Driving *e* sequences and sequence items from UVM-SV using UVM-ML-OA

February 11, 2015

Contents

1 The Example Architecture

In the example the xbus ***e***VC is instantiated under the SV env component. The SV testbench drives the xbus ***e***VC using a sequencer proxy, as if the xbus ***e***VC was implemented in SV.

The sequence layering is enabled by the ml_sequencer_proxy and ml_seqr_tlm_if components connected by several TLM ports. The sequencer proxy and the TLM interface work together, performing the necessary handshake and data transfer, to enable driving sequences and sequence items into the xbus ***e***VC.

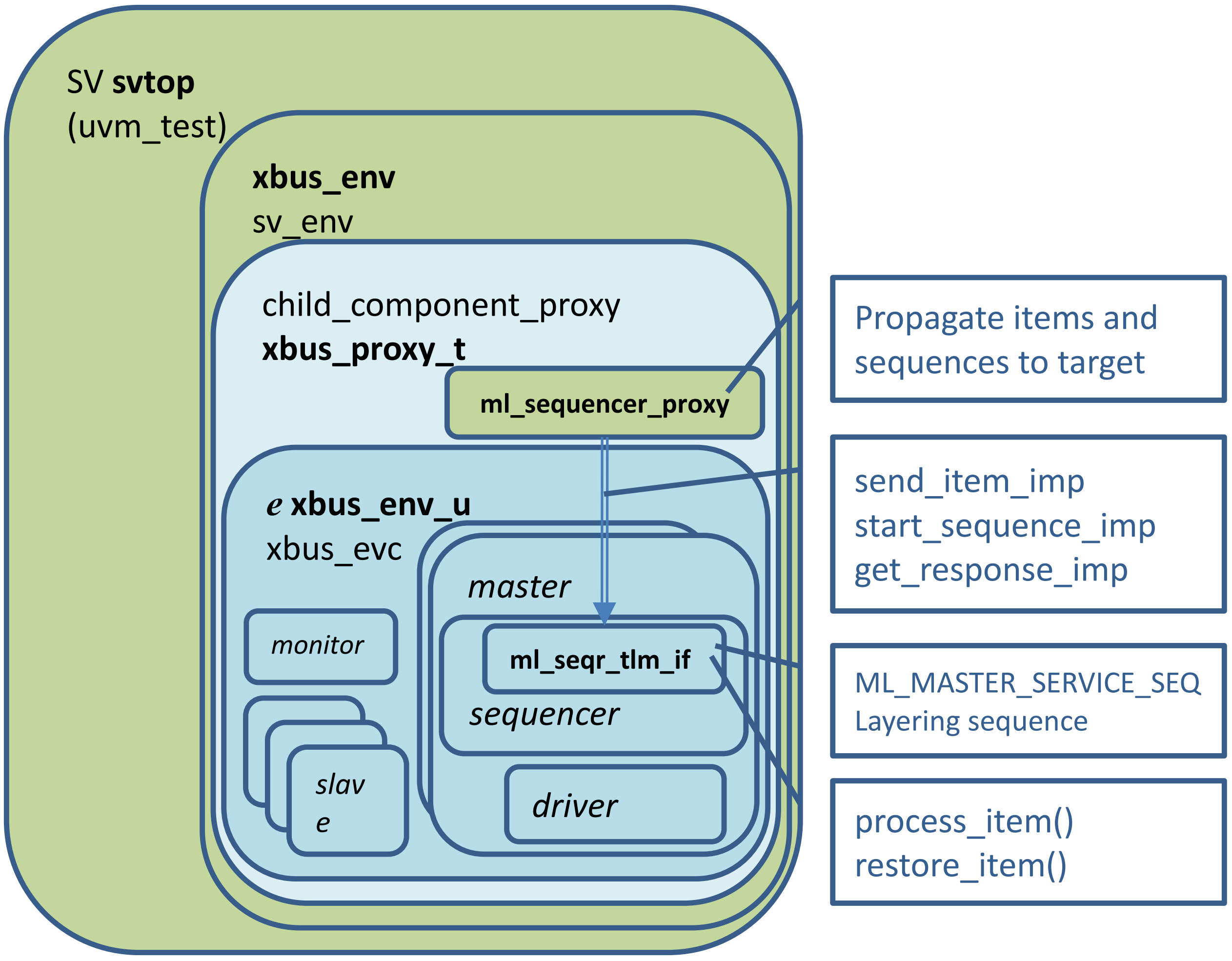

Figure 1. Architecture of the example

1.1 The Example

The SV code runs a single sequence xbus_seq driven from a virtual sequencer. It initiates 5 different transactions demonstrating different use cases:

- Writing a random 8 byte xbus sequence item
- Writing 4 bytes to a random address
- Reading back the 4 bytes in the same address
- Invoking an xbus sequence of WRITE_TRANSFER
- Invoking an xbus sequence of READ_TRANSFER

The xbus monitor reports the actual transactions on the bus. The activity can also be viewed using the waveform viewer.

1.2 Running the Example

Open the tar in a clean directory and set up UVM-ML-OA version 1.3 or later. To run the example:

```
setenv SEQ_LAYERING $UVM_ML_HOME/ml/examples/use_cases
```

```
cd $SEQ_LAYERING/sv_over_e
```

```
demo.sh IES
```

Or for GUI mode:

```
demo.sh IES -gui
```

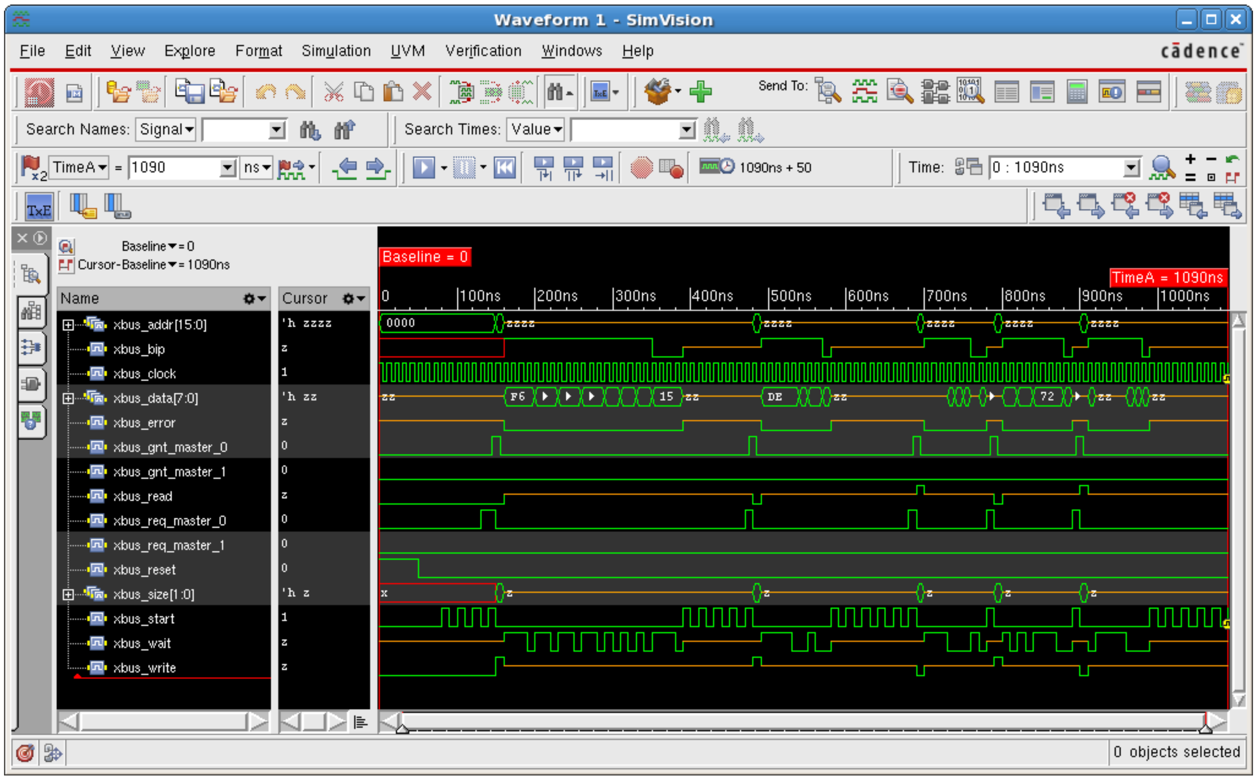

Figure 2. Waveform output from the example

2 Methodology

Sequence layering methodology in multi-language environment explains how to drive sequences and sequence items in one framework (e.g. UVM-*e*) from another framework (e.g. UVM-SV), using the UVM-ML-OA library.

2.1 Sequence Layering Principles

To facilitate the driving of an ***e*** sequencer from the outside, one must add TLM interfaces to provide bi-directional communication between the driving component and the sequencer. The example below makes use of a ***sequencer TLM interface*** which is a small unit instantiated in the target sequencer, containing the necessary TLM interfaces to drive this sequencer.

To provide full control from the outside, we override the default behavior of the ***e*** sequencer by shutting down the main sequence, thus only items and sequences initiated from the outside are executed.

On the SV side we instantiate a ***proxy sequencer*** in the child_component_proxy which represents the ***e*** environment. It acts like a normal sequencer except that it is driving the items and sequences directly to the target sequencer in ***e***. It does this through the TLM interfaces which are connected to the ***sequencer TLM interface*** in the target sequencer.

The ***sequencer TLM interface*** and ***proxy sequencer*** are included in the UVM-ML OA package. The sequence items are passed in push mode, i.e. the upper level component issues a non-blocking put() and the lower level schedules the item for execution. Sequences are also pushed from the upper layer and are spawned in a separate thread.

In addition to connecting the sequencers, one must also “export” the sequence item type and the sequences. Exporting the item is necessary because the type is used by the TLM interfaces and must be defined the same way in both languages. Exporting can be done manually as shown in the example or using ***mltypemap*** which is documented in the IES documentation in “mltypemap Utility: Automatic Generation of Data Types”. Exporting the sequences is needed to make those known to the ***proxy sequencer*** such that they can be used as native sequences in SV. Sequences must be translated manually as shown in the example.

2.2 Main Steps to Implement Sequence Layering

The following sections describe what needs to be done to enable sequence and item driving from SV to ***e***. The sections below describe the necessary steps to integrate the ***e*** VIP in an SV environment:

- Sending Sequence Items from SV
    - Exporting an *e* Sequencer
    - Exporting the Sequence Item
    - Sending Sequence Items
- Invoking e Sequences from SV
    - Exporting *e* Sequences
    - Invoking *e* Sequences

The first section deals with the basic use case where SV sequences generate sequence items to be executed in the ***e*** VIP. The second section deals with more complicated use cases where sequences in the VIP’s sequence library are exported to SV so they can be invoked like native sequences from SV.

The ML sequence driving solution described here suggests a combination of **sequencer proxy** in SV and a matching **sequencer TLM interface** in ***e***. The sequencer proxy and sequencer TLM interface are connected by several TLM1 ports:

- send_item_imp (non-blocking put from proxy sequencer)
- get_response_imp (blocking get from proxy sequencer)
- start_sequence_imp (blocking put from proxy sequencer) to be used when no return value is expected
- start_seq_imp (blocking transport from proxy sequencer) to be used when the sequence returns the result

This solution creates a convenient interface between the SV sequencer and the ***e*** sequencer, for passing items and activating sequences in the ***e*** sequencer from the SV code. You can see the code in sequencer_ proxy.sv and seqr_tlm_interface.e.

2.3 Sending Sequence Items from SV

To send items from SV to ***e***, one must connect the SV sequencer proxy to the corresponding ***e*** sequencer and translate the sequence item type to SV.

2.3.1 Exporting an *e* Sequencer

To export the ***e*** sequencer to SV, one must instantiate the sequencer proxy and sequencer TLM interface and connect them up.

The sequencer TLM interface is instantiated under the target sequencer in ***e*** using the provided template as shown below (see ml_sequencer.e). The template parameters are the type of the sequence item and the type of the layering sequence defined inside the sequencer TLM interface:

```
extend xbus_master_driver_u {
   tlm_if: ml_seqr_tlm_if of (xbus_trans_s,ML_MASTER_SERVICE_SEQ xbus_master_sequence)
           is instance;
     keep tlm_if.sequencer == me;
}; // extend xbus_master_driver_u
```

On the SV side derive a user defined proxy from the child_component_proxy class, which is the base class for foreign child components and contains the entire necessary infrastructure (e.g. the creation of the foreign component and the phase propagation mechanism). In this class add the **sequencer proxy** which will connect to the **sequencer TLM interface** in the target sequencer.

```
  class xbus_proxy_t extends child_component_proxy;
    ml_sequencer_proxy seqr_proxy_0;
    function new (string name, uvm_component parent=null);
      super.new(name,parent);
    endfunction
    function void build_phase(uvm_phase phase);
      super.build_phase(phase);
      seqr_proxy_0 = ml_sequencer_proxy::type_id::create("seqr_proxy_0", this);
    endfunction
    `uvm_component_utils(xbus_proxy_t)
  endclass
```

Next instantiate this user defined proxy and call its create_foreign_component() function, providing the framework and the type of the desired child component (see env.sv).

```
  class testbench extends uvm_env;
    xbus_proxy_t xbus_uvc;
    `uvm_component_utils(xbus_env)
    function new(string name, uvm_component parent=null);
      super.new(name,parent);
    endfunction
    function void build_phase(uvm_phase phase);
      super.build_phase(phase);
      xbus_uvc = xbus_proxy_t::type_id::create("xbus_uvc", this);
      assert(xbus_uvc.create_foreign_component("e", "xbus_env_u")==1);
    endfunction
    function void connect_phase(uvm_phase phase);
      super.connect_phase(phase);
      xbus_uvc.seqr_proxy_0.connect_proxy({xbus_uvc.get_full_name(),
                       ".active_masters[0].ACTIVE'MASTER'driver.tlm_if."});
    endfunction : connect_phase
  endclass : testbench
```

2.3.2 Exporting the Sequence Item

The xbus_trans_s type is defined in the xbus ***e***VC and the corresponding SV definition can be found in xbus_trans.sv. The developer can control which fields are generated in SV and which are left to be generated by the ***e***VC.

```
class xbus_trans_s extends uvm_sequence_item;
  rand xbus_trans_kind_t kind;
  rand xbus_addr_t addr;
  rand bit[1:0]  size_ctrl;
  rand xbus_read_write_t read_write;
  rand int unsigned size;
  rand byte unsigned data[];
  rand bit [3:0] MASTER__wait_states[];
  rand int MASTER__error_pos_master;
  rand int unsigned MASTER__transmit_delay;
  `uvm_object_utils_begin(xbus_trans_s)
   `uvm_field_enum(xbus_trans_kind_t, kind, UVM_ALL_ON)
    `uvm_field_int(addr, UVM_ALL_ON|UVM_DEC|UVM_UNSIGNED)
    `uvm_field_int(size_ctrl, UVM_ALL_ON|UVM_DEC|UVM_UNSIGNED)
    `uvm_field_enum(xbus_read_write_t, read_write, UVM_ALL_ON)
    `uvm_field_int(size, UVM_ALL_ON|UVM_DEC|UVM_UNSIGNED)
    `uvm_field_array_int(data, UVM_ALL_ON|UVM_DEC)
    `uvm_field_array_int(MASTER__wait_states, UVM_ALL_ON|UVM_DEC)
    `uvm_field_int(MASTER__error_pos_master, UVM_ALL_ON|UVM_DEC)
    `uvm_field_int(MASTER__transmit_delay, UVM_ALL_ON|UVM_DEC|UVM_UNSIGNED)
  `uvm_object_utils_end
```

```
   constraint default_master_kind{kind == MASTER;}
   constraint default_data_size{data.size() == size;}
   constraint default_direction{read_write inside {READ,WRITE};}
   constraint default_size_ctrl{size==1 -> size_ctrl==0;
                              size==2 -> size_ctrl==1;
                              size==4 -> size_ctrl==2;
                              size==8 -> size_ctrl==3;}
   constraint default_error_pos{MASTER__error_pos_master == -1;}
   constraint default_wait_size{MASTER__wait_states.size() == size;}
   constraint default_transmit_delay{MASTER__transmit_delay == 0;}
endclass : xbus_trans_s
```

In the ***e*** code, physical fields indicated by the % sign are deserialized by default. In this case we pass some non-physical fields from SV. To adjust the deserialization one must declare the non physical fields as follows:

```
extend uvm_ml_config {
    tlm_pass_field(f:rf_field) : bool is also {
        if ( f.get_declaring_struct().get_name() == "xbus_trans_s" ) {
            var n : string = f.get_name();
            if (n == "kind" or n == "size") {return TRUE;};
        };
        if ( f.get_declaring_struct().get_name() == "MASTER'kind xbus_trans_s" ) {
            var n : string = f.get_name();
            if (n == "wait_states" or
                n == "error_pos_master" or
                n == "transmit_delay") {return TRUE;};
        };
    };
};
```

2.3.2.1 *Using mltypemap to create the SV type*

The SV type definition above can be generated with the help of mltypemap as follows:

```
irun -snload ../../uvcs/xbus_simple/e/xbus_top.e -mltypemap_input do.tcl
```

With the following configuration (do.tcl):

```
configure_code -uvm_ml_oa
configure_type -type "e:MASTER cdn_xbus::xbus_trans_s" -skip_field driver \
-skip_field bus_name \
-skip_field check_error
configure_type -type "e:cdn_xbus::xbus_trans_s" -skip_field master_name \
-skip_field slave_name
```

```
configure_type -type "e:MONITOR cdn_xbus::xbus_trans_s" -skip_field waits \
-skip_field error_pos_mon
maptypes -from_type "e:cdn_xbus::xbus_trans_s" -base_name sn_xb -to_lang sv
```

2.3.3 Sending Sequence Items

Since this type is derived from uvm_sequence_item, it can be used in sequences running on the sequencer proxy. Doing an xbus_trans_s item in SV will result in doing the corresponding item in ***e*** as shown in seq_lib.sv:

```
      `uvm_do_with(xb_item, {addr == 'h1000;
                             read_write == WRITE;
                             size == 8;
                             })
```

On the ***e*** side the sequencer TLM interface can "do" an item created in SV using the process_item() method. The process_item() method can be simply a gen action using the item fields sent from SV. For example as shown in ml_sequencer.e:

```
   process_item(p : any_sequence_item) : MASTER xbus_trans_s is {
      var inp := p.as_a(MASTER xbus_trans_s);
      gen save_item keeping {
         .driver           == sequencer;
         .addr             == inp.addr;
         .data             == {inp.data};
         .size             == inp.size;
         .read_write       == inp.read_write;
         .size_ctrl        == inp.size_ctrl;
         .wait_states      == {inp.wait_states};
         .error_pos_master == inp.error_pos_master;
         .transmit_delay   == inp.transmit_delay;
      }; // gen save_item
      return save_item;
   };
```

The gen action above is needed to allow the ***e*** code generate valid values for fields that were not explicitly set from SV. If the generated item is not valid, a generation error is produced.

2.3.4 Getting Sequence Items Response

The sequence item does not get updated automatically. To receive the response from an item, for example the results of a read operation, the SV code must call update_done_item() which blocks until the item is done.

```
      uvm_sequence_item resp_item;
      void'(p_sequencer.update_done_item(resp_item));
      $cast(xb_item,resp_item);
```

After the item is executed and updated, the ***e*** code calls the hook method restore_item() which allows the user to process the modified item before returning the result to SV. By default it will just return the item (see ml_sequencer.e).

```
   restore_item() : MASTER xbus_trans_s is {
      result = save_item;
   }; // restore_item
```

2.4 Invoking *e* Sequences from SV

To be able to invoke ***e*** sequences from SV one must export them from the sequence library. Once these sequences are made available in SV, they can be used in SV sequences.

2.4.1 Exporting *e* Sequences

For each ***e*** sequence one must create a similar SV sequence (but without the body) derived from the ml_seq base class (which in turn is derived from uvm_sequence). This base class is responsible for forwarding the sequence to the ***e*** code. For example to be able to run the xbus WRITE_TRANSFER sequence from SV, define the following (see seq_lib.sv):

```
class WRITE_TRANSFER extends ml_seq;
   rand bit[15:0] base_addr;
   rand int unsigned size;
   rand bit[63:0] data;
  `uvm_object_utils_begin(WRITE_TRANSFER)
    `uvm_field_int(base_addr, UVM_ALL_ON)
    `uvm_field_int(size, UVM_ALL_ON)
    `uvm_field_int(data, UVM_ALL_ON)
  `uvm_object_utils_end
  `uvm_declare_p_sequencer(ml_proxy_sequencer #(ml_xbus_master_seq))
  function new(string name="WRITE_TRANSFER");
      super.new(name);
  endfunction
endclass : WRITE_TRANSFER
```

Every exported sequence must be declared and activated in the ML_MASTER_SERVICE_SEQ layering sequence as follows (see seq_lib.e):

```
struct WRITE_TRANSFER like ml_seq {
   % base_addr : uint(bits:16);
   % size      : uint;
   % data      : uint(bits:64);
}; // struct WRITE_TRANSFER
extend ML_MASTER_SERVICE_SEQ xbus_master_sequence {
   !wt_user_seq : WRITE_TRANSFER xbus_master_sequence;
   do_user_seq(p : ml_seq) @ driver.clock is also {
      if(p.kind == “WRITE_TRANSFER”) {
         do wt_user_seq keeping {
            .base_addr == p.as_a(WRITE_TRANSFER).base_addr;
            .size == p.as_a(WRITE_TRANSFER).size;
            .data == p.as_a(WRITE_TRANSFER).data;
         };
      };
   };
};
```

In the code above one can control which fields are set based on values from SV and which are left to the ***e***VC to be randomized. The fields set from SV are labeled as “physical” using the % sign. This way the fields are deserialized by default when received through the TLM interface.

2.4.2 Invoking *e* Sequences

Once exported, the sequence can be called in SV same as native sequences as shown in seq_lib.sv:

```
  READ_TRANSFER     rt_seq;    // xbus READ_TRANSFER sequence
  `uvm_do_with (rt_seq, {base_addr == target_addr; size == 4;})
  `uvm_info(get_type_name(),$sformatf("read %x", rt_seq.data),UVM_LOW);
```

Doing the WRITE_TRANSFER sequence in SV will result in doing the corresponding WRITE_TRANSFER sequence in ***e***.

By default the sequence interface waits for the sequence to complete and updates the sequence fields with the result, so the read sequence above will contain the data read from the DUT. However if the sequences do not require response, the interface can be configured to not update the sequence fields, thus achieving better performance. The following methods of the ml_sequencer_proxy control this capability:

```
seqr_proxy_0.set_seqr_resp_off(); // sequence response not required
seqr_proxy_0.set_seqr_resp_on(); // sequence response required
```