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

February 11, 2015

Contents

1 The Example Architecture

This document describes how to use a SystemVerilog VIP in an ***e*** environment. In particular we show how to connect and drive the VIP. The VIP example is the ubus which is also used in the UVM-SV release as an example.

In our example the ubus VIP is instantiated under the ***e*** 'ubus_env_proxy' component. The ***e*** testbench contains a virtual sequencer which drives the master sequencer in the ubus VIP. The remote driving from ***e*** is enabled by the sequencer-proxy/sequencer-TLM-interface components connected by several TLM ports. The **sequencer proxy** and **sequencer TLM interface** work together, performing the necessary handshake and data passing, to enable driving sequences and sequence items into the ubus VIP.

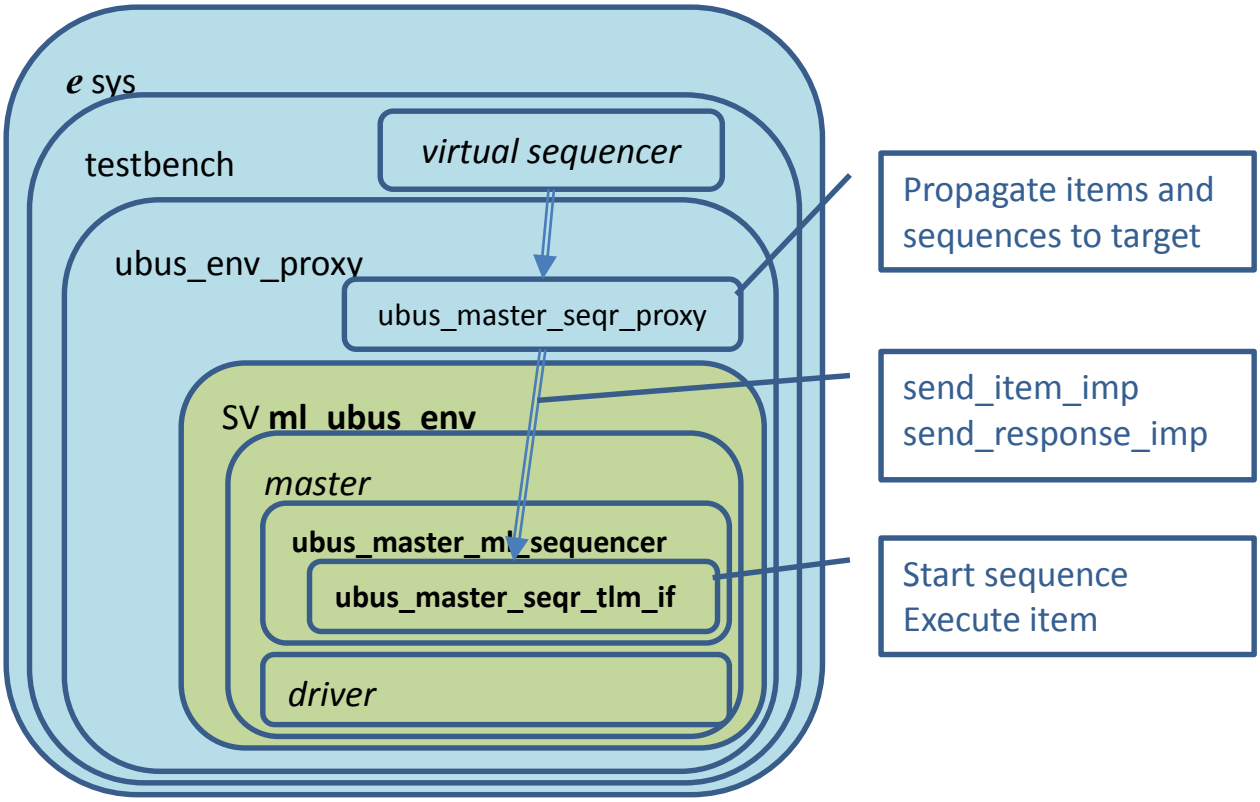

Figure 1. Architecture of the example

1.1 The Example

The ***e*** code has a virtual sequencer which drives the sequencer proxies. Each sequencer proxy runs a ubus sequence which does the following, demonstrating different use cases:

- Sending item to write 4 bytes to a random address
- Sending item to read back the 4 bytes
- Invoking a ubus sequence of write_word_seq
- Invoking a ubus sequence of read_word_seq

The ubus monitor logs 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/e_over_sv
demo.sh IES
```

Or for GUI mode:

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

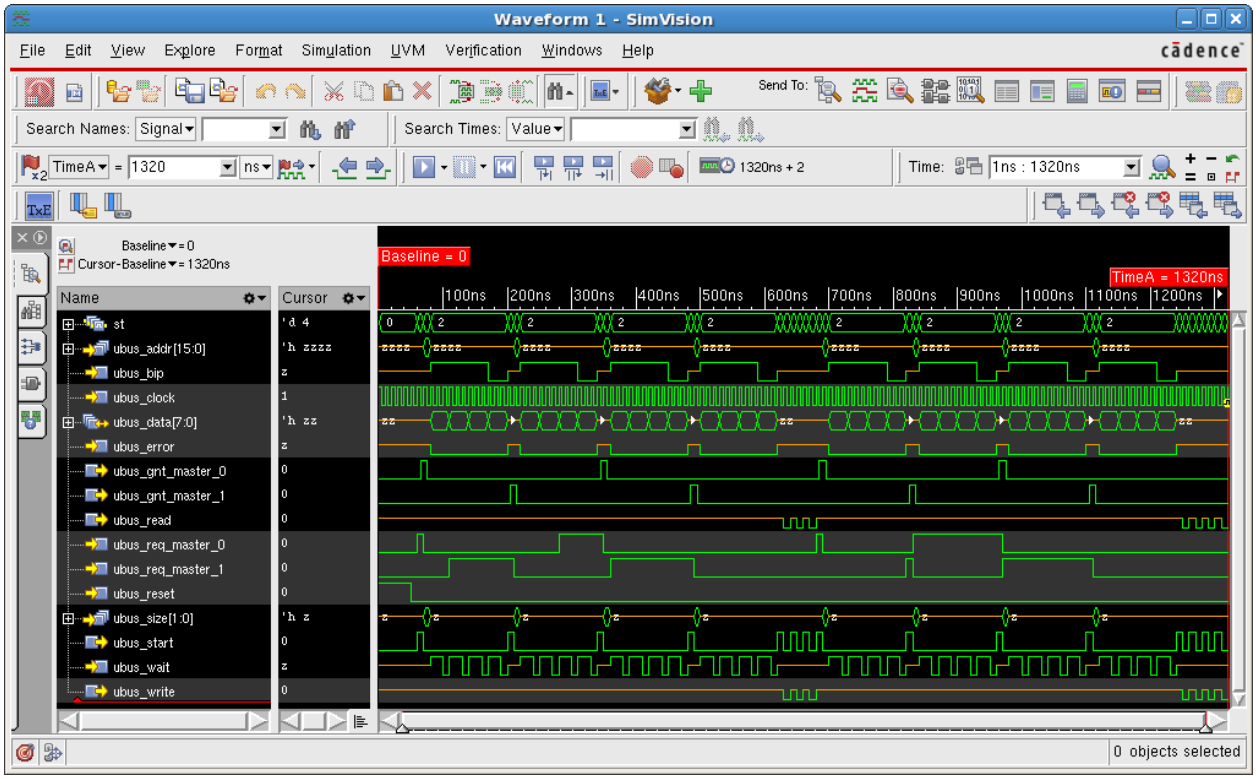

Figure 2. Waveform output from the example

2 Methodology

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

2.1 Sequence Layering Principles

To facilitate the driving of an SV 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 component instantiated in the target sequencer, containing the necessary TLM interfaces to drive this sequencer.

On the ***e*** side we instantiate a ***proxy sequencer*** which acts like a normal sequencer except it is driving the items and sequences directly to the target sequencer in SV. 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 pull mode, i.e. the lower level component issues a get() and waits for the item to be provided from the upper level. Sequences are 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 it is used by the TLM interfaces and must be defined the same way in both languages. Exporting the sequences is needed to make those known to the ***proxy sequencer*** such that they can be used as native sequences in ***e***. 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".

2.2 Main Steps to Implement Sequence Layering

The following sections describe the necessary steps to integrate the SV VIP into an ***e*** environment and how to enable sequence and item driving from ***e*** to SV. The sections below describe:

- Sending Sequence Items from *e*
    - Exporting an SV Sequencer
    - Exporting a Sequence Item

2.2.1.1 Using mltypemap to create the e type

The ***e*** type definition above can be generated with the help of run_mltypemap.sh with the following configuration (do.tcl):

```
configure_code -uvm_ml_oa

configure_type -type
"sv:uvm_pkg::uvm_sequence#(ubus_pkg::ubus_transfer,ubus_pkg::ubus_transfer)" -
use_existing

maptypes -from_type "sv:ubus_pkg::ubus_transfer" -from_type
"sv:ubus_pkg::write_word_seq" -from_type "sv:ubus_pkg::read_word_seq" -base_name sv_ub
-to_lang e
```

    - Sending Sequence Items
- Invoking SV Sequences from *e*
    - Exporting SV Sequences
    - Invoking SV Sequences

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

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

- send_item_imp (blocking get from sequencer TLM interface)
- get_response_imp (blocking put from sequencer TLM interface)
- start_sequence_p (blocking put from sequencer proxy)

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

2.3 Sending Sequence Items from *e*

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

2.3.1 Exporting an SV Sequencer

Each SV sequencer that you want to drive from ***e*** must be associated with a pair of sequencer-proxy/sequencer-TLM-interface as explained below.

2.3.1.1 Adding the TLM Interface to the Sequencer

On the SV side extend the sequencer by adding the sequencer TLM interface and replace the original sequencer as shown below (see ml_ubus_env.sv):

```
class ubus_master_ml_sequencer extends ubus_master_sequencer;
  ubus_master_seqr_tlm_if seqr_tlm_if;
  `uvm_component_utils(ubus_master_ml_sequencer)
  function new (string name, uvm_component parent);
    super.new(name, parent);
  endfunction : new
  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    seqr_tlm_if = ubus_master_seqr_tlm_if::type_id::create("seqr_tlm_if", this);
  endfunction
  // set the sequencer pointer of the TLM interface
  function void end_of_elaboration_phase(uvm_phase phase);
    void'(seqr_tlm_if.set_seqr(this));
  endfunction
endclass : ubus_master_ml_sequencer
...
set_type_override_by_type(uvm_sequencer#(ubus_transfer)::get_type(),
                          ubus_master_ml_sequencer::get_type());
```

The code for the sequencer TLM interface (see seqr_tlm_interface.sv) is derived from the seqr_tlm_if_imp base class (see seqr_tlm_if_imp.sv), and adds the ubus specific code as needed.

2.3.1.2 Instantiating the Sequencer Proxy

On the ***e*** side instantiate a sequencer proxy of type ubus_master_seqr_proxy and connect it to the sequencer TLM interface in SV as shown in testbench.e. The path to the target sequencer TLM interface is relative to the e_path. The ubus environment was created from this context as “vip_top” and the sequencer TLM interface was inserted in the sequencer of masters[0] as shown in the previous section.

```
   env_sequencer_0 : ubus_master_seqr_proxy is instance;
   connect_ports() is also {
      env_sequencer_0.connect_proxy(append(e_path(),
       ".uvc_top.masters[0].sequencer.seqr_tlm_if"));
   };
```

The sequencer proxy is derived from a template base class defined in ubus_master_seqr_proxy.e.

The code for ubus_master_seqr_proxy can be further enhanced with application specific content. For example if the data from a read operation 'rsp' needs to be propagated back to the original item 'req' in the ***e*** code, one can extend the sequencer proxy as shown in ubus_master_seq.e:

```
extend ubus_master_seqr_proxy {
   current : any_sequence_item;
   get(req : *any_sequence_item)@sys.any is only {
      req = me.get_next_item();
      current = req; // save original item
   };
   put(rsp : any_sequence_item) @sys.any is first {
      current.as_a(ubus_transfer).data = rsp.as_a(ubus_transfer).data;
   };
};
```

2.3.2 Exporting a Sequence Item

Data transfer between languages requires that both languages will have compatible definitions of the data type. This can be achieved either manually or with the help of mltypemap, which is available in the IES release.

The sequence item 'ubus_transfer' must be exported from SV to ***e***.

2.3.2.1 Map the Sequence Item

The sequence item (declared in ubus_transfer.sv) is derived from uvm_sequence_item. Since transferring sequence items (or any class derived from uvm_object) is handled automatically, we can ignore the fields of the base class when defining the corresponding type in ***e***. The struct we define has the same type name as the original SV class and the fields have identical names and same order as the SV class fields declared in the uvm_field_* macros.

The translated version of the ubus_transfer is as follows (see ubus_exported.e):

```
struct ubus_transfer like any_sequence_item {
  %addr : uint(bits:16);
  %read_write : ubus_read_write_enum;
  %size : uint;
  %data : list of byte;
  %wait_state : list of uint(bits:4);
```

```
  %error_pos : uint;
  %transmit_delay : uint;
  !%master : string;
  !%slave : string;
}; // end struct ubus_transfer
```

2.3.2.2 *Using mltypemap to create the e type*

The ***e*** type definition above can be generated with the help of run_mltypemap.sh with the following configuration (do.tcl):

```
configure_code -uvm_ml_oa
configure_type -type
"sv:uvm_pkg::uvm_sequence#(ubus_pkg::ubus_transfer,ubus_pkg::ubus_transfer)" -
use_existing
maptypes -from_type "sv:ubus_pkg::ubus_transfer" -from_type
"sv:ubus_pkg::write_word_seq" -from_type "sv:ubus_pkg::read_word_seq" -base_name sv_ub
-to_lang e
```

2.3.3 Sending Sequence Items

Once the sequencer is exported and the item is translated, the sequencer TLM interface can “do” an item created in ***e*** as shown in test.e:

```
extend WRITE_READ ubus_master_sequence {
   !req : ubus_transfer;
   body() @driver.clock is only {
      do req keeping {
         .read_write == WRITE;
         .addr in [0 .. 4095];
         .size == 4;
      };
```

The item is passed to the sequencer TLM interface in SV. On the way it is converted to the native SV type ubus_transfer which can be executed by the SV sequencer. Since the item is passed through a blocking TLM interface, it will wait until the sequencer is ready to execute it. The end result is doing the item in SV as a native sequence item.

2.4 Invoking SV Sequences from *e*

To be able invoking SV sequences from ***e*** one must export them from the sequence library. Once these sequences are made available in ***e***, they can be used in ***e*** sequences like native sequences. As explained below, we only export the fields of the sequences which were declared using the uvm_field_* macros. These fields can be set in ***e*** and then passed to SV to create a native SV sequence with the field values provided from ***e***.

2.4.1 Exporting SV Sequences

First we must select the sequences that we want to invoke from ***e*** from the sequence library.

For each exported sequence one must define a struct containing the fields of the sequence and the serializer in SV for this struct. The serializer is responsible for translating the data type when passed from one language to another.

For example the write_word_seq (defined in ubus_master_seq_lib.sv) will look in ***e*** as follows (see ubus_exported.e):

```
struct write_word_seq like ubus_base_sequence {
  %start_addr : uint(bits:16);
  %data0 : byte;
  %data1 : byte;
  %data2 : byte;
  %data3 : byte;
  %transmit_del : uint;
}; // end struct write_word_seq
```

The exported sequence struct must be wrapped by a native ***e*** sequence so that it can be used in ***e*** sequences. The wrapper is a WHEN subtype of the ubus_master_sequence, and it contains the exported struct as a member "exported_seq" as shown below.

```
extend ubus_master_sequence_kind : [WRITE_WORD];
extend WRITE_WORD ubus_master_sequence {
   exported_seq : write_word_seq;
   get_exported_seq() : ubus_base_sequence is {
      result = exported_seq;
   };
};
```

When doing this sequence in ***e***, the sequencer proxy will unwrap the struct to pass the relevant data to the sequencer TLM interface in a form that can be used as a native sequence in SV (see ubus_seq_lib.e):

The SV serializer for this sequence looks as follows (see ubus_exported.sv):

```
class ubus_pkg_write_word_seq_serializer extends uvm_ml::uvm_ml_class_serializer;
  function void deserialize(inout uvm_object obj);
    ubus_pkg::write_word_seq inst;
    $cast(inst,obj);
    inst.start_addr = unpack_field_int(16);
    inst.data0 = unpack_field_int(8);
    inst.data1 = unpack_field_int(8);
    inst.data2 = unpack_field_int(8);
    inst.data3 = unpack_field_int(8);
    inst.transmit_del = unpack_field_int(32);
  endfunction : deserialize
```

```
endclass : ubus_pkg_write_word_seq_serializer
// code for serializer registration
function int register_ubus_pkg_write_word_seq_serializer();
  ubus_pkg_write_word_seq_serializer inst;
  inst = new;
  return uvm_ml::register_class_serializer(inst,
                                           ubus_pkg::write_word_seq::get_type());
endfunction : register_ubus_pkg_write_word_seq_serializer
int dummy_ubus_pkg_write_word_seq_serializer =
       register_ubus_pkg_write_word_seq_serializer();
```

The code above show the serializer class which converts the ***e*** struct to an SV sequence. The class must be registered to associate it with the sequence class. The last line in the code above creates a call to the registration function to make sure the serializer is registered at the beginning of the simulation.

2.4.2 Invoking SV Sequences

Once exported, the sequence can be called in ***e*** same as native sequences as shown in test.e, except that the sequence parameters are constrained in the exported_seq member of the sequence:

```
extend ubus_master_sequence_kind : [WRITE_READ];
extend WRITE_READ ubus_master_sequence {
   !my_seq1 : WRITE_WORD ubus_master_sequence;
   body() @driver.clock is only {
      do my_seq1 keeping{
         .exported_seq.start_addr in [0 .. 4095];
         .exported_seq.transmit_del == 0;
      };
```

Doing the WRITE_WORD sequence in ***e*** will result in starting the corresponding write_word_seq sequence in SV.