Getting Started with OVM 2.0

Doulos, October 2008

Tutorial 1 – A First Example

A Series of Tutorials based on a set of Simple, Complete Examples

Introduction

In this tutorial, the emphasis is on getting a simple example working rather than on understanding the broad flow of the constrained random verification process and verification component reuse.

Compiling OVM Code

The two most important top level directories in the OVM release are **./src** and **./examples**, which contain the source code for the OVM library and a set of examples, respectively. The source code is structured into subdirectories, but you can ignore this for now.

In order to compile OVM applications, different approaches are recommended depending on your choice of simulator. For Cadence Incisive, the best approach is:

- to add **./src** to the include path on the command line, that is, **+incdir+/.../src**
- to add the following directive at the top of your various SystemVerilog files

```
`include "ovm.svh"
```

For Mentor Graphics QuestaSim, the best approach is:

- to add **./src** to the include path on the command line, that is, **+incdir+/.../src**
- to add **./src/ovm_pkg.sv** to the front of the list of files being compiled
- to add the following directives to your various SystemVerilog files

```
`include "ovm_macros.svh"
import ovm_pkg::*;
```

Make sure that you do *not* put **ovm_pkg.sv** on the command line if you include **"ovm.svh"** in the source files.

The following conditional code serves to make code portable between both simulators at the time of writing:

```
`ifdef INCA
  `include "ovm.svh"
`else
  `include "ovm_macros.svh"
  import ovm_pkg::*;
`endif
```

The **./examples** directory in the OVM release contains sample script files for both simulators that can be modified to compile this tutorial example.

Getting Started with OVM 2.0

Doulos, October 2008

Tutorial 1 – A First Example

The Verification Environment

This tutorial is based around a very simple example including a design-under-test, a verification environment (or test bench), and a test. Assuming you have written test benches in VHDL or Verilog, the structure should be reasonably obvious. The SystemVerilog code is structured as follows:

```
- Interface to the design-under-test

- Design-under-test (or DUT)

- Verification environment (or test bench)
  - Transaction
  - Sequencer (stimulus generator)
  - Driver
  - Top-level of verification environment
    - Instantiation of sequencer
    - Instantiation of driver

- Top-level module
  - Instantiation of interface
  - Instantiation of design-under-test
  - Test, which instantiates the verification environment
  - Process to run the test
```

Since this example is intended to get you started, all the code is placed in a single file. In practice, of course, the code would be spread across multiple files. Also, some pieces of the jigsaw are missing, most notably a verification component to perform checking and collect functional coverage information. It should be emphasised that the purpose of this tutorial is not to demonstrate the full power of OVM, but just to get you up-and-running.

Classes and Modules

In traditional Verilog code, modules are the basic building block used to structure designs and test benches. Modules are still important in SystemVerilog and are the main language construct used to structure RTL code, but classes are also important, particularly for building flexible and reusable verification environments and tests. Classes are best placed in packages, because packages enable re-use and also give control over the namespaces of the SystemVerilog program.

The example for this tutorial includes a verification environment consisting of a set of classes, most of which are placed textually within a package, a module representing the design-under-test, and a single top-level module coupling the two together. The actual link between the verification environment and the design-under-test is a SystemVerilog interface.

Hooking up the DUT

The SystemVerilog interface encapsulates the pin-level connections to the DUT.

```
interface dut_if();

  int addr;
  int data;
  bit r0w1;

  modport test (output addr, data, r0w1);
  modport dut  (input  addr, data, r0w1);

endinterface: dut_if
```

Of course, a real design would have several far more complex interfaces, but the same principle holds. Having written out all the connections to the DUT within the interface, the actual code for the outer layer of the DUT module becomes trivial:

```
module dut(dut_if.dut i_f);
  ...
endmodule: dut
```

As well us removing the need for lots of repetitive typing, interfaces are important because they provide the mechanism for hooking up a verification environment based on classes. In order to mix modules and classes, a module may instantiate a variable of class type, and the class object may then use hierarchical names to reference other variables in the module. In particular, a class may declare a *virtual interface*, and use a hierarchical name to assign the virtual interface to refer to the actual interface. In effect, a virtual interface is just a reference to an interface. The overall structure of the code is as follows:

```
package my_pkg;
  ...
  class my_driver extends ovm_driver;
    ...
    virtual dut_if m_dut_if;
    ...
  endclass

  class my_env extends ovm_env;
    my_driver m_driver;
    ...
  endclass
  ...
endpackage

module top;
  import my_pkg::*;

  dut_if dut_if1 ();
  dut    dut1 ( .i_f(dut_if1) );

  class my_test extends ovm_test;
    ...
    my_env m_env;
    ...
    virtual function void connect;
      m_env.m_driver.m_dut_if = dut_if1;
    endfunction: connect
    ...
  endclass: my_test
  ...
endmodule: top
```

If you study the code above, you will see that the connect method of the class **my_test** uses a hierarchical name to assign **dut_if1**, the actual DUT interface, to the virtual interface buried within the object hierarchy of the verification environment. In practice, the verification environment would consist of many classes scattered across many packages from multiple sources. The behavioral code within the verification environment can now access the pins of the DUT using a single virtual interface. The single line of code assigning the virtual interface is the only explicit dependency necessary between the verification environment and the DUT interface. In other words, the verification environment does not directly refer to the pins on the DUT, but only to the pins of the virtual interface.

Transactions

The verification environment consists of two verification components, a sequencer and a driver, and a third class representing the transaction passed between them. The sequencer creates random transactions, which are retrieved by the driver and used to stimulate the pins of the DUT. A transaction is just a collection of related data items that get passed around the verification environment as a single unit. You would create a user-defined transaction class for each meaningful collection of data items to be passed around your verification environment. Transaction classes are very often associated with busses and protocols used to communicate with the DUT.

In this example, the transaction class mirrors the trivial structure of the DUT interface:

```
class my_transaction extends ovm_sequence_item;

  rand int addr;
  rand int data;
  rand bit r0w1;

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

  constraint c_addr { addr >= 0; addr < 256; }
  constraint c_data { data >= 0; data < 256; }

  `ovm_object_utils_begin(my_transaction)
    `ovm_field_int(addr, OVM_ALL_ON + OVM_DEC)
    `ovm_field_int(data, OVM_ALL_ON + OVM_DEC)
    `ovm_field_int(r0w1, OVM_ALL_ON + OVM_BIN)
  `ovm_object_utils_end

endclass: my_transaction
```

The address, data and command (**r0w1**) fields get randomised as new transactions are created, using the constraints that are built into the transaction class. In OVM, all transactions, including sequence items, are derived from the class **ovm_transaction**, which provides some hidden machinery for transaction recording and for manipulating the contents of the transaction. But because the example uses a sequencer, the transaction class must be derived from the **ovm_sequence_item** class, which is a subclass of **ovm_transaction**. The constructor **new** is passed a string that is used to build a unique instance name for the transaction.

As transaction objects are passed around the verification environment, they may need to be copied, compared, printed, packed and unpacked. The methods necessary to do these things are created automatically by the **ovm_object_utils** and **ovm_field** macros. At first, it may seem like an imposition to be required to include macros repeating the names of all of the fields in the transaction, but it turns out that these macros provide a significant convenience because the of the high degree of automation they enable.

The flag OVM_ALL_ON indicates that the given field should be copied, printed, included in any comparison for equality between two transactions, and so on. The flags OVM_DEC and OVM_BIN indicate the radix of the field to be used when printing the given field.

Verification Components

In OVM, a verification component is a SystemVerilog object of a class derived from the base class **ovm_component**. Verification component instances form a hierarchy, where the top-level component or components in the hierarchy are derived from the class **ovm_env**. Objects of type **ovm_env** may themselves be instantiated as verification components within other **ovm_env**s. You can instantiate **ovm_env**s and **ovm_component**s from other **ovm_env**s and **ovm_component**s, but the top-level component in the hierarchy should always be an **ovm_env**.

A verification component may be provided with the means to communicate with the rest of the verification environment, and may implement a set of standard methods that implement the various phases of elaboration and simulation. One such verification component is the driver, which is described here line-by-line:

```
class my_driver extends ovm_driver #(my_transaction);
```

ovm_driver is derived from **ovm_component**, and is the base class to be used for user-defined driver components. There is a number of such *methodology base classes* derived from **ovm_component**, each of which has a name suggestive of its role. Some of these classes add very little functionality of their own, so it is also possible to derive the user-defined class directly from **ovm_component**.

ovm_driver is a parameterized class – so either you must specialize it when creating your driver class, or the user-defined class must also be parameterized. In this example we have taken the former approach and specialized for the user-defined transaction class.

```
`ovm_component_utils(my_driver)
```

The **ovm_component_utils** macro provides factory automation for the driver. The factory will be described below, but this macro plays a similar role to the **ovm_object_utils** macro we saw above for the transaction class. The important point to remember is to invoke this macro from every single verification component; otherwise bad things happen.

```
virtual dut_if m_dut_if;
```

The virtual interface is the means by which the driver communicates with the pins of the DUT, as described above.

```
function new(string name, ovm_component parent);
  super.new(name, parent);
endfunction: new
```

The constructor for an **ovm_component** takes two arguments, a string used to build the unique hierarchical instance name of the component and a reference to the parent component in the hierarchy. Both arguments should always be set correctly, and the user-defined constructor should always pass its arguments to the constructor of the superclass, **super.new**.

```
virtual task run;
  forever
  begin
    my_transaction tx;
    #10
    seq_item_port.get_next_item(tx);
    ovm_report_info("",$psprintf("Driving cmd = %s, addr = %d, data = %d}",
                                (tx.r0w1 ? "W" : "R"), tx.addr, tx.data));
    m_dut_if.r0w1 = tx.r0w1;
    m_dut_if.addr = tx.addr;
    m_dut_if.data = tx.data;
  end
endtask: run

endclass: my_driver
```

The **run** method is one of the standard hooks called back in each of the phases of elaboration and simulation. It contains the main behavior of the component to be executed during simulation. This **run** method contains an infinite loop to wait for some time, get the next transaction from the **seq_item_port**, then wiggle the pins of the DUT through the virtual interface mentioned above.

The **seq_item_port** is declared in the **ovm_driver** class itself – it is available through inheritance and its type is **ovm_seq_item_pull_port**. The driver uses the **seq_item_port** to communicate with the sequencer, which generates the transactions. It calls the **get_next_item** method through this port to fetch transactions of type **my_transaction** from the sequencer.

People sometimes express discomfort that this loop appears to run forever. What stops simulation? There are two aspects to the answer. Firstly, **get_next_item** is a *blocking* method. The call to **get_next_item** will not return until the next transaction is available. When there are no more transactions available, **get_next_item** will not return, and simulation is able to stop due to event starvation. Secondly, it is possible to set a global watchdog timer such that every run method will eventually return, even if it is starved of transactions. The timer is set by calling the method **set_global_timeout**.

The **run** method also makes a call to **ovm_report_info** to print out a report. This is a method of the report handling system, which provides a standard way of logging messages during simulation. The first argument to **ovm_report_info** is a message type, and the second argument the text of the message. Report handling can be customised based on the message type or severity, that is, information, warning, error or fatal. **ovm_report_info** generates a report with severity OVM_INFO.

The Sequencer

The sequencer's role is to generate a stream of transactions for the driver to use.

```
class my_sequencer extends ovm_sequencer #(my_transaction);
```

A sequencer is derived from **ovm_sequencer** and specialized for the transaction type, which in our example is **my_transaction**.

```
  `ovm_sequencer_utils(my_sequencer)
```

The **ovm_sequencer_utils** macro provides factory automation for the sequencer.

```
  function new (string name = "my_sequencer", ovm_component parent);
    super.new(name, parent);
    `ovm_update_sequence_lib_and_item(my_transaction)
  endfunction : new

endclass : my_sequencer
```

The constructor for an **ovm_sequencer** takes the same arguments as an **ovm_driver**'s constructor. A sequencer has a library of sequences, and the constructor must build a default sequence library. The **ovm_update_sequence_lib_and_item** macro does this. It also tells the sequencer to generate transactions of the **my_transaction** type.

Connecting Up The Environment

The driver and sequencer are specific verification components instantiated within the verification environment. Now we will look at how the environment in constructed from these and other components.

Tutorial 1 – A First Example

Again, we will describe the code line-by-line.

```
class my_env extends ovm_env;

`ovm_component_utils(my_env)
```

As mentioned above, **ovm_env** is the base class from which all user-defined verification environments are created. In a sense, an **ovm_env** is just another hierarchical building block, but one that may be instantiated as a top-level verification component.

```
my_sequencer m_sequencer;
my_driver    m_driver;
```

The environment contains the sequencer (stimulus generator) and driver. The default behavior of a sequencer is to generate a sequence of up to 10 randomized transactions.

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

The constructor should look familiar, but notice that in this case the parent argument may be **null** because the **ovm_env** could be a top-level component.

```
virtual function void build;
  super.build();

  m_sequencer = my_sequencer::type_id::create("m_sequencer", this);
  m_driver    = my_driver::type_id::create("m_driver", this);

endfunction: build
```

The **build** method creates instances of the **my_driver** and **my_sequencer** components, but *not* by calling their constructors. Instead, they are instantiated using the *factory*, which performs polymorphic object creation. In other words, the actual type of the object being created by the factory can be set at run time such that the lines of code shown above will not *necessarily* create components of type **my_driver** or **my_sequencer**. It is possible to override the type of component being created from the test in order to replace **my_driver** with a modified driver and **my_sequencer** with a modified sequencer. The factory is one of the most important mechanisms in OVM, permitting the creating of flexible and reusable verification components. This flexibility is not available when directly calling the constructor **new**, which always creates an object of a given type as determined at compile time.

The way in which the factory creates an instance is quite complicated. All you really need to know for now is what to write to create an instance, which involves calling a **create** function, as shown here. The arguments passed to the method **create** are: one, the local instance name of the component being instantiated; and two, a reference to the parent component, which in this case is the environment.

At this stage, the verification components have been created, but not connected together.

```
virtual function void connect;
  m_driver.seq_item_port.connect(m_sequencer.seq_item_export);
endfunction: connect
```

The **connect** method of **ovm_component** is the callback hook for another of the standard phases, and is used to connect ports to exports. Here, we are connecting the driver's seq_item_port to the sequencer's seq_item_export. The **connect** method is called *after* the **build** method, so you know that the components being connected will already have been instantiated, whether by **new** or by the factory. Notice that the method used to make the connections is also named connect.

```
task run();
  ovm_report_info("","Called my_env::run");
endtask: run

virtual function void report;
  ovm_report_info("", "Called my_env::report");
endfunction: report

endclass: my_env
```

The **run** and **report** callback hooks are also included for completeness, although here they do nothing but print out a message showing they have been called. **report** is called near the end of simulation, when the run phase is complete

Elaboration and Simulation Phases

Now we can summarise the standard elaboration and simulation phases. The following callback hooks are called in the order shown for classes derived from **ovm_component**:

1.	new()	The constructor
2.	build()	Create components using **new** or the factory
3.	connect()	Make port, export and implementation connections
4.	end_of_elaboration()	After all connections have been hardened
5.	start_of_simulation()	Just before simulation starts
6.	run()	Runs simulation
7.	extract()	Post-processing 1
8.	check()	Post-processing 2
9.	report()	Post-processing 3

Getting Started with OVM 2.0

Tutorial 1 – A First Example

The Test Class

We have looked at the verification environment, which generates a series of random transactions and drives them into the DUT, and at hooking up the verification environment to the DUT. We have also emphasised that a real verification environment would also have to deal with checking and coverage collection across multiple, reusable verification components. Now we look at the test, which configures the verification environment to apply a specific stimulus to the DUT. A practical verification environment must allow you to choose between multiple tests, which must be selectable and runnable with minimal overhead.

The test is represented by a class derived from the methodology base class **ovm_test**, and is described line-by-line below:

```
class my_test extends ovm_test;

  `ovm_component_utils(my_test)

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

The **ovm_test** class does not actually provide any functionality over-and-above an **ovm_component**, but the idea is that you use it as the base class for all user-defined tests.

```
my_env m_env;

virtual function void build;
  super.build();

  m_env = my_env::type_id::create("m_env", this);
  set_global_timeout(1us);
endfunction: build
```

The test instantiates the verification environment as a local component using the factory. It is reasonable that the test should depend on the environment, since each test will usually configure the environment to its own specific needs. The factory is used for consistency, although the flexibility provided by the factory is not needed in this case: we could equally well have used **new** to instantiate the environment.

The method **set_global_timeout** sets the watchdog timer referred to above to ensure that every **run** method will eventually return, particularly those **run** methods that are waiting on events or empty queues.

```
virtual function void connect;
  m_env.m_driver.m_dut_if = dut_if1;
endfunction: connect
```

The **connect** method hooks up the DUT interface to the virtual interface in the driver. This was already discussed above. This one assignment makes the connection between the structural SystemVerilog code, that is, modules and interfaces, and the class-based code of the verification environment.

Getting Started with OVM 2.0

Doulos, October 2008

Tutorial 1 – A First Example

In this simple example, the test itself doesn't seem to do anything. However, it does instantiate the environment; the sequencer will generate a default sequence of transactions and the driver will apply these to the DUT.

Running the Test

We have got the verification environment connected to the DUT, and we have got a test which could be one of many. The test is actually started by calling the **run_test** method. The name of the test to be run may be passed as an argument to **run_test** or may be given on the command line. Once again, here is the outline of the top-level module:

```
module top;
  ...
  dut_if dut_if1 ();
  dut    dut1 ( .i_f(dut_if1) );

  class my_test extends ovm_test;
    ...
  endclass: my_test

  initial
    run_test();

endmodule: top
```

The simulator can now be started from the command line as follows:

```
Incisive>  irun -f irun.f +OVM_TESTNAME=my_test

QuestaSim> qverilog -f questa.f -R +OVM_TESTNAME=my_test
```

When the simulation runs, you should see the following:

- A message saying that my_test is being run
- Many of the callback hooks print out a message saying that they have been called
- The driver prints out a series of up to 10 messages showing that it is driving transactions into the DUT. Each transaction obeys the constraints set in my_transaction.
- The DUT prints out a series of up to 10 messages showing that it has received the corresponding commands

You will find the source code for this example in the file ovm_getting_started_1.sv which is availabile to download at the bottom of this page on the Doulos website
www.doulos.com/knowhow/sysverilog/ovm/tutorial_1/