Exploiting the TLM-2 Features of VMM 1.2

John Aynsley

Doulos
Ringwood, UK

www.doulos.com

ABSTRACT

VMM 1.2 introduces a set of features for transaction-level communication inspired by the SystemC TLM-2.0 standard. At the same time, the VCS TLI has been upgraded to support the TLM-2.0 standard. This paper, written by the author of the TLM-2.0 standard itself, explains the significance of these new features to VMM and gives ideas on how best to exploit these features for communication between a SystemC reference model and a SystemVerilog test bench.

Table of Contents

1 Introduction

The VMM methodology provides a powerful, flexible and intuitive framework for the construction of SystemVerilog verification environments. However, many SystemVerilog users also have models written in C, C++, or sometimes SystemC. Furthermore, the emergence of the SystemC TLM-1 and TLM-2.0 transaction-level modeling standards is having an impact on communication styles within SystemVerilog verification environments.

VMM 1.2 introduces a set of features for transaction-level communication within a VMM verification environment inspired by the SystemC TLM-2.0 standard. The inclusion of these features was motivated in part by the need to connect a VMM test bench to a SystemC reference model. At the same time, the existing VCS TLI (Transaction-Level Interface), a means of bridging between SystemC and SystemVerilog at the transaction level, has been upgraded and refined to work with the TLM-2.0 standard.

This paper explains the new TLM-2 features of VMM 1.2 and how they integrate with the existing communication features of VMM. We also offer practical guidance on using the VCS TLI with the TLM-2.0 standard to integrate C, C++ and SystemC code into an existing VMM-based SystemVerilog test bench.

This paper will be of interest to verification engineers who have already adopted, or are planning to adopt, SystemVerilog as their primary verification language, and who also expect to make use of C/C++/SystemC code in their verification activity. The content assumes some familiarity with SystemVerilog for verification, and some C/C++ programming skills, but does not require any prior experience in using those languages together.

2 Overview of the TLM-2 features in VMM 1.2

2.1 The motivation for adding TLM-2 features to VMM

VMM has always used transaction-level communication for abstraction, speed and productivity.

Mixed-language simulation environments involving a VMM test bench and C/C++ or SystemC reference models are not unusual. Virtual platform models, as used for software development and architectural exploration, are growing in importance, and the SystemC TLM-2.0 standard is being used to achieve interoperability between the components of such a virtual platform model. If a constrained random VMM environment is to be used with a reference model that consists of a virtual platform adhering to the SystemC TLM-2.0 standard, then having TLM-2.0 support within VMM promises to make life easier for the VMM programmer. The adoption of a common TLM standard across both SystemVerilog test benches and SystemC reference models makes good sense for everyone.

Besides interoperability, the other main objective of the SystemC TLM-2.0 standard is simulation speed. The combination of speed and interoperability depends on the technical details of the way in which transactions are passed between components. Fortunately, those technical details are a good fit with the way communication has always worked in VMM. In particular, both VMM and TLM-2.0 support the idea that each transaction has a finite lifetime with a well-defined time-of-birth and time-of-death.

2.2 Styles of transaction-level communication

Transaction-level communication in current languages (i.e. SystemVerilog and SystemC) and methodologies (i.e. VMM, TLM-1, and TLM-2.0) differs along several dimensions.

All of the above-mentioned methodologies represent transactions as objects passed as arguments to object-oriented method calls, such as *put(trans)* and *get(trans)* in the case of VMM and TLM-1, or *b_transport(trans, delay)* in the case of TLM-2.0. Such method calls can either be *blocking,* meaning that the function may suspend execution and only returns when the transaction is complete (in some sense), or *non-blocking*, meaning that the function always returns immediately, and indicates back to the caller whether or not the transaction is complete. All of the above-mentioned methodologies support both blocking and non-blocking communication.

Transaction objects can be passed by value or by reference. With pass-by-value semantics, regardless of whether argument passing is actually implemented by taking a copy of an argument to a method call, the transaction object is notionally read-only and communication is unidirectional; any response must be returned using a separate transaction object. TLM-1 uses pass-by-value in this sense. With pass-by-reference, the transaction object has an existence independent of the method call, and hence the same transaction object can be passed through a series of method calls. VMM and TLM-2.0 use pass-by-reference.

Communication between a producer and a consumer can be direct, or can be mediated by a channel. With direct communication (aka the remote procedure call), an initiator calls a method that is implemented by a target. Direct communication is used by TLM-1 and TLM-2.0. When communication is mediated by a channel, a producer and a consumer make method calls to a common channel that serves as a transaction buffer and allows the choice of blocking versus non-blocking to be made independently at producer and consumer. TLM-1 and VMM support communication mediated by a channel.

Another difference lies in the completion model for transactions. The completion of a transaction can be signalled using an argument or return value of a method call, using an attribute of a transaction object, or can be implicit. Closely related to the completion model is the approach used to signal significant events during the lifetime of the transaction. VMM takes the approach of using event notifications embedded within the transaction object itself. TLM-2.0 takes the approach of using separate non-blocking method calls to mark each timing point during the lifetime of a single transaction object.

VMM provides a diverse set of communication mechanisms including channels, notifications, and callbacks. The primary mechanism for transaction-level communication within a VMM verification environment is the vmm_channel, which can be used to implement a true zero-length queue with a blocking completion model, a finite FIFO, or an infinite FIFO with a non-blocking completion model. VMM 1.2 adds TLM-2.0-style direct communication using ports and exports.

The vmm_channel carries transactions by reference. The lifetime of a given transaction can extend over several method calls to the channel. The concept of the "active slot" permits a transaction to be inspected and modified by a consumer without being removed from the channel; the consumer is permitted to modify the state of the transaction object in order to return a response back to the initiator. The notifications built into each transaction (EXECUTE / STARTED / ENDED) permit timing points to be signalled (to the producer or consumer) during the lifetime of the transaction. Additional timing points can be signalled by adding a vmm_notify in parallel with the vmm_channel, which is the role played by transaction phases in TLM-2.0.

Even prior to VMM 1.2, the model of a transaction lifetime in VMM had aspects that reflect the TLM-2.0 standard:

- The lifetime of a VMM transaction extends across multiple method calls, like TLM-2.0
- Method calls can be blocking or non-blocking, like TLM-2.0
- A VMM transaction can be modified while still in a vmm_channel, like TLM-2.0
- VMM can explicitly signal the end of a transaction using vmm_data::ENDED, rather like the TLM-2.0 END_RESP phase, but it is only the subsequent call to remove() or get() that actually makes space in the channel for the next transaction.

2.3 Overview of the new VMM 1.2 TLM base classes

A full description of the VMM 1.2 TLM base classes can be found in the VMM Standard Library User Guide [2]. The intent of this section is not to repeat the information found in the User Guide, but rather to give an overview of the more significant features and their intended use.

2.3.1 The transport interfaces

TLM-2 communication in VMM 1.2 uses the blocking and non-blocking transport interfaces, which are used to communicate between an initiator and a target. The initiator would typically be a VMM transactor that produces transactions, and the target a transactor that consumes transactions. The blocking transport interface consists of a single method b_transport called in the forward direction from initiator to target, where the entire transaction is completed in a single method call.

```
port.b_transport( tx, delay );  // Called from initiator on forward path
```

The value of the delay argument is added to the current simulation time to determine the time at which the transaction should be processed at the target. The transaction itself is passed by reference (the argument is a SystemVerilog object handle), and remains valid only until the return from the b_transport method call, after which the initiator is free to re-use the transaction object for some other purpose. The "b_" in the method name stands for "blocking", meaning that the body of the method may suspend by executing a SystemVerilog event control and only return to the caller at a later simulation time.

The non-blocking transport interface consists of a pair of methods nb_transport_fw and nb_transport_bw, called in the forward and backward directions respectively, where the progress of a single transaction may be described using multiple calls to these two methods. In the description below, the term *nb_transport* is used to indicate either nb_transport_fw or nb_transport_bw.

```
vmm_tlm::sync_e   status;
vmm_tlm::phase_e  phase;
status = port.nb_transport_fw( tx, phase, delay);     // Called from initiator on forward path

status = export.nb_transport_bw( tx, phase, delay);  // Called from target on backward path
```

The status and phase variables use the following type definitions:

```
class vmm_tlm;
   typedef  enum
      { TLM_REFUSED, TLM_ACCEPTED, TLM_UPDATED, TLM_COMPLETED }
         sync_e;
   typedef  enum
      { BEGIN_REQ, END_REQ, BEGIN_RESP, END_RESP }
         phase_e;
   ...
endclass: vmm_tlm
```

The advantage of the blocking transport interface is its simplicity; the transaction always completes in a single method call. With the non-blocking transport interface, significant timing points during the lifetime of a transaction (e.g. the start of the response phase) are indicated by calling nb_transport in either the forward or backward direction, the specific timing point being identified using the phase argument. Protocol-specific rules for reading or writing the attributes of a transaction object can be expressed relative to the phase. The four phases BEGIN_REQ, END_REQ, BEGIN_RESP, and END_RESP belong to the so-called *base protocol*, which is described in more detail below. The phase can be used for flow control, and for that reason may have a different value at each hop taken by a transaction (where the same transaction is passed through multiple transactors); the phase is not an attribute of the transaction object.

A call to nb_transport always represents a phase transition, whereas the return from nb_transport may or may not do so, as indicated by the status value; TLM_UPDATED means the return represents a phase transition, TLM_ACCEPTED means it does not (so that transaction and phase arguments should be ignored), and TLM_COMPLETED is a shortcut meaning that the transaction is complete (that is, has jumped to the final phase). Alternatively, the completion of the transaction can be indicated by making an explicit transition to the final phase.

By design, the transaction object itself does not contain any timing information or events. Delays are passed as arguments to b_transport/nb_transport. On the other hand, a transaction object would typically include a response status field that carries protocol-specific information about the success or failure of the transaction.

The point of using the transport interfaces in VMM is to provide a simple, uniform way of passing transaction objects between transactors with well-defined semantics for transaction completion and for describing timing points. Also, having the transport interfaces common between VMM and SystemC provides a natural starting point for cross-language working.

2.3.2 Ports and exports

When two or more VMM transactors communicate using the transport interfaces, they do so using the VMM TLM ports and exports. The purpose of ports and exports is to provide a structured way of making method calls between VMM transactors (or SystemC modules) such that the dependencies between each transactor and its environment can be minimized. To call a transport method, the code within the producer only need refer to the port and has no direct dependencies on any code outside that transactor. Similarly, to call a transport method implemented within a consumer, the environment only need refer to the export, and has no other direct dependencies. It is only when the port and export are connected within the connect_ph method of the environment that a specific dependency is established between the producer and consumer transactors.

VMM provides ports and exports dedicated to the blocking and non-blocking transport interfaces. As per the TLM-2.0 standard, VMM also provides so-called *sockets* that combine both transport interfaces in a single object (described later).

```
class  producer  extends  vmm_xactor;
   vmm_tlm_b_transport_port #(producer, my_tx)  m_port;
   my_tx  tx;
   ...
   m_port.b_transport(tx, delay);
   ...

class  consumer  extends  vmm_xactor;
   vmm_tlm_b_transport_export #(consumer, my_tx)  m_export;

   task  b_transport(int id = -1, my_tx trans, ref int delay);
      ...

class  my_env  extends  vmm_group;
   producer  m_producer;
   consumer  m_consumer;

   virtual  function  void  connect_ph;
      m_producer.m_port.tlm_bind( m_consumer.m_export );
   endfunction
   ...
```

In the above code, the producer is calling b_transport through a port, the consumer is providing an implementation of b_transport using an export, and the top-level environment is connecting (or "binding") the port to the export. The tlm_bind method is creating the link between the port and the export such that when the producer calls b_transport, it is the implementation of b_transport within the consumer that actually gets called.

Each port and export is an object that must be constructed explicitly (the calls to *new* are not actually shown in the example above). Both the port and the export declarations are parameterized with the type of the transactor (producer/consumer) and the type of the transaction (my_tx). You may notice that the implementation of b_transport has an extra *int id* argument. This so-called *peer id* can be used to distinguish between transactions arriving from different producers.

In addition to the tlm_bind method shown above, VMM also provides a tlm_unbind method to remove an existing connection and a tlm_import method for binding a port or export on a child transactor to a port or export on a parent transactor, respectively. Note that this child-to-parent binding can only be performed by calling tlm_import, and not by calling tlm_bind, which can only be used for peer-to-peer binding.

VMM also provides a family of methods for determining the existing bindings of a port or export, namely get_n_peers, get_peers, get_peer, get_peer_id, check_bindings, print_bindings and report_unbound. The details can be found in the User Guide [2]

2.3.3 Peer ids

Peer ids permit a VMM consumer to distinguish between incoming transactions arriving from multiple producers, as illustrated by the following example:

```
class  producer  extends  vmm_xactor;
   vmm_tlm_b_transport_port #(producer, my_tx) m _port;
   ...
   m_port.b_transport(tx, delay);
   ...

class  consumer  extends  vmm_xactor;
   vmm_tlm_b_transport_export #(consumer, my_tx) m_export;

   function  new (string inst, vmm_object parent = null);
      super.new(...);
      m_export = new(this, "m_export", 2);     // 3rd argument = max # bindings
   endfunction

   task  b_transport(int id = -1, my_tx trans, ref int delay);
      ...

class  my_env  extends  vmm_group;
   producer  m_producer_1;
```

```
    producer  m_producer_2;
    consumer  m_consumer;
    virtual  function  void  connect_ph;
       m_producer_1.m_port.tlm_bind( m_consumer.m_export, 0 );  // 2nd argument = id
       m_producer_2.m_port.tlm_bind( m_consumer.m_export, 1 );  // 2nd argument = id
    endfunction
    ...
```

The first thing to notice is the connect_ph method of the environment, which binds two separate ports to the same export. The tlm_bind method takes a second argument, the peer id, which allows transactions from the two ports to be distinguished.

The second thing to notice is that when the export is instantiated, the constructor *new* takes a third argument that specifies the maximum number of bindings to this export. The default value of 1 would be inadequate in this case, since the export is bound twice.

Finally, the first argument to the b_transport method implemented in the consumer is the peer id passed to the tlm_bind method. The implementation of b_transport can now use the peer id to distinguish between transactions from the two producers.

As an alternative to peer ids, for every relevant type of port, export and socket VMM provides a shorthand macro that allows multiple ports/exports/sockets with the same transaction type to coexist within the same transactor, as illustrated by the following example:

```
  class  consumer  extends  vmm_xactor;
    `vmm_tlm_b_transport_export(_1)  // Argument is suffix to name
    `vmm_tlm_b_transport_export(_2)
    vmm_tlm_b_transport_export_1 #(consumer, my_tx)  m_export_1;
    vmm_tlm_b_transport_export_2 #(consumer, my_tx)  m_export_2;

    task  b_transport_1(int id = -1, my_tx trans, ref int delay);
       ...
    task  b_transport_2(int id = -1, my_tx trans, ref int delay);
       ...
```

The argument passed to the macro is used as the suffix for a new type name and a new method name. Those new types are then used to create two separate exports, and the consumer contains two separate and differently named implementations of the b_transport method, one for each export. It is good practice to use the same suffix when naming the export members themselves (e.g. m_export_1), though this is not strictly necessary. Since peer ids are not being used, the id argument to b_transport will have the value 0 for both methods.

2.3.4 Analysis ports and exports

Analysis ports and exports are a variant on the TLM ports and exports discussed above. The main difference between analysis ports and regular ports is that a single analysis port can be bound to multiple analysis exports, in which case the same transaction is "broadcast" to each and every export or "observer" connected to the analysis port.

Analysis ports provide a mechanism for distributing transactions to passive components in a verification environment, such as checkers and scoreboards, and as such they provide an alternative to VMM callbacks for the situation where the transaction does not need to be modified by the observer. The User Guide [1] actually recommends the use of analysis ports rather than callbacks or vmm_notify when distributing transactions that do not need to be modified. Callbacks should still be used where the transaction does need to modified, and vmm_notify should still be used for broadcasting data-less synchronization events.

The following example shows an analysis port:

```
class  my_tx  extends  vmm_data;   // User-defined transaction class
   ...

class  transactor  extends  vmm_xactor;
   vmm_tlm_analysis_port #(transactor, my_tx)  m_ap;    // The analysis port
   ...
   virtual task main;
      my_tx  tx;
      ...
      m_ap.write(tx);     // Broadcast transaction to all observers
   ...
```

The transactor above sends a transaction tx out through an analysis port m_ap. The type of the analysis port is parameterized with the type of the transactor and of the transaction my_tx. The call to *write* sends the transaction to any object that has registered itself with the analysis port. There could be zero, one, or many such observers registered with the analysis port.

To continue the example, let us look at one observer:

```
class  observer  extends  vmm_object;
   vmm_tlm_analysis_export  #(observer,  my_tx)  m_export;

   function  new  (string inst,  vmm_object  parent = null);
      ...
      m_export = new(this, "m_export");  // Every port and export needs to be constructed
      ...
   function  void  write( int  id,  my_tx  tx );
      ...
```

The observer has an instance of an analysis export and must implement the *write* method that the export will provide to the transactors. Note that the observer extends vmm_object. Since an observer is passive, it need not extend vmm_xactor.

The analysis port may be bound to any number of observers in the surrounding environment, as shown in the following example:

```
class tb_env extends vmm_group;
  transactor   m_transactor;
  observer     m_observer_1;
  another      m_observer_2;
  yet_another  m_observer_3;

  virtual function void build_ph;
    m_transactor = new( "m_transactor", this );
    m_observer   = new( "m_observer",   this );
  endfunction

  virtual function void connect_ph;
    m_transactor.m_ap.tlm_bind( m_observer_1.m_export );
    m_transactor.m_ap.tlm_bind( m_observer_2.m_export );
    m_transactor.m_ap.tlm_bind( m_observer_3.m_export );
  ...
```

2.3.5 Sockets

As mentioned above, a socket combines multiple ports and exports into a single object so that calls to b_transport, nb_transport_fw and nb_transport_bw can be made through a single pair of sockets, one initiator socket and one target socket, as follows:

```
class producer extends vmm_xactor;
  vmm_tlm_initiator_socket #(producer, my_tx) m_socket;
  ...
  m_socket.b_transport(tx, delay);   // Call through socket
  ...
  status = m_socket.nb_transport_fw(tx, phase, delay);  // Call through socket
  ...
  virtual function vmm_tlm::sync_e nb_transport_bw(
    int id=-1, my_tx trans, ref vmm_tlm::phase_e ph, ref int delay);
    ...

class consumer extends vmm_xactor;
  vmm_tlm_target_socket #(consumer, my_tx) m_socket;
  ...
  status = m_socket.nb_transport_bw(tx, phase, delay);  // Call through socket
  ...
  task b_transport(int id = -1, my_tx trans, ref int delay);
    ...
  virtual function vmm_tlm::sync_e nb_transport_fw(
    int id=-1, my_tx trans, ref vmm_tlm::phase_e ph, ref int delay);
    ...
```

In the above code, note that the producer, which instantiates the initiator socket, must implement nb_transport_bw, and the consumer, which instantiates the target socket, must implement both b_transport and nb_transport_fw. Also note that the code above just shows the raw method calls; actual working code must make transport calls according to the rules of some protocol, and an initiator should not mix calls to b_transport and nb_transport_fw in a fine-grained manner.

The initiator socket must be bound to the target socket with a single call to tlm_bind at the top level, as shown below:

```
class my_env extends vmm_group;
  producer  m_producer;
  consumer m_consumer;
  virtual function void connect_ph;
    m_producer.m_socket.tlm_bind( m_consumer.m_socket );
  endfunction
  ...
```

2.3.6 Generic payload

The TLM-2.0 standard defines a generic payload and a base protocol to enhance interoperability for models with a memory-mapped bus interface. Although it is possible to use the transport interfaces described above with user-defined transaction types and protocols, for the sake of interoperability TLM-2.0 strongly recommends either using the base protocol off-the-shelf or creating models of specific protocols using the generic payload and base protocol as a starting point, then adding user-defined extensions as needed. The generic payload provides an extension mechanism for this purpose. All of these TLM-2.0 features are available in VMM 1.2.

In the world of virtual platform modeling, TLM-2.0 interfaces fall into one of two categories. Models that need to communicate by reading or writing a block of bytes at a certain address without regard for the fine details of the protocol will use the plain, unextended generic payload, and thus achieve a high degree of off-the-shelf interoperability. On the other hand, models that need to be concerned with the fine details of a specific protocol, perhaps because they require a high degree of timing accuracy, will need to extend the generic payload and phases, and by so doing will sacrifice interoperability with the base protocol.

The generic payload contains a set of attributes that are typical of memory-mapped busses. The details can be found in the OSCI TLM-2.0 LRM [4]. In VMM 1.2 a generic payload transaction can be created and transported as follows:

```
vmm_tlm_b_transport_port #(initiator, vmm_tlm_generic_payload)  m_port;
...
begin
  vmm_tlm_generic_payload  tx;
  int delay;
```

```
    assert( randomized_tx.randomize() with {
      m_command == vmm_tlm_generic_payload::TLM_WRITE_COMMAND;
      m_address >= 0 && m_address < 256;
      m_length == 4 || m_length == 8;
      m_data.size == m_length;          // Trick to randomize dynamic array
      m_byte_enable_length <= m_length;
      (m_byte_enable_length % 4) == 0;
      m_byte_enable.size == m_byte_enable_length;
      m_streaming_width == m_length;
    } )
    else `vmm_error(log, "tx.randomize() failed");

    $cast(tx, randomized_tx.copy());
    m_port.b_transport(tx, delay);
    assert( tx.m_response_status == vmm_tlm_generic_payload::TLM_OK_RESPONSE );
  end
```

A generic payload transaction has 10 attributes, the most important being the command, address, data array, data length, and response status. Other attributes include byte enables, streaming width, and extensions. There are two particular points to note from the example above; firstly, all attributes are set (or constrained to sensible values) before sending the transaction through the transport interface, and secondly, the response status is checked on return from b_transport. As a practical point, the byte enable length should be set to 0 if byte enables are not used, and the streaming width should be set equal to the data length (m_length) if streaming mode is not used.

The target should inspect and execute the incoming generic payload transaction as follows:

```
  vmm_tlm_b_transport_export #(target, vmm_tlm_generic_payload)  m_export;
  ...
  task  b_transport(int id = -1,  vmm_tlm_generic_payload  trans,  ref int  delay);
    vmm_tlm_generic_payload::tlm_command   cmd = trans.m_command;
    longint            adr = trans.m_address;
    int unsigned    len = trans.m_length;
    int unsigned    bel = trans.m_byte_enable_length;
    int unsigned   wid = trans.m_streaming_width;

    // Check whether the attribute values are supported by this target
    if (adr + len >= SIZE) begin
      trans.m_response_status =
        vmm_tlm_generic_payload::TLM_ADDRESS_ERROR_RESPONSE;
      return;
    end
    if (wid < len) begin
      trans.m_response_status =
        vmm_tlm_generic_payload::TLM_BURST_ERROR_RESPONSE;
      return;
    end
    if (cmd == vmm_tlm_generic_payload::TLM_READ_COMMAND)
      for (int unsigned i = 0; i < len; i++) begin
        `define REM(a, b) ((a)-(((a)/(b))*(b)))
        if ( bel == 0 || trans.m_byte_enable[ `REM(i, bel) ] )
          trans.m_data[i] = mem[adr + i];
      end
    else if (cmd == vmm_tlm_generic_payload::TLM_WRITE_COMMAND)
      for (int unsigned i = 0; i < len; i++) begin
        `define REM(a, b) ((a)-(((a)/(b))*(b)))
        if ( bel == 0 || trans.m_byte_enable[ `REM(i, bel) ] )
          mem[adr + i] = trans.m_data[i];
      end
    #10;
    trans.m_response_status = vmm_tlm_generic_payload::TLM_OK_RESPONSE;
  endtask : b_transport
```

There are several important points to note from the example above. Firstly, the command, address, data length, byte enable length, and streaming width attributes must all be checked to ensure that the transaction is valid for this particular target; otherwise, the target must set an error response in the transaction before returning. Secondly, this particular target is able to execute either a read or a write command with support for byte enables, where the byte enable array length is permitted to be less than the data array length (in which case access to the byte enable array wraps around using REM, a macro to calculate remainder on division). Finally, if the target is able to execute the transaction successfully, it must set the response status to OK before returning.

2.4 The relationship between VMM TLM and the SystemC TLM-2.0 standard

Although the implementation of TLM in VMM 1.2 is inspired by the SystemC TLM-2.0 standard, there are significant differences between the C++ and SystemVerilog languages that prevent it being a literal translation. Also, differences between the typical use cases for VMM and SystemC mean that some differences between the TLM-2.0 implementations would be desirable anyway. For anyone who is interested in both SystemVerilog and SystemC, this section highlights places where the VMM 1.2 implementation differs from the SystemC TLM-2.0 standard.

VMM only implements the transport interfaces, not the TLM-2.0 direct memory or debug transport interfaces. Although the direct memory and debug transport interfaces are useful in the context of a virtual platform model written in C++, they do not make much sense when exercising such a model from a SystemVerilog test bench.

In VMM, the nb_transport method adds a new return value TLM_REFUSED, which allows the implementation of the method to reject the transaction. In TLM-2.0 there is no ability to refuse a transaction at the API level; such an ability must be modeled using attributes of the transaction object.

The lack of multiple inheritance in SystemVerilog makes certain things harder to express. In particular, the hierarchical SystemC interface structure cannot be reproduced exactly. Unlike SystemC, VMM makes use of separate port and export types for each kind of transport interface (blocking and non-blocking). Like SystemC, VMM combines all of the transport interfaces into a pair of initiator and target sockets to simplify binding. Unlike SystemC, the VMM sockets are parameterized on the transaction and phase types separately, rather than using a single protocol traits class.

Language differences between C++ and SystemVerilog force a different approach to tagged sockets, i.e. allowing multiple instances of the same socket type within the same component, and to multi-ports and multi-sockets, i.e. binding multiple initiators to a single target and vice-versa. VMM achieves very similar results to TLM-2.0 using the *peer id* and shorthand macros (both described above).

TLM-2.0 handles conversion between blocking and non-blocking transport calls automatically using the simple target socket, which provides both blocking and non-blocking transport methods, only one of which actually needs to be implemented within the target. VMM does not provide this kind of automatic adaption, but achieves a similar result using the vmm_connect utility, which must be called explicitly in order to bind ports and exports of different interface types. For example:

```
vmm_connect #(.D(vmm_tlm_generic_payload))::tlm_transport_interconnect(
   m_initiator.m_b_port,    // Blocking transport port on initiator
   m_target.m_nb_export,  // Non-blocking transport export on target
   vmm_tlm::TLM_NONBLOCKING_EXPORT);
```

Unlike TLM-2.0, the vmm_tlm_generic_payload class does not define any get/set access methods. Instead, the generic payload data members are public and are declared as *rand*. This makes sense in the context of constrained random transaction generation; the VMM generic payload includes a set of default constraints to generate transactions with reasonable attribute values. (In contrast, the SystemC TLM-2.0 standard was developed with deterministic execution in mind, i.e. running system software.) Also, the VMM generic payload extends vmm_data, as would be the case for any other VMM transaction type, and makes use of the VMM field automation macros to define methods for copying, comparing, and printing transactions.

Unlike C++, SystemVerilog does not support pointers. Objects are uniformly accessed by reference, and SystemVerilog provides automatic garbage collection. Because of these language differences, the VMM generic payload does not provide methods for explicit memory management (i.e. acquire, release, reset).

Unlike C++, where the data and byte enable array attributes are pointers, in SystemVerilog they are dynamic arrays. This makes the transaction data part of the transaction object itself, which is a difference in philosophy with TLM-2.0.

VMM supports generic payload extensions. However, because SystemVerilog has no support for function templates, extensions have to be accessed by passing the extension ID as an argument to the set/get/clear_extension methods.

3 How to use TLM-2-style communication in VMM 1.2

3.1 Communication options in VMM 1.2

VMM 1.2 provides an array of options for communication: the channel (vmm_channel), the notification service (vmm_notify), callbacks (`vmm_callback), and now the blocking and non-blocking transport ports and analysis ports inspired by the TLM-2.0 standard. The User Guide [2] describes each of these in detail and provides a set of guidelines for when to use each of these options, which are summarized here:

- **vmm_tlm_b_transport_port** is preferred for *master-like* transactors that generate transactions to be consumed by other transactors. The benefit of blocking transport is that the completion model is very well-defined and is unrelated to the specific details of the transaction object, making it robust and easy-to-understand. Simply put, b_transport does not return until the transaction is complete.

- **vmm_channel** is preferred for *slave-like* transactors that consume transactions generated elsewhere. The vmm_channel provides maximum flexibility when writing this kind of transactor, because it can be connected to any kind of producer (i.e. one that uses ports or channels, or that makes blocking or non-blocking calls). Moreover, the vmm_channel gives the ability for the consumer to process the transaction while it remains in the *active slot* of the channel, to buffer multiple transactions, and to idle while waiting for the next transaction to become available.

- **vmm_tlm_analysis_port** is preferred for generating transactions sent to passive objects such as scoreboards and coverage collectors that do not need to modify the transaction object. Analysis ports have the benefit that they can broadcast transactions to any number of observers, and guarantee that the observers will not interfere with each other by modifying the transaction.

- **vmm_callback** is preferred when modifying transactions for the purpose of overriding the behavior of a transactor for a specific test case. Callbacks offer two significant advantages when compared to analysis ports in this scenario: firstly, and essentially, they permit the transaction object to be modified, and secondly they give control over the order in which multiple callbacks are made.

- **vmm_notify** is preferred for synchronization between transactors in the case where there is no data passed along with the synchronization. Exceptionally, vmm_notify may be used in parallel with vmm_channel when multiple synchronization events are to be associated with a transaction.

The non-blocking transport interface is not preferred in native VMM environments, but is provided for compatibility with the TLM-2.0 standard and for communication with SystemC models.

3.2 Analysis ports versus callbacks

In order to understand the consequences of choosing an analysis port versus a callback, consider the following example:

```
m_ap.write(tx);
```

versus

```
`vmm_callback(callback_facade, write(tx));
```

The effect is very similar, but there are differences. Unlike VMM callbacks, the name of the method called through an analysis port is fixed at *write*. A VMM callback method is permitted to modify the transaction object, whereas a transaction sent through an analysis port cannot be modified. When multiple callbacks are registered, the prepend_callback and append_callback methods allow you to determine the order in which the callbacks are made, whereas you have no control over the order in which write is called for multiple observers bound to an analysis port. Because of these differences, only VMM callbacks are appropriate for modifying the behavior of transactors. Analysis ports are only appropriate for sending transactions to passive components that will not attempt to modify the transaction object. On the other hand, that in itself is the feature and strength of analysis ports; they are only for analysis.

It can make sense to combine a VMM callback with an analysis port in the same transactor, using the callback to inject an error and the analysis port to send the modified transaction to a scoreboard, for example:

```
`vmm_callback(callback_facade, inject_error(tx));
m_ap.write(tx);
```

In this situation, the VMM recommendation is to make the analysis call after the callback, as shown above.

3.3 Example using blocking transport, analysis port, callback, and vmm_channel

Here we show a more extensive example of using a blocking transport port in a producer together with a vmm_channel in a consumer, as recommended above. In this example, the generic payload is replaced by a user-defined transaction type:

```
class  my_tx  extends  vmm_data;
   rand  int  addr;                                  // All protocol properties are rand and public
   rand  int  data;
   constraint c_addr { addr >= 0; addr < 256; } // Default constraints with sensible values
```

```
    constraint c_data { data >= 0; data < 256; }
    `vmm_typename(my_tx)                        // Automation macros...
    `vmm_class_factory(my_tx)
    `vmm_data_member_begin(my_tx)
      `vmm_data_member_scalar(addr, DO_ALL)
      `vmm_data_member_scalar(data, DO_ALL)
    `vmm_data_member_end(my_tx)
  endclass: my_tx

  typedef  vmm_channel_typed #(my_tx)  my_channel;
```

The producer sends transactions using several different mechanisms simultaneously for different purposes: through a blocking transport port to some downstream transactor, through an analysis port for monitoring, and through callbacks to allow modifications specific to particular test cases:

```
  class my_gen extends vmm_xactor;
    vmm_tlm_b_transport_port  #(my_gen,  my_tx)  m_port;
    vmm_tlm_analysis_port       #(my_gen,  my_tx)  m_ap;

    virtual  function  void  build_ph;
      m_port = new(this, "m_port");   // Construct transport port object
      m_ap   = new(this, "m_ap");      // Construct analysis port object
    endfunction:  build_ph
    ...
    virtual  task  main;
      super.main();
      repeat(10)
      begin: loop
        my_tx   tx;
        int       delay;
        assert( randomized_tx.randomize() )
        else `vmm_error(log, "tx.randomize() failed");
        $cast(tx, randomized_tx.copy());

        // Insert callbacks at all significant points in the transactor
        `vmm_callback(my_gen_callbacks,  pre_trans(this, tx));

        m_port.b_transport(tx, delay);          // b_transport call for master-like transactor

        `vmm_callback(my_gen_callbacks,  post_trans(this, tx));
        m_ap.write(tx);                              // Send tx through analysis port after callback
      end
      ...
```

There is no obligation either to connect the analysis port or to register any callbacks, but including these calls in the transactor provides the user with the flexibility to make use of the features later if needed.

The consumer receives incoming transactions through a vmm_channel. The channel acts as a buffer between producer and consumer, with the producer putting transactions into the tail of the channel and the consumer getting transactions from the head of the channel. The channel deliberately isolates the consumer from the details of the producer, such as whether the producer makes blocking or non-blocking calls (it could do either or both), or whether the producer uses a VMM TLM port or a vmm_channel (again, it could do either):

```
class  my_bfm  extends  vmm_xactor;
   my_channel   m_chan;    // Reference to input channel instantiated in the environment
   vmm_tlm_analysis_port  #(my_bfm, my_tx)  m_ap;
   ...
   virtual  task  main;
      super.main();
      fork
         forever
         begin: loop
            my_tx  tx;
            `vmm_callback(my_bfm_callbacks,  pre_trans(this, tx));

            // Consumer should use activate/start/complete/remove
            m_chan.activate(tx);
            m_chan.start();
            @(i_f.bus_cb);
            i_f.bus_cb.addr1  <=  tx.addr;   // Wiggle pins using clocking block
            i_f.bus_cb.data1  <=  tx.data;
            m_chan.complete();
            m_chan.remove();

            `vmm_callback(my_bfm_callbacks,  post_trans(this, tx));
            m_ap.write(tx);                    // Send tx through analysis port after callback
         end
      join_none
      ...
```

Note that the consumer is making use of the *active slot* in the vmm_channel by allowing the transaction to remain in the channel while it is being processed, only removing the transaction from the channel when it is complete. The consumer could update fields in the transaction object if desired, such as the response status attribute of the generic payload. Also note that the consumer, like the producer, makes use of callbacks and an analysis port.

The top-level env instantiates the producer and consumer transactors and binds the transport port to the vmm_channel using the connect utility:

```
class  tb_env  extends  vmm_group;
   my_gen       m_gen;                          // Producer
   my_bfm       m_bfm;                          // Consumer
   my_channel  m_tx_chan;                       // Consumer input channel
   ...
   virtual  function  void  build_ph;
      m_tx_chan  = new( "my_channel", "m_tx_chan" );
      m_tx_chan.reconfigure(1);                 // Input channel given full-level = 1
      m_gen       = new( "m_gen", this );   // Construct transactor objects
      m_bfm       = new( "m_bfm", this );
   endfunction: build_ph

   function  void  connect_ph();
      // Bind blocking transport port to vmm_channel
      vmm_connect #(.D(my_tx))::tlm_bind( m_tx_chan,  m_gen.m_port,
                                               vmm_tlm::TLM_BLOCKING_EXPORT );

      // Recommended to connect channel in consumer from connect_ph of parent
      m_bfm.m_chan = m_tx_chan;

      // Register channel with vmm_consensus to stop test when channel is empty
      vote.register_channel(m_tx_chan);
   endfunction:connect_ph
endclass: tb_env
```

There are two key points to note in the above. Firstly, the channel is reconfigured to have a full level of 1. This ensures that the blocking transport call does indeed block. If the full level were greater than 1, the first call to b_transport will return immediately before the transaction had completed, which would contradict the semantics of the blocking transport interface. As things stand, the implementation of b_transport within the vmm_channel will not return until the transaction has been removed from the channel by the consumer. Setting the full level of the channel to 1 ensures that it behaves as a zero-length queue (aka rendezvous). Note that even when the consumer removes the transaction from the channel, the transaction object itself remains in existence, and indeed, b_transport must subsequently return a reference to the transaction object back to the producer.

Secondly, the vmm_connect utility is used within the connect_ph method to bind the blocking transport port to the vmm_channel. This connect utility must be used when binding VMM TLM objects to channels, and can also used in order to bind TLM ports and exports of differing interface types (e.g. a blocking port to a non-blocking export). The third argument to the tlm_bind method indicates that the connection is being made from the port in the producer to a blocking export within the channel.

4 Connecting a VMM test bench to a SystemC reference model

4.1 Dealing with timing differences between test bench and reference model

A SystemVerilog VMM test bench typically exercises an RTL model of a Design-Under-Test (DUT) by pin wiggling, that is, by making low level assignments to individual Verilog wires with more-or-less precise timing (the timing could be accurate to the picosecond or merely clock-cycle-accurate). The pin wiggling is usually encapsulated within driver and monitor components that communicate with the rest of the test bench using transactions. In VMM this is termed the command layer, in which transactions are simple and atomic. These atomic transactions are generated from higher layers of the test bench which combine these atomic transactions into more complex higher-level transactions according to the details of the interface or protocol being modeled.

Transactions may be sent to a so-called scoreboard which checks for functional correctness. It is within the scoreboard that a test bench may need to invoke a reference model to calculate the expected values of the DUT or to analyze the actual values generated by the DUT. The scoreboard is typically expected to receive both stimulus sent to the DUT and the actual response from the DUT transaction-by-transaction at some appropriate abstraction level.

It is increasingly the case that SystemC models conform to the OSCI TLM-2.0 standard. However, a C/C++ reference model may not be structured to receive transactions one-by-one. Rather, the programming interface to the reference model may consist of a single function call that carries with it an entire dataset, or the dataset may be read from an external file. Thus it may be necessary for the scoreboard to collect a set of incoming transaction and then pass them to the reference model for batch processing. Theoretically this buffering could occur on the SystemVerilog or the C/C++ side, depending on the ease with which transaction can be passed between the two languages. Because most facilities for passing data and control between languages have limitations (i.e. the SystemVerilog DPI and the VCS TLI), the most practical approach can be to pass simple, standard transactions across the language boundary and do any necessary buffering on the C/C++ side.

4.2 How to use the VCS TLI (Transaction-Level Interface)

4.2.1 Overview of the TLI

The VCS TLI (Transaction-Level Interface) is an off-the-shelf mechanism for transaction-level communication between SystemVerilog and SystemC. The motivation for the TLI is to provide standard transaction-level communication between a SystemVerilog test bench and a SystemC model. In principle, this can be achieved using the SystemVerilog Direct Programming Interface (DPI), as described in [5]. However, the DPI has no native support for SystemC interface method calls or for VMM channels, so using the DPI to call SystemC methods from VMM is non-trivial.

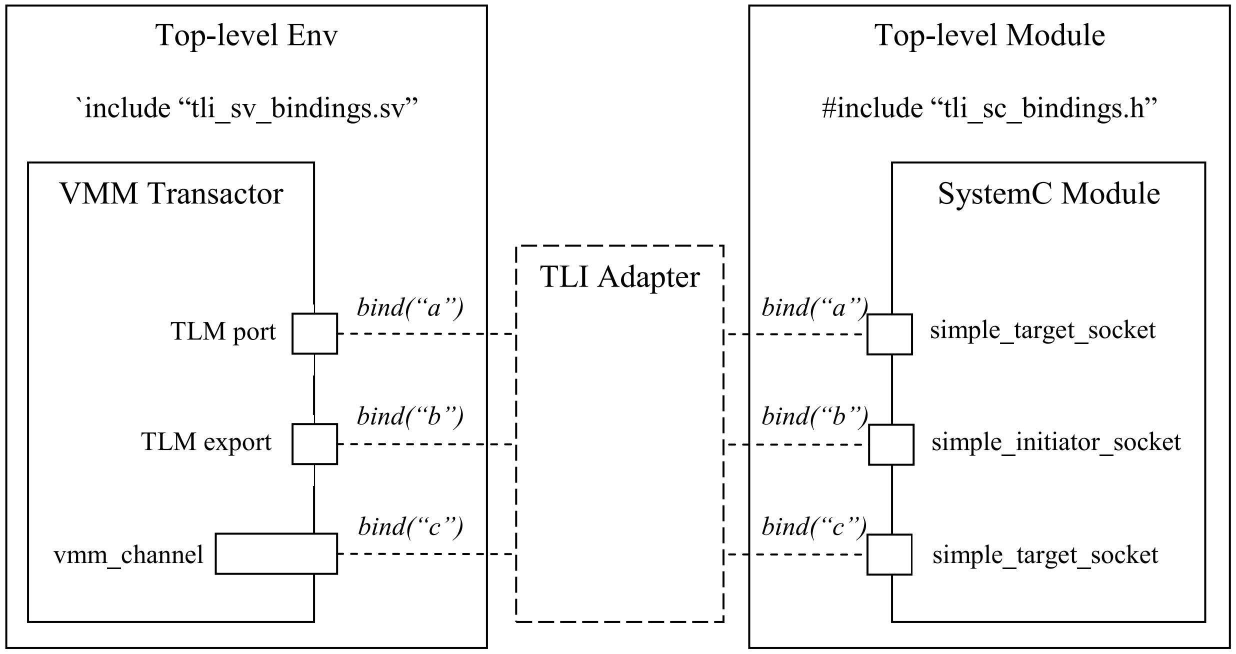
The latest release of the TLI from Synopsys has explicit support for VMM 1.2 and for the Sys-temC TLM-2.0 standard, making it easy to integrate the TLM features of VMM described in this paper with a SystemC TLM-2.0 model. On the SystemVerilog side, the TLI supports two styles of communication: either VMM TLM ports/exports (transport or analysis) or vmm_channels. On the SystemC side, the TLI supports the standard TLM-2.0 simple sockets and analysis ports/interfaces with both the LT (loosely-timed) and AT (approximately-timed) coding styles. The TLI supports method calls in both directions, that is, either side can be the producer or the consumer of transactions, giving many possible permutations:

- SystemVerilog blocking port, SystemC LT target
- SystemVerilog non-blocking port, SystemC AT target
- SystemVerilog vmm_channel, SystemC LT target
- SystemVerilog vmm_channel, SystemC AT target
- SystemVerilog analysis port, SystemC analysis subscriber

- SystemC LT initiator, SystemVerilog blocking export
- SystemC AT initiator, SystemVerilog non-blocking export
- SystemC LT initiator, SystemVerilog vmm_channel
- SystemC AT initiator, SystemVerilog vmm_channel
- SystemC analysis port, SystemVerilog analysis subscriber

At the time of writing, the TLI does not allow both blocking and non-blocking transport calls to be mixed through the same port or socket.

In order to use the TLI, it is only necessary to add a few new import/include directives and bind calls on both the SystemVerilog and SystemC sides.



The bindings between the VMM and the SystemC ports/exports/channels/sockets are made by calling the appropriate methods anywhere in the SystemVerilog and SystemC source code, respectively. Note that these methods are not literally named *bind* as shown on the diagram above; see the example below. As well as referring to the appropriate object instance, each bind call is passed a string that must uniquely identify the specific binding. The correspondence between the SystemVerilog and SystemC sides of the TLI adapter is established using these strings.

4.2.2 TLI example

As an example, we will show the case of making a b_transport call from SystemVerilog to SystemC. All the other cases follow a similar pattern.

The VMM transactor makes a b_transport call to send a transaction out through a TLM port:

```
class  my_xactor  extends  vmm_xactor;
   vmm_tlm_b_transport_port    #(my_xactor, vmm_tlm_generic_payload, tli_phase_e)
       m_b_port;
   ...
   m_b_port.b_transport(tx, delay);
```

The top-level VMM env binds the port to the TLI adapter:

```
`include "tli_sv_bindings.sv"
import  vmm_tlm_binds::*;

class  my_env  extends  vmm_group;
  my_xactor  m_xactor;
  function  void  connect_ph();
     tli_tlm_bind( m_xactor.m_b_port,
                         vmm_tlm::TLM_BLOCKING_EXPORT,     "sv_tlm_lt");
   ...
```

On the SystemC side, the SystemC module has a simple_target_socket:

```
struct  scmod:  sc_module
{
    tlm_utils::simple_target_socket<scmod>    targ_socket_lt;
   ...
    targ_socket_lt.register_b_transport (this, &scmod::b_transport);
```

Finally, the top-level SystemC module binds the target socket to the TLI adapter using the same identifier as the SystemVerilog side, which was “sv_tlm_lt”:

```
#include "tli_sc_bindings.h"

struct  sctop:  sc_module
```

```
{
  scmod  *m_scmod;
  SC_CTOR(sctop) {
    m_scmod = new scmod("m_scmod");
    tli_tlm_bind_target   (m_scmod->targ_socket_lt, LT, "sv_tlm_lt");
    ...
```

As shown above, the only changes to the user's code are the include directives and the bind calls. The user just has to ensure that every bind is given a globally unique string id to establish the correspondence across the language boundary. The TLI itself is implemented by a single global adapter that needs to be compiled into the user's environment using a very simple script. The TLI adapter is not explicitly instantiated in the user's code. To run the code, the user needs to compile the TLI adapter itself and add the appropriate include paths to the VCS command line. This is easily accomplished by copying the examples provided by Synopsys.

4.2.3 User-defined transactions and protocols

The above example shows a generic payload transaction being passed between a VMM test bench and a SystemC model. It is also possible to customize the TLI adapter to pass a user-defined transaction type, though this may require considerably more effort. In the case of non-blocking transport and the AT coding style, the behavior of the four phases BEGIN_REQ, END_REQ, BEGIN_RESP, END_RESP of the base protocol is built into the TLI adapter, which is thus able to communicate with a SystemC model that is compliant with the TLM-2.0 base protocol. With the current TLI implementation, it is possible to pass transaction types other than tlm_generic_payload by writing a user-defined conversion function, but it is not possible to pass user-defined phases between SystemVerilog and SystemC.

Another approach to passing non-standard transaction types between languages would be to use the SystemVerilog DPI directly, as described in [5].

5 Conclusions

This paper has described the TLM-2 features in VMM 1.2, the relationship between these and the original SystemC TLM-2.0 standard, and the upgraded VCS TLI. We have explained the motivation for each of the new features, and the advantages and disadvantages of each approach.

The new features of VMM-1.2 provide an enhanced communication model within VMM that makes it easier to integrate SystemC reference models. The upgraded VCS TLI simplifies the mechanics of passing transactions across the language boundary between SystemC and System-Verilog.

Both TLM-2-within-VMM and the VCS TLI have a place when integrating a SystemC reference model into a VMM test bench, depending on the SystemVerilog coding style adopted. Furthermore, the introduction of transaction-level communication into VMM makes the problem of integrating verification components developed using different methodologies more tractable.

6 References

[1] Verification Methodology Manual for SystemVerilog; Bergeron, Cerny, Hunter, Nightingale; Springer, ISBN-13: 978-0-387-25538

[2] VMM Standard Library User Guide, version D-2009.12, December 2009

[3] IEEE Std 1666-2005, SystemC Language Reference Manual

[4] OSCI TLM-2.0 Language Reference Manual, version JA32

[5] SystemVerilog Meets C++: Re-use of Existing C/C++ Models Just Got Easier, J.Aynsley, Design & Verification Conference (DVCon) San Jose, February 2010.

[6] VCS TLI Adapter User Guide, 3-November-2009