UVM Test [uvm_test]

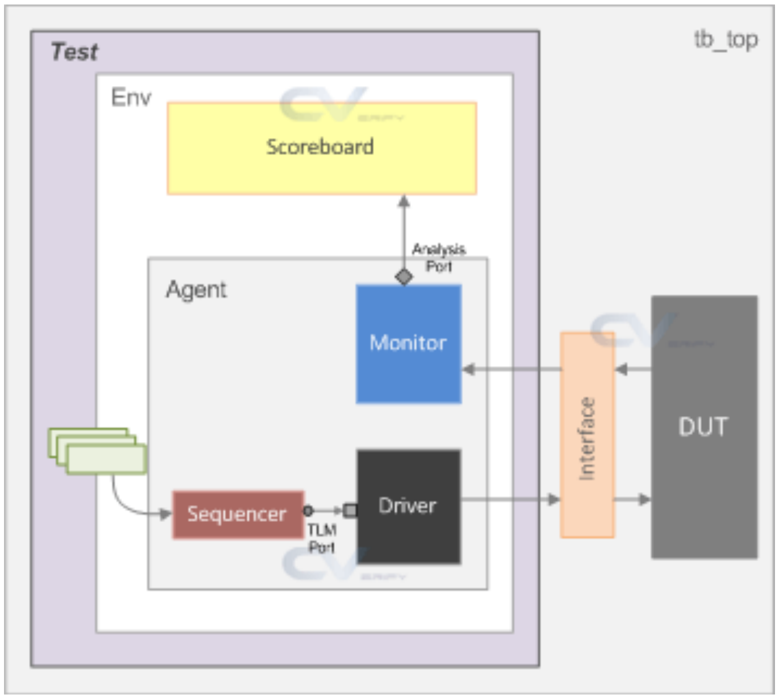
```
Build a derivative test that launches a different sequence -> only run_phase is
required but

want to run the previous derived sequence on different config of env. in this case
we have another derivative test of the previous class and define its build_phase in
a different way
```

What is a testcase ?

A **testcase** is a pattern to check and verify specific features and functionalities of a design. A verification plan lists all the features and other functional items that needs to be verified, and the tests neeeded to cover each of them.

A lot of different tests, hundreds or even more, are typically required to verify complex designs.

Instead of writing the same code for different testcases, we put the entire testbench into a container called an Environment, and use the same environment with a different configuration for each test. Each testcase can override, tweak knobs, enable/disable agents, change variable values in the configuration table and run different sequences on many sequencers in the verification environment.



Steps to write a UVM Test

1. Create a custom class inherited from `uvm_test`**, register it with factory and call function** `new`

```
// Step 1: Declare a new class that derives from "uvm_test"
// my_test is user-given name for this class that has been derived from "uvm_test"
class my_test extends uvm_test;

    // [Recommended] Makes this test more re-usable
    `uvm_component_utils (my_test)

    // This is standard code for all components
    function new (string name = "my_test", uvm_component parent = null);
      super.new (name, parent);
    endfunction

    // Code for rest of the steps come here
endclass
```

2. Declare other environments and verification components and build them

```
// Step 2: Declare other testbench components - my_env and my_cfg are assumed to be
defined
      my_env   m_top_env;              // Testbench environment that contains other
agents, register models, etc
      my_cfg   m_cfg0;                 // Configuration object to tweak the
environment for this test

      // Instantiate and build components declared above
      virtual function void build_phase (uvm_phase phase);
         super.build_phase (phase);

         // [Recommended] Instantiate components using "type_id::create()" method
instead of new()
         m_top_env  = my_env::type_id::create ("m_top_env", this);
         m_cfg0     = my_cfg::type_id::create ("m_cfg0", this);

         // [Optional] Configure testbench components if required, get virtual
interface handles, etc
         set_cfg_params ();

         // [Recommended] Make the cfg object available to all components in
environment/agent/etc
         uvm_config_db #(my_cfg) :: set (this, "m_top_env.my_agent", "m_cfg0",
m_cfg0);
      endfunction
```

3. Print UVM topology if required

```
// [Recommended] By this phase, the environment is all set up so its good to just
```

```
print the topology for debug
      virtual function void end_of_elaboration_phase (uvm_phase phase);
         uvm_top.print_topology ();
      endfunction
```

4. Start a virtual sequence

```
// Start a virtual sequence or a normal sequence for this particular test
virtual task run_phase (uvm_phase phase);

        // Create and instantiate the sequence
        my_seq m_seq = my_seq::type_id::create ("m_seq");

        // Raise objection - else this test will not consume simulation time*
        phase.raise_objection (this);

        // Start the sequence on a given sequencer
        m_seq.start (m_env.seqr);

        // Drop objection - else this test will not finish
        phase.drop_objection (this);
endtask
```

How to run a UVM test

A test is usually started within testbench top by a task called `run_test`.

This global task should be supplied with the name of user-defined UVM test that needs to be started. If the argument to `run_test` is blank, it is necessary to specify the testname via command-line options to the simulator using `+UVM_TESTNAME`.

```
// Specify the testname as an argument to the run_test () task
initial begin
   run_test ("base_test");
end
```

Definition for `run_test` is given below.

```
// This is a global task that gets the UVM root instance and
// starts the test using its name. This task is called in tb_top
task run_test (string test_name="");
  uvm_root top;
  uvm_coreservice_t cs;
  cs = uvm_coreservice_t::get();
  top = cs.get_root();
```

```
   top.run_test(test_name);
 endtask
```

How to run *any* UVM test

This method is preferred because it allows more flexibility to choose different tests without modifying testbench top every time you want to run a different test. It also avoids the need for recompilation since contents of the file is not updated.

If `+UVM_TESTNAME` is specified, the UVM factory creates a component of the given test type and starts its phase mechanism. If the specified test is not found or not created by the factory, then a fatal error occurs. If no test is specified via command-line and the argument to the `run_test()` task is blank, then all the components constructed before the call to *run_test()* will be cycled through their simulation phases.

```
// Pass the DEFAULT test to be run if nothing is provided through command-line
initial begin
   run_test ("base_test");
   // Or you can leave the argument as blank
   // run_test ();
end

// Command-line arguments for an EDA simulator
$> [simulator] -f list +UVM_TESTNAME=base_test
```

UVM Base Test Example

In the following example, a custom test called base_test that inherits from `uvm_test` is declared and registered with the factory.

Testbench environment component called m_top_env and its configuration object is created during the `build_phase` and setup according to the needs of the test. It is then placed into the configuration database using `uvm_config_db` so that other testbench components within this environment can access the object and configure sub components accordingly.

```
// Step 1: Declare a new class that derives from "uvm_test"
class base_test extends uvm_test;

          // Step 2: Register this class with UVM Factory
   `uvm_component_utils (base_test)

   // Step 3: Define the "new" function
   function new (string name, uvm_component parent = null);
      super.new (name, parent);
   endfunction
   // Step 4: Declare other testbench components
   my_env   m_top_env;              // Testbench environment
   my_cfg   m_cfg0;                 // Configuration object


   // Step 5: Instantiate and build components declared above
   virtual function void build_phase (uvm_phase phase);
      super.build_phase (phase);

      // [Recommended] Instantiate components using "type_id::create()" method
instead of new()
      m_top_env  = my_env::type_id::create ("m_top_env", this);
      m_cfg0     = my_cfg::type_id::create ("m_cfg0", this);

      // [Optional] Configure testbench components if required
      set_cfg_params ();

      // [Optional] Make the cfg object available to all components in
environment/agent/etc
      uvm_config_db #(my_cfg) :: set (this, "m_top_env.my_agent", "m_cfg0",
m_cfg0);
   endfunction

   // [Optional] Define testbench configuration parameters, if its applicable
   virtual function void set_cfg_params ();
      // Get DUT interface from top module into the cfg object
      if (! uvm_config_db #(virtual dut_if) :: get (this, "", "dut_if",
m_cfg0.vif)) begin
         `uvm_error (get_type_name (), "DUT Interface not found !")
      end

      // Assign other parameters to the configuration object that has to be used in
testbench
      m_cfg0.m_verbosity    = UVM_HIGH;
      m_cfg0.active         = UVM_ACTIVE;
   endfunction

         // [Recommended] By this phase, the environment is all set up so its good
to just print the topology for debug
   virtual function void end_of_elaboration_phase (uvm_phase phase);
      uvm_top.print_topology ();
   endfunction

   function void start_of_simulation_phase (uvm_phase phase);
```

```
      super.start_of_simulation_phase (phase);

      // [Optional] Assign a default sequence to be executed by the sequencer or 
look at the run_phase ...
      uvm_config_db#
(uvm_object_wrapper)::set(this,"m_top_env.my_agent.m_seqr0.main_phase",
                                        "default_sequence",
base_sequence::type_id::get());

   endfunction

   // or [Recommended] start a sequence for this particular test
   virtual task run_phase (uvm_phase phase);
        my_seq m_seq = my_seq::type_id::create ("m_seq");

        super.run_phase(phase);
        phase.raise_objection (this);
        m_seq.start (m_env.seqr);
        phase.drop_objection (this);
   endtask
endclass
```

The UVM topology task `print_topology` displays all instantiated components in the environment and helps in debug and to identify if any component got left out.

A test sequence object is built and started on the environment virtual sequencer using its start method.

Derivative Tests

A base test helps in the setup of all basic environment parameters and configurations that can be overridden by derivative tests. Since there is no definition for `build_phase` and other phases that are defined differently in dv_wr_rd_register, its object will inherently call its parent's `build_phase` and other phases because of inheritance. Function new is required in all cases and simulation will give a compilation error if its not found.

```
// Build a derivative test that launches a different sequence
// base_test <- dv_wr_rd_register_test
class dv_wr_rd_register_test extends base_test;
        `uvm_component_utils (dv_wr_rd_register_test)

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

        // Start a different sequence for this test
        virtual task run_phase(uvm_phase phase);
```

```
                wr_rd_reg_seq   m_wr_rd_reg_seq =
wr_rd_reg_seq::type_id::create("m_wr_rd_reg_seq");

                super.run_phase(phase);
                phase.raise_objection(this);
                m_wr_rd_reg_seq.start(m_env.seqr);
                phase.drop_objection(this);
        endtask
endclass
```

In this case, only `run_phase` will be overriden with new definition in derived test and its `super` call will invoke the `run_phase` of the base_test.

Assume that we now want to run the same sequence as in dv_wr_rd_register_test but instead want this test to be run on a different configuration of the environment. In this case, we can have another derivative test of the previous class and define its `build_phase` in a different way.

```
// Build a derivative test that builds a different configuration
// base_test <- dv_wr_rd_register_test <- dv_cfg1_wr_rd_register_test

class dv_cfg1_wr_rd_register_test extends dv_wr_rd_register_test;
        `uvm_component_utils (dv_cfg1_wr_rd_register_test)

        function new(string name = "dv_cfg1_wr_rd_register_test");
                super.new(name);
        endfunction

        // First calls base_test build_phase which sets m_cfg0.active to ACTIVE
        // and then here it reconfigures it to PASSIVE
        virtual function void build_phase(uvm_phase phase);
                super.build_phase(phase);
                m_cfg0.active = UVM_PASSIVE;
        endfunction
endclass
```