The OVM-VMM Interoperability Library: Bridging the Gap

Tom Fitzpatrick
Mentor Graphics Corp.
tom_fitzpatrick@mentor.com

Adam Erickson
Mentor Graphics Corp.
adam_erickson@mentor.com

ABSTRACT

The Accellera Verification IP Technical Subcommittee (VIP-TSC) has spent the past year-and-a-half developing an interoperability class library to allow OVM- and VMM[1]-based VIP to work together in a single environment. This paper will describe the primary obstacles we encountered while bridging these two independently-developed methodologies. It will then provide examples of how best to apply the resulting adapters, converters and other infrastructure comprising the VIP Interoperability reference library to enable OVM users to incorporate VMM-based IP in their environments and vice-versa. Since the VIP-TSC has published a Best Practices Document[1], this paper will not rehash the contents of that document. Rather, it will attempt to provide some broader context in which to examine the choices made in developing the library.

The paper will also describe how the library can be easily extended to accommodate even tighter integration among OVM and VMM-based components. Example uses of these extensions in the area of stimulus generation and VMM env reuse in an OVM environment are discussed.

1. INTRODUCTION

As testbenches comprise increasing numbers of verification IP (VIP) to meet the demands of shrinking schedules and the growing scale of system-level verification environments, the key to verification reuse is the ability to easily integrate VIP developed from multiple independent sources. A well-designed verification methodology provides guidelines that allow components to be developed independently yet work together when integrated within a larger testbench environment. The Open Verification Methodology (OVM) provides such guidelines and a supporting library that help ensure any OVM-based VIP will work with any other OVM IP. Similarly, the Verification Methodology Manual (VMM), encourages the development of VIP that can be used with other VMM-based IP.

A challenge for users arises when a testbench environment requires a mixture of both OVM-based and VMM-based IP. Preserving the reuse potential enjoyed by components deployed in homogeneous environments requires a reliable and standard mechanism for integrating such components in a mixed environment.

2. INTEGRATION ISSUES

When integrating OVM and VMM VIP, there are a number of issues that must be considered. In general, both OVM and VMM support similar concepts, such as phasing, transaction-based communication, configuration, messaging and so on, but there are important differences in the implementation of these concepts.

2.1. Phasing Differences

Both OVM and VMM partition test execution into a series of predefined phases. However, there are differences in the number of phases, the roles they are intended to perform, and the manner in which they are executed.

OVM	VMM
build	gen_cfg
connect	build
end_of_elaboration	reset_dut
start_of_simulation	cfg_dut
run	start
extract	wait_for_end
check	stop
report	cleanup
	report

Figure 1. OVM and VMM Predefined Phases

In OVM, the ovm_component base class—from which *all* user-defined components derive—defines a set of virtual methods corresponding to each phase. Users can override any or all of them in their derived component classes. The test flow is started by calling `run_test()`, which automatically calls all the phase methods in all components in the proper order. The next phase isn't started until all components have completed the previous phase.

In VMM, the concept of phasing via virtual method overrides exists only in the `vmm_env` class, of which there may be at most one instance in simulation. Phasing of the user's environment is managed by its `vmm_env` base class, but it is up to the user to manually initialize, configure, and start any children components in overrides of the env's phase methods.

There is some—but not complete—alignment between the set of OVM and VMM phases (see Figure 1.). Certain phases in OVM have no corresponding phase in VMM and vice versa. Where there is correspondence, there can be semantic differences, such as whether the phase is a task or a function and whether it is executed top-down or bottom-up in the hierarchy. Critically, the manner in which the time-consuming phases are executed are quite different between OVM and VMM. These differences in phasing lead to integration issues when instantiating, connecting and executing the environment.

For example, VMM envs define a *gen_cfg* and task-based *report* phase, neither of which are present in OVM. To enable VMM env integration in an OVM environment, the interoperability library's `avt_ovm_vmm_env` adapter uses OVM's flexible phasing

---

[1] All work done by the VIP-TSC was done using OVM 2.0[.*] and VMM 1.1[.*]

mechanism to register two new custom phases, *vmm_gen_cfg* and *vmm_report*, whose default implementations call the underlying VMM env's `gen_cfg()` and `report()` methods, respectively. Thus, whenever the `avt_ovm_vmm_env` adapter is used—that is, whenever a VMM env is integrated in an OVM environment—the VMM-specific phases are added to the list of phases that OVM will execute during simulation.

In both OVM and VMM, it is a rule that no phase may be executed before completion of the previous phase. In OVM, when a parent's `build()` method returns after creating one or more new child components, the phasing mechanism will recognize the new children and cycle them through any user-defined phases leading up to the *build* phase, such as the custom `vmm_gen_cfg` phase. Essentially, new children are "caught up" before OVM proceeds with the rest of the *build* phase. Thus, even when integrated in an OVM environment, the VMM env's `gen_cfg()` method will always be called before its build() method.

2.2. Testbench Construction Differences

Both OVM and VMM prescribe a process for assembling a verification environment whereby a parent component (which may be the top-level testbench) instantiates one or more children components, and then configures and connects each of them.

In OVM, the build process is partitioned into two phases, *build* and *connect*. These phases are implemented in the virtual `build()` and `connect()` methods in every OVM component.

The `build()` method performs child component instantiation and configuration, and the `connect()` method makes the connections that enable them to communicate with each other and with other components external to the parent.

Configuration is fetched and components are created in the *build* phase to allow users to override the types and number of components created without changing the parent container class. It doesn't have to be this way. A component may create children components in its constructor, but then testbench topology construction is less flexible overall, which limits reuse.

The following is a typical implementation of the `build()` and `connect()` phase methods.

```
class child extends ovm_component;
  ...
  int max_trans = 10; // default=10

  virtual function void build();
    // get user config, if any
    get_config_int(“max_trans”, max_trans);
  endfunction

endclass

class parent extends ovm_component;
  ...
  child c[$];

  virtual function void build();
    int num_child = 5;
    string name;

    // get config for this object
    get_config_int(“num_child”,num_child);

    // instantiate, based on config
    for (int i=0;i<num_child;i++) begin
      name = $sformatf(“c%0d”,i);
      c[i] = child::type_id::create(name, this);
    end

    // configure
    set_config_int(“c1”,”max_trans”,3);
  endfunction

  virtual function void connect();
    // connect
    child1.put_port.connect(child2.put_export);
  endfunction

endclass
```

```
class my_child extends child;
  // any user extensions of child class
endclass

class my_test extends ovm_component;

  parent my_env;

  virtual function void build();
    child::type_id::set_type_override(
                         my_child::get_type());
    set_config_int(“env”,”num_child”,10);
    my_env = parent::type_id::create(“env”,this);
  endfunction
  ...
endclass
```

The `create()` method in the `parent` is a call to the OVM factory requesting an instance of type `child`. Users may configure the factory to instead return any *extension* of type `child`, as is done with the call to `set_type_override()` in the `my_test` class Using the factory, we are able to substitute the types that get instantiated without actually changing the definition of the `parent` class..

The `set_config_int()` and `get_config_int()` calls in the example are example usages of OVM's configuration mechanism. The `set_config_*` methods store configuration settings in a table for future look-up by child components in calls to `get_config_*`. Deferring execution of configuration settings in this manner allows a parent to configure a child before the child even exists and without use of hierarchical references. Furthermore, we are able to dynamically specify the topology of the testbench—the number of child components, in this example—without changing the definition of the `parent` class.[2]

In contrast to OVM's build process, the VMM build process comprises a single pass via a call to the top-level vmm_env's `build()` method, which coordinates the instantiation, configuration, and connection of all components in the environment. Creation of its children occurs via direct calls to `new()`, which, in turn, create their children via direct calls to `new()`, and so on. Configuration and connection occur using special constructor arguments, direct assignment of properties, or calling of methods through hierarchical references.

---

[2] OVM's set/get_config can be used at any time, allowing dynamic parameter setting throughout the test. Topological configuration, however, must be done before the *end_of_elaboration* phase, typically in the *build* phase.

[4] Note that, since the OVM package is included via the interoperability library, VMM components may use the OVM factory mechanism to allocate new OVM types within a VMM environment, e.g. child1 = ovm_child::type_id::create().

```
class my_env extends vmm_env;
  ...
  virtual function void build();

    super.build();
    gen = new(..., my_chan);
    gen.randomized_obj = my_obj;

    subenv = new(..., my_consensus);
    ...
    subenv.configured();

    driver = new(...,custom_args,...);

  endfunction

endclass

class gen extends vmm_xactor;
  child child1;
  function void new(..., vmm_channel chan);
    super.new(...);
    child1 = new(...);
    ...
  endfunction

endclass
```

The differences in how OVM and VMM manage the build process leads to differences in how to handle instantiation of child components from the other library, depending on which methodology is the parent.

For an OVM parent instantiating a VMM child, the process is straightforward. All construction, configuration, and connection of the VMM child can be handled in the OVM parent's `build()` method in much the same manner that a `vmm_env` would do in its `build()` method.

```
class ovm_parent extends ovm_component;
  vmm_child child;

  virtual function void build();
    super.build();
    child = new(“vmm_child”,...);
    ...
  endfunction
  ...
endclass
```

By contrast, a VMM parent instantiating an OVM child must account for the separate phasing in the OVM build process. Since VMM does not employ a two-phase build process or the factory and configuration facilities, the constructor is the only means of allocating child components and configuring varying topologies.

First, the VMM parent instantiates the OVM child components normally. If the parent is a `vmm_env`, this occurs in the `build()` method. Otherwise, allocation occurs in the parent's constructor via a call to `new()` or the OVM factory's `create() method`.[4]

To account for the OVM build process, the VMM user's env is requred to extend from the interoperability library's `avt_vmm_ovm_env` adapter, not `vmm_env`. This “underpinning” allows the user env to inherit the `ovm_build()` method and other infrastructure needed in a mixed VMM-OVM environment. Thus defined, the user env's implementation of the `build()` method must call the inherited `ovm_build()` method *after* all VMM and OVM children have been allocated and *before* any OVM component connections.

At the completion of `ovm_build()`, all OVM component children and all their descendants will be built and connected. At this point, the VMM parent is free to connect and configure the OVM children. The following code illustrates:

```
class vmm_parent extends `VMM_ENV;

  ovm_child child1, child2;

  `ovm_build

  virtual function void build();
    super.build();
    child1 = new(“ovm_child1”,null);
    child2 = new(“ovm_child2”,null);
    ovm_build();
    child1.put_port.connect(child2.put_export);
    ...
  endfunction

  ...
endclass
```

Note the `` `ovm_build `` macro invoked within the env class definition. This macro declares an instance-specific version of the `ovm_build()` method, which ensures that the `ovm_build()` method will be called only once should the env ever be extended.

2.3. Transaction-Level Communication

Both OVM and VMM support the concept of transaction-level communication. OVM supports it through an implementation of the OSCI SystemC Transaction-Level Modeling (TLM) standard while VMM supports it through a proprietary implementation. To support interoperability between the two, the Accellera VIP-TSC developed a set of adapters that implement OVM functionality on one side and VMM functionality on the other.

2.3.1. Communication Semantics

In OVM, TLM connections are handled via *ports* and *exports*. A port is an object in an initiator component that specifies the interface (i.e. a set of methods and their semantics) required to communicate, while an export is an object in a target component that provides the implementation of an interface. The connection from a port to a compatible export is typically accomplished by calling the port's `connect()` method in the parent component's `connect()` phase method. OVM automatically checks for interface and transaction type compatibility between the given port and export. At run-time, connections in OVM are thus correct-by-construction.

```
virtual function void connect();
  child1.put_port.connect(child2.put_export);
endfunction
```

In VMM, connections between components occur through the `vmm_channel` component. The `vmm_channel` supports a unidirectional connection between two components, the producer and the consumer. Because the put and get portions of the channel interface are not partitioned via different exports, it is incumbent on the user to ensure that the producer uses only the channel's put-side methods and the consumer uses only the get-side methods, and that the producer and consumer support the same completion model.

The interoperability library provides a set of adapters to support generic channel-based communication. The `avt_tlm2channel` enables communication between an OVM producer and a VMM

---

consumer. It provides a set of OVM ports and exports for connecting to any OVM producer type. In the `connect()` method of the producer's and adapter's parent component, the producer 's port or export is connected to a compatible export or port in the adapter, and the remaining ports and exports left unconnected.

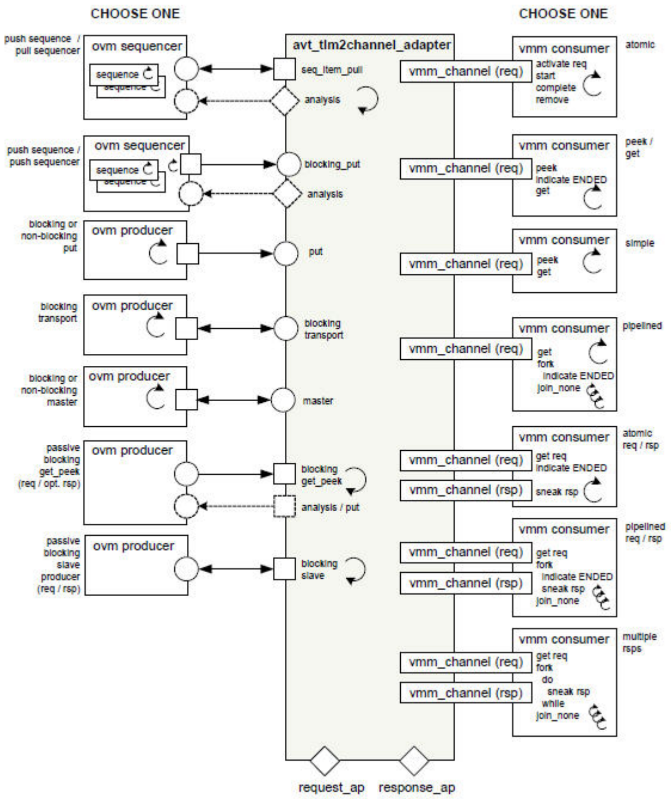

Figure 2. avt_tlm2channel adapter

On the VMM side, as with most VMM transactors, the adapter contains an internal `vmm_channel` for requests and another for responses. For protocols where the consumer will annotate the request with response information, only the request channel needs to be connected, but the `req_is_rsp` bit of the adapter must be set. If a `vmm_channel` is passed in as a constructor argument, then that channel is used, otherwise a new channel is allocated internally by the adapter. It is up to the user to ensure that the semantics required by the OVM producer are provided by the attached VMM consumer.

The `avt_channel2tlm` enables communication between a VMM producer and an OVM consumer. As with all OVM components, the OVM consumer's port or export is connected to one of the exports and ports provided by the adapter. If the consumer expects sequence_items, for example, we connect the consumer's `seq_item_pull_port` to the adapter's `seq_item_pull_export`. As with most VMM components, the VMM producer connects to the adapter via a shared `vmm_channel`, a handle to which is supplied as an argument to the producer's and/or adapter's constructor. Again, it is up to the user to ensure that the expected semantics of the producer match the semantics defined by the OVM ports/exports of the consumer.

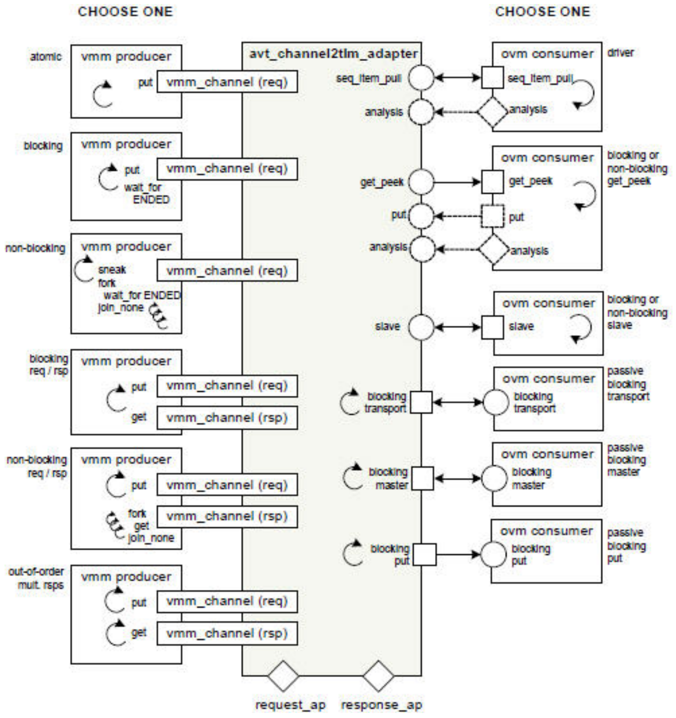

Figure 3. avt_channel2tlm adapter

The interoperability library also provides an `avt_analysis_channel` adapter that contains a `vmm_channel`, `ovm_analysis_port` and `ovm_analysis_export`. To connect a VMM producer to one or more OVM subscriber consumers, you connect the adapter's `ovm_analysis_port` to each of the subscriber's `ovm_analysis_exports`. To connect an OVM producer (e.g. a monitor) to one or more VMM consumers, you create an adapter instance for each VMM consumer, then connect the OVM component's `ovm_analysis_port` to each adapter's `ovm_analysis_export`. In all cases, you must also ensure that each adapter instance shares a common `vmm_channel` instance with their associated VMM component.

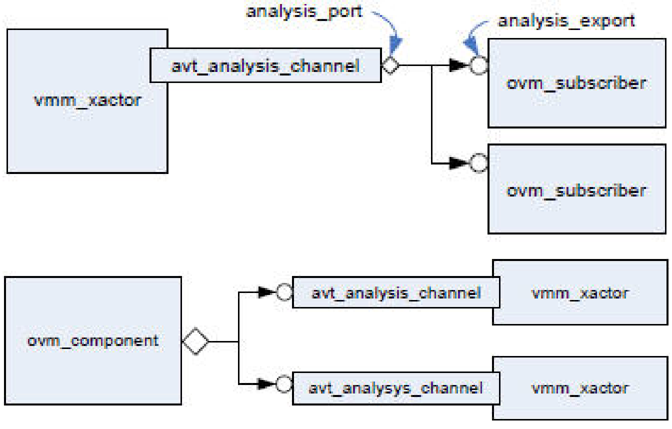

Figure 4. avt_analysis_channel adapter usages

2.3.2. Datatype Conversion

The adapters facilitate communication between OVM and VMM components by mapping the interface of OVM's TLM ports and exports to VMM's `vmm_channel`. Since OVM and VMM use different class types for the transaction data being communicated, it is also necessary to convert between them as the data is transferred from one methodology to the other (and back). The conversion is done by the user defining a unidirectional converter class as follows:

```
class apb_rw_ovm2vmm;
  static function
         vmm_apb_rw convert(ovm_apb_rw from,
                            vmm_apb_rw to=null);
    if (to == null)
      convert = new;
    else
      convert = to;

    case (from.cmd)
      ovm_apb_rw::RD :
        convert.kind = vmm_apb_rw::READ;

      ovm_apb_rw::WR :
        convert.kind = vmm_apb_rw::WRITE;
    endcase

    convert.addr = from.addr;
    convert.data = from.data;
    convert.data_id = from.get_transaction_id();
    convert.scenario_id = from.get_sequence_id();
  endfunction

endclass
```

We use a static `convert()` function to allow it to be called by the adapters without having to instantiate the converter class itself.

Each adapter in the interoperability kit is parameterized to the data types and converter types needed to get an OVM and VMM component talking to each other. Users specify the actual types when instantiating the adapter.

```
class avt_analysis_channel#(
         type OVM=int,
              VMM=int,
              OVM2VMM=avt_converter #(OVM,VMM),
              VMM2OVM=avt_converter #(VMM,OVM))
      extends ovm_component;
  ...
  function void write(OVM ovm_t);
    VMM vmm_t;
    if (ovm_t == null)
      return;
    vmm_t = OVM2VMM::convert(ovm_t);
    chan.sneak(vmm_t);
  endfunction


endclass

class ovm_producer; ... endclass

class vmm_consumer; ... endclass

class ovm_env extends ovm_component;

  ovm_producer producer;
  ovm_consumer consumer;
  avt_analysis_channel #(ovm_apb_rw,vmm_apb_rw,
       apb_rw_ovm2vmm, apb_rw_vmm2ovm) adapter;

  virtual function void build();
    producer = ovm_producer::type_id::create
                               (“producer”,this);
    vmm_consumer consumer = new(...);
    adapter = new(“adapter”,null,consumer.out_chan);
  endfunction

  virtual function void connect();
    producer.analysis_port.connect(
                        adapter.analysis_export);
  endfunction

endclass
```

3. REUSING VMM_ENV IN OVM

Having discussed the challenges we faced during development of the OVM-VMM interoperability library, we will explore applications of the library not found in the kit provided by Accellera.

In VMM, there is a single `vmm_env` instance that serves as the top-level component. The following example demonstrates simple instantiation of a VMM env within an OVM component, which allows the env to be reused anywhere within the OVM hierarchy.

Figure 5. Encapsulating a VMM env in an OVM component

```
class user_ovm_component extends ovm_component;

  ...

  avt_ovm_vmm_env #(user_vmm_env) env;

  function void build();
    env = new("user_vmm_env",this);
    env.auto_stop_request = 1;
  endfunction

endclass
```

Here, we've contained the `user_vmm_env` in an OVM component using the interoperability library's `avt_ovm_vmm_env` adapter, which allows us to integrate VMM envs such that their phases are synchronized with those of other OVM components in the testbench. From the standpoint of the VMM env itself, it is phased just as it would in a native VMM testbench. The only difference is that we call `run_test()` to kick off simulation rather than `my_vmm_env.run()`.

In the next example, we show a VMM env that has been more fully integrated in an OVM environment.

4. VMM ENV AS OVM COMPONENT

In this example, we define an extension to the `avt_ovm_vmm_env` adapter, which enables us to more fully integrate the VMM env it contains. VMM envs, when fully integrated as OVM components, can reside deep in the component hierarchy as a mere sub-components of a much larger OVM environment. In fact, any number of VMM envs can be instantiated in an OVM testbench using this technique. The VMM envs look and behave like OVM components, which frees the environment designer and verification engineer from having to learn more than one methodology.

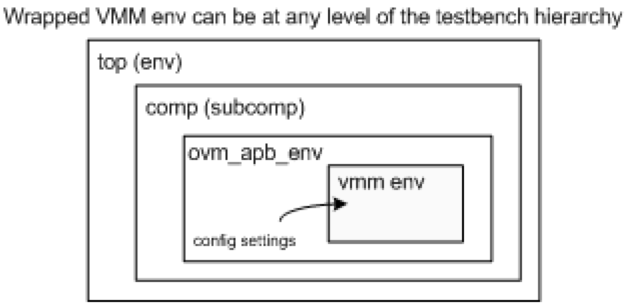

Figure 6. A more fully integrated VMM env in an OVM environment

As stated previously, the `vmm_gen_cfg()` and `build()` methods in the `avt_ovm_vmm_env` adapter call the underlying VMM env's `gen_cfg()` and `build()` methods. Users can derive extensions of `avt_ovm_vmm_env` and override either of these methods to perform actions both before and after calling `super.vmm_gen_cfg()` and/or `super.build()`. For example, in `vmm_gen_cfg()`, we can call `super.vmm_gen_cfg()` to generate the underlying VMM env's configuration object, and then modify the configuration based on settings retrieved from OVM's configuration mechanism. In `build()`, we could call `super.build()` to construct the underlying VMM env, and then create and connect OVM components to some embedded vmm_xactors using the appropriate interoperability adapters.

Before we define our custom VMM env wrapper, we define a simple container class for delivering `vmm_data`-based objects via OVM configuration mechanism.

```
class vmm_data_wrap #(type T=vmm_data)
                          extends ovm_object;
  typedef vmm_data_wrap #(T) this_type;
  `ovm_object_param_utils(this_type)
  T obj;
endclass
```

Next, we define `ovm_apb_env` as an extension of the `avt_ovm_vmm_env` adapter.

```
class ovm_apb_env
         extends avt_ovm_vmm_env #(vmm_apb_env);

  `ovm_component_utils(ovm_apb_env)

  ovm_analysis_port #(vmm_apb_rw) ap;

  function new (string name="ovm_apb_env",
                ovm_component parent=null);
    super.new(name,parent);
    ap = new("analysis_port",this);
    // stop_request when wait_for_end returns
    auto_stop_request = 1;
  endfunction



  virtual function void vmm_gen_cfg();
    // do stuff before generating config here
    super.vmm_gen_cfg();
    // do post config generation here
  endfunction


  virtual function void build();
    ovm_object obj;
    vmm_data_wrap #(vmm_apb_rw) prototype;
    super.build(); // build VMM env

    // configure env's xactors post-build

    void'(get_config_int("num_trans",
               env.gen.stop_after_n_insts));

    if (get_config_object("prototype",obj,0) &&
             $cast(prototype,obj))
      env.gen.randomized_obj = prototype.obj;
    else
     `ovm_error(...)
  endfunction
endclass
```

The `build()` method builds our wrapped VMM env. Because the underlying VMM env's `gen_cfg()` has been called by now, we can modify the VMM env's config object before calling `super.build()`. After calling `super.build()`, we can modify other aspects that depend on the env being built, such as `gen.stop_after_n_insts` in this example.

Now that we've encapsulated the VMM env in an OVM component wrapper, we can now integrate it into an OVM environment as any other OVM component. Below, we define a basic OVM testbench where the VMM env is not a top-level component but a *grandchild* of the overall OVM environment.

```
class subcomp extends ovm_component;
  `ovm_component_utils(subcomp)
  ovm_apb_env apb_env;

  function new (string name="subcomp",
                ovm_component parent=null);
    super.new(name,parent);
  endfunction

  virtual function void build();
      apb_env = new("apb_env",this);
  endfunction
endclass

class env extends ovm_component;
  `ovm_component_utils(env)
  subcomp comp;

  function new (string name="env",
                ovm_component parent=null);
    super.new(name,parent);
  endfunction

  virtual function void build();
    comp = new("comp",this);
  endfunction
endclass

module example_09_subenv;
  `include "vmm_apb_env.sv" // the VMM env
  env top = new("top");
  vmm_data_wrap #(vmm_apb_rw) apb_ext = new;
  vmm_apb_rw_extend      my_prototype = new;

  initial begin
    apb_ext.obj = my_prototype;

    // set number of transaction to 5
    set_config_int("top.comp.apb_env",
                   "num_trans",5);

    // set the type of transactions to produce
    // to a special extension of the apb_rw.
    set_config_object("top.comp.apb_env",
                   "prototype_obj",apb_ext,0);
```

```
    run_test();
  end
endmodule
```

5. VMM SCENARIOS AS OVM SEQUENCES

This example uses an `ovm_scenario2sequence` adapter (see [4]) to wrap an instance of a vmm_scenario. The adapter allows you to run scenarios alongside OVM sequences and have the OVM sequencer manage the arbitration among them all.

The adapter contains a `vmm_scenario` and a `vmm_channel` into which the `vmm_scenario` puts transactions. A background process continually gets transactions from this channel, converts them to the corresponding OVM transaction type, and then presents them to the sequencer for execution as any OVM sequence would do.

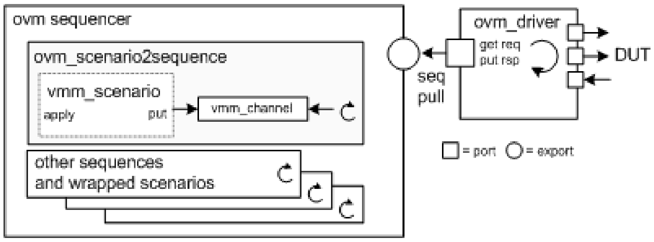

Figure 7. Encapsulating a VMM scenario as an OVM sequence

In the following example, note how you may choose to randomize the embedded scenario with in-line constraints before starting the sequence.

```
// typedef a scenario wrapper class for apb
typedef ovm_scenario2sequence
  #(vmm_apb_rw_scen, // the VMM scenario
   ovm_apb_rw,apb_rw,
   apb_rw_convert_ovm2vmm,
   apb_rw_convert_vmm2ovm) vmm_apb_rw_scen_seq;

class env extends ovm_component;

  `ovm_component_utils(env)

  ovm_sequencer #(ovm_apb_item) seqr;
  ovm_driver_req drv;

  function new (string name="env",
                ovm_component parent=null);
    super.new(name,parent);
  endfunction

  virtual function void build();
    seqr = new("o_seq", this);
    drv  = new("o_drv", this);
  endfunction

  virtual function void connect();
    drv.seq_item_port.connect
             (seqr.seq_item_export);
  endfunction

  virtual task run();

    // create 3 scenarios wrapped in sequences
    vmm_apb_rw_scen_seq seq1 = new("seq1");
    vmm_apb_rw_scen_seq seq2 = new("seq2");
    vmm_apb_rw_scen_seq seq3 = new("seq3");

    // randomize them as needed
    seq1.randomize() with
        { seq1.scenario.addr == 'h111;
          seq1.scenario.length == 9; };
    seq2.randomize() with
        { seq2.scenario.addr == 'h222;
          seq2.scenario.length == 7; };
    seq3.randomize() with
        { seq3.scenario.addr == 'h333;
          seq3.scenario.length == 5; };

    // start them up concurrently (in this case)
    fork
      seq1.start(seqr);
      seq2.start(seqr);
      seq3.start(seqr);
    join

    // we’re done, so stop the run phase
    ovm_top.stop_request();

  endtask

endclass

module example_08_scenario2sequence;
  env e = new;
  initial run_test();
endmodule
```

6. VMM MULTI-STREAM SCENARIOS AS OVM SEQUENCES

This example uses the `ovm_ms_scenario2sequence` adapter to encapsulate a VMM multi-stream scenario. Although this example does not drive multiple sequencers or channels, it is a simple matter of programming.

Multi-stream scenarios operate differently from their single-stream counterparts and are more difficult to integrate as an OVM sequence.

Single-stream scenarios are not dependent on the generator that selects them for execution. Nor are they required to fetch a channel it will use from an external object; the channel handle is passed as an argument to the `apply` method.

Multi-stream scenarios and the channels they put or sneak into must be pre-allocated and pre-registered with a multi-stream scenario generator before they can be used. Then, in the scenario's `execute()` method, the channel handle is retrieved from the associated ms generator by name (string) lookup.

The following defines an OVM parent sequence that concurrently executes an OVM child sequence and wrapped VMM scenario:

```
// typedef a ms scenario wrapper class for apb
typedef ovm_ms_scenario2sequence
 #(vmm_apb_rw_ms_scen, // the VMM ms scenario
 ovm_apb_rw, apb_rw,
 apb_rw_convert_ovm2vmm,
 apb_rw_convert_vmm2ovm) vmm_apb_rw_ms_scen_seq;


class my_sequence extends
                  ovm_sequence #(ovm_apb_rw);

  `ovm_object_utils(my_sequence)

  function new(string name="my_sequence");
    super.new(name);
  endfunction

  vmm_apb_rw_ms_scen_seq vmm_ms_seq;
```

```
  virtual task body();

    ovm_apb_rw_seq ovm_seq = new("ovm_seq");

    // concurrently execute OVM sequence
    // and VMM scenario-sequence
    ovm_seq.randomize() with { addr == 3; };
    vmm_ms_seq.randomize() with {
                    scenario.addr == 256; };

    fork
     ovm_seq.start(this.m_sequencer, this);
     vmm_seq.start(this.m_sequencer, this);
    join

  endfunction
endclass
```

Note that the `` `ovm_do_* `` macros, which embed synchronization and allocation, can not be used for multi-stream scenarios.

To run a multi-stream scenario as a sequence, we first allocate the scenario, scenario adapter, and multi-stream scenario generator in the `build()` method. Then, we register the scenario and channel(s) it uses with the multi-stream scenario generator.

```
class env extends ovm_component;

  `ovm_component_utils(env)

  ovm_sequencer #(ovm_apb_item) sequencer;
  ovm_driver_req driver;
  vmm_ms_scenario_gen vmm_scen_gen;

  my_sequence vseq;

  function new (string name="my_env",
                ovm_component parent=null);
   super.new(name,parent);
  endfunction

  virtual function void build();

   sequencer = new("OVM_Sequencer", this);
   driver    = new("OVM_Driver", this);
   vmm_ms_scenario_gen vmm_scen_gen= new(“gen“);

   // ms scenarios must be pre-allocated
   // and registered with its ms_scenario_gen
   vmm_apb_rw_ms_scen_seq vmm_ms_seq=new(“seq”);


   vmm_scen_gen.register_ms_scenario
                 ("vmm_seq",vmm_seq.scenario);


   // register the channel so the VMM scenario
   // can get a reference to it via get_channel
   vmm_scen_gen.register_channel("apb_rw_chan",
                    vmm_seq.chan);

   // create sequence using factory
   vseq = my_sequence::type_id::create
                   ("my_sequence",this);

   vseq.vmm_seq = vmm_seq;

  endfunction

  virtual function void connect();
     driver.seq_item_port.connect
            (sequencer.seq_item_export);
  endfunction

  virtual task run();
    vseq.start(sequencer);
    ovm_top.stop_request();
   endtask

endclass

module example_09_ms_scenario2sequence;
  env e = new;
  initial run_test();
endmodule
```

7. CONCLUSION

This paper provided insight into the challenges we faced while developing the interoperability library and detailed information on the various adapters that are available to interconnect OVM and VMM components. The Accellera VIP-TSC based its work on the assumption that the engineer who is actually doing the work of integrating OVM and VMM IP must know enough about both methodologies in order to apply the interoperability library effectively. This paper provided several advanced applications of the library to assist integrators in gaining this knowledge: encapsulating VMM envs with OVM component wrappers, thereby allowing them to be integrated and reused as any other component in an OVM environment, and adapting VMM scenarios to run as and alongside other OVM sequences, thereby enhancing reuse of existing VMM-based stimulus generation.

8. REFERENCES

[1] Verification Intellectual Property (VIP) Recommended Practices, Version 1.0, August 25, 2009. Accellera; See http://sourceforge.net/projects/acc-vip-iop.

[2] Bergeron, Janick, Cerny, Eduard, Hunter, Alan, Nightingale, Andrew, *Verification Methodology Manual for SystemVerilog*, Springer, 2006

[3] Glasser, Mark, Open Verification Methodology Cookbook, Springer, 2009

[4] OVM World, OVM’s online community; http://www.ovmworld.org

[5] VMM Central, VMM’s online community; http://vmmcentral.com