQ &req ) {
  RSP rsp;

  mutex.lock();

  request_fifo.put( req );
  response_fifo.get( rsp );

  mutex.unlock();
  return rsp;

}
```

This simple function provides a key link between the bidirectional and sequential world as represented by the transport function and the timed, unidirectional world as represented by tlm_fifo. We will explain this in detail in the transactor (3.4) and arbiter (4.2) examples below.

2.4 Summary of the Core TLM Proposal

The ten methods split into five classes described in Section 2.2 form the basis of the OSCI TLM proposal. On the basis of this simple transport mechanism, we can build models of software and hardware, generic routers and arbiters, pipelined and non pipelined buses, and packet based protocols. We can model at various different levels of timing and data abstraction and we can also provide channels to connect one abstraction level to another. Because they are based on the interfaces to sc_fifo, they are easily understood, safe and efficient.

Users can and should design their own channels implementing some or all of these interfaces, or they can implement them directly in the target using sc_export. The transport function in particular will often be directly implemented in a target when used to provide fast programmers view models for software prototyping.

In addition to the core interfaces, we have defined two standard channels, tlm_fifo<T> and tlm_req_rsp_channel<REQ,RSP>.

These two channels can be used to model a wide variety of timed systems, with the tlm_req_rsp_channel class providing an easy to use bridge between the untimed and timed domains.

3. Modeling a simple peripheral bus at various levels of abstraction

In this section, we will describe how to take an abstract single master / single slave model down through various levels of abstraction to an rtl only implementation. This ordering will be familiar to readers who are from a software background and want to understand how to incorporate real time hardware behaviour into their TLM models. We also discuss how to build a modeling architecture so that a meaningful interface can be presented to application experts while the underlying protocols can be accurately modeled. Hardware designers may find it easier to read this section backwards, starting with the rtl and abstracting to reach the programmers view model.

3.1 A Programmers View Architecture

In many organizations, there are two distinct groups of engineers. In fact, one of the primary goals of TLM is to provide a mechanism that allows these two groups of people to exchange models. The first group understands the application domain very well, but is not particularly expert in C++ nor interested in the finer details of the TLM transport layer or the signal level protocol used to communicate between modules. The second group does not necessarily understand the application domain well but does understand the underlying protocols and the C++ techniques needed to model them. Because of this divide in expertise and interests, it is often useful (but by no means compulsory) to define a protocol specific boundary between these two groups of engineers.

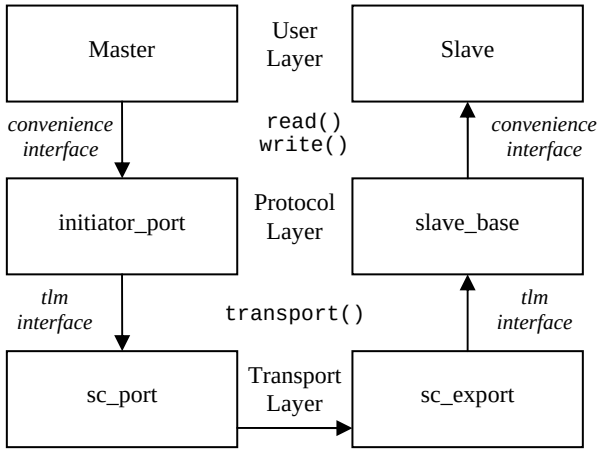


Figure 1 : Modeling Architecture

This interface is sometimes called the convenience interface. It will typically consist of methods that make sense to users of the protocol in question : for example, read, write, burst read and burst write. A user will use initiator ports that supply these interfaces, and define target modules which inherit from the these interfaces. The infrastructure team will implement the protocol layer for the users. This consists of the request and response classes that encapsulate the protocol, an initiator port that translates from the convenience functions to calls to RSP transport(const &REQ) in the port, and a slave base class that

implements RSP transport(const REQ &) in the slave. The infrastructure team then publishes the initiator port and slave base class to the users, who are then protected from the transport layer completely. In effect, we have a three layer protocol stack.

In all the subsequent examples, we use this architecture when we are modeling at the PV level. The consequence of this is that we can reuse the master code shown below while we refine the slave from an abstract implementation down to rtl.

```
void master::run()
{
  DATA_TYPE d;
  for( ADDRESS_TYPE a = 0; a < 20; a++ )
  {
    initiator_port.write( a , a + 50 );
  }
  for( ADDRESS_TYPE a = 0; a < 20; a++ )
  {
    initiator_port.read( a , d );
  }
}
```

In order to achieve this level of reuse and usability at the user level, the implementation team has to define an initiator port, and slave base class and protocol that allows these two classes to communicate.

3.1.1 The Protocol

At this abstract modeling level, the request and response classes used to define the protocol have no signal level implementation, they simply describe the information going in to the slave in the request and the information coming out of the slave in the response.

```
template< typename ADDRESS , typename DATA >
class basic_request
{
public:
  basic_request_type type;
  ADDRESS a;
  DATA d;
};

template< typename DATA >
class basic_response
{
public:
  basic_request_type type;
  basic_status status;
  DATA d;
};
```

3.1.2 The Initiator Port

On the master side, infrastructure team supplies an initiator port which translates from the convenience layer to the transport layer.

```
basic_status read( const ADDRESS &a , DATA &d ) {
    basic_request<ADDRESS,DATA> req;
    basic_response<DATA> rsp;
    req.type = READ;
    req.a = a;
    rsp = (*this)->transport( req );
    d = rsp.d;
    return rsp.status;
}
```

The write method is implemented in a similar fashion.

3.1.3 Slave Base Class

In the slave base class, we translate back from the transport layer to the convenience layer.

```
basic_response<DATA>
transport( const basic_request<ADDRESS,DATA>
&request ) {
    basic_response<DATA> response;
    switch( request.type ) {
    case READ :
      response.status = read( request.a ,
response.d );
      break;
    case WRITE:
      response.status = write( request.a ,
request.d );
      break;
    …
    }
    return response;
}
```

The read and write functions are pure virtual in the slave base class, and are supplied by the user's implementation which inherits from the slave base class.

3.1.4 Only Request and Response Classes are Compulsory

It is worth re-emphasizing that this modeling architecture is not compulsory. In this case, the infrastructure team only supplies the protocol itself and not the initiator port and slave base class. The consequence of this is that each master and each slave may have to do the translation to and from the underlying protocol described in the preceding two sections. While the examples below have been coded using a convenience layer, they could have been implemented directly on top of the transport layer.

3.2 PV Master / PV Slave

This example uses a single thread in the master to send a sequence of writes and reads to the slave. Both write and read transactions are bidirectional (although a write doesn't return data it does return protocol specific status information) so we use the bidirectional blocking interface, tlm_transport_if.

Using the protocol and the modeling architecture described above, we can produce a simple PV master / PV slave arrangement as shown below.[i]

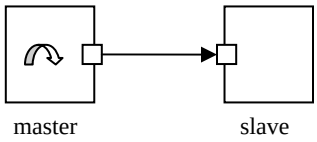


Figure 2 : PV Master / PV Slave

There is only one thread in this system, on the master side. The methods in the slave are run in the context of this thread, having been called directly by the master using the sc_export mechanism.

The user only has to do two things in the slave : bind its interface to the sc_export so that the master can bind to it as in the diagram above, and define read() and write().

```
mem_slave::mem_slave(     const     sc_module_name
&module_name , int k ) :
  sc_module( module_name ) ,
  target_port("iport")
{
  target_port( *this );
  memory = new ADDRESS_TYPE[ k * 1024 ];
}

basic_status
mem_slave::
read( const ADDRESS_TYPE &a , DATA_TYPE &d )
{
  d = memory[a];
  return basic_protocol::SUCCESS;
}

basic_status
mem_slave::
write( const ADDRESS_TYPE &a, const DATA_TYPE &d)
{
  memory[a] = d;
  return basic_protocol::SUCCESS;
}
```

[i] See Section 6 for the graphical conventions used in these examples

3.3 PV Master / tlm_req_rsp_channel / unidirectional slave

This example shows how to connect a master using the bidirectional transport interface to a slave which has unidirectional interfaces. As described above, to do this we use tlm_req_rsp_channel which implements the transport function as blocking calls to a request and a response fifo. The different modules are connected together as shown.

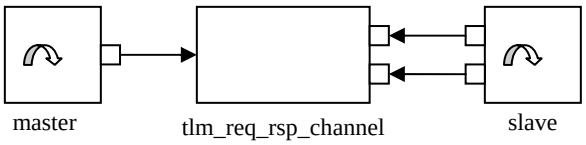


Figure 3 : PV master / unidirectional slave

The slave now models the separate request and response phases of the transaction, which is closer to the final implementation than the previous example. However, it pays a price in performance because we now have two threads in the system and have to switch between them.

The master is unchanged from the previous example, but the slave has a two sc_ports and a thread as shown below.

```
void mem_slave::run()
{
  basic_request<ADDRESS_TYPE,DATA_TYPE> request;
  basic_response<DATA_TYPE> response;

  for(;;)
  {
    request = in_port->get();
    response.type = request.type;

    switch( request.type )
    {
    case basic_protocol::READ :
      response.d = memory[request.a];
      response.status = basic_protocol::SUCCESS;
      break;
    case basic_protocol::WRITE:
    …
    }
    out_port->put( response );
  }
}
```

3.4 PV Master / transactor / rtl slave

We can now refine the slave further to a genuine register transfer level implementation. The example shows this rtl implementation

in SystemC, although in reality it may be in a verilog or vhdl and linked to SystemC using a commercial simulator.

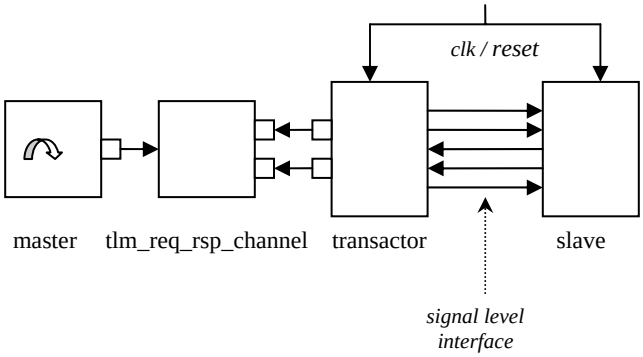


Figure 4 : PV Master / rtl slave

The key component in this system is the transactor. It gets an abstract request from the request fifo in the tlm_req_rsp_channel and waits for an opportunity to send this out over the rtl level bus. It then waits until it sees a response on the rtl bus, at which point it puts the abstract response into the response fifo in the tlm_req_rsp_channel. The master thread will then unblock because it has a response available to be "got" from the fifo.

In order to do all this, the transactor has to implement at least one state machine to control the bus., usually in an SC_METHOD statically sensitive to the clock. A consequence of using SC_METHOD is that we need to use the non blocking interfaces when accessing the fifos in tlm_req_rsp_channel.

If the slave and transactor use SC_METHODs, then the only thread in this system is the master.

3.5 RTL Master / RTL Slave

This example is a conventional rtl master talking across a simple peripheral bus to an rtl slave. While the example is implemented in SystemC using SC_METHODs statically sensitive to a clock, it could also be implemented entirely in vhdl or verilog and simulated using a commercial simulator.

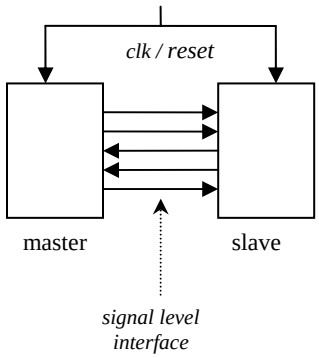


Figure 5 : RTL Master / RTL Slave

4. Typical SoC Modeling Patterns using the TLM Proposal

In the previous section, we showed how to refine a single master and single slave down to rtl using simple non pipelined peripheral bus. In this section we show how to model typical patterns found in more complicated SoC modeling problems.

4.1 Router

The first case we will look at is a common problem found in almost any SoC : how to route the traffic generated by one master to one of many slaves connected to it. When it comes to the final implementation, the decoding may be centralized or decentralized. In 4.4 we discuss how to do decentralized decoding. However, modeling the decoding as a centralized router is easier to code and faster to execute, so we discuss the router pattern first.

The basic pattern is shown below. The address map, router module and router port used in the diagram below are generic components. Provided a protocol satisfies some minimal requirements, this router module is capable of routing any protocol.

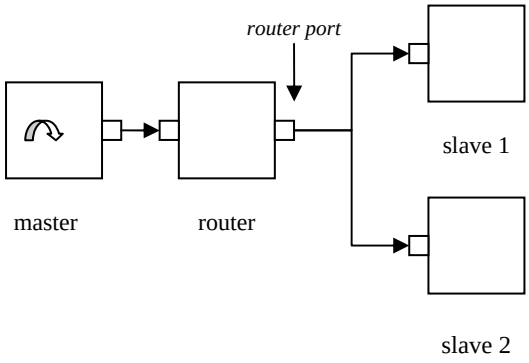


Figure 6 : Master, Router, Multiple Slaves

```
// an example address map
// slave one is mapped to [ 0 , 0x10 )
// slave two is mapped to [ 0x10, 0x20 )
slave_1.iport 0 10
slave_2.iport 10 20
```

The address map in the router is a mapping from an address range to a port id. In order to build up this mapping from a file such as the one above, we need to be able to ask the port in the router how the names of the slaves that it is connected to map to port ids. The generic component router_port<IF> adds this functionality to sc_port. Because router_port inherits from sc_port, in all other respects it behaves like an sc_port.

The router receives a request from the master. It attempts to find an address range which contains this address. If it is unable to find such a range, it returns a suitable protocol error. If it is successful, it subtracts the base address of the slave from the

request, forwards the adjusted request to the correct slave and finally sends the response back to the master.

```
RSP transport( const REQ &req ) {
  REQ new_req = req;
  int port_index;
  if( !amap.decode( new_req.get_address() ,
                    new_req.get_address() ,
                    port_index ) ) {
      return RSP();
  }
  return router_port[port_index]->
         transport( new_req );
}
```

As can be seen from the code above, we need to make two assumptions about the protocol to make it routable : the request must have a get_address function which returns a reference to the address, and the response's default constructor must initialize the response to an error state. We also need to assume the address is reasonably well behaved (eg it is copyable and has <, << and >> operators defined).

4.2 Arbiter

Arbitration is not quite as common a pattern as the routing pattern, since by definition we only need to arbitrate between two simultaneous requests when we have introduced time into our model. In a pure PV model which executes in zero time, arbitration is a meaningless concept. However, pure PV models are in fact quite rare and arbitration is often needed in TLM models.

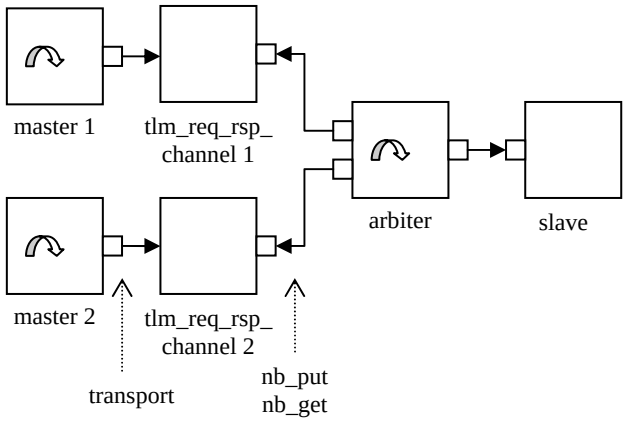


Figure 7 : Arbitration between Multiple Masters

The masters put a request into their respective tlm_req_rsp_channels and wait for a corresponding response. A separate thread in the arbiter polls all the request fifos (using nb_get) , decides which is the most important, and forwards the request to the slave. When the slave responds, the arbiter puts the response into the reponse fifo in the relevant tlm_req_rsp_channel. The master then picks up the response and completes the transaction. The key thread in the arbiter is below.

```
virtual void run() {
  port_type *port_ptr;
  multimap_type::iterator i;
  REQ req;
  RSP rsp;
  for( ;; ) {
    port_ptr = get_next_request( i , req );
    if( port_ptr != 0 ) {
      rsp = slave_port->transport( req );
      (*port_ptr)->put( rsp );
    }
    wait( arb_t );
  }
}
```

get_next_request() iterates over a multimap of sc_ports. These ports have been sorted in priority order, although the precise operation of get_next_request can be changed according to the arbitration scheme. A very naïve starvation inducing arbitration scheme is shown below, for illustration purposes, although we

would expect that this virtual function is overridden to do something more realistic.

```
virtual port_type *
get_next_request( multimap_type::iterator &i,
                  REQ &req ) {
  port_type *p;
  for( i = if_map.begin();
       i != if_map.end();
       ++i )
  {
      p = (*i).second;
      if( (*p)->nb_get( req )  ) {
       return p;
    }
  }
  return 0;
}
```

The multimap is a one to many mapping from priority level to port, which can be configured and reconfigured at any time. The code above will always get the highest priority port which has a pending request. Multimap is in the stl library and comes with many access functions useful for arbitration – for example, it is easy to find all ports of the same priority level for use in a prioritized round robin arbitration scheme.

4.3 Multiple Slaves with Decentralized Decoding

As discussed above, the easiest and most efficient way to model decoding is by using a centralized router. However, there are occasions when this technique diverges too far from the implementation it is modeling. In these cases, we need to use the decentralized decoding pattern.

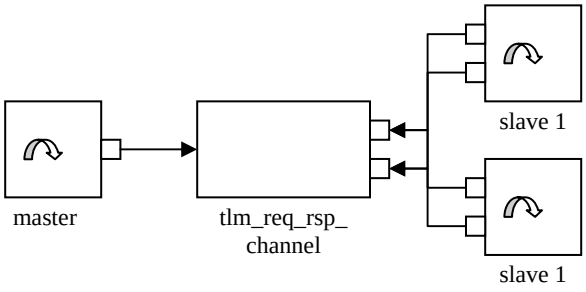


Figure 8 : Decentralised Decoding

The master is connected to the tlm_req_rsp_channel in the normal way. All of the slaves monitor all of the requests coming in. If the request is successfully decoded locally, one of the slaves tells the master that the request has been decoded. This unblocks the request fifo in the tlm_req_rsp_channel. The slave then goes on to process the request, and send a response to the response fifo in the tlm_req_rsp_channel, from where the master can pick it up.

4.3.1 The Moded Extensions to the Core TLM Interfaces

In the description of how the slave works above, it is apparent that the core tlm interfaces do not provide the functionality needed. The get() functions in the core interface all do three things. They *get* the data, they *consume* the transaction – ie successive calls to get will return a different transaction – and *notify* the master that they have done so, all in one function. The moded extensions split these three aspects up into separate functions.

In this example, each slave needs to peek at the same request to see if it can be decoded. If it can be decoded, only one of the slaves consumes the request, notifies the master that it has done so, and sends back a response. So for this example, we need one function to get the data without consuming the data or notifying the master, and a separate function to consume and notify. In other examples, we may want to get and consume in one function and notify but not consume in a separate one.

Similarly, on the put side, the moded extensions allow a master to overwrite the most recently put transaction, without waiting for the slave.

The put and get modes are :

```
enum tlm_get_type {
  NORMAL_GET , // consumes and notifies
  SHRINK ,     // consumes but doesn't notify
  PEEK ,       // neither consumes nor notifies
};

enum tlm_finish_get_type {
  UNSHRINK ,  // notifies
  POP         // consumes and notifies
};

enum tlm_put_type {
  NORMAL_PUT ,
  OVERWRITE
};
```

and they are used in the moded tlm interfaces :

```
template < typename T >
class tlm_moded_get_if :
public virtual sc_interface
{
public:
virtual bool get( T & , tlm_get_type ) = 0;
virtual bool nb_get( T & , tlm_get_type ) = 0;

virtual bool notify_got( tlm_finish_get_type )=0;
virtual bool nb_notify_got( tlm_finish_get_type )
= 0;
virtual bool nb_notify_got( const sc_time & ,
                        tlm_finish_get_type ) = 0;
```

```
};

template < typename T >
class tlm_moded_put_if : public virtual
sc_interface
{
public:
  virtual bool put( const T & ,
                    tlm_put_type ) = 0;
  virtual bool nb_put( const T & ,
                       tlm_put_type ) = 0;

};
```

4.3.2 Decentralised Decoding Slaves

As already described, the slaves need to get without consuming or notifying, and then consume and notify. The get type we need is PEEK, and the finish get type we need is POP. The code is shown below.

```
while( true ) {
  request_port->get( req , PEEK );
  if( decode( req.a ) {
    request_port->nb_notify_got( POP );
    rsp = process_req( req );
    response_port->put( rsp );
  }
  wait( request_port->ok_to_get() );
}
```

4.4 Pipeline

In Section 3, we started with a PV master connected to a PV slave, and refined first the slave and then the master down to an rtl description. If we use a weak definition of a Progammers View model (ie a model which does call wait() but which does not advance time) this example can be described as a PV model. We will leave it to the reader to do the refinement in a similar fashion to Section 3.

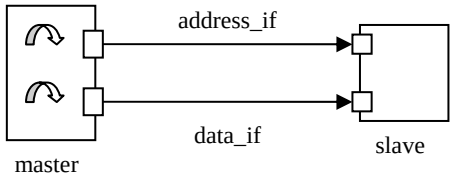


Figure 9 : Pipeline

The basic topology is shown above. Since the protocol now has separate address and data phases, we need two separate threads in the master. We also need a new protocol definition, or more accurately, we need two new protocols, one for each phase.

```
enum address_phase_status {
  ADDRESS_OK ,
  ADDRESS_ERROR
};

template < typename ADDRESS >
struct address_phase_request {
  pipelined_protocol_type type;
  ADDRESS a;
};

template < typename DATA >
struct data_phase_request {
  pipelined_protocol_type type;
  DATA wr_data;
};

template < typename DATA >
struct data_phase_response {
  pipelined_protocol_type type;
  DATA rd_data;
  data_phase_status status;
};
```

Since the template parameters for the two phases are completely different, we can implement both the address phase and the data phase transport functions at the top level in the slave.

To make sure that we issue and process requests in the correct (ie, pipelined) order, we have fifos in both master and slave. In the master, whenever we issue an address phase request, we put the corresponding data phase request into a fifo to be processed

later when the pipeline is full. The slave stores the requests as they come in, so that when it responds to a data phase it knows what request it is responding to.

The critical lines of code, which control the correct operation of the pipeline, are in the slave data phase :

```
while( pipeline->nb_can_put() ) {
  wait( pipeline->ok_to_get() );
}
pipeline->nb_get( pending , PEEK );
if( pending.type != req.type ) {
  rsp.status = DATA_ERROR;
  return rsp;
}
pipeline->nb_notify_got( POP );

rsp.status = DATA_OK;
rsp.type = req.type;
switch( req.type ) {
case READ :
  rsp.rd_data = memory[pending.a];
  break;
…
```

This code ensures that a data phase request is not responded to until the pipeline is full. It also checks that the address request just leaving the pipeline is of the same type as the data request. If this check fails, we do not process the request. If the check is ok, we go on to do the appropriate read or write.

This is a particular, abstract model of an in-order pipeline. To understand the example properly, it may be necessary to look at the code in www.cadence.com/systemc/whitepaper/examples. Of course, there are many other kinds of pipelines, most of which will modeled at the rtl level or close to it. However, like this example, they will all need two threads either in the master or slave to connect to a TLM model, and will all need to store the unfinished transactions in some kind of buffer on the slave side as they proceed down the pipeline.

5. Architectural Exploration

The patterns in sections 3 and 4 have been presented in their most simple form, in order to clarify the main issues associated with each pattern. In real TLMs, various patterns and levels of abstraction will combined. In this section, we show how these basic components can be combined and recombined to explore different architectures for a switch. Because the basic components are very generic and use the proposed TLM interfaces, switching from one architecture to another is very easy and requires little if any disruption to the SoC model as a whole.

5.1 Hub and Spoke

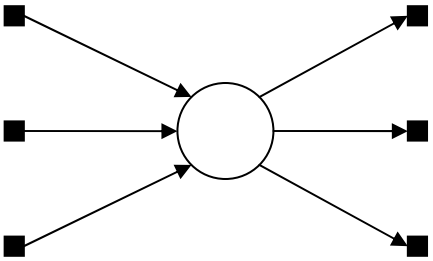


Figure 10 : Hub and Spoke Architecture

The first switch architecture we shall consider is a hub and spoke arrangement. In this architecture, all transactions pass through a central hub before being routed to their final destination. As a result, we have to arbitrate between the various requests for control of this central hub. While this is not the most efficient architecture in terms of throughput, it is efficient in terms of silicon area and therefore cost, since we only need one arbiter. Because all the transactions go through a central hub, its behavior is also more predictable than other switch architectures.

We use the arbiter in 4.2 followed by the router in 4.3 to implement this architecture.

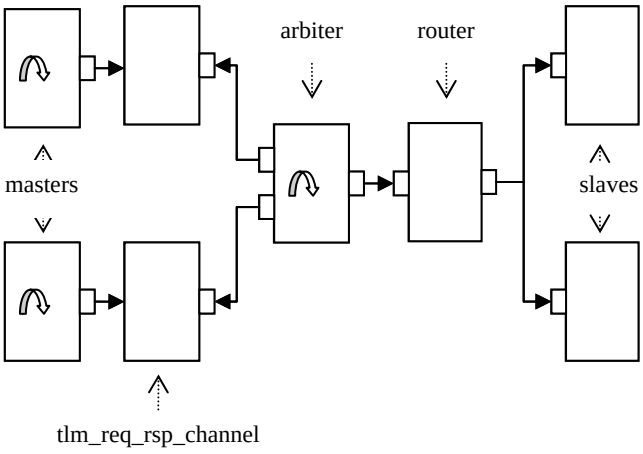


Figure 11 : 2 * 2 Hub and Spoke Implementation

5.2 Cross Bar Switch

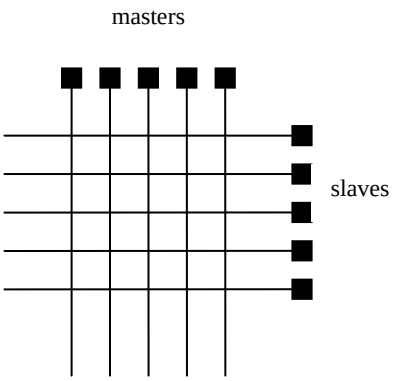


Figure 12 : Cross Bar Architecture

The advantage of a cross bar architecture is that we are able to make more than one connection across the switch at the same time. If a slave is available, a master may connect to it whatever else is going on in the system at the same time. A disadvantage is that there is no central arbitration, so every slave has to arbitrate between all the masters. This makes this architecture more expensive and also less predictable. However, the overall throughput is much greater than for the hub and spoke.

To move from the hub and spoke to the cross bar architecture, we need to make no changes at all to the masters and slaves. In terms of the modeling architecture in 2.1, we simply rearrange the components in the transport layer.

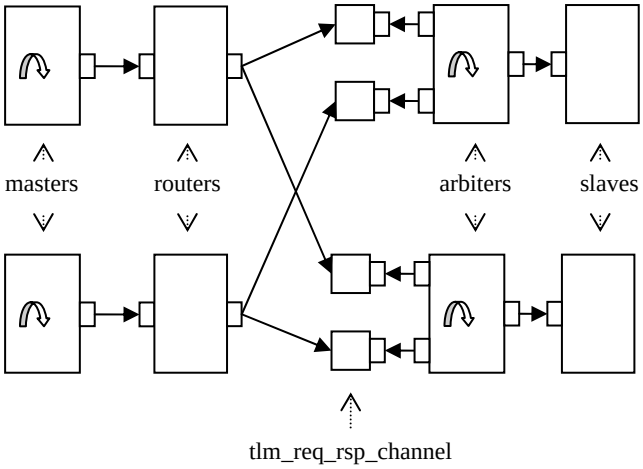


Figure 13 : 2 * 2 Cross Bar Implementation

5.3 Summary

The intention of sections 3,4 and 5 is to show how to use the relatively simple transport mechanism provided by the tlm proposal to coordinate between different teams of engineers, how to combine different levels of abstraction in the same TLM, and how to approach common modeling problems. It is not intended to be prescriptive. Rather, it summarizes many of the discussions that have taken place in and around the OSCI TLM working group. We hope that many more discussions along these lines will take place in the future.

6. References

[1] Clouard A, Jain K, Ghenassia F, Laurent Maillet-Contoz L, Strassen J-P "SystemC, Methodologies and Applications" Chapter 2. Edited by Muller, Rosenstiel and Ruf, published by Kluwer, ISBN 1402074794.

[2] Pierce, J.L, Erickson A, Swan S, and Rose A.D, "Standard SystemC Interfaces for Hardware Functional Modeling and Verification", Internal Cadence Document, 2004

[3] Burton, M and Donlin, A, "Transaction Level Modeling : Above RTL design and methodology", Feb 2004, internal OSCI TLM WG document.

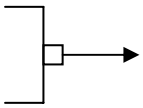
[4] "Functional Specification for SystemC 2.0", Version 2.0-Q, April 5th 2002 available at www.systemc.org

[5] multimap<Key, Data, Compare, Alloc> at http://www.sgi.com/tech/stl/Multimap.html

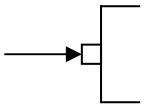
[6] T. Groetker, S. Liao, G. Martin, S. Swan, "System Design with SystemC", book available at www.amazon.com

7. Notes on the Graphical Representation of sc_port, sc_export and channels

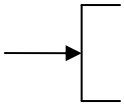
Throughout the examples in Sections 3, 4, and 5 we adopt the following graphical conventions.



A small square with an arrow leaving it is an **sc_port**



A small square with an arrow arriving at it is an **sc_export**



An arrow arriving at a module with no small square indicates a **channel**

This symbol represents a **thread**

Appendix A : sc_export

SystemC 1.0 provided sc_signal to connect an sc_in to an sc_out. This was primarily used to model at or close to the register transfer level.

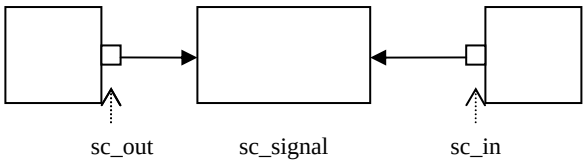


Figure 14 : rtl level binding in SystemC 1.0 and 2.0

For higher levels of abstraction, SystemC 2.0 generalised this pattern by introducing sc_port<IF> and channels. A channel implements one or more interfaces, and sc_ports are bound to that interface.

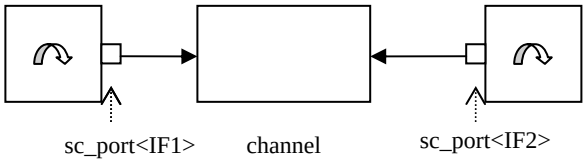


Figure 15 : Binding to a Channel in SystemC 2.0

The advantage of using channels is that there is a very clear boundary between behaviour and communication. The disadvantage is that we are forced to use two threads, one in each of the modules on either side of the channel.

In SystemC 2.1, sc_export was introduced. This allows direct port to export binding, as shown below.

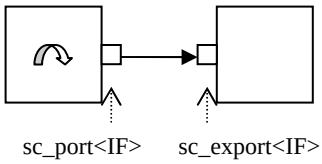


Figure 16 : Binding to an sc_export in SystemC 2.1

An sc_port assumes that it is bound to an interface. In Figure 16, this interface is supplied by sc_export. The port is bound to sc_export, and in turn sc_export is bound to an implementation of the interface somewhere inside the target block. In software engineering terms, we would describe sc_export<IF> as a proxy for the interface. The main reason for the introduction of sc_export is execution speed. We can now call the interface method in the target directly from within the initiator, so there is no need for a separate thread in the target and no reduction in performance associated with switching between threads.

An important use of sc_export is to allow sc_ports to connect to more than one implementation of the same interface in the same top level block.

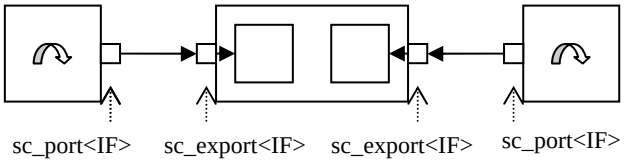


Figure 17 : exporting two copies of the same interface

Finally, an sc_export can be bound to another sc_export, provided the template parameter is the same. This mechanism is used to give access to an interface defined lower down in the sc_object hierarchy.

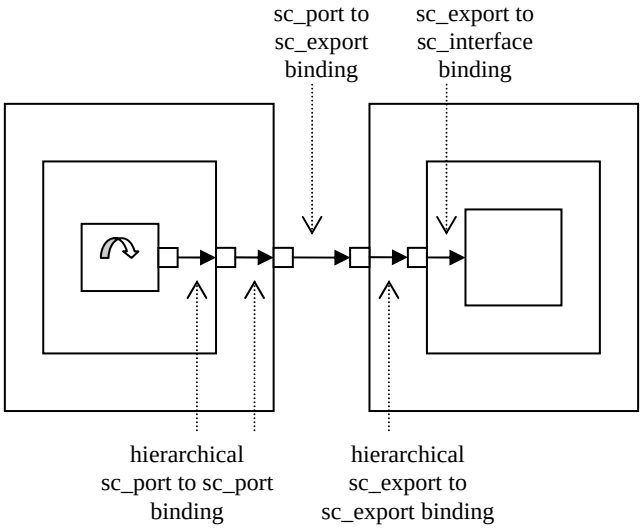


Figure 18 : sc_port, sc_export and hierarchy

The diagram above shows how a thread in a low level sub module inside an initiator directly calls a method in low level sub module in a target, using a chain of sc_ports to traverse up the initiator hierarchy, and a chain of sc_exports to traverse down the target hierarchy.

Appendix B : Safety in a Concurrent SystemC Environment

There have been many discussions relating to safety in the TLM WG.[ii] By safety we mean protection from premature deletion or editing of transaction data by one process while that transaction is being used elsewhere in the TLM. We also mean safety from unintended memory leaks. This appendix offers guidelines for the safe use of the TLM interfaces presented in this paper.

The TLM interfaces follow the style of the sc_fifo interfaces, which in turn are similar to many other C++ interfaces. Data going in to a method is always passed by const reference. Data coming back is passed by value if we can guarantee that there will always be data there eg the blocking get and transport calls. However, if we cannot guarantee that data will come back, we return the status by value and pass in a non const reference into the method, which will have data assigned to it if data is available. We do not pass by pointer, and we do not use a non const reference to pass data into a method. Since this style is widely used, in sc_fifo, throughout SystemC, and elsewhere in the C++ world, it is easily understood and used.

However, SystemC is a co-operative multi-threaded environment, so some care does need to be taken when using these interfaces over and above the usual precautions when programming in a single threaded environment. When we say co-operative, we mean that a thread is only suspended when the thread itself calls wait. Hence if we are safe when we call a method, and we can guarantee that we do not call wait inside that method, then the transaction data in that method is safe. For this reason, we know that all non blocking interface methods are safe, whatever their signatures.

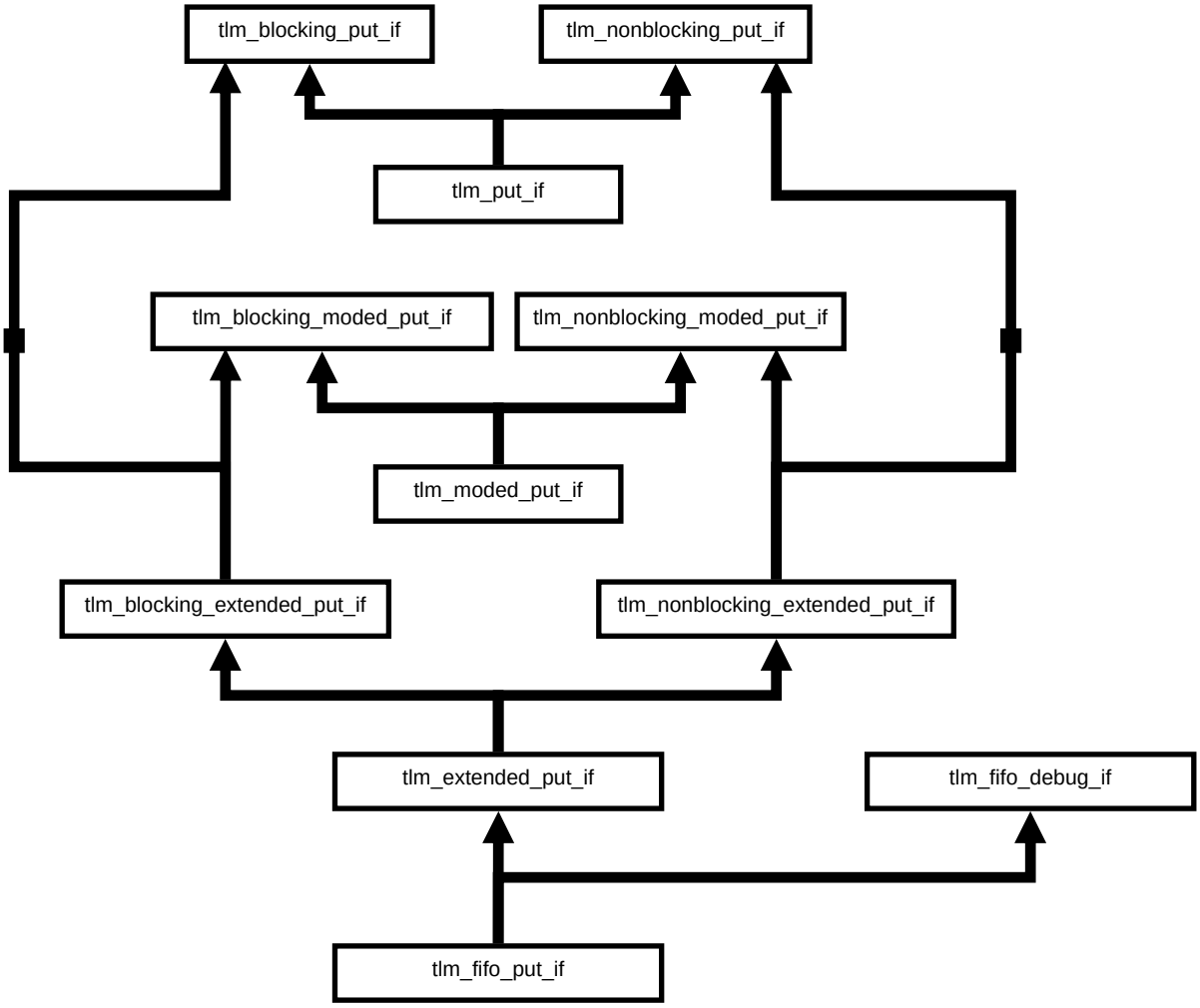
In all the examples discussed in this paper, the transaction data is allocated and owned by the thread which calls the tlm interface function. Whether or not there is a wait in the target, if the master owns the data in this way, the transaction data is safe from premature deletion and unintended editing.

It is a REQUIREMENT of the TLM standard that objects passed into blocking (or even potentially blocking) interface funtions are owned in the manner described above. With this requirement, implementations of blocking TLM API functions can safely assume that data passed into them by reference will not be prematurely deleted, even if these implementations call wait().

In some cases where large objects are being passed, the effective pass by value semantics of the TLM API may become a significant overhead. In such cases the user may wish to leverage C++ smart pointers and containers to gain efficiency. For example, large objects can be safely and efficiently passed using the boost shared_ptr template using the form shared_ptr<const T>, where T is the underlying type to be passed.

[ii] Thanks to Maurizio Vitale from Philips for stimulating the discussion around this issue.

Appendix C : Unidirectional TLM Interfaces



Core TLM Interfaces classes are

- tlm_transport_if<REQ,RSP>
- tlm_blocking_put_if, tlm_nonblocking_put_if, tlm_put_if
- tlm_blocking_get_if, tlm_nonblocking_get_if, tlm_get_if

Extended TLM Interface classes are

- tlm_blocking_moded_put_if, tlm_nonblocking_moded_if, tlm_moded_put_if
- tlm_blocking_extended_put_if, tlm_nonblocking_extended_if, tlm_extended_put_if
- and their get equivalents

Also Provided are

- tlm_master_if - combines tlm_put_if<REQ> and tlm_get_if<RSP>
- tlm_slave_if – combines tlm_get_if<REQ> and tlm_put_if<REQ>

Fifo Specific Interface classes are

- tlm_fifo_put_if, tlm_fifo_get_if, tlm_fifo_debug_if

inherits from

NB get interface hierarchy is not shown but follows the same pattern