2014 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION INDIA

How to Reuse Sequences with the UVM-ML Open Architecture library

Hannes Fröhlich, Cadence Design Systems, Bracknell, UK (*hannes@cadence.com*)

Kishore Sur, Cadence Design Systems, Bangalore, India (*kishore@cadence.com*)

***Abstract*—Reuse and control of data stimuli is one of the major aspects of VIP reuse. The UVM-ML Open Architecture library (UVM-ML OA) contains several features which enable sequence reuse in multi-language verification environments. There are of course several ways to reuse sequences and some are explored here. The trade-off between detailed top level controllability and the complexity of the solution needs to be understood in order to have a successful reuse experience. Finally there will be a detailed exploration of using special primitives for sequence reuse provided in the UVM-ML OA library.**

Keywords— UVM-ML OA, sequence reuse, system integration, Verification-IP(VIP)

I. INTRODUCTION

The UVM-ML OA library [1] has been available since July 2012. As established in [2] and other user experiences, this library contains a host of services and features which help to solve a real problem of today's verification projects: how to deal with a mixture of Verification-IP (VIP) which are developed using different languages and/or methodologies. The main building blocks of UVM-ML OA are the open source backplane and various framework specific adapters (e.g. UVM SV adapter, UVM SC adapter). Together they enable a number of features, including multi-framework construction, TLM communication, UVM-based phasing and UVM-like configuration.

The UVM-ML OA library itself does not have any specific services or features to enable reuse of data stimuli. However, the features of the library give users the ability to reuse and control stimuli in a number of different ways. This paper will first shortly explain how data stimulus is modeled and controlled in UVM. We then briefly explain how multi-language environments are constructed and configured. Next we will introduce different methods of sequence reuse, highlighting the trade offs of controllability versus effort. Finally we will explore in more detail how sequences and sequence items can be controlled in a multi-language environment using a combination of library provided primitives, type/factory overrides and TLM communication.

II. CONTROL OF VERIFICATION STIMULI

VIPs enable users to control the stimuli by way of constraining stimuli items and sequences. A stimuli item is an atomic transfer that contains all the information required to control an interface to the device under test (DUT) and transmit a specific piece of information. Examples are interface bus transfers, data packets and processor instructions. A sequence enables users to describe stimuli with respect to multiple items and their relationships, for example a write and read-back sequence of bus transfers. Sequences also can contain other sequences, which enables you to build up specific stimuli patterns with some parameters that can still be randomized. Sequences are generally located in a sequence library, which test writers can use to construct specific test cases.

When reusing VIPs, users generally expect to reuse sequences from the sequence library. In some cases specific items need to be controlled and additional sequences need to be defined. The actual tests generally control the first sequence that is started (e.g. default sequence in UVM SV, or MAIN sequence in UVM e), which in turn controls what other sequences and items will happen in this test.

III. MULTI-LANGUAGE ENVIRONMENT CONSTRUCTION

Most commercial simulators support multiple languages, so you can construct your testbench by just creating testbench components separately in each language. This is sometimes referred to as parallel tree construction. The benefit of this approach is that components can be added without having to modify your main testbench. The drawback is that you cannot configure foreign components (the ones in other languages) from the testbench. They are configured in their own domain. So you have to manually check on the correct configuration.

The UVM-ML OA library enables users to instantiate foreign components in a single hierarchy. This is referred to in [2] as unified hierarchy. The main benefits of this approach are the ability to configure components during the construction process from the top level testbench and the ability to phase various services, like the build phases, based on hierarchical order. The drawback is that users have to modify their testbench to construct foreign components.

The example below shows how a component modeled in e can be constructed from a UVM SV component. The construction looks pretty similar to regular UVM SV with the additional call to the create_foreign_component() function defined in UVM-ML OA.

```
class my_env extends uvm_env;
   child_component_proxy my_uvc;
   . . .
   virtual function void build_phase(uvm_phase phase);
      super.build_phase(phase);
      my_uvc =child_component_proxy::type_id::create("my_uvc",this);
      assert (my_uvc.create_foreign_component("e","ex_env_u") == 1);
   endfunction : build_phase
   . . .
```

Figure 1, Creation of an e component in UVM SV with the UVM-ML OA library

IV. MULTI-LANGUAGE ENVIRONMENT CONFIGURATION

As mentioned before, if the multi-language environment is constructed of parallel trees you have to configure VIPs within the language they are written in. This can lead to a kind of distributed configuration, which has to be checked carefully to ensure that all components of the multi-language environment are configured correctly. A much better way is available if components are constructed in the unified hierarchy using the UVM-ML OA library. In this case configuration can be done hierarchically. This enables top components to configure other components lower down in the environment hierarchy. Thus test writers can uniformly configure the whole verification environment from the unified top hierarchy. The configuration enabled by the library follows the model used in UVM SV. Users can place scalar values or configuration objects in a data base and tag to which components and members in the hierarchy those values or objects should be applied to. Below is an example showing how an e component can be configured from UVM SV.

```
class my_env extends uvm_env;
  child_component_proxy my_uvc;
  . . .
  virtual function void build_phase(uvm_phase phase);
     super.build_phase(phase);
     uvm_config_int::set(this,"my_uvc","nr_channels", 4);
     my_uvc =child_component_proxy::type_id::create("my_uvc",this);
     assert (my_uvc.create_foreign_component("e","ex_env_u") == 1);
  endfunction : build_phase
  . . .
```

```
extend ex_env_u {
   . . .
   nr_channels: uint;
   keep uvm_config_get(nr_channels);
};
```

Figure 2, Multi-language configuration (setting in UVM SV, getting in UVM e)

V. SEQUENCE REUSE OPTIONS

There are several ways of reusing sequences from a VIP in multi-language verification environments

- (1) Side by side - Use existing sequences without control from the top-level testbench by natively (same language domain) configuring sequences to be executed in reused VIP.
- (2) Unified configuration - Use the configuration mechanism of the UVM-ML OA library to pass the default/main sequence to be started in the reused VIP.
- (3) Unified control - Use additional primitives provided in the UVM-ML OA library to facilitate control of sequences and sequence items of the reused VIP. This includes the use of proxy sequencers and additional TLM ports to exchange the required information and synchronize the sequence and sequence item execution.

Options (1) and (2) enable users to pick the default sequence to be executed. This gives users the ability to control the traffic in general, but it does not enable fine-grained control or interaction with other sequences being driven in the testbench. However, it requires very little effort and can be easily achieved.

A. Side by side

This option can be used if you have parallel trees in a multi-language environment. As with configuration, you would also natively define what sequence is the default sequence, or what traffic is started in the MAIN sequence in UVM e. Again, you don't have a central point of control, so users have to carefully check that the right files are compiled and included in the current to ensure the correct traffic will be driven on the interfaces.

B. Unified configuration

As shown in section (VI) the UVM-ML OA library enables multi-language configuration in the single hierarchy. You can use this feature to also configure sequences executed in VIPs from the top level of your testbench. Below is an example based on the previous example, which showed hierarchical instantiation of foreign components.

```
class test_2 extends uvm_test;
  my_env tb_top;
  . . .
  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    tb_top = new("my_env", this);
    uvm_config_string::set(this,
       "*.my_uvc.masters[1].ACTIVE'driver","default_sequence",
       "WRITE_AND_READ");
  . . .
```

Figure 3, Controlling the sequence kind using the UVM-ML OA configuration mechanism

```
extend ex_master_sequencer_u {
   default_sequence: string;
   keep uvm_config_get(default_sequence);
   . . .
};
extend MAIN ex_sequence {
   !def_seq: ex_sequence;
   body()@driver.clock is only {
      do def_seq keeping {
        it.kind == driver.default_sequence.as_a(my_seq_kind)
      };
   };
};
```

Figure 4, Controlling traffic in the MAIN sequence using a configuration attribute

C. Unified control

Some users require more detailed control, and need to integrate the sequencer of a reused VIP with a top level virtual sequencer or test scheme. The solution for this case would be to use proxy sequencers and TLM based interface type provided as part of the UVM-ML OA library.

Sequencers generally interact with drivers/BFMs via a transaction based interface. This fact can be exploited as follows

- Instead of sending sequences/sequence items to the driver/BFM, one could send them via TLM port to another sequencer. This is what the proxy sequencer does. It looks like a sequencer to the outside world, but it doesn't communicate with a driver/BFM, but with a remote sequencer instead.
- The foreign sequencer can be enhanced to contain TLM ports to receive sequences and sequence items. In addition the sequencer functionality can be extended to consume those sequences and items to send to a connected driver/BFM.

The figure below outlines the overall scheme of using proxy sequencer and TLM ports for multi-language sequence layering.

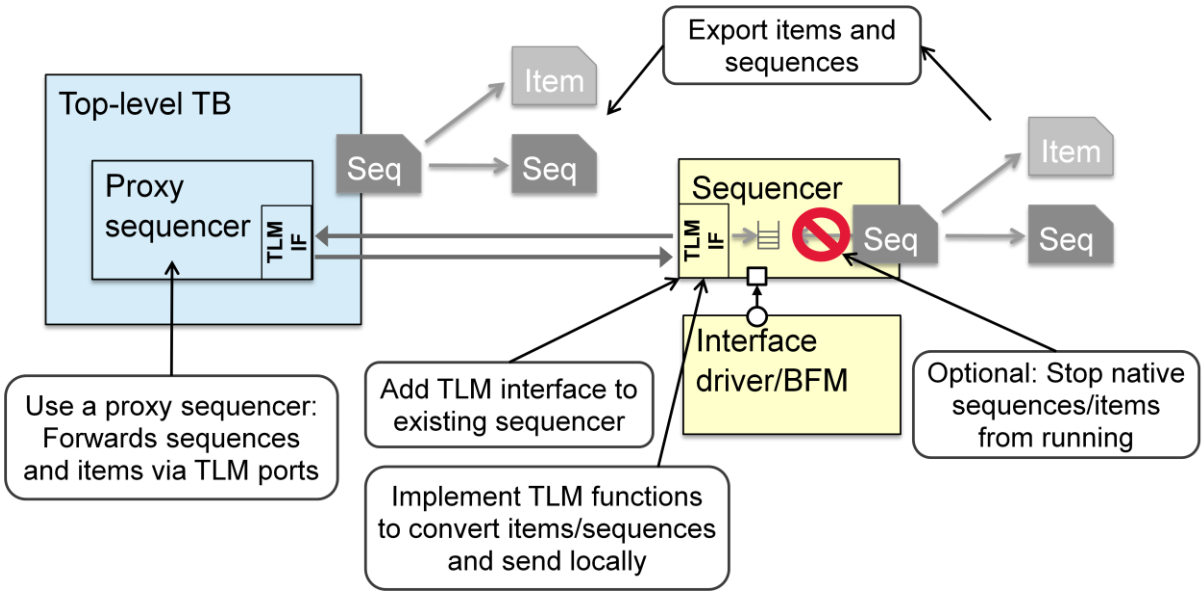

Figure 5, Overall flow of controlling sequences using a proxy sequencer and TLM ports

In the following sections we will explain in more detail how the proxy sequencer works and how to integrate the TLM interfaces with an existing sequencer.

VI. PROXY SEQUENCER DETAILS

The proxy sequencer inherits from a normal sequencer. It generally behaves like one, and users can integrate it in a top level testbench like any other sequencer.

The specific implementation makes the following changes. For sequence items, you generally have 4 distinct steps - the allocation step, the randomization step, the driving step and the "wait for item done" step. The proxy sequencer overrides the driving step by a call to an item-forwarding TLM port (non-blocking put). The "wait for item done" step is modified to also call a TLM port (blocking get) to receive the updated item when it is done. Below are some code fragments from the UVM-SV proxy sequencer supplied in the UVM-ML OA library.

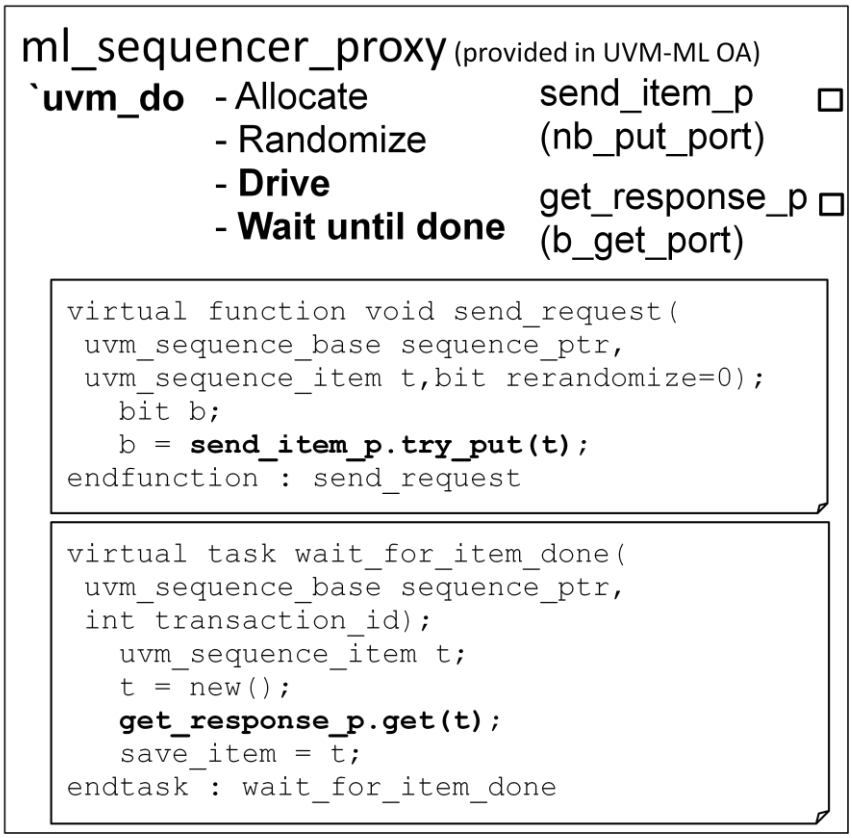

Figure 6, Code fragments from the UVM-SV proxy sequencer implementation

VII. TLM INTERFACE DETAILS

The TLM interface is a collection of TLM ports, which are also supplied in the UVM-ML OA library. Users need to integrate this TLM interface into the sequencer which needs to be controlled (i.e. the foreign language sequencer). The main integration steps are the instantiation and setting of a sequence pointer in the TLM interface to the sequencer. The TLM interface class itself already defines the TLM port implementation functions and tasks. Below is an example, implemented in UVM e.

The example shows how the item forwarding is implemented. Once a sequence item is passed via the port it will be executed on the sequencer. Note that there is an additional method call (process_item()), which is a user extendable hook to control how randomization is handled. This will be discussed in later section. Similarly on the response path there is also a hook to enable users to intercept transactions before sending them back to the top level sequencer. For sequence forwarding there is one additional TLM port which also connects to the TLM interface. It passes the sequence information and causes a sequence of a specific kind to be started on the foreign sequencer.

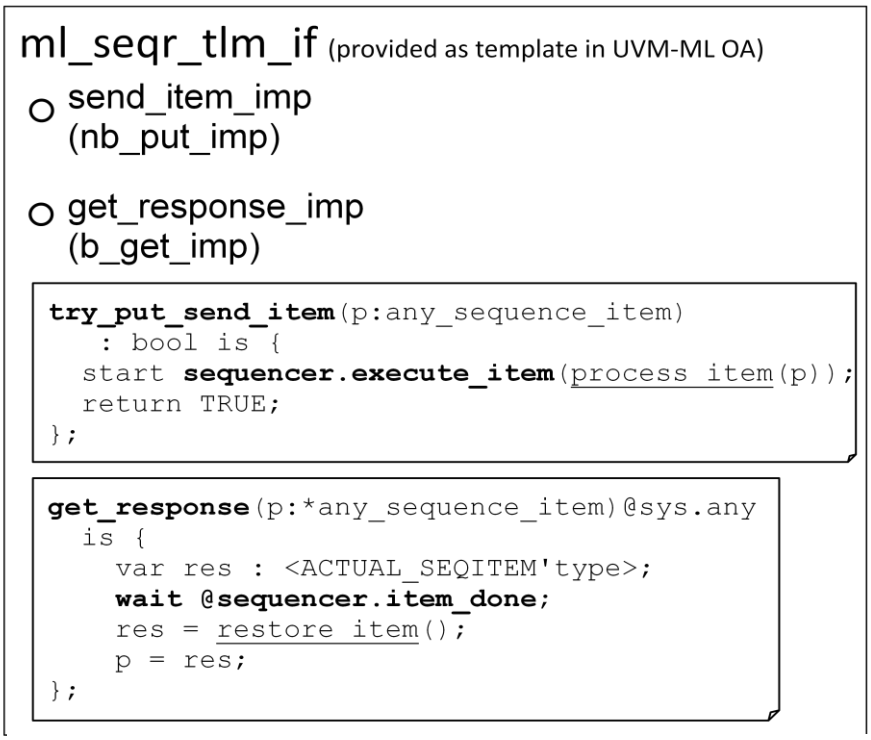

Figure 7, Code fragments from the UVM e TLM interface template struct

VIII. RANDOMIZATION CONTROL

In figure (5) it was shown that sequences and sequence items need to be exported from the foreign description into the top level language/methodology. Users basically have to create equivalent classes in the top level language (type mapping). This needs to be done to be able to control and constrain the relevant attributes from the top level testbench. Note that not all attributes need to be exported. For example if for bus transfers only the burst size and address are important, but not the actual data, then data would not have to be exported and would be randomly created in the VIP that is reused. This entirely depends on the verification project and requirements for a specific testbench.

All exported attributes can be controlled and constrained in top level tests. This means randomization is done in the top level language and then the transfers are passed into the reused VIP. There the user has the following choices regarding randomization:

- Disable randomization in the reused VIP and use attributes as is. Randomization can be controlled per attribute.
- Randomize locally, using the attribute as additional constraining input to keep the transfer in line with top level control and low level constraints that make a transaction legal/valid.

IX. TOP LEVEL INTEGRATION

Once the sequence items and sequences are exported and you've integrated the proxy sequencer and TLM interface structures, the VIP can be controlled via the proxy sequencer just like controlling a native sequencer. The figure below shows the proxy sequencer integration along with a call to the predefined connect_proxy() function, which connects all 3 TLM ports in one call.

```
class my_proxy_t extends child_component_proxy;
 // ml_sequencer_proxy (provided in the UVM-ML OA library)
 ml_sequencer_proxy seqr_proxy;
 // build phase
 seqr_proxy = ml_sequencer_proxy::type_id::create("seqr_proxy", this);
 . . .
endclass
class my_env extends uvm_component;
  my_proxy_t my_uvc;
  . . .
  // connect_phase
  my_uvc.seqr_proxy.connect_proxy({my_uvc.get_full_name(),
                          ".masters[0].ACTIVE'MASTER'driver.tlm_if."});
  . . .
endclass
```

Figure 8, Proxy sequencer instantiation and TLM port connection

Users can also use the UVM-SV configuration database [6] to enable tests to get a handle to the sequencer. An example of this is shown here.

```
class top_env extends uvm_env;
   my_env e_env;

   // connect_phase
   uvm_config_db#(ml_sequencer_proxy)::set(null, "",
                                          "proxy",e_env.my_uvc.seqr_proxy);
endclass
```

Figure 9, Making sequencer handle available via UVM SV configuration database

This enables users to easily get a handle to the sequencer from their tests. The example below shows how the proxy sequencer can be used. The example also makes use of the return path, calling the update_done_item() function to synchronize with the "wait for item done" step.

```
class my_seq extends uvm_sequence;
  my_trans_s        my_item;
  uvm_sequence_item resp_item;
  . . .
  virtual task body();
     // get pointer to ML sequencer
     ml_sequencer_proxy seqr_proxy;
     void'(uvm_config_db#(ml_sequencer_proxy)::get(null,"",
                                                   "proxy",seqr_proxy));
     // Write to random addresses
     `uvm_do_on_with(my_item, seqr_proxy, {addr inside {[0:'h07fff]};
                                     direction == WRITE;
                                     })
     void'(seqr_proxy.update_done_item(resp_item)); // (optional)
   . . .
endclass
```

Figure 10, Example test making use of the proxy sequencer

X. SUMMARY

In this paper we have shown multiple ways to control sequences in multi-language environments. We've also highlighted how UVM-ML OA library features enable sequence reuse and shown some of the implementation details as well as examples on how to implement a proxy sequencer based scheme. In general users can select from different approaches to sequence reuse, trading off between detailed controllability and integration overhead.

Since the reuse factor is the most important one, we do expect users to select the path of least resistance, i.e. we anticipate that users try to reduce the integration effort. However, if more detailed control is required, the sequence forwarding solution (unified control) does provide an easy way to control the traffic by sending sequences and sequence items from a proxy sequencer which is layered of the sequencer in the reused VIP.

REFERENCES

[1] UVM-ML Open Architecture (Overview and download) http://forums.accellera.org/files/file/65-uvm-ml-open-architecture

[2] B. Sniderman, V Yankelevich, "Multi-Language Verification: Solutions for Real World Problems", DVCon 2014, San Jose, CA

[3] IEEE-1800-2012, http://standards.ieee.org/findstds/standard/1800-2012.html

[4] IEEE-1647-2011, http://standards.ieee.org/findstds/standard/1647-2011.html

[5] Universal Verification Methodology – UVM, http://www.accellera.org/downloads/standards/uvm

[6] V. Cooper, "Demystifying the UVM Configuration Database", DVCon 2014, San Jose, CA