A Beginner's Guide to Using SystemC TLM-2.0 IP with UVM

Dr David Long and John Aynsley
Doulos
Ringwood, UK

Doug Smith
Doulos
Austin, Tx, USA

www.doulos.com

ABSTRACT

UVM 1.x includes support for the communication interfaces defined by the SystemC TLM-2.0 standard, although some implementation details differ. This enables integration of SystemC TLM-2.0 IP into a SystemVerilog UVM verification environment. The connection between SystemC and SystemVerilog currently requires a tool-specific language interface such as Synopsys TLI, since it is not yet implemented as part of UVM. This paper begins with a brief overview of TLM-2.0 aimed at novice users. It then discusses the steps required to add a SystemC TLM-2.0 model into a SystemVerilog UVM environment and simulate it with VCS. At each step, issues that users will face are explored and suggestions made for practical fixes, showing the relevant pieces of code. Finally, the paper gives a summary of areas where the UVM implementation of TLM-2.0 differs from the SystemC standard and proposes workarounds to ensure correct communication between the SystemVerilog and SystemC domains.
# Table of Contents

1. Introduction .................................................................................................................. 3

2. Overview of the TLM-2.0 features in UVM ................................................................ 3
   2.1 The motivation for adding TLM-2.0 features to UVM ............................................... 3
   2.2 Styles of transaction-level communication ................................................................. 4
   2.3 Overview of the UVM TLM classes ........................................................................... 5
      The transport interfaces ............................................................................................... 5
      Ports and exports ......................................................................................................... 7
      Analysis ports and exports ......................................................................................... 8
      Sockets ......................................................................................................................... 9
      Generic payload ......................................................................................................... 10
   2.4 The relationship between UVM TLM and the SystemC TLM-2.0 standard ................. 12

3. How to use TLM-2-style communication in UVM .......................................................... 14
   3.1 Communication options in UVM ............................................................................... 14
   3.2 Analysis ports versus callbacks .............................................................................. 15

4. Connecting a UVM test bench to a SystemC reference model ........................................ 15
   4.1 Dealing with timing differences between test bench and reference model ............... 15
   4.2 How to use the VCS TLI (Transaction-Level Interface) .......................................... 16
      Overview of the TLI .................................................................................................. 16
      TLI example .............................................................................................................. 17
      User-defined transactions and protocols ................................................................... 18

5. Conclusions ................................................................................................................... 19

6. References ..................................................................................................................... 19
1. Introduction

The Universal Verification Methodology (UVM) provides a powerful, flexible and intuitive framework for the construction of SystemVerilog verification environments. It is also an Accel-lera Systems Initiative standard. However, many SystemVerilog users also have algorithm design models (IP) written in C, C++, or sometimes SystemC. Furthermore, the emergence of the SystemC TLM-1 and TLM-2.0 transaction-level modeling standards has influenced the communication styles within SystemVerilog verification environments.

UVM introduces a set of features for transaction-level communication within a UVM verification environment inspired by the SystemC TLM-2.0 standard (now part of IEEE 1666-2011 [2]). The inclusion of these features was motivated in part by the need to connect a UVM test bench to a SystemC reference model. The major EDA tool vendors each have a mechanism that supports connection of SystemC and SystemVerilog models at the transaction level, such as SystemVerilog Direct Programming Interface(DPI). The VCS TLI (Transaction-Level Interface), which is the Synopsys solution, has TLM-2.0 features supported.

This paper explains the TLM-2.0 features of UVM and how they integrate with the other more generally used communication features of UVM. We also offer practical guidance on using the VCS TLI with the TLM-2.0 standard to integrate C, C++ and SystemC code into a UVM-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 IP 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-2.0 features in UVM

2.1 The motivation for adding TLM-2.0 features to UVM

UVM was developed from the earlier OVM and VMM methodologies that have always used transaction-level communication for abstraction, speed and productivity.

Mixed-language simulation environments involving a UVM 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 standard is being used to achieve interoperability between the components of such a virtual platform model. If a constrained random UVM 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 UVM promises to make life easier for the UVM 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 works in UVM. In particular, both UVM and TLM-2.0 support the idea that each transaction can have a finite lifetime with a well-defined time-of-birth and time-of-death that makes the usage of SystemC models more flexible compared with the TLM-1 standard.

2.2 Styles of transaction-level communication

Transaction-level communication in current languages (i.e. SystemVerilog and SystemC) and methodologies (i.e. VMM, OVM, UVM and TLM-2.0) differs in several respects.

All of the above-mentioned methodologies represent transactions as objects passed as arguments to object-oriented method calls. These are the put(trans) and get(trans) methods in the case of TLM-1, vmm_channel and the sequencer-driver connection found in OVM/UVM, 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. UVM, VMM, OVM 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, vmm_channel and the OVM/UVM tlm_fifo 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.

UVM provides a diverse set of communication mechanisms including channels, notifications, and callbacks. The primary mechanism for transaction-level communication within a UVM verification environment are pairs of matching ports and exports (uvm_*_port and uvm_*_export where * is put/get/peek/etc). A port calls an external interface method from within a component,
an export provides access to an interface method provided by the component. Where multiple pending transactions need to be queued, the queue could be provided by the component having the export (as is the case with the `uvm_sequencer`) or a dedicated channel such as `uvm_tlm_fifo` may be used between components to provide a true zero-length queue with a blocking completion model, a finite FIFO, or an infinite FIFO with a non-blocking completion model. UVM also supports TLM-2.0-style direct communication using sockets.

### 2.3 Overview of the UVM TLM classes

A full description of the UVM TLM classes can be found in the UVM Class Reference [1]. The intent of this section is not to repeat the information found in the standard documentation, but rather to give an overview of the more significant features and their intended use.

#### The transport interfaces

TLM-2.0 communication in UVM 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 UVM component that produces transactions, and the target another UVM component 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.

```c
void 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`.

```c
uvm_tlm_sync_e status;
uvm_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:
typedef enum
{
  UVM_TLM_ACCEPTED,
  UVM_TLM_UPDATED,
  UVM_TLM_COMPLETED
} uvm_tlm_sync_e;

typedef enum
{
  UNINITIALIZED_PHASE,
  BEGIN_REQ,
  END_REQ,
  BEGIN_RESP,
  END_RESP
} uvm_tlm_phase_e;

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

When two or more UVM components communicate using the transport interfaces, they do so using the UVM TLM ports and exports. The purpose of ports and exports is to provide a structured way of making method calls between UVM components (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 component. 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_phase method of a higher level component that a specific dependency is established between the producer and consumer components.

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

```plaintext
class producer extends uvm_component;
   uvm_tlm_b_transport_port #(my_tx) m_port;
   my_tx tx;
   ...
   m_port.b_transport(tx, delay);
   ...

class consumer extends uvm_component;
   uvm_tlm_b_transport_imp #(my_tx, consumer) m_export;

   task b_transport(my_tx trans, uvm_tlm_time delay);
   ...

class my_agent extends uvm_agent;
   producer m_producer;
   consumer m_consumer;

   virtual function void connect_phase(uvm_phase phase);
      m_producer.m_port.connect( 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 a special form of an export call an “imp” (short for “implementation”), and the higher-level component (in this case an “agent”) is connecting the port to the export. The connect 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 same type of the transaction (my_tx).

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 easier to use alternative to callbacks for the situation where the transaction does not need to be modified by the observer (this was the approach used in VMM). Note that UVM still provides a similar callback mechanism to VMM to ease the integration of existing VMM models. Callbacks should also be used where the transaction does need to modified.

The following example shows an analysis port:

```verbatim
class my_tx extends uvm_transaction; // User-defined transaction class
...

class my_monitor extends uvm_monitor;
    uvm_analysis_port #(my_tx) m_ap; // The analysis port
    ...
    virtual task main(uvm_phase phase);
        my_tx tx;
        ...
        m_ap.write(tx); // Broadcast transaction to all observers
    ...
```

The component 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 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:

```verbatim
class observer extends uvm_component;
    uvm_analysis_imp #(my_tx, observer) m_export;

    function new (string inst, uvm_component parent);
        ...
        m_export = new(this, "m_export"); // Every port and export needs to be constructed
    ...
```
The observer has an instance of an analysis imp export and so must implement the write method that the export will provide to the component that called it.

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

```cpp
class tb_env extends uvm_env;
my_monitor m_monitor;
observer m_observer_1;
another m_observer_2;
yet_another m_observer_3;

virtual function void build_phase;
  m_monitor = my_monitor::type_id::create( "m_transactor", this );
  m_observer_1 = observer::type_id::create( "m_observer_1", this );
  m_observer_2 = another::type_id::create( "m_observer_2", this );
  m_observer_3 = yet_another::type_id::create( "m_observer_3", this );
endfunction

virtual function void connect_phase;
  m_transactor.m_ap.connect(m_observer_1.m_export);
  m_transactor.m_ap.connect(m_observer_2.m_export);
  m_transactor.m_ap.connect(m_observer_3.m_export);
...```

Sockets

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

```cpp
class producer extends uvm_component;
  uvm_tlm_nb_initiator_socket #(producer,my_tx) m_socket;
...
  m_socket = new(this,"m_socket");  // Create socket instance in function new
...  status = m_socket.nb_transport_fw(tx, phase, delay);  // Call through socket in run phase
...
  // Must provide implementation of nb_transport_bw in producer
  virtual function uvm_tlm_sync_e nb_transport_bw(
    my_tx trans, ref uvm_tlm_phase_e ph, uvm_tlm_time delay);
...```
class consumer extends uvm_component;
    uvm_nb_target_socket #(consumer, my_tx) m_socket;
...
status = m_socket.nb_transport_bw(tx, phase, delay); // Call through socket in run phase
...
m_socket = new(this,"m_socket"); // Create socket instance in function new
//@ Must provide implementation of nb_transport-fw in consumer
virtual function uvm_tlm_sync_e nb_transport_fw(
    my_tx trans, ref uvm_tlm_phase_e ph, uvm_tlm_time 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 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 a protocol. The TLM sockets in UVM differ from those in the SystemC standard in that there are different socket types for blocking and non-blocking transport methods – it is not possible to call both through a single socket.

The initiator socket must be bound to the target socket with a single call to connect at the upper level that instantiated both producer and consumer, as shown below:

class my_env extends uvm_env;
    producer m_producer;
    consumer m_consumer;
    virtual function void connect_phase(uvm_phase phase);
        m_producer.m_socket.connect( m_consumer.m_socket );
    endfunction
...

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 UVM.

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 SystemC LRM (IEEE 1666-2011) [2]. In UVM a generic payload transaction can be created and transported as follows:

```verbatim
uvm_tlm_b_initiator_socket #(initiator, uvm_tlm_generic_payload) m_port;
...
begin
  uvm_tlm_generic_payload tx;
  uvm_tlm_time delay;

  assert( tx.randomize() with {
    m_command == UVM_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 uvm_error(“TLM”, “tx.randomize() failed”);

  m_port.b_transport(tx, delay);
  assert( tx.m_response_status == UVM_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 within its b_transport task:

```verbatim
task b_transport(uvm_tlm_generic_payload trans, uvm_tlm_time delay);
  uvm_command_e cmd = trans.m_command;
  bit [63:0] 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 =
        UVM_TLM_ADDRESS_ERROR_RESPONSE;
    return;
end
if (wid < len) begin
    trans.m_response_status =
        UVM_TLM_BURST_ERROR_RESPONSE;
    return;
end
if (cmd == UVM_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 == UVM_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 = UVM_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 UVM TLM and the SystemC TLM-2.0 standard

Although the implementation of TLM in UVM 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 UVM 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 UVM implementation differs from the SystemC TLM-2.0 standard.
UVM 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.

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, UVM makes use of separate port and export types for each kind of transport interface (blocking and non-blocking). Like SystemC, UVM combines all of the transport interfaces into a pair of initiator and target sockets to simplify binding. Unlike SystemC, the UVM 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. UVM achieves very similar results to TLM-2.0 using the `uvm_blocking_transport_imp_decl macro.

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. UVM does not provide this kind of automatic adaption.

In TLM-2.0, the generic_payload class defines get/set methods that must be used to access the members. In UVM, the generic payload data members are public and are declared as rand so the set/get methods are not required, although they are still provided. The UVM approach makes sense in the context of constrained random transaction generation; the UVM 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 UVM generic payload extends uvm_sequence_item, as would be the case for most other UVM transaction types, and makes use of the UVM field automation macros to define methods for copying, comparing, and printing the payload contents.

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 UVM generic payload does not provide methods for explicit memory management (i.e. acquire, release, reset).

Unlike SystemC, where the data and byte enable array attributes are pointers, in UVM they are dynamic arrays. This makes the transaction data part of the transaction object itself, which is a difference in philosophy from TLM-2.0.
Like TLM-2.0, UVM supports adding generic extensions to the generic payload class. 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 UVM

3.1 Communication options in UVM

UVM provides various options for communication: ports, exports and channels based around TLM 1.0 (e.g. uvm_tlm_req_rsp_channel), event notifications (using uvm_event or uvm_barrier), callbacks (uvm_callback), and the blocking and non-blocking transport ports and analysis ports inspired by the TLM-2.0 standard. The UVM Class Reference [1] describes each of these in detail. Some guidelines for when to use each of these options are given here:

- **uvm_tlm_b_initiator_socket** or **uvm_tlm_b_transport_port** may be used for master-like components that generate transactions to be consumed by other components. 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.

- **TLM-1.0 ports, exports and channels** are preferred for components that have a tightly coupled relationship between generated and consumed transaction objects (such as between a uvm_sequencer and uvm_driver). A TLM-1.0 port (e.g. uvm_put_port) provides maximum flexibility when writing a component that initiates the transactions, because it can be connected directly to a passive or reactive target with a corresponding export or indirectly to an active target via an appropriate channel (and can make blocking or non-blocking calls).

- **uvm_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.

- **uvm_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.

- **uvm_event** is preferred for synchronization between transactors in the case where there is no data passed along with the synchronization.

The non-blocking transport interface is not generally used in native UVM 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

\[ `uvm_{-}do\_callbacks(my_{-}cb,my\_{-}comp, write(this,tx)); \]

The effect is very similar, but there are differences. Unlike UVM callbacks, the name of the method called through an analysis port is fixed at `write`. A UVM 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 optional `ordering` argument allows 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 UVM callbacks are appropriate for modifying the behavior of components. 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 UVM callback with an analysis port in the same component, using the callback to inject an error and the analysis port to send the modified transaction to a scoreboard. In this situation, the UVM recommendation is to make the analysis call after the callback, as shown above.

4. Connecting a UVM test bench to a SystemC reference model

4.1 Dealing with timing differences between test bench and reference model

A SystemVerilog UVM 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 UVM these low-level transactions are generated by a low-level sequence. This low-level sequence is in turn typically generated by a higher-level sequence using a "layered sequence" or controlled from a "virtual" sequence.

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 transactions 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)

#### Overview of the TLI

The VCS TLI (Transaction-Level Interface) is an off-the-shelf mechanism for transaction-level communication between SystemVerilog and SystemC (see the VCS User Guide [4]). 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 [3]. However, the DPI has no native support for SystemC interface method calls or for UVM channels, so using the DPI to call SystemC methods from UVM is non-trivial.

The most recent release of the TLI from Synopsys has explicit support for UVM and for the SystemC TLM-2.0 standard, making it easy to integrate the TLM features of UVM described in this paper with a SystemC TLM-2.0 model. On the SystemVerilog side, the TLI supports communication using either UVM TLM ports/exports (transport or analysis). 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 several possible permutations:

- SystemVerilog blocking port, SystemC LT target
- SystemVerilog non-blocking port, SystemC AT target
- SystemVerilog analysis port, SystemC analysis subscriber
- SystemC LT initiator, SystemVerilog blocking export
- SystemC AT initiator, SystemVerilog non-blocking export
- 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 UVM and the SystemC 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.

**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 UVM component makes a *b_transport* call to send a transaction out through a TLM port:

```verilog
class my_initiator extends uvm_component;
    uvm_tlm_b_initiator_socket #(payload) m_initiator_socket;

    ...
    m_initiator_socket.b_transport(tx, delay);
```

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

```verilog`
'include "initiator.sv"
'include "target.sv"
```
class my_env extends uvm_env;
    my_initiator m_initiator1;
    function void connect_phase(uvm_phase phase);
        uvm_tlm2_sv_bind#(payload)::connect(m_initiator1.m_initiator_socket,
            UVM_TLM_B_TARGET,
            "port0");
    endfunction
...

On the SystemC side, the SystemC module has TLM-2.0 target/ initiator sockets:

class target : public sc_module
    .public tlm::tlm_fw_transport_if<my_payload_types>
{
    tlm::tlm_target_socket<32,my_payload_types> target_socket;
...

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 "uvm_tlm2_sc_bind.h"

class sc_top: sc_module
{
    target *m_trgt;
    SC_CTOR(sc_top) {
        m_trgt = new target("m_trgt");
        uvm_tlm2_bind_sc_target(m_trgt->target_socket,UVM_TLM_B,"port0",true);
    ...

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.

User-defined transactions and protocols

The above example shows a generic payload transaction being passed between a UVM 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 [3].

5. Conclusions

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

The features of UVM provide a communication model within UVM that makes it easier to integrate SystemC reference models. A future version of UVM is likely to include a DPI-based interface to SystemC (at the time of writing, one EDA vendor has recently announced their own DPI-based solution but it is not clear whether a similar approach will be adopted for UVM by the Accellera Systems Initiative Verification IP Technical Committee). In the meantime, the VCS TLI simplifies the mechanics of passing transactions across the language boundary between SystemC and SystemVerilog.

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

6. References