Why not “Connect” using UVM Connect: Mixed Language communication got easier with UVMC

Vishal Baskar
Product Engineer
Siemens SISW
Vishal.Baskar@siemens.com

***Abstract*- Today's world deals with a lot of designs involving mixed languages like SystemVerilog (SV) and SystemC (SC). This paper describes an easy method of integrating these two languages, using TLM connections made via UVM Connect (UVMC). Using a UVMC example[1], this paper will demonstrate how to build, connect and execute a verification simulation with SystemVerilog and SystemC.**

I. INTRODUCTION

With the increasing use of mixed language in today’s semiconductor and design industry, the question arises of how to effectively verify such complex designs. Thanks to the merger between Open SystemC Initiative (OSCI) and Accellera, SystemC and UVM can be easily connected via TLM-1.0 and 2.0. This allows complex models in SystemC to help verify complex designs in SystemVerilog.

UVM Connect was developed to make this process consistent and easy to debug. The Hybrid SC-SV model enables abstraction refinements at various levels, SystemC being a sweet spot for high-level modeling. Reusing the stimulus generation agents in SV to verify models in SC can be made possible. Also, a key feature to be noted is that when there is no single language restriction for Verification IPs, either SystemVerilog or SystemC versions may be used interchangeably. The strength of each language can be used to provide random verification for your model. And you can leverage the speed and capacity of SC for verifying untimed or loosely timed system-level environments. In this paper, we will discuss two examples, one is a simple example dealing with TLM-1.0 and a complex example with TLM-2.0.

II. BACKGROUND: TLM-1.0 AND TLM-2.0

Transaction Level Modeling (TLM) uses transaction-based methods which can be used for communication between modules. UVM offers TLM libraries such as ports, sockets, imp, and interface ports, The libraries branch into 2 versions, TLM-1.0 and TLM-2.0. As quoted in the IEEE std 1666-2011 IEEE Standard for Standard SystemC Language Reference Manual[3] Page.415/Sec.10,

a). TLM-1.0 has no standard transaction class, so each application must create its non-standard classes, resulting in very poor interoperability between models from different sources. TLM-2.0 addresses this shortcoming with the generic payload.

b) TLM-1.0 has no explicit support for timing annotation, so no standardized way of communicating timing information between models. TLM-2.0 addresses this shortcoming with the addition of timing annotation function arguments to the blocking and non-blocking transport interface.

c) The TLM-1.0 interfaces require all transaction objects and data to be passed by value or const reference, and models created using it must generate a delay using "wait", which may slow down simulation in certain use cases. TLM-2.0 passes transaction objects by non-const reference, which is a fast solution for modeling memory-mapped buses.

TLM-2.0 library provides model interoperability for memory-mapped bus modeling, and it recommended that core interfaces, sockets, generic payload, and base protocol be used together in concert. They are collectively known as the interoperability layer. If the generic payload is inappropriate, the core interface and both the initiator and target connector can be used with a different type of transaction. TLM-2.0 communication is pass-by-reference, which we emulate in UVM Connect by copying the changes made to the original transaction object on return from each interface method call. TLM-2.0 rules require that the same transaction object be used until transaction execution is complete.

This improves the efficiency of the execution. This paper discusses two examples. One is a simple example of data transfer by port connection and the other is data transfer of different transaction types and sizes on both the SV and SC side.

III. DIAGRAM OF THE EXAMPLE

In both the examples discussed in the paper, the transaction is initiated in the form of packets from the SV side and is sent using UVM Connect. The SC model here acts as a monitor, enabling unpacking and packing of the bits ensuring that there is no loss of data and the packet is sent back to the SV side using UVM connect. The received bits are then compared using a comparator or a checker to make sure the same bits that were sent are received. The only difference in the second example is the addition of data and address inverted from the SC side and the received packet is verified on the SV side.

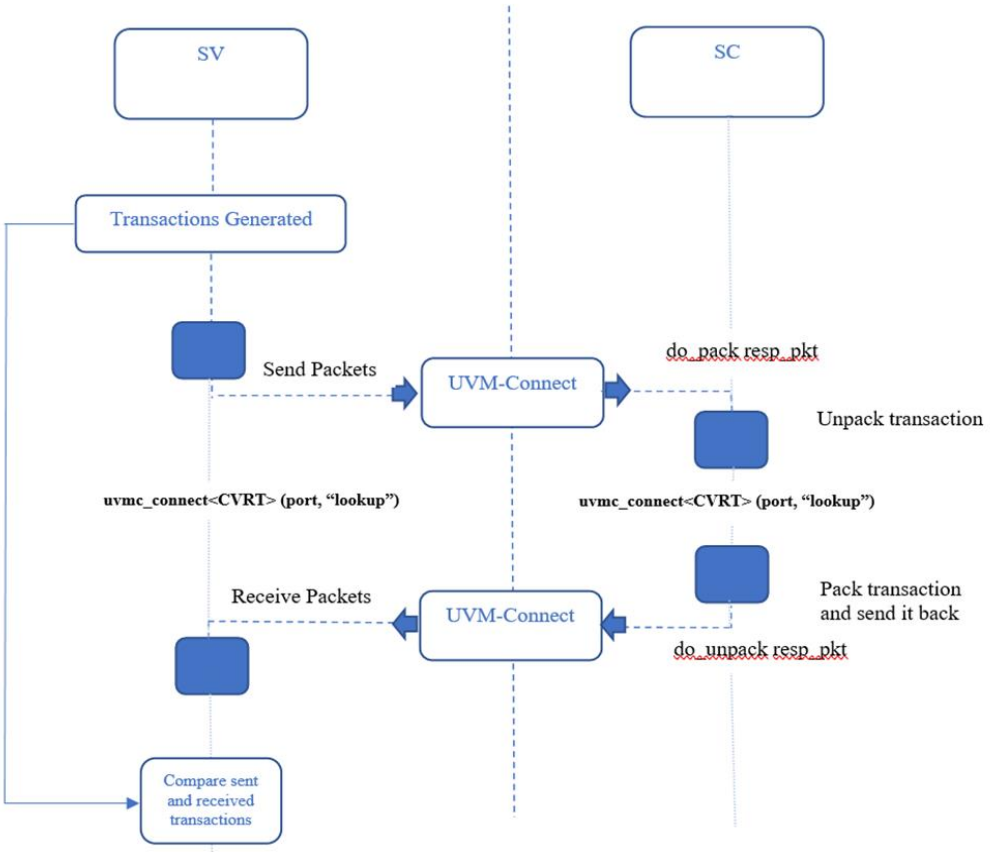


Figure 1: Diagram of Example

IV. EXAMPLE OF SIMPLE PORT CONNECTION

In this example, packets that are sent to both the SV and SC ports are of the same type. A producer_loop_back component initiates the transfers while a consumer_loop_back component returns the same transaction that is received. The goal at the end of the test is to ensure that there is no loss of information after packing and unpacking bits, and all packets that are sent are compared on the SV side with the packets that are received. The SC part here acts as a monitor, receiving packed bits of data and address, from the SV side, and sending them back by unpacking them using tlm ports.

Connections

One needs a handful of elements. A SystemC model, a SystemVerilog Testbench, TLM Connections, and UVMC Libraries. The example in this paper deals with packets of data being transferred from the SV side to the SC side via TLM 1.0 blocking transport port using the UVM connections, and unpacking is done on the SC side, and the bits are then sent back to the SV side, ensuring no loss of data. This paper will show the basic building blocks of SystemC model as a Monitor in a UVM testbench and connections are made via TLM ports, using UVMC, sending a data type, like an ethernet packet.

To communicate, verification components must agree on the data they are exchanging, and the interface used to exchange that data. The datatype for the connections should look the same regardless of the types of the TLM interfaces being connected. The only requirement is that the port types be compatible. Using UVMC we get port type matching for free from the C++ or SystemVerilog compilers. This example uses the SystemC side to be the monitor which can be scaled further involving the UVM environment and scoreboard.

SV Side: **uvmc_tlm1 #(request, response)::connect(out, "lookup")**
SC Side: **uvmc_connect (port, "lookup")**

Ports: The port's hierarchical name will be registered as a lookup string for matching against other port registrations within both SV and SC. A string match between two registered ports results in those ports being connected. Lookup strings are global across both SC and SV. A lookup string can be anything you wish if it is unique to other UVMC connections. Just before UVM's end_of_elaboration phase, UVM Connect will establish the actual cross-language connection. Also, on the SystemC side, when this function is called, you pass in a reference to the TLM instance and an optional lookup string. During elaboration, UVMC will connect the ports whose registered lookup strings match. In the example below, "transport_port_sv_out" is the lookup string and the same look up string will be used from the sc side.

```
    class producer_with_transport_port extends uvm_component;
          `uvm_component_utils(producer_with_transport_port)

          uvm_blocking_transport_port #(req_packet, rsp_packet) out;
               .
               .
          req_packet out_q[$];
          rsp_packet in_q[$];
          bit compare_status;
               .
               .
          virtual function void connect_phase(uvm_phase phase);
              super.connect_phase(phase);
              uvmc_tlm1 #(req_packet, rsp_packet)::connect(out, "transport_port_sv_out");
          endfunction
```

REQ and ***RSP:*** They are used to specify the request and response transaction types for bidirectional TLM1 ports. The default RSP type is the REQ type, so the RSP must be specified only if it is different from REQ. These are the required parameters for SC connections. On the SC side, you do not typically need to specify any type of parameters.

```
class req_packet extends uvm_sequence_item;

   rand bit rnw;
   rand bit [32-1:0] addr;
   rand logic [32-1:0] wdata;
   bit [32-1:0] rdata;
  .
  .
  .
endclass : req_packet
```

```
class rsp_packet extends uvm_sequence_item;

 rand bit [32-1:0] addr;
 rand bit [32-1:0] rdata;
.
.
.
endclass : rsp_packet
```

Comparator: The comparator function checks the transaction queues out_q and in_q and then checks for the size of the bits in the transactions, on the SV side. Compare function here is to check the data and address received matches.

```
function bit compare_req_rp_packet(req_packet req, rsp_packet rsp);
 if ( (req.addr == rsp.addr) && (req.wdata == rsp.rdata) ) begin
    return 1;
 end else begin
     return 0;
 end
endfunction
```

```
        virtual function void do_check();
              int unsigned min_size;
               `uvm_info(get_type_name(),    $psprintf("out_q    size:    %1d,    in_q    size:
%1d",out_q.size(),in_q.size()), UVM_LOW)
              if (out_q.size() == in_q.size()) begin
                  compare_status = 1;
              end else begin
                  compare_status = 0;
              end
              min_size = in_q.size();
              if (min_size > out_q.size()) begin
                  min_size = out_q.size();
              end
              for (int unsigned i_ = 0; i_ < min_size; i_++) begin
                  bit compare_status_l;
                  req_packet out_tx;
                  rsp_packet in_tx;
                  out_tx = out_q[i_];
                  in_tx = in_q[i_];
                  compare_status_l = compare_req_rp_packet(out_tx, in_tx);
                  compare_status &= compare_status_l;
                  if (compare_status_l == 0) begin
                  `uvm_warning(get_type_name(), $psprintf("index '%1d' does not match", i_))
                  end
              end
          endfunction
```

Packer: On the SC Side, we declare a simple transaction class with different sizes of bits and vectors. Once the transaction in the form of packets is received from the SV side to the SystemC side, the packet is unpacked and packed using the <do_pack> and <do_unpack> functions and sent back to the SV side.

```
  class req_packet
    {
  public:
          bool rnw;
          sc_bv < 32 > addr;
          sc_lv < 32 > wdata;
          sc_bv < 32 > rdata;

      req_packet() { }
      virtual ~req_packet() { }
      virtual void do_pack(uvmc_packer& p) const {
          p << rnw        ;
          p << addr       ;
          p << wdata      ;
          p << rdata      ;
      }

      virtual void do_unpack(uvmc_packer& p) {
          p >> rnw        ;
          p >> addr       ;
          p >> wdata      ;
          p >> rdata      ;
      }
};
```

```
  class rsp_packet
    {
  public:
          sc_bv < 32 > addr;
          sc_bv < 32 > rdata;
      rsp_packet() { }
      virtual ~rsp_packet() { }

  virtual  void  do_pack(uvmc_packer&  p)
const {
          p << addr       ;
          p << rdata      ;
      }

  virtual void do_unpack(uvmc_packer& p) {
          p >> addr       ;
          p >> rdata      ;
      }
  };
```

A consumer with a transport port on the SC side is created to receive the packet on the "put" port (**sv_out**) and send it back to the analysis port (**sv_in**) on the SV side. The idea here is to check we do not lose any information after packing and unpacking.

```
  class consumer_with_transport_port:
          public sc_module, public tlm::tlm_transport_if< req_packet, rsp_packet >
  {
  public:

    sc_export<tlm::tlm_transport_if < req_packet, rsp_packet > > sv_out;
    rsp_packet tx_rsp;

          consumer_with_transport_port(sc_module_name nm) :
        sc_module(nm), sv_out("sv_out"), tx_rsp()
          {
                  sv_out(*this);
          }
          rsp_packet transport(const req_packet &tx) {
                  cout << "received input in sc" << endl;
                  tx.do_print(cout);
          tx_rsp.addr = tx.addr;
          tx_rsp.rdata = tx.wdata;
          cout << "sent output to sv" << endl;
          tx_rsp.do_print(cout);
          return tx_rsp;
          }

  };
```

CVRT, CVRT_REQ, CVRT_RST: Here there is a converter declared for packing/unpacking the rsp and req type for the transport port. It is optional to use a converter policy class for this connection. In SV, you don't need to specify a converter for transaction types that extend uvm_object and implement the <do_pack> and <do_unpack> methods

```
  struct req_rsp_packet_converter : public uvmc_converter<req_packet> {
    public:
    static void do_pack(const req_packet &t, uvmc_packer &packer) {
        t.do_pack(packer);
    }
    static void do_unpack(req_packet &t, uvmc_packer &packer) {
        t.do_unpack(packer);
    }
  };
```

```
  int sc_main(int argc, char* argv[]) {
          consumer_with_transport_port cons("cons");

        uvmc_connect<req_rsp_packet_converter,    req_rsp_packet_converter    >    (cons.sv_out,
"transport_port_sv_out");
          sc_start();
          return 0;
  }
```

Once the transaction has been sent back to the SV side from the SC side, they are checked using a comparator function, do_check mentioned in sv_main. The transcript window prints the below data.

```
# Registering SV-side 'prod.out' and lookup string 'transport_port_sv_out' for later connection
with SC Connected SC-side 'sc_main/cons/sv_out' to SV-side 'prod.out'
  # UVM_INFO sv2sc_transport.sv(109) @ 0: prod [producer_with_transport_port] sending item '0':
  # ------------------------------------------
  # Name            Type        Size  Value
  # ------------------------------------------
  # req_seq_item_0  req_packet  -     @492
  #   rnw           integral    1     'h1
  #   addr          integral    32    'h2cdbf045
  #   wdata         integral    32    'h7ef09f51
  #   rdata         integral    32    'h0
  # ------------------------------------------
  received input in sc:
  rnw                 1
  addr                02cdbf045
  wdata               07ef09f51
  rdata               000000000
  sent output to sv:
  addr                02cdbf045
  rdata               07ef09f51
```

V. COMPLEX EXAMPLE USING CONVERTERS

In the previous example, we saw how transactions can be passed by simply connecting the ports of both the SC and SV sides using built-in support for the TLM 1.0 transaction type, using a blocking transport. But that holds good if the transactions type on both sides is the same. When they are different, meaning the types other than the generic payload are needed, one can define a conversion algorithm for all connections of a transaction type or design a custom-made conversion algorithm for each connection using TLM-2.0 libraries.

Suppose the two components are developed so that both the component agrees with the same content of the transaction, but this time their transaction definitions are of different types. This condition always occurs between components written in two different languages; they may not be able to share a common transaction definition. For these components to communicate with each other, one would need an adapter or converter, which converts between the transaction types defined in each language.

The members (properties) of the transaction classes do not have to have the same declaration number, type, and order in both languages. The converter can adapt different transaction definitions and at the same time serialize the data. The following are valid and compatible UVM Connect transaction definitions, assuming a properly coded converter:

```
SV                                  SC
class C;                            struct C {
  cmd_t cmd;                         long addr;
  shortint unsigned address;         vector<char> data;
  int payload[MAX_LEN];              bool write;
endclass                            };
```

Conversion in the SV side:

For this example, I have used the conversion algorithm within the transaction class itself. A transaction in UVM is derived from <uvm_sequence_item>, which defines the virtual methods <do_pack> and <do_unpack> that allow users to implement this conversion functionality. UVMC's default converter for SV works for these types of transactions. These macros, that expand into two or more lines of code are more efficient than using the packer's API directly. This option is the recommended option for SV-based transactions. Most transactions in SV should be defined this way as prescribed by UVM and it works with UVM Connect's standard SV converter.

```
//In-Transaction SV Conversion
  class packet_base extends uvm_sequence_item;

    `uvm_object_utils(packet_base)

    typedef enum { WRITE, READ } cmd_t;
    rand cmd_t cmd;
    rand int   addr;
    rand byte  data[$];
    rand int   q[$];
    function new(string name="");
      super.new(name);
    endfunction

    virtual function void do_pack(uvm_packer packer);
      `uvm_pack_enum(cmd)
      `uvm_pack_int(addr)
      `uvm_pack_queue(data)
      `uvm_pack_queue(q)
    endfunction
```

```
    virtual function void do_unpack(uvm_packer packer);
      `uvm_unpack_enum(cmd,cmd_t)
      `uvm_unpack_int(addr)
      `uvm_unpack_queue(data)
      `uvm_unpack_queue(q)
    endfunction

  endclass
```

Here we have defined a transaction class, Packet extended from packet_base. This class indirectly extends uvm_object. This also defines a generic producer model via ‘include. All transactions and components in the user library must be written out of context; i.e do not assume a UVMC or any other external connection.

```
  class packet extends packet_base;

    `uvm_object_utils(packet)

    rand int extra_int;
    function new(string name="");
      super.new(name);
    endfunction

    virtual function void do_pack(uvm_packer packer);
      super.do_pack(packer);
      `uvm_pack_int(extra_int)
    endfunction

    virtual function void do_unpack(uvm_packer packer);
      super.do_unpack(packer);
      `uvm_unpack_int(extra_int)
    endfunction
  `include "producer.sv"
endpackage : user_pkg
```

In the connect phase, we register the producer's output port for the UVMC connection using the search string "stimulus". The SC side registers its consumers’ port with the same search string. UVMC will match these two strings and complete the cross-language connection, i.e. SV producer’s <out> port will be bound to the SC consumer’s <in> export. Since our package class implements the required <do_pack> and <do_unpack> methods, we can take advantage of UVMC's standard converter, which delegates these methods.

```
module sv_main;
    .
    .
    .
    function void connect_phase(uvm_phase phase);
      uvmc_tlm #(packet)::connect(prod.out, "stimulus");
    endfunction
    .
    .
    .
  endclass

  initial begin
    env = new("env");
    run_test();
  end
endmodule
```

This is an example of a generic producer parameterized on the transaction type. The packets that are sent from the SV side are checked with the ones that are received from the SC side and are checked for inverted data and addresses. This ensures that there is no loss in packets during the process.

```
class producer #(type T=int) extends uvm_component;
   uvm_tlm_b_transport_port #(T) out;
   int num_pkts;

   `uvm_component_param_utils(producer #(T))

   function new(string name, uvm_component parent=null);
      super.new(name,parent);
      out = new("out", this);
      num_pkts = 10;
   endfunction : new

   task run_phase (uvm_phase phase);
     uvm_tlm_time delay = new("delay",1.0e-12);

     phase.raise_objection(this);

     for (int i = 1; i <= num_pkts; i++) begin
       int unsigned exp_addr;
       byte exp_data[$];

       T pkt = new(); //$sformatf("packet%0d",i));
       assert(pkt.randomize());
       delay.set_abstime(1,1e-9);

       exp_addr = ~pkt.addr;              //Checking for inverted address match from SC
       foreach (pkt.data[i])
         exp_data[i] = ~pkt.data[i];      //Checking for inverted data match from SC

       `uvm_info("PRODUCER/PKT/SEND_REQ",
          $sformatf("SV producer request:\n   %s", pkt.convert2string()), UVM_MEDIUM)

       out.b_transport(pkt,delay);

       `uvm_info("PRODUCER/PKT/RECV_RSP",
          $sformatf("SV producer response:\n   %s\n", pkt.convert2string()), UVM_MEDIUM)

     if (exp_addr != pkt.addr)
         `uvm_error("PRODUCER/PKT/RSP_MISCOMPARE",
           $sformatf("SV producer expected returned address to be %h, got back %h",
                   exp_addr,pkt.addr))
     if (exp_data != pkt.data)
         `uvm_error("PRODUCER/PKT/RSP_MISCOMPARE",
           $sformatf("SV producer expected returned data to be %p, got back %p",
                     exp_data,pkt.data))
     end
     `uvm_info("PRODUCER/END_TEST","Dropping objection to ending the test",UVM_LOW)
     phase.drop_objection(this);

   endtask
endclass
```

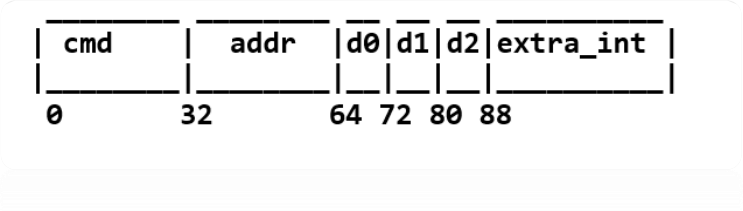
Conversion in the SC Side:

```
class packet_base extends uvm_sequence_item:
   typedef enum {WRITE, READ} cmd_t;
   cmd_t cmd;
   int   addr;
   byte  data[$];
endclass

class packet:
 int extra_int;
endclass
```

Ideally, we would have this mirror the transaction types on the SV-side. In some cases, we would not be able to mirror the types of declaration order and types. We can choose a nominal way to write our custom converter by allowing the pack and unpack method to occur as is in the SV side and implement a sub-type to the SC converter specialization to convert according to how the bits are received in the SV side. We will define a converter for this packet, then connect an instance of the consumer with an SV-side producer using a blocking transport interface conveying that transaction.

From the SV Side, the packet transaction will be packed normally with cmd, addr, data, and extra_int, assuming 3 bytes in the data array, which looks like this:



It is a simple class ‘packet’ transaction on the SC side. Hence, we can adapt in the SC side as follows:

- map 32-bit cmd from SV to a single bool in SC
- map 32-bit addr from SV into two: addr_lo and addr_hi 16-bit values in SC
- map data byte array data from SV to an integer array in SC

```
  class packet
{   public:
    short addr_hi;
    short addr_lo;
    unsigned int payload[4];
    char  len;
    unsigned int sc_q[4];
    bool  write; // 1=write, 0=read
};
```

Custom Converter:

With the usage of a separate converter class, one is not limited to member-to-member, bit compatible packing, and unpacking. In some cases, like our example here, we can't use the default converter, the reason is that it delegates to the transaction pack and unpack methods that the package class user_pkg doesn't have. When one implements the converter's < do_pack> and <do_unpack> functions, you will stream members of your transaction to and from the <packer> variable. This is an instance of < uvmc_packer > that inherits from an inner base class.

```
#include "uvmc.h"
using namespace uvmc;
using namespace user_lib;

struct packet_converter : public uvmc_converter<packet>
{
  static void do_pack(const packet &t, uvmc_packer &packer) {
    int cmd_tmp;
    if (t.write)
      cmd_tmp = 0;
    else
      cmd_tmp = 1;
    packer << cmd_tmp
           << t.addr_lo << t.addr_hi
           << (int)(t.len) << t.payload << t.sc_q;
  }

  static void do_unpack(packet &t, uvmc_packer &packer) {
    int cmd_tmp;
    vector<unsigned char> data_tmp;
    packer >> cmd_tmp >> t.addr_lo >> t.addr_hi >> data_tmp;
      .
      .
      .
};
UVMC_PRINT_4(packet,addr_hi,addr_lo,len,write)
```

We can instantiate a generic consumer module in the top-level testbench environment with a tlm2 blocking port. We also register the consumer's <in> export to have a UVMC connection with a lookup string 'stimulus'. The SV-side will also register its producer's 'out' port with the same string 'stimulus'.

```
#include "consumer2.cpp"

class sc_env: public sc_module
{
  public:
  consumer<packet> cons;
  sc_env(sc_module_name nm) : cons("cons")
     {uvmc_connect<packet_converter>(cons.in,"stimulus");
  }
};

int sc_main(int argc, char* argv[])
{
  sc_env env("env");
  sc_start();
  return 0;
}
```

On looking at the consumer2.cpp code. We can see that it takes the input transaction from the SV side, inverts the data and its address before returning it to the SV side. The producer will check the data and the address has been inverted, which concludes that the transaction packets have been successful and not lost from the SV side and back.

```
template <class T>
class consumer : public sc_module, public tlm_blocking_transport_if<T> {

  public:
  sc_export<tlm_blocking_transport_if<T> > in;

  consumer(sc_module_name nm) : in("in")
  {
    in(*this);
  }
  virtual void b_transport(T& t, sc_core::sc_time& delay) {

    cout << sc_time_stamp() << " SC consumer executing packet:"
       << endl << "  " << t << " payload:{ "  ;

    for (int i=0; i<4; i++){
      cout << hex << t.payload[i];
      if (i != 3)
        cout << ", ";
   }
    cout << " }" << endl ;

   cout << sc_time_stamp() << " SC consumer queue:" << endl <<  " sc_q :{ " ;
     for (int j=0; j<4; j++){
        cout << hex << t.sc_q[j];
        if (j != 3)
        cout<< ", "; }
        cout << " }" << endl  ;
    wait(delay);
// invert address
    t.addr_lo = ~t.addr_lo;
    t.addr_hi = ~t.addr_hi;

// invert data
    for (int i=0; i<4; i++)
      t.payload[i] = ~t.payload[i];

    cout << sc_time_stamp() << " SC consumer packet executed:"
       << endl << "  " << t << " payload:{ ";

    for (int i=0; i < 4; i++) {
      cout << hex << t.payload[i];
      if (i != 3)
        cout << ", ";
    }
    cout << " }" << endl;
    delay = SC_ZERO_TIME;
  }
};
```

The transcript window shows the address and the data sent from the SV side, address and data inverted, and sent it back to the SV side as well.

```
  # UVM_INFO producer.sv(62) @ 0: env.prod [PRODUCER/PKT/SEND_REQ] SV producer request:
  # cmd:READ addr:c4eb4c43 data:'{-80, 23, -101, -127, -106, -104} extra_int:f1075ad5

  0 s SC consumer executing packet:
    '{addr_hi:c4eb addr_lo:4c43 len:6 write:0 } payload:{ 819b17b0, 9896, 0, 0 }
  1 ns SC consumer packet executed:
'{addr_hi:3b14 addr_lo:b3bc len:6 write:0 } payload:{ 7e64e84f, ffff6769, ffffffff, ffffffff }

  # UVM_INFO producer.sv(67) @ 1000: env.prod [PRODUCER/PKT/RECV_RSP] SV producer response:
  # cmd:READ addr:3b14b3bc data:'{79, -24, 100, 126, 105, 103} extra_int:ffffffff
```

VI. PRIOR CHALLENGES

- One burden that the DPI places on the user is that the objects on the SC side should align equivalent data types and fields with that of the SystemVerilog side. For example, an int on the C function side should be declared as an int in the DPI import SystemVerilog side.
  But this has been resolved by using a converter or an adapter that translates between the transaction types defined in each language irrespective of the transacting data types and sizes as seen in Section V
- Sending datatypes that are complex like dynamic arrays is not simple. This was solved using tlm2.0 using a generic payload for supporting built-in types, arrays, and even sub-objects as properties of your transaction class. The ~uvmc_packer~ supports packing and unpacking the following types. <bool>, < char, unsigned char>, <short, unsigned short>, sc_bv<N>, enums, vector<T>, list<T>, where KEY and T are among the mentioned types. This is also mentioned in [10]
- Earlier versions of the UVMC library “do_pack” and “do _unpack” functions implemented as methods of transactions had their shortcomings in payload length limiting it to not more than 4K Bytes. But with the latest UVMC library 2.3.1, they have not only removed fixed limitations on the data payload sizes but have also added the support for “fast packer converters” which greatly improves the performance. The latest library is available to download at verificationacademy.com [11]

VII. SUMMARY

UVMC is filled with libraries that are indeed powerful in linking SystemC and SystemVerilog. It bridges both SV and SC language boundaries by providing TLM-1.0 and TLM-2.0 connectivity between components in these two languages. Many C-program users write tests and can record their tests with SystemC directly into the UVM test bench. The UVMC libraries along with the interface layer and SV UVM components can be used to independently design and communicate without directly referencing each other, and can further be integrated into both native and mixed-language environments without modifications, making them reusable. It also provides a means of directly accessing and controlling UVM simulation via the UVM Command API. The UVMC library is open and is distributed under Apache license and the latest library is available in verificationacademy.com[12]. This modeling can naturally be scaled up by using this SC as a scoreboard model and can be reused by plugging it with UVM testbenches. With the increasingly complex designs that are emerging in today's world, it is safe to say that UVMC has the flexibility to connect different UVM components involving SV and SC models with ease.

VIII. REFERENCES

[1] Adam Erickson https://s3.amazonaws.com/verificationacademy-news/DVCon2013/Papers/MGC_DVCon_13_Transaction-Level_Friending_An_Open-Source_Standards-Based_Library.pdf
[2] IEEE Std 1800-2005 “IEEE Standard for SystemVerilog – Unified Hardware Design, Specification, and Verification Language,”
[3] IEEE Std 1666-2005 “IEEE Standard SystemC Language Reference Manual”
[4] OSCI TLM-2.0 Language Reference Manual, version JA32, 2009
[5] https://www.amiq.com/consulting/2017/09/08/how-to-align-systemverilog-to-systemc-tlm-transactions-definitions/
[6] https://sutherland-hdl.com/papers/2004-SNUG-Europe-paper_SystemVerilog_DPI_with_SystemC.pdf
[7] https://verificationacademy.com/resources/technical-papers/transaction-level-friending--an-open-source-standards-based-library-for connecting-tlm-models-in-systemc-and-systemverilog
[8] UVM Cookbook - http://verificationacademy.com/uvm-ovm
[9] https://staging.doulos.com/media/1297/systemverilog-meets-cplusplus.pdf: SystemVerilog Meets C++: Re-use of Existing C/C++ Models Just Got Easier

[10] A. Moursi, R. Samhoud, Y. Kamal, M. Magdy, S. El-Ashry and A. Shalaby, "Different Reference Models for UVM Environment to Speed Up the Verification Time," 2018 19th International Workshop on Microprocessor and SOC Test and Verification (MTV), 2018, pp. 67-72, doi: 10.1109/MTV.2018.00023.
[11] https://verificationacademy.com/verification-methodology-reference/uvmc-2.3/docs/html/files/examples/converters/README-txt.html#Default_Converters
[12] Verification Academy. http://verificationacademy.com A free site for learning about almost any verification topic