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

Vishal Baskar

Siemens Industry Software, 46871 Bayside Parkway, Fremont, CA 94538 Vishal.Baskar@siemens.com

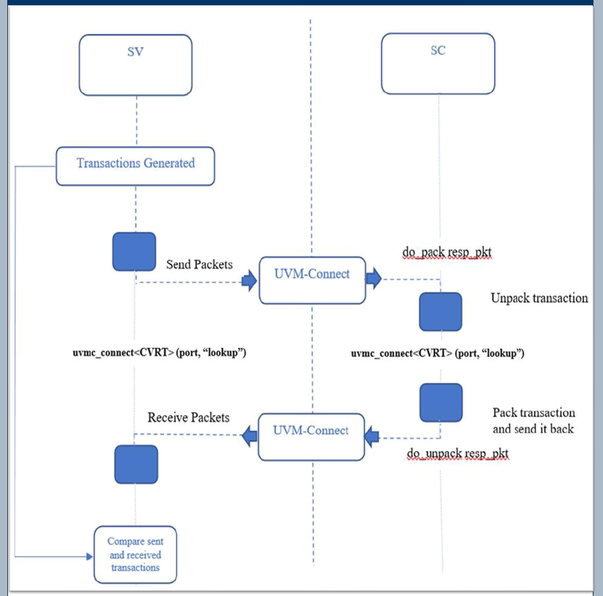
INTRODUCTION

Abbreviations
SC: SystemC
SV: SystemVerilog
TLM: Transaction Level Modelling
UVMC: Universal Verification Methodology Connect

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

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. 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. TLM uses transaction-based methods which can be used for communication between modules.

DIAGRAM OF THE EXAMPLE



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.

EXAMPLE OF SIMPLE PORT CONNECTION

In this example, packets of data are 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. To communicate, the datatype for the connections should look the same regardless of the types. During elaboration, in the SV side, 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.

```
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
```

SV Side:

```
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
```

SC 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 class with a transport port in 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.

```
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);
      }
```

```
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;
}
```

A **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.

```
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
```

COMPLEX EXAMPLE USING CONVERTERS

When the data types other than the generic payload are required, 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. Although both the component agrees with the same content of the transaction, this time their transaction definitions are of different types. A **converter** can adapt different transaction definitions and at the same time serialize the data. In the SV side, a conversion algorithm is declared within transaction class itself which is derived from **uvm_sequence_item.**

```
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
```

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 consumer's 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

SV Side:

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

A generic producer is 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 ensuring no loss in packets during the process.

Producer:

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

  `uvm_component_param_utils(producer #(T))

    .
    .
```

Ideally, we would have this mirror the transaction types on the SV-side. Or we can choose a nominal way to write our **custom converter.** 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.

SC Side - Custom Converter:

```
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;
```

```
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;
}
```

Consumer:

```
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);
  } .
    .
    .
```

CHALLENGES AND THE FUTURE

- Earlier, the objects on the SC side should have **equivalent data types** and **fields** with that of the SV side. This was resolved by using a **converter or an adapter** that translates between the transaction types irrespective of the data types and sizes.
- 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.
- UVMC **bridges** both language boundaries by providing TLM-1.0 and TLM-2.0 connectivity between components. The UVMC libraries and SV UVM components can be used to **independently design and communicate** without referencing each other and can further be integrated into both native and mixed-language environments without modifications, making them **reusable**. 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.

TEMPLATE DESIGN © 2008
www.PosterPresentations.com