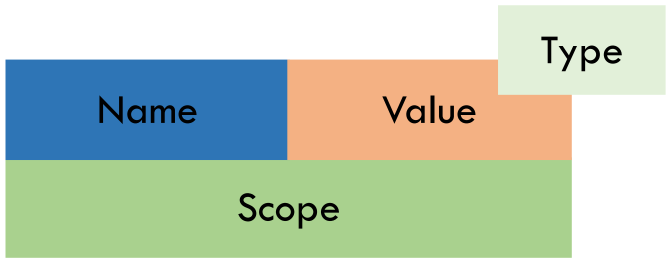
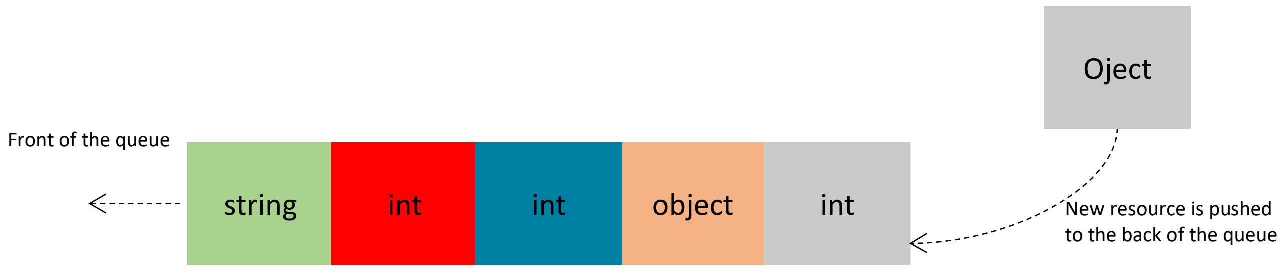
Lab 10: UVM

1. Configuration Database

A resource is a parameterized container that holds arbitrary data. Resources can be used to configure components, supply data to sequences, or enable sharing of information across disparate parts of the testbench. They are stored using scoping information so their visibility can be constrained to certain parts of the testbench. You can put any data type into the resource database, and any other component retrieves it later at some point in simulation, which is a very convenient feature to have. Such a configurable testbench provides a degree of freedom to the verification engineer to use provided information in various parts of the testbench.



Name	The "name" by which this resource is stored in the database. The same "name" has to be supplied to retrieve it later.
Value	The value that should be stored in the database for the given "name"
Scope	A regular expression that specifies the scope over which this resource is visible to other components in the testbench.
Type	The data type of the object that this resource contains. It can be a string, int, virtual interface class obiect, or any other valid Systemverdog data-type



The global resource database has both a name table and a type table into which each resource is entered. So, the same resource can be retrieved later by name or type. Multiple resources with the same name/type are stored in a queue and hence those which were pushed in earlier have more precedence over those placed later. In the image above, if a request to retrieve an item of type string is made, the queue is traversed from front to back, and first occurrence of an object of string type will be returned. Now, consider the case where items 2(red) and 3(blue) in the queue have the same scope, and a get by type(method is called for that particular scope. Then item 2(red) will be returned since that sits earlier in the queue.

UVM has an internal database table in which we can store values under a given name and can be retrieved later by some other testbench component. The uvm_config_db class provides a convenience interface on top of the uvm_resource_db to simplify the basic interface used for uvm component instances.

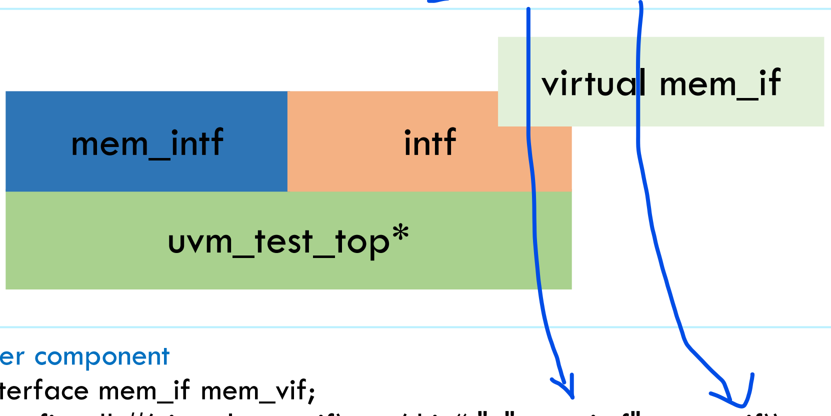
Note that all the functions are static and must be called using the scope operator ::

Such a configuration database allows us to store different configuration settings under different names that could potentially configure testbench components when required without modifying the actual testbench code.

Example(1): In the top testbench module we instantiate the virtual interface and store it in the Configuration database to allow other UVM components to retrieve it.

Set method: after creating an instance from the interface mem_if called it intf, we store it in the configuration database under “mem_intf” name and virtual mem_if type. The first argument in the set is the hierarchical starting point of where the database entry is accessible, and since we are in the top module so we put it “NULL”. The second argument is the hierarchical path that limits the accessibility of the database entry (Who can access the stored data), “ * ” means all components can access it.

```
//In top module
mem_if intf(clk,reset);  //interface instance
uvm_config_db#(virtual mem_if)::set(null,"*","mem_intf",intf);  //set method
```



```
//In driver component
virtual interface mem_if mem_vif;
if( !uvm_config_db#(virtual mem_if)::get(this,“","mem_intf", mem_vif))
     `uvm_fatal(get_full_name(),{"virtual interface must be set for:",".mem_vif"} );
```

If get() fails to find the “mem_intf” string field_name in the database, a fatal will be reported. Even though the fatal check is not mandatory, it is recommended to use it for debugging purposes.

Example(2): In the below example, the control bit is stored in the database which acts as an enable condition to create an object for my_component in the component_B. The new resource with the name "control" of type bit is added in the resource pool from the test case.

```
// In component_A:
uvm_config_db #(bit)::set(null, "*", "control", 1);
//where
//cntxt is the hierarchical starting point of where the database entry is
accessible.
//uvm_component cntxt= null;
// hierarchical path that limits the accessibility of the database entry.
//string inst_name = "*"
//String field_name = "control";
//T value = 1;

// In component_B:
if(!uvm_config_db #(bit)::get(this, "*", "control", ctrl))
  `uvm_fatal(get_type_name(), "get failed for resource in this scope");
```

The resource is retrieved using the get static function which lookup in the database with the field_name as "control". If get() fails to find the "control" string field_name in the database, a fatal will be reported.

```
`include "uvm_macros.svh"
import uvm_pkg::*;

class component_A #(parameter ID_WIDTH = 8) extends uvm_component;
  bit [ID_WIDTH-1:0] id;
  `uvm_component_param_utils(component_A #(ID_WIDTH))

  function new(string name = "component_A", uvm_component parent = null);
    super.new(name, parent);
    id = 1;
  endfunction

  function display();
    `uvm_info(get_type_name(), $sformatf("inside component_A: id = %0d", id),
UVM_LOW);
  endfunction
endclass
```

```
class mycomponent #(parameter ID_WIDTH = 8) extends uvm_component;
  bit [ID_WIDTH-1:0] id;
  `uvm_component_param_utils(mycomponent #(ID_WIDTH))

  function new(string name = "mycomponent", uvm_component parent = null);
    super.new(name, parent);
    id = 2;
  endfunction

  function display();
    `uvm_info(get_type_name(), $sformatf("inside mycomponent: id = %0d", id),
UVM_LOW);
  endfunction
endclass

class component_B #(int ID_WIDTH = 8) extends component_A #(ID_WIDTH);
  bit ctrl;
  bit [ID_WIDTH-1:0] id;
  mycomponent #(8) my_comp;
  `uvm_component_param_utils(component_B #(ID_WIDTH))

  function new(string name = "component_B", uvm_component parent = null);
    super.new(name, parent);
    id = 3;
  endfunction

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    if(!uvm_config_db #(bit)::get(this, "*", "control", ctrl))
      `uvm_fatal(get_type_name(), "get failed for resource in this scope");
    if(ctrl)  my_comp = mycomponent #(8)::type_id::create("my_comp", this);

  endfunction

  function display();
    `uvm_info(get_type_name(), $sformatf("inside component_B: id = %0d, ctrl
= %0d", id, ctrl), UVM_LOW);
    if(ctrl) void'(my_comp.display());
  endfunction
endclass
```

```
class my_test extends uvm_test;
  bit control;
  `uvm_component_utils(my_test)
  component_A #(32) comp_A;
  component_B #(16) comp_B;

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

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    comp_A = component_A #(32)::type_id::create("comp_A", this);
    comp_B = component_B #(16)::type_id::create("comp_B", this);

    uvm_config_db #(bit)::set(null, "*", "control", 1);
    //or
    //uvm_config_db #(bit)::set(this, "*", "control", 1);
  endfunction

  function void end_of_elaboration_phase(uvm_phase phase);
    super.end_of_elaboration_phase(phase);
    uvm_top.print_topology();
  endfunction

  task run_phase(uvm_phase phase);
    super.run_phase(phase);
    void'(comp_A.display());
    void'(comp_B.display());
  endtask
endclass

module tb_top;
  initial begin
    run_test("my_test");
  end
endmodule
```

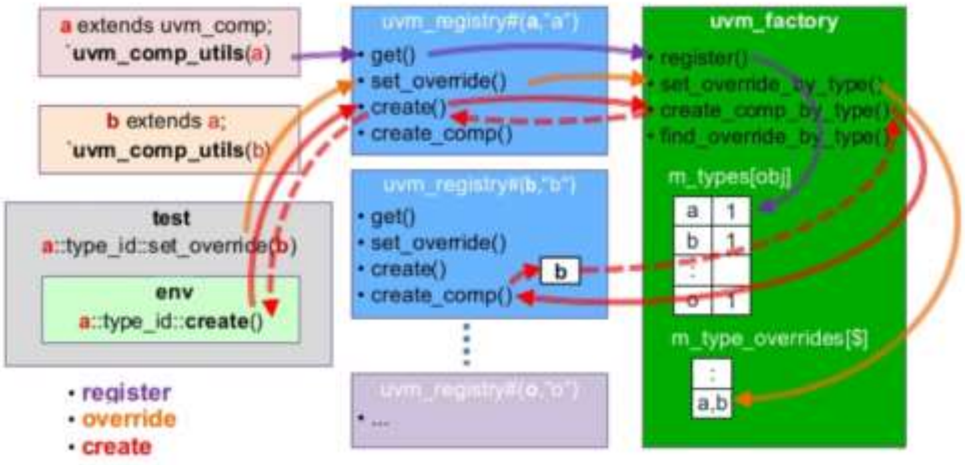
2. Factory

UVM factory gives mechanism to improve flexibility, scalability and reusability of the testbench by allowing the user to substitute an existing class object by any of its inherited child class objects. User can use "factory overriding" feature given by UVM to swap instances of an old class with instances of a new class. Moreover, UVM requires each user defined class to be registered in the factory.

There are UVM macros that allow classes to be registered with the factory, and methods that allow certain types and instances of class objects to be overridden by its derived types.

In order to register a class in factory, two macros are used:

- `uvm_component_utils & `uvm_component_param_utils (for parameterized classes): This macro registers classes' names which are derived from uvm_component base class.
- `uvm_object_utils & `uvm_object_param_utils (for parameterized classes):: This macro registers classes' names which are derived from uvm_transaction, uvm_sequence etc.



Why is UVM factory required?

The new function is used in SystemVerilog to create a class object and is perfectly valid to be used in UVM as well. Assume that an existing testbench uses Wishbone v1.0 protocol data packet class and is used throughout the testbench in components like driver, monitor, scoreboard, and many other sequences. If Wishbone v2.0 is released and the testbench is required to update and start using packet definition for the new protocol, there would be many places in the testbench that would require an update in code which can prove to be cumbersome.

UVM has a factory feature which allows users to modify or substitute type of the item created by the factory without having to modify existing class instantiations. Instead of text substitution of class name for the existing data packet, a child class object can be created that makes necessary modifications for Wishbone v2.0 and the factory can be used to return the newly defined class object in all places within the testbench instead of the first one. So, the preferred method of object creation in UVM testbench is by using create() method. This is known as the UVM factory override mechanism.

The create() method of the wrapper class is used to create objects for the uvm_object and uvm_component class. The build_phase is used to create component instances and build component hierarchy. While the uvm objects are created in the run_phase.

Syntax for Component creation:

```
<instance_name> = <type>::<type_id>::create("<name>", <parent>)
```

Syntax for Object creation:

```
<instance_name> = <type>::<type_id>::create("<name>")
```

Factory Overriding

Based on the requirement, the behavior of the testbench can be changed by substituting one class with another when it is constructed. This facility of the uvm factory allows users to override the class without editing or recompiling the code.

How factory overriding happens?

A factory can be thought of as a look-up table and a factory component wrapper class can be accessed using type_id which is used in create method and returns a resultant handle. Using the polymorphism concept, the factory override mechanism returns a derived type handle using a base type handle. This means when the create method is being called for the base class type, uvm_factory will return a pointer to an object of a derived class type.

Factory overriding ways

1. Type Override
2. Instance Override

Type override

In a type override, a substitute component class type is created instead of an original component class in the testbench hierarchy. This applies to all instances of that component type.

Methods:

1. set_type_override_by_type

```
function void set_type_override_by_type (uvm_object_wrapper original_type,
                                          uvm_object_wrapper override_type,
                                          bit replace = 1)
```

2. set_type_override_by_name

```
function void set_type_override_by_name (string original_type_name,
                                          string override_type_name,
                                          bit replace = 1)
```

```
`include "uvm_macros.svh"
import uvm_pkg::*;

class component_A extends uvm_component;
  `uvm_component_utils(component_A)

  function new(string name = "component_A", uvm_component parent =
null);
    super.new(name, parent);
  endfunction

  virtual function display();
    `uvm_info(get_type_name(), $sformatf("inside component_A"),
UVM_LOW);
  endfunction
endclass

class component_B extends component_A;
  `uvm_component_utils(component_B)

  function new(string name = "component_B", uvm_component parent =
null);
    super.new(name, parent);
  endfunction

  function display();
    `uvm_info(get_type_name(), "inside component_B", UVM_LOW);
  endfunction
endclass
```

```
class my_test extends uvm_test;
  `uvm_component_utils(my_test)
  component_A comp_A;

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

  function void build_phase(uvm_phase phase);
      ……….
  endfunction

  function void end_of_elaboration_phase(uvm_phase phase);
    super.end_of_elaboration_phase(phase);
    uvm_top.print_topology();
  endfunction

  task run_phase(uvm_phase phase);
    super.run_phase(phase);
    comp_A.display();
  endtask
endclass

module tb_top;
  initial begin
    run_test("my_test");
  end
endmodule
```

Method 1

```
function void build_phase(uvm_phase phase);
    uvm_factory factory = uvm_factory::get();
    super.build_phase(phase);

    set_type_override_by_type(component_A::get_type(),
component_B::get_type());
    comp_A = component_A::type_id::create("comp_A", this);
    factory.print();
  endfunction
```

Method 2

```
function void build_phase(uvm_phase phase);
    uvm_factory factory = uvm_factory::get();
    super.build_phase(phase);

    factory.set_type_override_by_name("component_A", "component_B");
    comp_A = component_A::type_id::create("comp_A", this);
    factory.print();
endfunction
```

Instance Override

Unlike type override does override all instances of the type, instance override does override only specified positions in the uvm component hierarchy.

Instance override methods

1. set_inst_override_by_type

```
function void set_inst_override_by_type (uvm_object_wrapper original_type,
                                         uvm_object_wrapper override_type,
                                         string full_inst_path)
```

2. set_inst_override_by_name

```
function void set_inst_override_by_name (string original_type_name,
                                         string override_type_name,
                                         string full_inst_path)
```

Method 1

```
function void build_phase(uvm_phase phase);
    uvm_factory factory = uvm_factory::get();
    super.build_phase(phase);

    set_inst_override_by_type(component_A::get_type(),
component_B::get_type(), "*");
    comp_A = component_A::type_id::create("comp_A", this);

    factory.print();
  endfunction
```

Method 2

```
function void build_phase(uvm_phase phase);
    uvm_factory factory = uvm_factory::get();
    super.build_phase(phase);

    factory.set_inst_override_by_name("component_A", "component_B",
"*");
    comp_A = component_A::type_id::create("comp_A", this);
    factory.print();
  endfunction
```

There are also similar two methods used that behave similarly as mentioned above.

1. set_type_override
   Syntax:

```
<original_type>::type_id::set_type_override(<substitute_type>::get_type(),
replace);
```

```
function void build_phase(uvm_phase phase);
    uvm_factory factory = uvm_factory::get();
    super.build_phase(phase);

    component_A::type_id::set_type_override(component_B::get_type());
    comp_A = component_A::type_id::create("comp_A", this);
    factory.print();
  endfunction
```

2. set_inst_override
   Syntax:

```
<original_type>::type_id::set_inst_override(<substitute_type>::get_type(),
<path_string>);
```

```
function void build_phase(uvm_phase phase);
    uvm_factory factory = uvm_factory::get();
    super.build_phase(phase);

    component_A::type_id::set_inst_override(component_B::get_type(),
"*");
    comp_A = component_A::type_id::create("comp_A", this);
    factory.print();
  endfunction
```

Object Overriding in UVM factory

The uvm_object or sequence overriding is similar to the uvm_component overriding factory mechanism that returns the derived object handle using a base class handle. For overriding uvm_object or sequences, type overriding is recommended to use since instance overriding requires a hierarchical path. But instance override is also possible provided the correct instance path has been mentioned either by arbitrary string or get_full_name() call.

3. Phasing

The phases are an important concept in uvm that applies to all testbench components.

Each testbench component is derived from uvm_component that has predefined phases. They are represented as callback methods. Hence, the user may implement these callbacks.

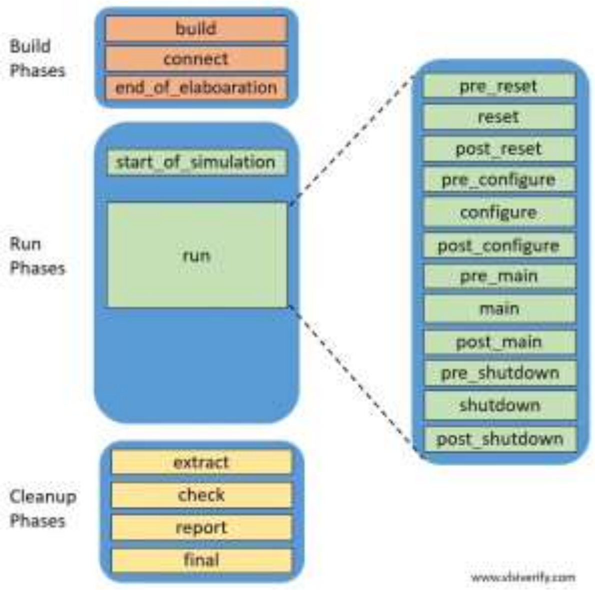
Each component cannot move to the next phase unless the current phase execution is completed for all the components. This provides proper synchronization between all the components.

UVM phases are executed in a certain order and all are virtual methods.

Few phases that consume simulation time for execution are implemented as tasks and other phases that do not consume any simulation time are implemented as functions.

Main categories in UVM phases:

a) Build phases: Used to configure or construct the testbench.
b) Run-time phases: Time-consuming testbench activity like running the test case.
c) Clean up phases: Collect and report the results of the simulation.



How does UVM phase execution start?

The run_test() method is required to call from the static part of the testbench. This will trigger up the UVM testbench. It is usually called in the initial block from the top-level testbench module. The run_test() method call to construct the UVM environment root component and then initiates the UVM phasing mechanism.

Build phases

Phases in this categorize are executed at the start of the UVM testbench simulation, where the testbench components are constructed, configured and testbench components are connected. All the build phase methods are functions and therefore execute in **zero simulation time**.

Phase Name	Phase Type	Description	Execution approach
build_phase	function	Build or create testbench component	Top to down
connect_phase	function	Connect different testbench component using the TLM port mechanism	Bottom to top
end_of_elaboration_phase	function	Before simulation starts, this phase is used to make any final adjustment to the structure, configuration, or connectivity of the testbench. It also displays UVM topology.	Bottom to top

Run-time phases

Once testbench components are created and connected in the testbench, it follows run phases where actual simulation time consumes. After completing, start_of_simulation phase, there are two paths for run-time phases. The run_phase and pre_reset phase both start at the same time.

Phase Name	Phase Type	Description	Execution approach
start_of_simulation_phase	function	Used to display testbench topology or configuration.	Bottom to top
run_phase	task	Used for the stimulus generation, checking activities of the testbench, and consumes simulation time cycles.	The run_phase for all components are executed in parallel.

Clean up phases

The clean-up phases are used to collect information from functional coverage monitors and scoreboards to see whether the coverage goal has been reached or the test case has passed. The cleanup phases will start once the run phases are completed. They are implemented as functions and work from the bottom to the top of the component hierarchy. The extract, check, and report phase may be used by analysis components.

Phase Name	Phase Type	Description	Execution approach
extract	function	Used to retrieve and process the information from functional coverage monitors and scoreboards. This phase may also calculate any statistical information that will be used by report_phase.	Bottom to top
check	function	Checks DUT behavior and identity for any error that occurred during the execution of the testbench.	Bottom to top
report	function	Used to display simulation results. It can also write results to the file.	Bottom to top
final	function	Used to complete any outstanding actions that are yet to be completed in the testbench.	Top to down

Note:

Remember that in Verilog, as modules are static, users don’t have to care about their creation as they would have already created at the beginning of the simulation. In the case of UVM based System Verilog testbench, class objects can be created at any time during the simulation based on the requirement. Hence, it is required to have proper synchronization to avoid objects/components being called before they are created, The UVM phasing mechanism serves the purpose of synchronization.

4. Reporting

As we saw before, most of the verification components are inherited from UVM report object and hence they already have functions and methods to display messages. UVM Reporting or Messaging has a rich set of message-display commands & methods to alter the numbers & types of messages that are displayed without re-compilation of the design. UVM Reporting also includes the ability to mask or change the severity of the message to adapt the required environment condition.

UVM Reporting has the concepts of Severity, Verbosity and Simulation Handing Behavior. Each of them can be independently specified and controlled.

Now let's see what each of these indicates:

Severity

- Severity indicates importance
- Examples are Fatal, Error, Warning & Info

Verbosity

- Verbosity indicates filter level
- Examples are None, Low, Medium, High, Full & Debug

Simulation Handling Behavior

- Simulation handling behavior controls simulator behavior
- Simulation Handling Behavior in-fact is the Action taken by the Simulator which is dependent on Severity being produced by the Verification Environment.
- Examples are Exit, Count, Display, Log, Call Hook & No Action

UVM Reporting provides Macros to embed report messages. Followings are the Macros to be used:

- `` `uvm_info(string ID, string MSG, verbosity); ``
- `` `uvm_error(string ID, string MSG); ``
- `` `uvm_warning(string ID, string MSG); ``
- `` `uvm_fatal(string ID, string MSG); ``

From the syntax of the above mentioned Macros it is very well evident how we can embed the message of our choice with a particular macro. The provided message ID helps in easy search, e.g. grep, in the simulation log. In addition, total count of a particular severity can also be seen using message ID at the end of the simulation log.

Simulation Handling Behavior:

Now, when we've seen the usage of the different severity macros, let see the associated default simulator action with each of these Severity.

Severity	Simulator Action
uvm_fatal	uvm_display & uvm_exit
uvm_error	uvm_display & uvm_count
uvm_warning	uvm_display
uvm_info	uvm_display

There are in total six Simulator behaviors which can be associated with a particular UVM Reporting Severity depending upon the users choice. Let's see the description for each of Simulation Actions:

Simulator Action	Description
uvm_exit	Exit from simulation immediately
uvm_count	Increment global error count
uvm_display	Display message on console
uvm_log	Captures messages in a named file
uvm_call_back	Calls callback method
uvm_no_action	Do nothing

Fundamentally the Verbosity level describes how verbose a Testbench can be. The default Verbosity is UVM_MEDIUM. There are different Verbosity levels being supported by UVM, **which is required only for uvm_info**.

Verbosity level	Value
UVM_NONE	0
UVM_LOW	100
UVM_MEDIUM	200
UVM_HIGH	300
UVM_FULL	400
UVM_DEBUG	500

5. Objections

The uvm_objection class provides a mechanism to coordinate status information between two or more components, objects. The uvm_objection class is extended from uvm_report_object. The objection deals with the concept of raise and drop objection which means the internal counter is increment and decrement respectively. Each participating component and object may raise or drop objections asynchronously. When all objections are dropped, the counter value will become zero. The objection has to be raised before starting any process and drop it once it is completed. Objections are generally used in components and sequences.

UVM Objection Usage

1. UVM phasing mechanism uses objections to coordinate with each other and the phase should be ended when all objections are dropped. They can be used in all UVM phases.
2. It allows proceeding for the "End of test". The simulation time-consuming activity happens in the run phase. If all objections dropped for run phases, it means simulation activity is completed. The test can be ended after executing clean up phases.

```
task reset_phase( uvm_phase phase);
  phase.raise_objection(this);
  ...
  phase.drop_objection(this);
endtask

task run_phase(uvm_phase phase);
  phase.raise_objection(this, "Raised Objection");
  ...
  phase.drop_objection(this, "Dropped Objection");
endtask
```

Methods in uvm_objection

Methods	Description
raise_objection (uvm_object obj = null, string description = " " , int count = 1)	Raises number of objections for corresponding object with default count = 1
drop_objection (uvm_object obj = null, string description = " ", int count = 1)	Drops number of objections for corresponding object with default count = 1
set_drain_time (uvm_object obj=null, time drain)	set_drain_time (uvm_object obj=null, time drain)

Drain Time: The amount of wait time between all objections has been dropped and calling all_dropped() callback is called drain time.

The callback methods are defined in the uvm_objection_callback class which is derived from the uvm_callback class.

Callback Hooks	**Description**
raised	Called when an objection is raised for this component or its derived classes.
dropped	Called when an objection is dropped for this component or its derived classes.
all_dropped	Called when all objections are dropped for this component and its derived classes.

Example:

```
class seq_item extends uvm_sequence_item;
  rand bit[15:0] addr;
  rand bit[15:0] data;
  `uvm_object_utils(seq_item)

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

endclass

class base_seq extends uvm_sequence #(seq_item);
  seq_item req;
  `uvm_object_utils(base_seq)

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

  task body();
    `uvm_info(get_type_name(), "Base seq: Body started", UVM_LOW);
    req = seq_item::type_id::create("req");
    // send to the driver
    #20;
    `uvm_info(get_type_name(), "Base seq: Body completed", UVM_LOW);
  endtask
endclass
```

```
class base_test extends uvm_test;
  base_seq bseq;

  `uvm_component_utils(base_test)

  function new(string name = "base_test", uvm_component parent = null);
    super.new(name, parent);
  endfunction

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
  endfunction

  task run_phase(uvm_phase phase);
    super.run_phase(phase);
    phase.raise_objection(this, "Raise Objection");
    bseq = base_seq::type_id::create("bseq");
    bseq.start(null);
    phase.drop_objection(this, "Drop Objection");
  endtask
endclass

module tb_top;
  initial begin
    run_test("base_test");
  end
endmodule
```

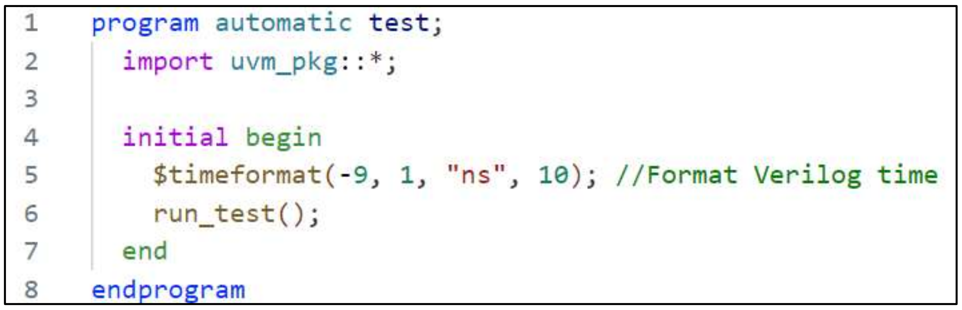
The objection can also be raised and dropped either in body tasks or pre_body/ post_body tasks.

```
class base_seq extends uvm_sequence #(seq_item);
  ...
  task pre_body();
    if(starting_phase != null) starting_phase.raise_objection(this);
  endtask

  task body();
    req = seq_item::type_id::create("req");
    // send to the driver
    #20;
    ...
  endtask

  task post_body();
    if(starting_phase != null) starting_phase.drop_objection(this);
  endtask
endclass
```

6. Lab Task

Part 1: Build a simple UVM test

1) Enter the following code in a new file "**test.sv**":



```
program automatic test;
  import uvm_pkg::*;

  initial begin
    $timeformat(-9, 1, "ns", 10); //Format Verilog time
    run_test();
  end
endprogram
```

2) Compile using VCS by entering the following command:

   ➢ vcs -sverilog -ntb_opts uvm-1.2 test.sv

   Note: The -ntb_opts uvm-1.2 switch enables VCS to look for the uvm_pkg in the VCS installation directory.

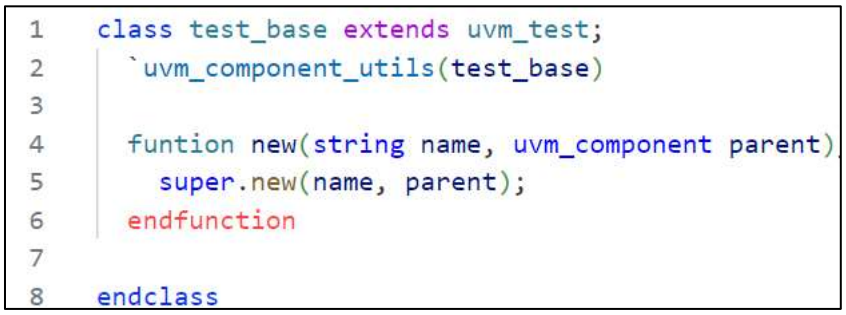
3) Simulate this UVM testbench and store the output in "**simv.log**" file:

   ➢ ./simv -l simv.log

4) You should see a fatal error report because:
   a. You do not have any UVM test class
   b. If you do have a UVM test class, you need to specify it using the +UVM_TESTNAME switch.

5) Create a UVM test class.

   Enter the following code in a new file and name it "**test_collection.sv**":



```
class test_base extends uvm_test;
  `uvm_component_utils(test_base)

  funtion new(string name, uvm_component parent)
    super.new(name, parent);
  endfunction

endclass
```

6) Include the "**test_collection.sv**" in your test program by adding the following line:

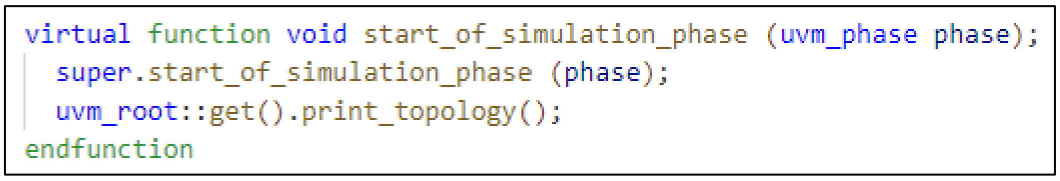
```
`include "test_collection.sv"
```

7) Compile and simulate the code again while specifying the specific test you need the UVM execution manager to run:

   ➢ vcs – sverilog -ntb_opts uvm-1.2 test.sv

- ./simv simv.log +UVM_TESTNAME=test_base

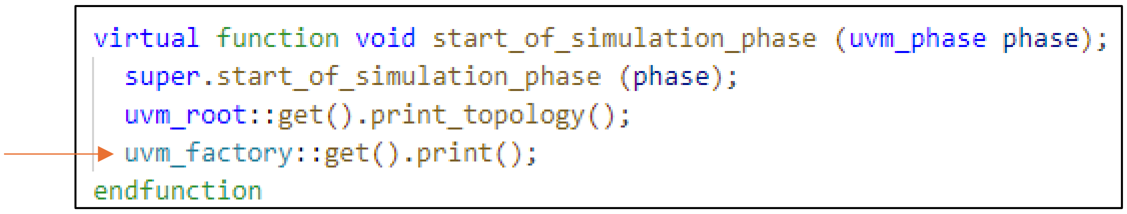
8) UVM execution manager is aware of the entire UVM test hierarchy from top to bottom. You can use the uvm_root class, and the class handle uvm_root::get() to print the test structural topology.
   a. Open the "**test_collection.sv**" file
   b. Add the following start of simulation method to the test_base class



```
virtual function void start_of_simulation_phase (uvm_phase phase);
  super.start_of_simulation_phase (phase);
  uvm_root::get().print_topology();
endfunction
```

9) Compile and simulate the test again. You should now see the test topology printed in the "**simv.log**" file.

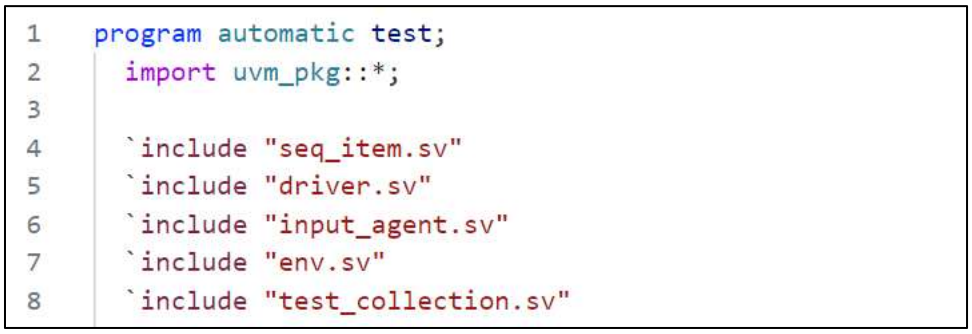
10) It is also useful for debugging to see all the user classes registered in the UVM factory. You can get this information by adding the following line to the start_of_simulation_phase method:



```
virtual function void start_of_simulation_phase (uvm_phase phase);
  super.start_of_simulation_phase (phase);
  uvm_root::get().print_topology();
  uvm_factory::get().print();
endfunction
```

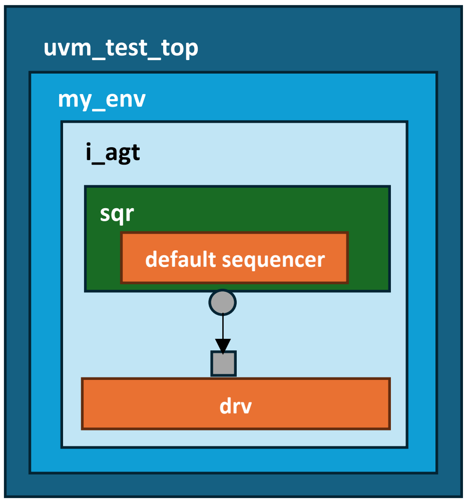
Part 2: Build a UVM testbench Hierarchy

1) Download lab 10 from Moodle. Add “**env.sv**”, “**input_agent.sv**”, “**driver.sv**”, and “**seq_item.sv**” to your simulation directory.
2) Add the following include statements in your testbench:



```
program automatic test;
  import uvm_pkg::*;

  `include "seq_item.sv"
  `include "driver.sv"
  `include "input_agent.sv"
  `include "env.sv"
  `include "test_collection.sv"
```

3) Open “**driver.sv**” file:
   a. Register the class in the UVM factory.
   b. Define the constructor method.
4) Instantiate and construct a driver and a sequencer object in the agent
   a. Open the “**input_agent.sv**” file:
   b. Use factory to create a driver and a sequencer object in the build phase.
5) Instantiate and construct an environment object in the test class
   a. Open the “**test_collection.sv**” file
   b. Use factory to create an environment object in the build phase
6) Compile and simulate your UVM testbench.
7) Check the “**simv.log**” file, you should now see:
   a. The full topology as shown in the figure below
   b. The factory content


uvm_test_top
my_env
i_agt
sqr
default sequencer
drv