INDEX

- Hierarchical Sequences
- Sequential Sequences
- Parallel Sequences

UVM SEQUENCE 4

- Sequencer Arbitration
- Setting The Sequence Priority

UVM SEQUENCE 5

- Sequencer Registration Macros
- Setting Sequence Members

UVM SEQUENCE 6

- Exclusive Access
- Lock-Unlock
- Grab-Ungrab

UVM TLM 1

- Port Based Data Transfer
- Task Based Data Transfer
- Operation Supported By Tlm Interface
- Methods
- Tlm Terminology
- Tlm Interface Compilation Models
- Interfaces
- Direction
- All Interfaces In Uvm

UVM TLM 2

- Analysis
- Tlm Fifo
- Example

UVM CALLBACK

- Driver And Driver Callback Class Source Code
- Testcase Source Code
- Testcase 2 Source Code
- Testcase 3 Source Code
- Testcase 4 Source Code
- Methods
- Macros

INTRODUCTION

The UVM (Universal Verification Methodology) Class Library provides the building blocks needed to quickly develop well-constructed and reusable verification components and test environments in SystemVerilog.

UVM library contains:

- Component classes for building testbench components like generator/driver/monitor etc.
- Reporting classes for logging,
- Factory for object substitution.
- Synchronization classes for managing concurrent process.
- Policy classes for printing, comparing, recording, packing, and unpacking of uvm_object based classes.
- TLM Classes for transaction level interface.
- Sequencer and Sequence classes for generating realistic stimulus.
- And Macros which can be used for shorthand notation of complex implementation.

In this tutorial, we will learn some of the UVM concepts with examples.

Installing Uvm Library

1)Go to http://www.accellera.org/activities/vip/
2)Download the uvm*.tar.gz file.
3)Untar the file.
4)Go to the extracted directory : cd uvm*\uvm\src
5)Set the UVM_HOME path : setenv UVM_HOME `pwd`
(This is required to run the examples which are downloaded from this site)
6)Go to examples : cd ../examples/hello_world/uvm/
7)Compile the example using :
your_tool_compilation_command -f compile_<toolname>.f
(example for questasim use : qverilog -f compile_questa.f)
8)Run the example.

UVM TESTBENCH

Uvm components, uvm env and uvm test are the three main building blocks of a testbench in uvm based verification.

Uvm_env

uvm_env is extended from uvm_component and does not contain any extra functionality. uvm_env is used to create and connect the uvm_components like driver, monitors , sequeners etc. A environment class can also be used as sub-environment in another environment. As there is no difference between uvm_env and uvm_component , we will discuss about uvm_component, in the next section.

Verification Components

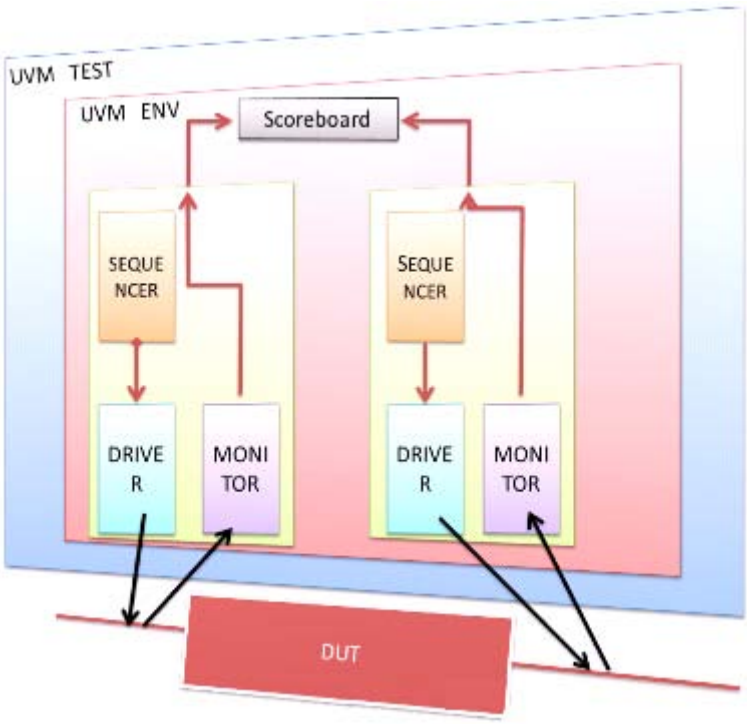
uvm verification component classes are derived from uvm_component class which provides features like hierarchy searching, phasing, configuration , reporting , factory and transaction recording.

Following are some of the uvm component classes

- uvm_agent
- uvm_monitor
- uvm_scoreboard
- uvm_driver
- uvm_sequencer

NOTE: uvm_env and uvm_test are also extended from uvm_component.

A typical uvm verification environment:



An agent typically contains three subcomponents: a driver, sequencer, and monitor. If the agent is active, subtypes should contain all three subcomponents. If the agent is passive, subtypes should contain only the monitor.

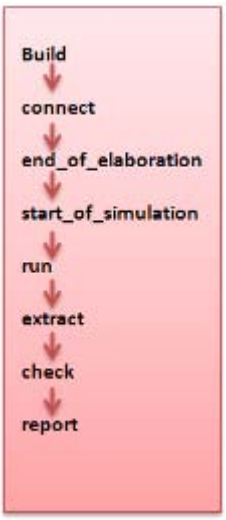
About Uvm_component Class:

uvm_compoenent class is inherited from uvm_report_object which is inherited from uvm_object.

As I mentioned previously, uvm_component class provides features like hierarchy searching, phasing, configuration , reporting , factory and transaction recording. We will discuss about phasing concept in this section and rest of the features will be discussed as separate topics.

UVM phases

UVM Components execute their behavior in strictly ordered, pre-defined phases. Each phase is defined by its own virtual method, which derived components can override to incorporate component-specific behavior. By default , these methods do nothing.



--> **virtual function void** build()

This phase is used to construct various child components/ports/exports and configures them.

--> **virtual function void** connect()

This phase is used for connecting the ports/exports of the components.

--> **virtual function void** end_of_elaboration()

This phase is used for configuring the components if required.

--> **virtual function void** start_of_simulation()

This phase is used to print the banners and topology.

--> **virtual task** run()

In this phase , Main body of the test is executed where all threads are forked off.

--> **virtual function void** extract()

In this phase, all the required information is gathered.

--> **virtual function void** check()

In this phase, check the results of the extracted information such as un responded requests in scoreboard, read statistics registers etc.

--> **virtual function void** report()

This phase is used for reporting the pass/fail status.

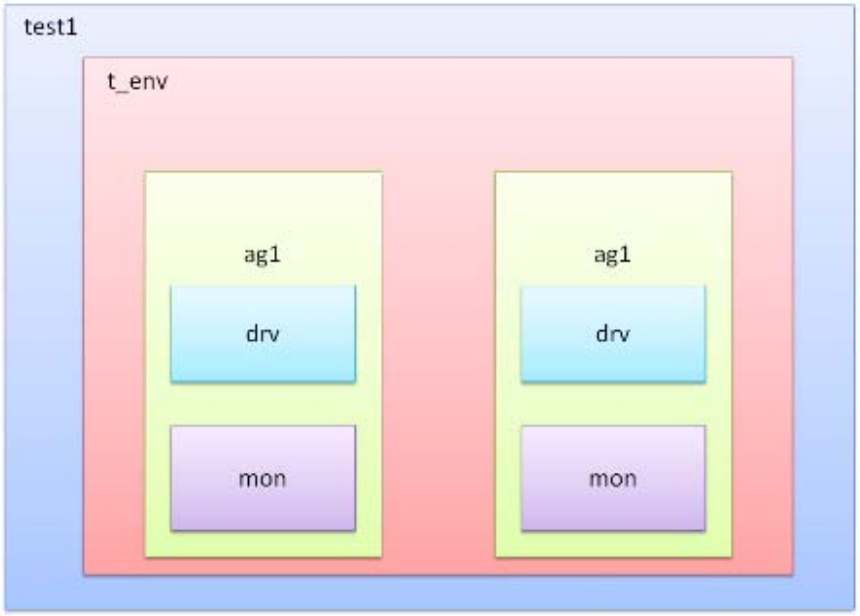
Only build() method is executed in top down manner. i.e after executing parent build() method, child objects build() methods are executed. All other methods are executed in bottom-up manner. The run() method is the only method which is time consuming. The run() method is forked, so the order in which all components run() method are executed is undefined.

Uvm_test

uvm_test is derived from uvm_component class and there is no extra functionality is added. The advantage of used uvm_test for defining the user defined test is that the test case selection can be done from command line option +UVM_TESTNAME=<testcase_string> . User can also select the testcase by passing the testcase name as string to uvm_root::run_test(<testcase_string>) method.

In the above <testcase_string> is the object type of the testcase class.

Lets implement environment for the following topology. I will describe the implementation of environment , testcase and top module. Agent, monitor and driver are implemented similar to environment.



1)Extend uvm_env class and define user environment.

```
class env extends uvm_env;
```

2)Declare the utility macro. This utility macro provides the implementation of create() and get_type_name() methods and all the requirements needed for factory.

```
`uvm_component_utils(env)
```

3)Declare the objects for agents.

```
agent ag1;
agent ag2;
```

4)Define the constructor. In the constructor, call the super methods and pass the parent object. Parent is the object in which environment is instantiated.

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

5)Define build method. In the build method, construct the agents.

To construct agents, use create() method. The advantage of create() over new() is that when create() method is called, it will check if there is a factory override and constructs the object of override type.

```
function void build();
  super.build();
  uvm_report_info(get_full_name(),"Build", UVM_LOW);
  ag1 = agent::type_id::create("ag1",this);
  ag2 = agent::type_id::create("ag2",this);
endfunction
```

6)Define connect(),end_of_elaboration(),start_of_simulation(),run(),extract(),check(),report() methods.

Just print a message from these methods, as we dont have any logic in this example to define.

```
function void connect();
  uvm_report_info(get_full_name(),"Connect", UVM_LOW);
endfunction
```

Complete code of environment class:

```
class env extends uvm_env;

  `uvm_component_utils(env)
  agent ag1;
  agent ag2;

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

  function void build();
    uvm_report_info(get_full_name(),"Build", UVM_LOW);
    ag1 = agent::type_id::create("ag1",this);
    ag2 = agent::type_id::create("ag2",this);
  endfunction

  function void connect();
    uvm_report_info(get_full_name(),"Connect", UVM_LOW);
  endfunction

  function void end_of_elaboration();
    uvm_report_info(get_full_name(),"End_of_elaboration", UVM_LOW);
  endfunction

  function void start_of_simulation();
    uvm_report_info(get_full_name(),"Start_of_simulation", UVM_LOW);
  endfunction

  task run();
    uvm_report_info(get_full_name(),"Run", UVM_LOW);
  endtask

  function void extract();
    uvm_report_info(get_full_name(),"Extract", UVM_LOW);
  endfunction

  function void check();
    uvm_report_info(get_full_name(),"Check", UVM_LOW);
  endfunction

  function void report();
    uvm_report_info(get_full_name(),"Report", UVM_LOW);
  endfunction
endclass
```

Now we will implement the testcase.

1)Extend uvm_test and define the test case

```
class test1 extends uvm_test;
```

2)Declare component ustilits using utility macro.

```
`uvm_component_utils(test1)
```

2)Declare environment class handle.

```
env t_env;
```

3)Define constructor method. In the constructor, call the super method and construct the environment object.

```
function new (string name="test1", uvm_component parent=null);
  super.new (name, parent);
  t_env = new("t_env",this);
endfunction : new
```

4)Define the end_of_elaboration method. In this method, call the print() method. This print() method will print the topology of the test.

```
function void end_of_elaboration();
  uvm_report_info(get_full_name(),"End_of_elaboration", UVM_LOW);
  print();
endfunction
```

4)Define the run method and call the global_stop_request() method.

```
task run ();
  #1000;
  global_stop_request();
endtask : run
```

Testcase source code:

```
class test1 extends uvm_test;

  `uvm_component_utils(test1)
  env t_env;
```

```
  function new (string name="test1", uvm_component parent=null);
    super.new (name, parent);
    t_env = new("t_env",this);
  endfunction : new

  function void end_of_elaboration();
    uvm_report_info(get_full_name(),"End_of_elaboration", UVM_LOW);
    print();
  endfunction

  task run ();
    #1000;
    global_stop_request();
  endtask : run

endclass
```

Top Module:

To start the testbench, run_test() method must be called from initial block. Run_test() mthod Phases all components through all registered phases.

```
module top;

  initial
  run_test();

endmodule
```

Download the source code

uvm_phases.tar
Browse the code in uvm_phases.tar

Command to run the simulation

VCS Users : make vcs
Questa Users: make questa

Log file:

```
[RNTST] Running test test1...
uvm_test_top.t_env [uvm_test_top.t_env] Build
uvm_test_top.t_env.ag1 [uvm_test_top.t_env.ag1] Build
uvm_test_top.t_env.ag1.drv [uvm_test_top.t_env.ag1.drv] Build
uvm_test_top.t_env.ag1.mon [uvm_test_top.t_env.ag1.mon] Build
uvm_test_top.t_env.ag2 [uvm_test_top.t_env.ag2] Build
uvm_test_top.t_env.ag2.drv [uvm_test_top.t_env.ag2.drv] Build
uvm_test_top.t_env.ag2.mon [uvm_test_top.t_env.ag2.mon] Build
uvm_test_top.t_env.ag1.drv [uvm_test_top.t_env.ag1.drv] Connect
uvm_test_top.t_env.ag1.mon [uvm_test_top.t_env.ag1.mon] Connect
uvm_test_top.t_env.ag1 [uvm_test_top.t_env.ag1] Connect
uvm_test_top.t_env.ag2.drv [uvm_test_top.t_env.ag2.drv] Connect
uvm_test_top.t_env.ag2.mon [uvm_test_top.t_env.ag2.mon] Connect
uvm_test_top.t_env.ag2 [uvm_test_top.t_env.ag2] Connect
uvm_test_top.t_env [uvm_test_top.t_env] Connect
uvm_test_top.t_env.ag1.drv [uvm_test_top.t_env.ag1.drv] End_of_elaboration
uvm_test_top.t_env.ag1.mon [uvm_test_top.t_env.ag1.mon] End_of_elaboration
uvm_test_top.t_env.ag1 [uvm_test_top.t_env.ag1] End_of_elaboration
uvm_test_top.t_env.ag2.drv [uvm_test_top.t_env.ag2.drv] End_of_elaboration
uvm_test_top.t_env.ag2.mon [uvm_test_top.t_env.ag2.mon] End_of_elaboration
uvm_test_top.t_env.ag2 [uvm_test_top.t_env.ag2] End_of_elaboration
uvm_test_top.t_env [uvm_test_top.t_env] End_of_elaboration
uvm_test_top [uvm_test_top] End_of_elaboration
----------------------------------------------------------------
Name Type Size Value
----------------------------------------------------------------
uvm_test_top test1 - uvm_test_top@2
t_env env - t_env@4
ag1 agent - ag1@6
drv driver - drv@12
rsp_port uvm_analysis_port - rsp_port@16
sqr_pull_port uvm_seq_item_pull_+ - sqr_pull_port@14
mon monitor - mon@10
ag2 agent - ag2@8
drv driver - drv@20
rsp_port uvm_analysis_port - rsp_port@24
sqr_pull_port uvm_seq_item_pull_+ - sqr_pull_port@22
mon monitor - mon@18

uvm_test_top.t_env.ag1.drv[uvm_test_top.t_env.ag1.drv]Start_of_simulation
uvm_test_top.t_env.ag1.mon[uvm_test_top.t_env.ag1.mon]Start_of_simulation
uvm_test_top.t_env.ag1[uvm_test_top.t_env.ag1]Start_of_simulation
uvm_test_top.t_env.ag2.drv[uvm_test_top.t_env.ag2.drv]Start_of_simulation
```

```
uvm_test_top.t_env.ag2.mon[uvm_test_top.t_env.ag2.mon]Start_of_simulatio

..
..
..
..
```

Observe the above log report:

1)Build method was called in top-down fashion. Look at the following part of message.

```
uvm_test_top.t_env [uvm_test_top.t_env] Build
uvm_test_top.t_env.ag1 [uvm_test_top.t_env.ag1] Build
uvm_test_top.t_env.ag1.drv [uvm_test_top.t_env.ag1.drv] Build
uvm_test_top.t_env.ag1.mon [uvm_test_top.t_env.ag1.mon] Build
uvm_test_top.t_env.ag2 [uvm_test_top.t_env.ag2] Build
uvm_test_top.t_env.ag2.drv [uvm_test_top.t_env.ag2.drv] Build
uvm_test_top.t_env.ag2.mon [uvm_test_top.t_env.ag2.mon] Build
```

2)Connect method was called in bottopm up fashion. Look at the below part of log file,

```
uvm_test_top.t_env.ag1.drv [uvm_test_top.t_env.ag1.drv] Connect
uvm_test_top.t_env.ag1.mon [uvm_test_top.t_env.ag1.mon] Connect
uvm_test_top.t_env.ag1 [uvm_test_top.t_env.ag1] Connect
uvm_test_top.t_env.ag2.drv [uvm_test_top.t_env.ag2.drv] Connect
uvm_test_top.t_env.ag2.mon [uvm_test_top.t_env.ag2.mon] Connect
uvm_test_top.t_env.ag2 [uvm_test_top.t_env.ag2] Connect
uvm_test_top.t_env [uvm_test_top.t_env] Connect
```

3)Following part of log file shows the testcase topology

```
----------------------------------------------------------------------
Name                    Type                Size                Value
----------------------------------------------------------------------
uvm_test_top            test1               -           uvm_test_top@2
  t_env                 env                 -                  t_env@4
    ag1                 agent               -                    ag1@6
      drv               driver              -                   drv@12
        rsp_port        uvm_analysis_port   -              rsp_port@16
        sqr_pull_port   uvm_seq_item_pull_* -         sqr_pull_port@14
      mon               monitor             -                   mon@10
    ag2                 agent               -                    ag2@8
      drv               driver              -                   drv@20
        rsp_port        uvm_analysis_port   -              rsp_port@24
        sqr_pull_port   uvm_seq_item_pull_* -         sqr_pull_port@22
      mon               monitor             -                   mon@18
```

UVM REPORTING

The uvm_report_object provides an interface to the UVM reporting facility. Through this interface, components issue the various messages with different severity levels that occur during simulation. Users can configure what actions are taken and what file(s) are output for individual messages from a particular component or for all messages from all components in the environment.

A report consists of an id string, severity, verbosity level, and the textual message itself. If the verbosity level of a report is greater than the configured maximum verbosity level of its report object, it is ignored.

Reporting Methods:

Following are the primary reporting methods in the UVM.

```
virtual function void uvm_report_info
(string id,string message,int verbosity=UVM_MEDIUM,string filename="",int line=0)

virtual function void uvm_report_warning
(string id,string message,int verbosity=UVM_MEDIUM,string filename="",int line=0)

virtual function void uvm_report_error
(string id,string message,int verbosity=UVM_LOW, string filename="",int line=0)

virtual function void uvm_report_fatal
(string id,string message,int verbosity=UVM_NONE, string filename="",int line=0)
```

Arguments description:

- id -- a unique id to form a group of messages.
- message -- The message text
- verbosity -- the verbosity of the message, indicating its relative importance. If this number is less than or equal to the effective verbosity level, then the report is issued, subject to the configured action and file descriptor settings.
- filename/line -- If required to print filename and line number from where the message is issued, use macros, `__FILE__ and `__LINE__.

Actions:

These methods associate the specified action or actions with reports of the given severity, id, or severity-id pair.

Following are the actions defined:

- UVM_NO_ACTION -- Do nothing
- UVM_DISPLAY -- Display report to standard output
- UVM_LOG -- Write to a file
- UVM_COUNT -- Count up to a max_quit_count value before exiting
- UVM_EXIT -- Terminates simulation immediately
- UVM_CALL_HOOK -- Callback the hook method .

Configuration:

Using these methods, user can set the verbosity levels and set actions.

```
function void set_report_verbosity_level
        (int verbosity_level)
function void set_report_severity_action
        (uvm_severity severity,uvm_action action)
function void set_report_id_action
        (string id,uvm_action action)
function void set_report_severity_id_action
        (uvm_severity severity,string id,uvm_action action)
```

UVM Reporting API		
Severity	Types	Methods
	UVM_INFO	uvm_report_info
	UVM_WARNING	uvm_report_warning
	UVM_ERROR	uvm_report_error
	UVM_FATAL	uvm_report_fatal
Filtering	set_report_verbosity_level	
Actions	Types	Methods
	UVM_NO_ACTION	set_report_id_action
	UVM_DISPLAY	
	UVM_LOG	set_report_severity_action
	UVM_COUNT	
	UVM_EXIT	set_report_severity_id_action
	UVM_CALLBACK_HOOK	
Outputfile	set_report_default_file	
	set_report_severity_file	
	set_report_id_file	
	set_report_severity_id_file	
Query	get_report_verbosity_level	
	get_report_action	
	get_report_file_handle	

Example

Lets see an example:

In the following example, messages from rpting::run() method are of different verbosity level. In the top module, 3 objects of rpting are created and different verbosity levels are set using set_report_verbosity_level() method.

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

class rpting extends uvm_component;

  `uvm_component_utils(rpting)

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

  task run();
    uvm_report_info(get_full_name(),
      "Info Message : Verbo lvl - UVM_NONE ",UVM_NONE,`__FILE__,`__LINE__);

    uvm_report_info(get_full_name(),
      "Info Message : Verbo lvl - UVM_LOW ",UVM_LOW);

    uvm_report_info(get_full_name(),
      "Info Message : Verbo lvl - 150 ",150);

    uvm_report_info(get_full_name(),
      "Info Message : Verbo lvl - UVM_MEDIUM",UVM_MEDIUM);

    uvm_report_warning(get_full_name(),
      "Warning Messgae from rpting",UVM_LOW);

    uvm_report_error(get_full_name(),
      "Error Message from rpting \n\n",UVM_LOW);
  endtask
endclass


module top;
  rpting rpt1;
  rpting rpt2;
  rpting rpt3;
```

```
  initial begin
    rpt1 = new("rpt1",null);
    rpt2 = new("rpt2",null);
    rpt3 = new("rpt3",null);

    rpt1.set_report_verbosity_level(UVM_MEDIUM);
    rpt2.set_report_verbosity_level(UVM_LOW);
    rpt3.set_report_verbosity_level(UVM_NONE);
    run_test();
  end

endmodule
```

Download the source code

uvm_reporting.tar
Browse the code in uvm_reporting.tar

Command to run the simulation

VCS Users : make vcs
Questa Users: make questa

Log file:

```
UVM_INFO reporting.sv(13)@0:rpt1[rpt1]Info Message:Verbo lvl - UVM_NONE
UVM_INFO @0:rpt1[rpt1] Info Message : Verbo lvl - UVM_LOW
UVM_INFO @0:rpt1[rpt1] Info Message : Verbo lvl - 150
UVM_INFO @0:rpt1[rpt1] Info Message : Verbo lvl - UVM_MEDIUM
UVM_WARNIN@0:rpt[rpt1] Warning Messgae from rpting
UVM_ERROR @0:rpt1[rpt1] Error Message from rpting

UVM_INFOreporting.sv(13)@0:rpt2[rpt2]Info Message:Verbo lvl - UVM_NONE
UVM_INFO@ 0:rpt2[rpt2] Info Message : Verbo lvl - UVM_LOW
UVM_WARNING@0:rpt2[rpt2] Warning Messgae from rpting
UVM_ERROR@0:rpt2[rpt2] Error Message from rpting

UVM_INFOreporting.sv(13)@0:rpt3[rpt3]Info Message:Verbo lvl - UVM_NONE
UVM_ERROR @ 9200 [TIMOUT] Watchdog timeout of '23f0' expired.
```

UVM TRANSACTION

A transaction is data item which is eventually or directly processed by the DUT. The packets, instructions, pixels are data items. In uvm, transactions are extended from uvm_transactions class or uvm_sequence_item class. uvm_transaction is a typedef of uvm_sequence_item.

Example of a transaction:

```
class Packet extends uvm_transaction;
  rand bit [7:0] da;
  rand bit [7:0] sa;
  rand bit [7:0] length;
  rand bit [7:0] data[];
  rand byte fcs;
endclass
```

Core Utilities:

uvm_transaction class is extended from uvm_object. uvm_transaction adds more features line transaction recording , transaction id and timings of the transaction.

The methods used to model, transform or operate on transactions like print, copying, cloning, comparing, recording, packing and unpacking are already defined in uvm_object.

Utilities Method	Description
print	Prints this object's properties
record	Records this object's properties
copy	Returns a deep copy of this object.
compare	Deep compares this data object with the object provided in the rhs (right-hand side) argument.
pack	This methods bitwise-concatenate this object's properties into an array
unpack	Extract property values from an array
clone	Clone method creates and returns an exact copy of this object. The default implementation calls create method followed by copy method.
create	This method allocates a new object of the same type as this object and returns it via a base uvm_object handle.

FIG:UVM OBJECT UTILITIES

User Defined Implementations:

User should define these methods in the transaction using do_<method_name> and call them using <method_name>. Following table shows calling methods and user-defined hook do_<method_name> methods. Clone and create methods, does not use hook methods concepts.

Functionality	User defined hook methods	Calling methods
Printing	do_print	print
		sprint
		convert2string
Recording	do_record	record
Copying	do_copy	copy
Comparing	do_compare	compare
Packing	do_pack	pack
		pack_bytes
		pack_ints
Unpacking	do_unpack	unpack
		unpack_bytes
		unpack_ints
Cloning	clone	clone
Create	create	create

Shorthand Macros:

Using the field automation concept of uvm, all the above defines methods can be defined automatically.
To use these field automation macros, first declare all the data fields, then place the field automation macros between the `uvm_object_utils_begin and `uvm_object_utils_end macros.

Example of field automation macros:

```
class Packet extends uvm_transaction;

  rand bit [7:0] da;
```

```
  rand bit [7:0] sa;
  rand bit [7:0] length;
  rand bit [7:0] data[];
  rand byte fcs;

  `uvm_object_utils_begin(Packet)
  `uvm_field_int(da, UVM_ALL_ON|UVM_NOPACK)
  `uvm_field_int(sa, UVM_ALL_ON|UVM_NOPACK)
  `uvm_field_int(length, UVM_ALL_ON|UVM_NOPACK)
  `uvm_field_array_int(data, UVM_ALL_ON|UVM_NOPACK)
  `uvm_field_int(fcs, UVM_ALL_ON|UVM_NOPACK)
 `uvm_object_utils_end

endclass.
```

For most of the data types in systemverilog, uvm defined corresponding field automation macros. Following table shows all the field automation macros.

Type	Macros
Scalar	`uvm_field_int `uvm_field_object `uvm_field_string `uvm_field_enum `uvm_field_real `uvm_field_event
Static Array	`uvm_field_sarray_int `uvm_field_sarray_object `uvm_field_sarray_string `uvm_field_sarray_enum
Dynamic array	`uvm_field_array_int `uvm_field_array_object `uvm_field_array_string `uvm_field_array_enum
Queue	`uvm_field_queue_int `uvm_field_queue_object `uvm_field_queue_string `uvm_field_queue_enum
Associative array with string index	`uvm_field_aa_int_string `uvm_field_aa_object_string `uvm_field_aa_string_string
Associative array with integral index	`uvm_field_aa_object_int `uvm_field_aa_int_int `uvm_field_aa_int_int_unsigned `uvm_field_aa_int_integer `uvm_field_aa_int_integer_unsigned `uvm_field_aa_int_byte `uvm_field_aa_int_byte_unsigned `uvm_field_aa_int_shortint `uvm_field_aa_int_shortint_unsigned `uvm_field_aa_int_longint `uvm_field_aa_int_longint_unsigned `uvm_field_aa_int_key `uvm_field_aa_int_enumkey

Each `uvm_field_* macro has at least two arguments: ARG and FLAG.
ARG is the instance name of the variable and FLAG is used to control the field usage in core utilities operation.

Following table shows uvm field automation flags:

FLAG	DESCRIPTION
UVM_ALL_ON	Set all operations
UVM_DEFAULT	Use the default flag settings.
UVM_NOCOPY	Do not copy this field.
UVM_NOCOMPARE	Do not compare this field.
UVM_NOPRINT	Do not print this field.
UVM_NODEFPRINT	Do not print the field if it is the same as its
UVM_NOPACK	Do not pack or unpack this field.
UVM_PHYSICAL	Treat as a physical field.
UVM_ABSTRACT	Treat as an abstract field.
UVM_READONLY	Do not allow setting of this field from the set * local methods.

By default, FLAG is set to UVM_ALL_ON. All these flags can be ored. Using NO_* flags, can turn of particular field usage in a paerticuler method. NO_* flags takes precedency over other flags.

Example of Flags:

```
`uvm_field_int(da, UVM_ALL_ON|UVM_NOPACK)
```

The above macro will use the field "da" in all utilities methods except Packing and unpacking methods.

Lets see a example:

In the following example, all the utility methods are defined using field automation macros except Packing and unpacking methods. Packing and unpacking methods are done in do_pack() amd do_unpack() method.

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

//Define the enumerated types for packet types
typedef enum { GOOD_FCS, BAD_FCS } fcs_kind_t;

class Packet extends uvm_transaction;
```

```
rand fcs_kind_t fcs_kind;

rand bit [7:0] length;
rand bit [7:0] da;
rand bit [7:0] sa;
rand bit [7:0] data[];
rand byte fcs;

constraint payload_size_c { data.size inside { [1 : 6]};}

constraint length_c { length == data.size; }

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

function void post_randomize();
  if(fcs_kind == GOOD_FCS)
    fcs = 8'b0;
  else
    fcs = 8'b1;
    fcs = cal_fcs();
endfunction : post_randomize

///// method to calculate the fcs /////
virtual function byte cal_fcs;
  integer i;
  byte result ;
  result = 0;
  result = result ^ da;
  result = result ^ sa;
  result = result ^ length;
  for (i = 0;i< data.size;i++)
    result = result ^ data[i];
    result = fcs ^ result;
  return result;
endfunction : cal_fcs

`uvm_object_utils_begin(Packet)
`uvm_field_int(da, UVM_ALL_ON|UVM_NOPACK)
`uvm_field_int(sa, UVM_ALL_ON|UVM_NOPACK)
`uvm_field_int(length, UVM_ALL_ON|UVM_NOPACK)
`uvm_field_array_int(data, UVM_ALL_ON|UVM_NOPACK)
`uvm_field_int(fcs, UVM_ALL_ON|UVM_NOPACK)
```

```
  `uvm_object_utils_end

  function void do_pack(uvm_packer packer);
    super.do_pack(packer);
    packer.pack_field_int(da,$bits(da));
    packer.pack_field_int(sa,$bits(sa));
    packer.pack_field_int(length,$bits(length));
    foreach(data[i])
      packer.pack_field_int(data[i],8);
      packer.pack_field_int(fcs,$bits(fcs));
  endfunction : do_pack

  function void do_unpack(uvm_packer packer);
    int sz;
    super.do_pack(packer);

    da = packer.unpack_field_int($bits(da));
    sa = packer.unpack_field_int($bits(sa));
    length = packer.unpack_field_int($bits(length));

    data.delete();
    data = new[length];
    foreach(data[i])
    data[i] = packer.unpack_field_int(8);
    fcs = packer.unpack_field_int($bits(fcs));
  endfunction : do_unpack

endclass : Packet


////////////////////////////////////////////////////////////
//// Test to check the packet implementation ////
////////////////////////////////////////////////////////////
module test;

  Packet pkt1 = new("pkt1");
  Packet pkt2 = new("pkt2");
  byte unsigned pkdbytes[];

  initial
    repeat(10)
    if(pkt1.randomize)   begin
      $display(" Randomization Sucessesfull.");
      pkt1.print();
```

```
    uvm_default_packer.use_metadata = 1;
    void'(pkt1.pack_bytes(pkdbytes));
    $display("Size of pkd bits %d",pkdbytes.size());
    pkt2.unpack_bytes(pkdbytes);
    pkt2.print();
    if(pkt2.compare(pkt1))
      $display(" Packing,Unpacking and compare worked");
    else
      $display(" *** Something went wrong in Packing or Unpacking or compare *** \n \n");
  end
  else
    $display(" *** Randomization Failed ***");

endmodule
```

Download the source code

uvm_transaction.tar
Browse the code in uvm_transaction.tar

Command to run the simulation

VCS Users : make vcs
Questa Users: make questa

Log report:

```
 Randomization Sucessesfull.
----------------------------------------------------------------------
Name                    Type              Size                Value
----------------------------------------------------------------------
pkt1                    Packet            -                   pkt1@3
  da                    integral          8                    'h1d
  sa                    integral          8                    'h26
  length                integral          8                     'h5
  data                  da(integral)      5                       -
    [0]                 integral          8                    'hb1
    [1]                 integral          8                    'h3f
    [2]                 integral          8                    'h9e
    [3]                 integral          8                    'h38
    [4]                 integral          8                    'h8d
  fcs                   integral          8                    'h9b
----------------------------------------------------------------------
Size of pkd bits          9
----------------------------------------------------------------------
Name                    Type              Size                Value
----------------------------------------------------------------------
pkt2                    Packet            -                   pkt2@5
  da                    integral          8                    'h1d
  sa                    integral          8                    'h26
  length                integral          8                     'h5
  data                  da(integral)      5                       -
    [0]                 integral          8                    'hb1
    [1]                 integral          8                    'h3f
    [2]                 integral          8                    'h9e
    [3]                 integral          8                    'h38
    [4]                 integral          8                    'h8d
  fcs                   integral          8                    'h9b
----------------------------------------------------------------------
 Packing,Unpacking and compare worked
```

UVM CONFIGURATION

Configuration is a mechanism in UVM that higher level components in a hierarchy can configure the lower level components variables. Using set_config_* methods, user can configure integer, string and objects of lower level components. Without this mechanism, user should access the lower level component using hierarchy paths, which restricts reusability. This mechanism can be used only with components. Sequences and transactions cannot be configured using this mechanism. When set_config_* method is called, the data is stored w.r.t strings in a table. There is also a global configuration table.
Higher level component can set the configuration data in level component table. It is the responsibility of the lower level component to get the data from the component table and update the appropriate table.

Set_config_* Methods:

Following are the method to configure integer , string and object of uvm_object based class respectively.

```
function void set_config_int (string inst_name,
                              string field_name,
                              uvm_bitstream_t value)

function void set_config_string (string inst_name,
                                 string field_name,
                                 string value)

function void set_config_object (string inst_name,
                                 string field_name,
                                  uvm_object value, bit clone = 1)
```

Arguments description:

- string inst_name: Hierarchical string path.
- string field_name: Name of the field in the table.
- bitstream_t value: In set_config_int, a integral value that can be anything from 1 bit to 4096 bits.
- bit clone : If this bit is set then object is cloned.

inst_name and field_name are strings of hierarchal path. They can include wile card "*" and "?" characters. These methods must be called in build phase of the

component.

"*" matches zero or more characters
"?" matches exactly one character

Some examples:

"*" -All the lower level components.

"*abc" -All the lower level components which ends with "abc".
Example: "xabc","xyabc","xyzabc" ....

"abc*" -All the lower level components which starts with "abc".
Example: "abcx","abcxy","abcxyz" ....

"ab?" -All the lower level components which start with "ab" , then followed by one more character.
Example: "abc","abb","abx" ....

"?bc" -All the lower level components which start with any one character ,then followed by "c".
Example: "abc","xbc","bbc" ....

"a?c" -All the lower level components which start with "a" , then followed by one more character and then followed by "c".
Example: "abc","aac","axc" â€¦

There are two ways to get the configuration data:
1)Automatic : Using Field macros
2)Manual : using gte_config_* methods.

Automatic Configuration:

To use the atomic configuration, all the configurable fields should be defined using uvm component field macros and uvm component utilities macros.

uvm component utility macros:
For non parameterized classes
`uvm_component_utils_begin(TYPE)
`uvm_field_* macro invocations here

`uvm_component_utils_end

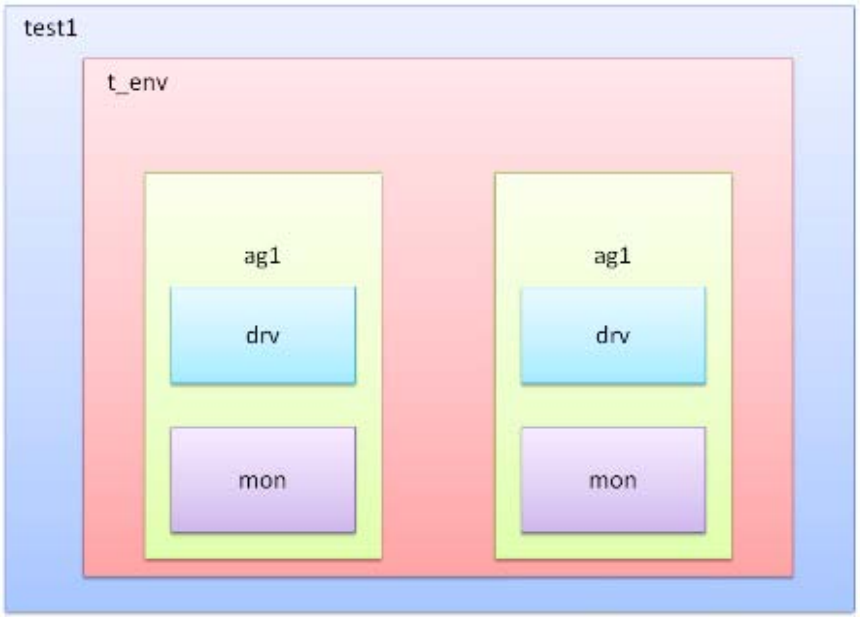
For parameterized classes.

`uvm_component_param_utils_begin(TYPE)
`uvm_field_* macro invocations here
`uvm_component_utils_end

For UVM Field macros, Refer to link
UVM_TRANSACTION

Example:
Following example is from link
UVM_TESTBENCH

2 Configurable fields, a integer and a string are defined in env, agent, monitor and driver classes. Topology of the environment using these classes is



Driver class Source Code:

Similar to driver class, all other components env, agent and monitor are define.

```
class driver extends uvm_driver;

  integer int_cfg;
  string str_cfg;

  `uvm_component_utils_begin(driver)
```

```
  `uvm_field_int(int_cfg, UVM_DEFAULT)
  `uvm_field_string(str_cfg, UVM_DEFAULT)
  `uvm_component_utils_end

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

  function void build();
    super.build();
  endfunction

endclass
```

Testcase:

Using set_config_int() and set_config_string() configure variables at various hierarchal locations.

```
//t_env.ag1.drv.int_cfg
//t_env.ag1.mon.int_cfg
set_config_int("*.ag1.*","int_cfg",32);

//t_env.ag2.drv
set_config_int("t_env.ag2.drv","int_cfg",32);

//t_env.ag2.mon
set_config_int("t_env.ag2.mon","int_cfg",32);

//t_env.ag1.mon.str_cfg
//t_env.ag2.mon.str_cfg
//t_env.ag1.drv.str_cfg
//t_env.ag2.drv.str_cfg
set_config_string("*.ag?.*","str_cfg","pars");

//t_env.str_cfg
set_config_string("t_env","str_cfg","abcd");
```

Download the source code

uvm_configuration_1.tar
Browse the code in uvm_configuration_1.tar

Command to run the simulation

VCS Users : make vcs
Questa Users: make questa

From the above log report of th example, we can see the variables int_cfg and str_cfg of all the components and they are as per the configuration setting from the testcase.

Manual Configurations:

Using get_config_* methods, user can get the required data if the data is available in the table.

Following are the method to get configure data of type integer , string and object of uvm_object based class respectively.

```
function bit get_config_int (string field_name,
                             inout uvm_bitstream_t value)

function bit get_config_string (string field_name,
                                inout string value)

function bit get_config_object (string field_name,
                                inout uvm_object value,
                                input bit clone = 1)
```

If a entry is found in the table with "field_name" then data will be updated to "value" argument . If entry is not found, then the function returns "0". So when these methods are called, check the return value.

Example:

Driver class code:

```
class driver extends uvm_driver;

  integer int_cfg;
  string str_cfg;

  `uvm_component_utils(driver)
```

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

  function void build();
    super.build();
    void'(get_config_int("int_cfg",int_cfg));
    void'(get_config_string("str_cfg",str_cfg));
    uvm_report_info(get_full_name(),
    $psprintf("int_cfg %0d : str_cfg %0s ",int_cfg,str_cfg),UVM_LOW);
  endfunction

endclass
```

Download the source code

uvm_configuration_2.tar
Browse the code in uvm_configuration_2.tar

Command to run the simulation

VCS Users : make vcs
Questa Users: make questa

Log file

```
UVM_INFO @ 0: uvm_test_top.t_env
int_cfg x : str_cfg abcd
UVM_INFO @ 0: uvm_test_top.t_env.ag1
int_cfg x : str_cfg
UVM_INFO @ 0: uvm_test_top.t_env.ag1.drv
int_cfg 32 : str_cfg pars
UVM_INFO @ 0: uvm_test_top.t_env.ag1.mon
int_cfg 32 : str_cfg pars
UVM_INFO @ 0: uvm_test_top.t_env.ag2
int_cfg x : str_cfg
UVM_INFO @ 0: uvm_test_top.t_env.ag2.drv
int_cfg 32 : str_cfg pars
UVM_INFO @ 0: uvm_test_top.t_env.ag2.mon
int_cfg 32 : str_cfg pars
```

Configuration Setting Members:

print_config_settings

```
function void print_config_settings
( string field = "",
uvm_component comp = null,
bit recurse = 0 )
```

This method prints all configuration information for this component.

If "field" is specified and non-empty, then only configuration settings matching that field, if any, are printed. The field may not contain wildcards. If "recurse" is set, then information for all children components are printed recursively.

print_config_matches

```
static bit print_config_matches = 0
```

Setting this static variable causes get_config_* to print info about matching configuration settings as they are being applied. These two members will be helpful to know while debugging.

Download the source code

uvm_configuration_3.tar
Browse the code in uvm_configuration_3.tar

Command to run the simulation

VCS Users : make vcs
Questa Users: make questa

Log file

When print_config_settings method is called

```
uvm_test_top.t_env.ag1.drv
uvm_test_top.*.ag1.* int_cfg int 32
uvm_test_top.t_env.ag1.drv.rsp_port
uvm_test_top.*.ag?.* str_cfg string pars
uvm_test_top.t_env.ag1.drv.rsp_port
uvm_test_top.*.ag1.* int_cfg int 32
uvm_test_top.t_env.ag1.drv.sqr_pull_port
uvm_test_top.*.ag?.* str_cfg string pars
uvm_test_top.t_env.ag1.drv.sqr_pull_port
uvm_test_top.*.ag1.* int_cfg int 32
```

When print_config_matches is set to 1.

```
UVM_INFO @ 0: uvm_test_top.t_env [auto-configuration]
Auto-configuration matches for component uvm_test_top.t_env (env).
Last entry for a given field takes precedence.

Config set from Instance Path Field name Type Value
------------------------------------------------------------------------------
uvm_test_top(test1) uvm_test_top.t_env str_cfg string abcd
```

UVM FACTORY

The factory pattern is an well known object-oriented design pattern. The factory method design pattern defining a separate method for creating the objects. , whose subclasses can then override to specify the derived type of object that will be created.

Using this method, objects are constructed dynamically based on the specification type of the object. User can alter the behavior of the pre-build code without modifying the code. From the testcase, user from environment or testcase can replace any object which is at any hierarchy level with the user defined object.

For example: In your environment, you have a driver component. You would like the extend the driver component for error injection scenario. After defining the extended driver class with error injection, how will you replace the base driver component which is deep in the hierarchy of your environment ? Using hierarchical path, you could replace the driver object with the extended driver. This could not be easy if there are many driver objects. Then you should also take care of its connections with the other components of testbenchs like scoreboard etc.

One more example: In your Ethernet verification environment, you have different drivers to support different interfaces for 10mbps,100mps and 1G. Now you want to reuse the same environment for 10G verification. Inside somewhere deep in the hierarchy, while building the components, as a driver components ,your current environment can only select 10mmps/100mps/1G drivers using configuration settings. How to add one more driver to the current drivers list of drivers so that from the testcase you could configure the environment to work for 10G.

Using the uvm fatroy, it is very easy to solve the above two requirements. Only classs extended from uvm_object and uvm_component are supported for this.

There are three basic steps to be followed for using uvm factory.

- 1) Registration
- 2) Construction
- 3) Overriding

The factory makes it is possible to override the type of uvm component /object or instance of a uvm component/object in2 ways. They are based on uvm component/object type or uvm compoenent/object name.

Registration:

While defining a class , its type has to be registered with the uvm factory. To do this job easier, uvm has predefined macros.

```
`uvm_component_utils(class_type_name)
`uvm_component_param_utils(class_type_name #(params))
`uvm_object_utils(class_type_name)
`uvm_object_param_utils(class_type_name #(params))
```

For uvm_*_param_utils are used for parameterized classes and other two macros for non-parameterized class. Registration is required for name-based overriding , it is not required for type-based overriding.

EXAMPLE: Example of above macros

```
class packet extends uvm_object;
  `uvm_object_utils(packet)
endclass

class packet #(type T=int, int mode=0) extends uvm_object;
  `uvm_object_param_utils(packet #(T,mode))
endclass

class driver extends uvm_component;
  `uvm_component_utils(driver)
endclass

class monitor #(type T=int, int mode=0) extends uvm_component;
  `uvm_component_param_utils(driver#(T,mode))
endclass
```

Construction:

To construct a uvm based component or uvm based objects, static method create() should be used. This function constructs the appropriate object based on the overrides and constructs the object and returns it. So while constructing the uvm based components or uvm based objects , do not use new() constructor.

Syntax :

```
static function T create(string name,
                         uvm_component parent,
                         string context = " ")
```

The Create() function returns an instance of the component type, T, represented by this proxy, subject to any factory overrides based on the context provided by the parents full name. The context argument, if supplied, supersedes the parents context. The new instance will have the given leaf name and parent.

EXAMPLE:

```
class_type object_name;

object_name = clss_type::type_id::creat("object_name",this);
```

For uvm_object based classes, doesnt need the parent handle as second argument.

Overriding:

If required, user could override the registered classes or objects. User can override based of name string or class-type.

There are 4 methods defined for overriding:

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

The above method is used to override the object instances of "original_type" with "override_type" . "override_type" is extended from"original_type".

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

Original_type_name and override_type_name are the class names which are registered in the factory. All the instances of objects with name "Original_type_name" will be overriden with objects of name "override_type_name" using set_inst_override_by_name() method.

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

Using the above method, request to create an object of original_type can be overriden with override_type.

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

Using the above method, request to create an object of original_type_name can be overriden with override_type_name.

When multiple overrides are done , then using the argument "replace" , we can control whether to override the previous override or not. If argument "replace" is 1, then previous overrides will be replaced otherwise, previous overrides will remain.

print() method, prints the state of the uvm_factory, registered types, instance overrides, and type overrides.

Now we will see a complete example. This example is based on the environment build in topic UVM TESTBENCH . Refer to that section for more information about this example.

Lets look at the 3 steps which I discussed above using the example defined in UVM TESTBENCH

1) Registration

In all the class, you can see the macro `uvm_component_utils(type_name)

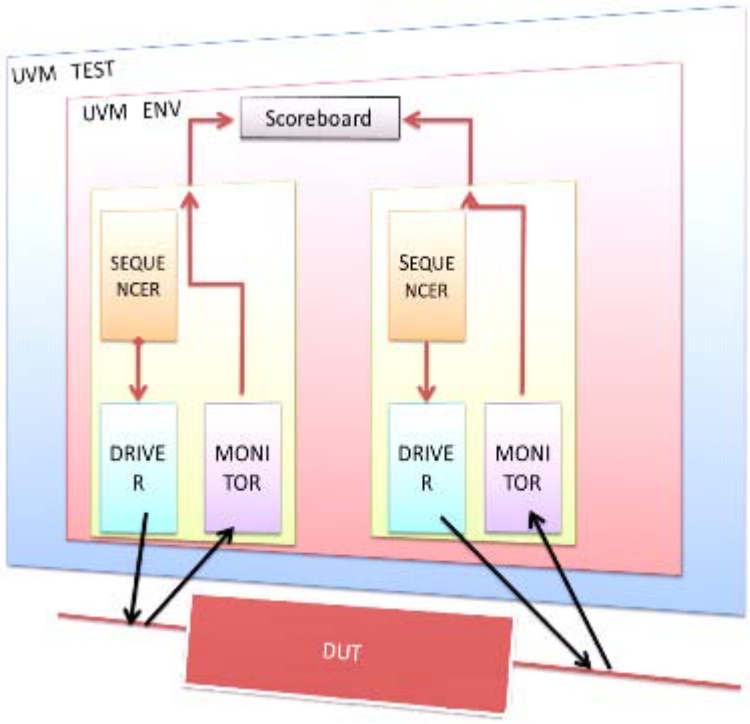
2) Construction

In file agant.sv file, monitor and driver are constructed using create() method.

```
mon = monitor::type_id::create("mon",this);
drv = driver::type_id::create("drv",this);
```

3)In this example, a one testcase is already developed in topic UVM_TESTBENCH. There are no over rides in this test case.

Topology of this test environment is shown below.



In this example, there is one driver class and one monitor class. In this testcase , By extending driver class , we will define driver_2 class and by extending monitor class, we will define monitor_2 class.

From the testcase , Using set_type_override_by_type, we will override driver with driver_2 and Using set_type_override_by_name, we will override monitor with monitor_2.

To know about the overrides which are done, call factory.print() method of factory class.

```
class driver_2 extends driver;

  `uvm_component_utils(driver_2)

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

endclass
```

```
class monitor_2 extends monitor;

  `uvm_component_utils(monitor_2)

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

endclass


class test_factory extends uvm_test;

  `uvm_component_utils(test_factory)
  env t_env;

  function new (string name="test1", uvm_component parent=null);
    super.new (name, parent);

    factory.set_type_override_by_type(driver::get_type(),driver_2::get_type(),"*");
    factory.set_type_override_by_name("monitor","monitor_2","*");
    factory.print();
    t_env = new("t_env",this);
  endfunction : new

  function void end_of_elaboration();
    uvm_report_info(get_full_name(),"End_of_elaboration", UVM_LOW);
    print();
  endfunction : end_of_elaboration

  task run ();
    #1000;
    global_stop_request();
  endtask : run

endclass
```

Download the example:

uvm_factory.tar
Browse the code in uvm_factory.tar

Command to simulate

Command to run the example with the testcase which is defined above:
VCS Users : make vcs
Questa Users: make questa

Method factory.print() displayed all the overrides as shown below in the log file.

```
#### Factory Configuration (*)

No instance overrides are registered with this factory

Type Overrides:

  Requested Type  Override Type
  --------------  -------------
  driver          driver_2
  monitor         monitor_2
```

In the below text printed by print_topology() method ,we can see overridden driver and monitor.

```
----------------------------------------------------------------------
Name                    Type                Size                Value
----------------------------------------------------------------------
uvm_test_top            test_factory        -          uvm_test_top@2
  t_env                 env                 -                 t_env@4
    ag1                 agent               -                   ag1@6
      drv               driver_2            -                   drv@12
        rsp_port        uvm_analysis_port   -              rsp_port@16
        sqr_pull_port   uvm_seq_item_pull_+ -         sqr_pull_port@14
      mon               monitor_2           -                   mon@10
    ag2                 agent               -                   ag2@8
      drv               driver_2            -                   drv@20
        rsp_port        uvm_analysis_port   -              rsp_port@24
        sqr_pull_port   uvm_seq_item_pull_+ -         sqr_pull_port@22
      mon               monitor_2           -                   mon@18
```

In the below text printed by print_topology() method ,with testcase test1 which does note have overrides.

Command to run this example with test1 is
VCS Users : make vcs
Questa Users: make questa

```
----------------------------------------------------------------------
Name                    Type                Size                Value
----------------------------------------------------------------------
uvm_test_top            test1               -           uvm_test_top@2
  t_env                 env                 -                  t_env@4
    ag1                 agent               -                    ag1@6
      drv               driver              -                  drv@12
        rsp_port        uvm_analysis_port   -              rsp_port@16
        sqr_pull_port   uvm_seq_item_pull_+ -         sqr_pull_port@14
      mon               monitor             -                  mon@10
    ag2                 agent               -                    ag2@8
      drv               driver              -                  drv@20
        rsp_port        uvm_analysis_port   -              rsp_port@24
        sqr_pull_port   uvm_seq_item_pull_+ -         sqr_pull_port@22
      mon               monitor             -                  mon@18
----------------------------------------------------------------------
```

UVM SEQUENCE 1

Introduction

A sequence is a series of transaction. User can define the complex stimulus. sequences can be reused, extended, randomized, and combined sequentially and hierarchically in various ways.

For example, for a processor, lets say PUSH_A,PUSH_B,ADD,SUB,MUL,DIV and POP_C are the instructions. If the instructions are generated randomly, then to excursing a meaningful operation like "adding 2 variables" which requires a series of transaction "PUSH_A PUSH_B ADD POP_C " will take longer time. By defining these series of "PUSH_A PUSH_B ADD POP_C ", it would be easy to exercise the DUT.

Advantages of uvm sequences :

- Sequences can be reused.
- Stimulus generation is independent of testbench.
- Easy to control the generation of transaction.
- Sequences can be combined sequentially and hierarchically.

A complete sequence generation requires following 4 classes.

1- Sequence item.
2- Sequence
3- Sequencer
4- Driver

- uvm_sequence_item :

User has to define a transaction by extending uvm_sequence_item. uvm_sequence_item class provides the basic functionality for objects, both sequence items and sequences, to operate in the sequence mechanism. For more information about uvm_sequence_item Refer to link

UVM_TRANSACTION

- uvm_sequence:

User should extend uvm_sequence class and define the construction of sequence of transactions. These transactions can be directed, constrained randomized or fully randomized. The uvm_sequence class provides the interfaces necessary in order to create streams of sequence items and/or other sequences.

```
virtual class uvm_sequence #(
  type REQ = uvm_sequence_item,
  type RSP = REQ
)
```

uvm_sequencer:

uvm_sequencer is responsible for the coordination between sequence and driver. Sequencer sends the transaction to driver and gets the response from the driver. The response transaction from the driver is optional. When multiple sequences are running in parallel, then sequencer is responsible for arbitrating between the parallel sequences. There are two types of sequencers : uvm_sequencer and uvm_push_sequencer

```
class uvm_sequencer #(
  type REQ = uvm_sequence_item,
  type RSP = REQ
)

class uvm_push_sequencer #(
  type REQ = uvm_sequence_item,
  type RSP = REQ
)
```

uvm driver:

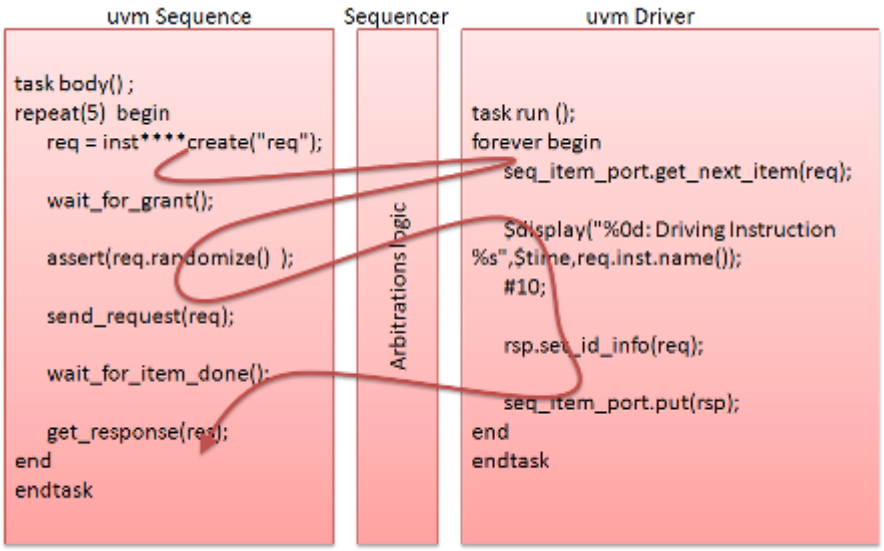
User should extend uvm_driver class to define driver component. uvm driver is a component that initiate requests for new transactions and drives it to lower level components. There are two types of drivers: uvm_driver and uvm_push_driver.

```
class uvm_driver #(
  type REQ = uvm_sequence_item,
  type RSP = REQ
)

class uvm_push_driver #(
  type REQ = uvm_sequence_item,
  type RSP = REQ
)
```

In pull mode , uvm_sequencer is connected to uvm_driver , in push mode uvm_push_sequencer is connectd to uvm_push_driver.

uvm_sequencer and uvm_driver are parameterized components with request and response transaction types. REQ and RSP types by default are uvm_sequence_type types. User can specify REQ and RSP of different transaction types. If user specifies only REQ type, then RSP will be REQ type.

Sequence And Driver Communication:



The above image shows how a transaction from a sequence is sent to driver and the response from the driver is sent to sequencer. There are multiple methods called during this operation.

First when the body() method is called

1) A transaction is created using "create()" method. If a transaction is created using "create()" method, then it can be overridden if required using uvm factory.

2) After a transaction is created, wait_for_grant() method is called. This method is blocking method.

3) In the run task of the driver, when "seq_item_port.get_next_item()" is called, then the sequencer un blocks wait_for_grant() method. If more than one sequence is getting executed by sequencer, then based on arbitration rules, un blocks the wait_for_grant() method.

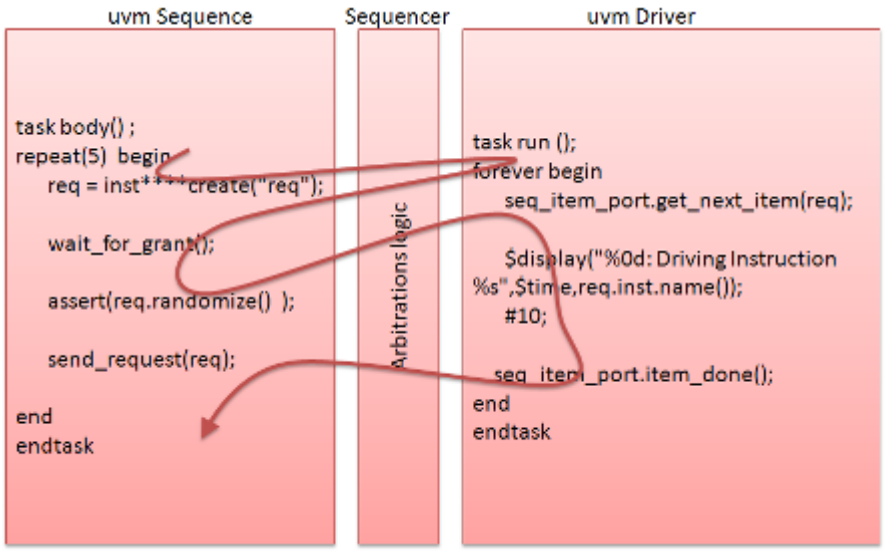
4) After the wait_for_grant() un blocks, then transaction can be randomized, or its properties can be filled directly. Then using the send_request() method, send the transaction to the driver.

5) After calling the send_request() method, "wait_for_item_done()" method is called. This is a blocking method and execution gets blocks at this method call.

6) The transaction which is sent from sequence , in the driver this transaction is available as "seq_item_port.get_next_item(req)" method argument. Then driver can drive this transaction to bus or lower level.

7) Once the driver operations are completed, then by calling "seq_item_port.put(rsp)", wait_for_item_done() method of sequence gest unblocked. Using get_responce(res), the response transaction from driver is taken by sequence and processes it.

After this step, again the steps 1 to 7 are repeated five times.



If a response from driver is not required, then steps 5,6,7 can be skipped and item_done() method from driver should be called as shown in above image.

Simple Example

Lest write an example: This is a simple example of processor instruction. Various instructions which are supported by the processor are PUSH_A,PUSH_B,ADD,SUB,MUL,DIV and POP_C.

Sequence Item

1) Extend uvm_sequence_item and define instruction class.

```
class instruction extends uvm_sequence_item;
```

2) Define the instruction as enumerated types and declare a variable of instruction enumerated type.

```
typedef enum {PUSH_A,PUSH_B,ADD,SUB,MUL,DIV,POP_C} inst_t;
rand inst_t inst;
```

3) Define operational method using uvm_field_* macros.

```
`uvm_object_utils_begin(instruction)
`uvm_field_enum(inst_t,inst, UVM_ALL_ON)
`uvm_object_utils_end
```

4) Define the constructor.

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

Sequence item code:

```
class instruction extends uvm_sequence_item;
  typedef enum {PUSH_A,PUSH_B,ADD,SUB,MUL,DIV,POP_C} inst_t;
  rand inst_t inst;

  `uvm_object_utils_begin(instruction)
  `uvm_field_enum(inst_t,inst, UVM_ALL_ON)
  `uvm_object_utils_end

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

endclass
```

Sequence

We will define a operation addition using uvm_sequence. The instruction sequence should be "PUSH A PUSH B ADD POP C".

1) Define a sequence by extending uvm_sequence. Set REQ parameter to "instruction" type.

```
class operation_addition extends uvm_sequence #(instruction);
```

2) Define the constructor.

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

3) Lets name the sequencer which we will develop is "instruction_sequencer".
Using the `uvm_sequence_utils macro, register the "operation_addition" sequence with "instruction_sequencer" sequencer. This macro adds the sequence to the sequencer list. This macro will also register the sequence for factory overrides.

```
`uvm_sequence_utils(operation_addition, instruction_sequencer)
```

4) In the body() method, first call wait_for_grant(), then construct a transaction and set the instruction enum to PUSH_A . Then send the transaction to driver using send_request() method. Then call the wait_for_item_done() method. Repeat the above steps for other instructions PUSH_B, ADD and POP_C.

For construction of a transaction, we will use the create() method.

```
virtual task body();
  req = instruction::type_id::create("req");
  wait_for_grant();
  assert(req.randomize() with {
    inst == instruction::PUSH_A;
  });
  send_request(req);
  wait_for_item_done();
  //get_response(res); This is optional. Not using in this example.

  req = instruction::type_id::create("req");
  wait_for_grant();
  req.inst = instruction::PUSH_B;
  send_request(req);
  wait_for_item_done();
  //get_response(res);

  req = instruction::type_id::create("req");
  wait_for_grant();
  req.inst = instruction::ADD;
  send_request(req);
  wait_for_item_done();
  //get_response(res);

  req = instruction::type_id::create("req");
  wait_for_grant();
  req.inst = instruction::POP_C;
  send_request(req);
  wait_for_item_done();
  //get_response(res);
endtask
```

Sequence code

```
class operation_addition extends uvm_sequence #(instruction);

  instruction req;

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

  `uvm_sequence_utils(operation_addition, instruction_sequencer)

  virtual task body();
    req = instruction::type_id::create("req");
    wait_for_grant();
    assert(req.randomize() with {
      inst == instruction::PUSH_A;
    });
    send_request(req);
    wait_for_item_done();
    //get_response(res); This is optional. Not using in this example.

    req = instruction::type_id::create("req");
    wait_for_grant();
    req.inst = instruction::PUSH_B;
    send_request(req);
    wait_for_item_done();
    //get_response(res);

    req = instruction::type_id::create("req");
    wait_for_grant();
    req.inst = instruction::ADD;
    send_request(req);
    wait_for_item_done();
    //get_response(res);

    req = instruction::type_id::create("req");
    wait_for_grant();
    req.inst = instruction::POP_C;
    send_request(req);
    wait_for_item_done();
    //get_response(res);
  endtask
```

```
endclass
```

Sequencer:

uvm_sequence has a property called default_sequence. Default sequence is a sequence which will be started automatically. Using set_config_string, user can override the default sequence to any user defined sequence, so that when a sequencer is started, automatically a user defined sequence will be started. If over rides are not done with user defined sequence, then a random transaction are generated. Using "start_default_sequence()" method, "default_sequence" can also be started.

uvm sequencer has seq_item_export and res_export tlm ports for connecting to uvm driver.

1) Define instruction_sequencer by extending uvm_sequencer.

```
class instruction_sequencer extends uvm_sequencer #(instruction);
```

2) Define the constructor.
Inside the constructor, place the macro `uvm_update_sequence_lib_and_item().
This macro creates 3 predefined sequences. We will discuss about the predefined sequences in next section.

```
function new (string name, uvm_component parent);
  super.new(name, parent);
  `uvm_update_sequence_lib_and_item(instruction)
endfunction
```

3) Place the uvm_sequencer_utils macro. This macro registers the sequencer for factory overrides.

```
  `uvm_sequencer_utils(instruction_sequencer)
```

Sequencer Code;

```
class instruction_sequencer extends uvm_sequencer #(instruction);

  function new (string name, uvm_component parent);
    super.new(name, parent);
    `uvm_update_sequence_lib_and_item(instruction)
  endfunction

  `uvm_sequencer_utils(instruction_sequencer)
endclass
```

Driver:

uvm_driver is a class which is extended from uvm_componenet. This driver is used in pull mode. Pull mode means, driver pulls the transaction from the sequencer when it requires.

uvm driver has 2 TLM ports.

1) Seq_item_port: To get a item from sequencer, driver uses this port. Driver can also send response back using this port.

2) Rsp_port : This can also be used to send response back to sequencer.

Seq_item_port methods:

Method	Type	Description
Task get_next_item(output REQ req_arg);	Blocking	Retrieves the next available item from a sequence.
task try_next_item(output REQ req_arg)	Non blocking	Retrieves the next available item from a sequence if one is available.
void item_done(RSP rsp_arg = null)	Non Blocking	Indicates that the request is completed to the sequencer.
task get(output REQ req_arg)	Blocking	Retrieves the next available item from a sequence.
task put(RSP rsp_arg)	Non Blocking	Sends a response back to the sequence that issued the request.
task peek(output REQ req_arg)	Blocking	Returns the current request item if one is in the sequencer fifo.

Lets implement a driver:

1) Define a driver which takes the instruction from the sequencer and does the processing. In this example we will just print the instruction type and wait for some delay.

```
class instruction_driver extends uvm_driver #(instruction);
```

2) Place the uvm_component_utils macro to define virtual methods like get_type_name and create.

```
`uvm_component_utils(instruction_driver)
```

3) Define Constructor method.

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

4) Define the run() method. Run() method is executed in the "run phase". In this methods, transactions are taken from the sequencer and drive them on to dut interface or to other components.
Driver class has a port "seq_item_port". Using the method seq_item_port.get_next_item(), get the transaction from the sequencer and process it. Once the processing is done, using the item_done() method, indicate to the sequencer that the request is completed. In this example, after taking the transaction, we will print the transaction and wait for 10 units time.

```
task run ();
  while(1) begin
    seq_item_port.get_next_item(req);
    $display("%0d: Driving Instruction %s",$time,req.inst.name());
    #10;
    seq_item_port.item_done();
  end
endtask

endclass
```

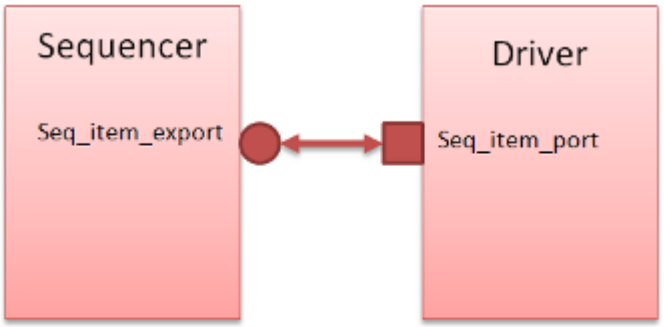
Driver class code:

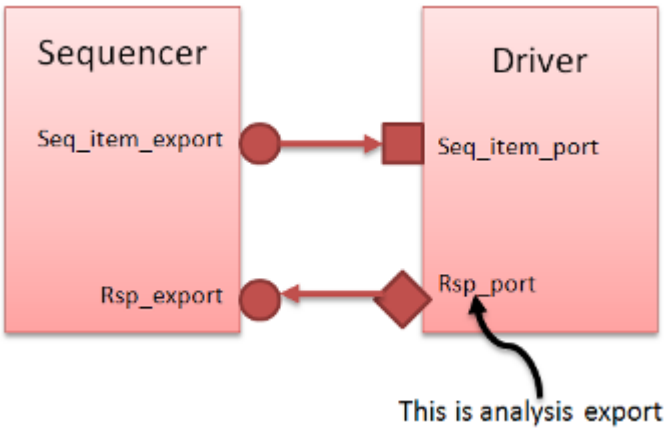
```
class instruction_driver extends uvm_driver #(instruction);

  // Provide implementations of virtual methods such as get_type_name and create
  `uvm_component_utils(instruction_driver)

  // Constructor
  function new (string name, uvm_component parent);
    super.new(name, parent);
  endfunction

  task run ();
    forever begin
      seq_item_port.get_next_item(req);
      $display("%0d: Driving Instruction %s",$time,req.inst.name());
```

```
        #10;
        // rsp.set_id_info(req); These two steps are required only if
        // seq_item_port.put(esp); responce needs to be sent back to sequence
        seq_item_port.item_done();
      end
    endtask

endclass
```

Driver And Sequencer Connectivity:

Deriver and sequencer are connected using TLM. uvm_driver has seq_item_port which is used to get the transaction from uvm sequencer. This port is connected to uvm_sequencer seq_item_export Using "<driver>.seq_item_port.connect(<sequencer>.seq_item_export);" driver and sequencer can be connected. Simillarly "res_port" of driver which is used to send response from driver to sequencer is connected to "res_export" of the sequencer using ""<driver>.res_port.connect(<sequencer>.res_export);".





Testcase:

This testcase is used only for the demo purpose of this tutorial session. Actually, the sequencer and the driver and instantiated and their ports are connected in a agent component and used. Lets implement a testcase

1) Take instances of sequencer and driver and construct both components.

```
sequencer = new("sequencer", null);
sequencer.build();
driver = new("driver", null);
driver.build();
```

2)
Connect the seq_item_export to the drivers seq_item_port.

```
driver.seq_item_port.connect(sequencer.seq_item_export);
```

3) Using set_confg_string() method, set the default sequence of the sequencer to "operation_addition". Operation_addition is the sequence which we defined previous.

```
set_config_string("sequencer", "default_sequence", "operation_addition");
```

4) Using the start_default_sequence() method of the sequencer, start the default sequence of the sequencer. In the previous step we configured the addition operation as default sequene. When you run the simulation, you will see the PUSH_A,PUSH_B ADD and POP_C series of transaction.

```
sequencer.start_default_sequence();
```

Testcase Code:

```
module test;

  instruction_sequencer sequencer;
  instruction_driver driver;

  initial begin
    set_config_string("sequencer", "default_sequence", "operation_addition");
    sequencer = new("sequencer", null);
    sequencer.build();
    driver = new("driver", null);
    driver.build();
```

```
    driver.seq_item_port.connect(sequencer.seq_item_export);
    sequencer.print();
    fork
      begin
        run_test();
        sequencer.start_default_sequence();
      end
      #2000 global_stop_request();
    join
  end

endmodule
```

Download the example:

uvm_basic_sequence.tar
Browse the code in uvm_basic_sequence.tar

Command to simulate

VCS Users : make vcs
Questa Users: make questa

Log file Output

```
UVM_INFO @ 0 [RNTST] Running test ...
0: Driving Instruction PUSH_A
10: Driving Instruction PUSH_B
20: Driving Instruction ADD
30: Driving Instruction POP_C
```

From the above log , we can see that transactions are generates as we defined in uvm sequence.

UVM SEQUENCE 2

Pre Defined Sequences:

Every sequencer in uvm has 3 pre defined sequences. They are

- 1)uvm_random_sequence
- 2)uvm_exhaustive_sequence.
- 3)uvm_simple_sequence

All the user defined sequences which are registered by user and the above three predefined sequences are stored in sequencer queue.

id	Sequences[$]
0	uvm_random_sequence
1	uvm_exhaustive_sequence
2	uvm_simple_sequence
3	User defined sequence 1
4	User defined sequence 2
5	User defined sequence 3
.	...
.	...

uvm_random_sequence :

This sequence randomly selects and executes a sequence from the sequencer sequence library, excluding uvm_random_sequence itself, and uvm_exhaustive_sequence. From the above image, from sequence id 2 to till the last sequence, all the sequences are executed randomly. If the "count" variable of the sequencer is set to 0, then non of the sequence is executed. If the "count" variable of the sequencer is set to -1, then some random number of sequences from 0 to "max_random_count" are executed. By default "max_random_count" is set to 10. "Count" and "max_random_count" can be changed using set_config_int().

The sequencer when automatically started executes the sequence which is point by default_sequence. By default default_sequence variable points to uvm_random_sequence.

uvm_exhaustive_sequence:

This sequence randomly selects and executes each sequence from the sequencers

sequence library once in a randc style, excluding itself and uvm_random_sequence.

uvm_simple_sequence:

This sequence simply executes a single sequence item.

In the previous example from UVM_SEQUENCE_1 section.

The print() method of the sequencer in that example printed the following

```
----------------------------------------------------------------------
Name                    Type                Size                Value
----------------------------------------------------------------------
sequencer               instruction_sequen+ -             sequencer@2
  rsp_export            uvm_analysis_export -           rsp_export@4
  seq_item_export       uvm_seq_item_pull_+ -      seq_item_export@28
  default_sequence      string              18     operation_addition
  count                 integral            32                     -1
  max_random_count      integral            32                   'd10
  sequences             array               4                      -
    [0]                 string              19    uvm_random_sequence
    [1]                 string              23   uvm_exhaustive_sequ+
    [2]                 string              19    uvm_simple_sequence
    [3]                 string              18     operation_addition
  max_random_depth      integral            32                    'd4
  num_last_reqs         integral            32                    'd1
  num_last_rsps         integral            32                    'd1
----------------------------------------------------------------------
```

Some observations from the above log:

The count is set to -1. The default sequencer is set to operations_addition. There are 3 predefined sequences and 1 user defined sequence.

Lets look at a example: In the attached example, in file sequence.sv file, there are 4 seuqneces, they are operation_addition, operation_subtraction, operation_multiplication.

In the testcase.sv file, the "default_seuence" is set to "uvm_exhaustive_sequence" using the set_config_string.

set_config_string("sequencer", "default_sequence", "uvm_exhaustive_sequence");

Download the example

uvm_sequence_1.tar
Browse the code in uvm_sequence_1.tar

Command to run the summation

VCS Users : make vcs
Questa Users: make questa

Log File

```
0: Driving Instruction PUSH_B

10: Driving Instruction PUSH_A
20: Driving Instruction PUSH_B
30: Driving Instruction SUB
40: Driving Instruction POP_C

50: Driving Instruction PUSH_A
60: Driving Instruction PUSH_B
70: Driving Instruction MUL
80: Driving Instruction POP_C

90: Driving Instruction PUSH_A
100: Driving Instruction PUSH_B
110: Driving Instruction ADD
120: Driving Instruction POP_C
```

From the above log , we can see that all the 3 user defined sequences and predefined uvm_simple_sequence are executed.

Sequence Action Macro:

In the previous sections, we have seen the implementation of body() method of sequence. The body() method implementation requires some steps. We have seen these steps as Creation of item, wait for grant, randomize the item, send the item.

All these steps have be automated using "sequence action macros". There are some

more additional steps added in these macros. Following are the steps defined with the "sequence action macro".

	Phase	Operation
1	Create	Construct a transaction using create() method.
2	Synchronize	Call wait_for_grant() method
3	Pre_do	Call pre_do() method.
4	Randomize	Optionally Randomize transaction.
5	Mid_do	Call mid_do() method.
6	Post_Synchronize	Call send_request() and wait_for_item_done() methods.
7	Post_do	Optionally call post_do() and get_response() methods.

Pre_do(), mid_do() and post_do() are callback methods which are in uvm sequence. If user is interested , he can use these methods. For example, in mid_do() method, user can print the transaction or the randomized transaction can be fined tuned. These methods should not be clled by user directly.

Syntax:

```
virtual task pre_do(bit is_item)
virtual function void mid_do(uvm_sequence_item this_item)
virtual function void post_do(uvm_sequence_item this_item)
```

Pre_do() is a task , if the method consumes simulation cycles, the behavior may be unexpected.

Example Of Pre_do,Mid_do And Post_do

Lets look at a example: We will define a sequence using `uvm_do macro. This macro has all the above defined phases.

1)Define the body method using the `uvm_do() macro. Before and after this macro, just call messages.

```
virtual task body();
  uvm_report_info(get_full_name(),
  "Seuqnce Action Macro Phase : Before uvm_do macro ",UVM_LOW);
  `uvm_do(req);
  uvm_report_info(get_full_name(),
```

```
    "Seuqnce Action Macro Phase : After uvm_do macro ",UVM_LOW);
  endtask
```

2)Define pre_do() method. Lets just print a message from this method.

```
  virtual task pre_do(bit is_item);
    uvm_report_info(get_full_name(),
    "Seuqnce Action Macro Phase : PRE_DO ",UVM_LOW);
  endtask
```

3)Define mid_do() method. Lets just print a message from this method.

```
  virtual function void mid_do(uvm_sequence_item this_item);
    uvm_report_info(get_full_name(),
    "Seuqnce Action Macro Phase : MID_DO ",UVM_LOW);
  endfunction
```

4)Define post_do() method. Lets just print a message from this method.

```
  virtual function void post_do(uvm_sequence_item this_item);
    uvm_report_info(get_full_name(),
    "Seuqnce Action Macro Phase : POST_DO ",UVM_LOW);
  endfunction
```

Complet sequence code:

```
class demo_uvm_do extends uvm_sequence #(instruction);

  instruction req;

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

  `uvm_sequence_utils(demo_uvm_do, instruction_sequencer)

  virtual task pre_do(bit is_item);
    uvm_report_info(get_full_name(),
    "Seuqnce Action Macro Phase : PRE_DO ",UVM_LOW);
  endtask

  virtual function void mid_do(uvm_sequence_item this_item);
    uvm_report_info(get_full_name(),
```

```
    "Seuqnce Action Macro Phase : MID_DO ",UVM_LOW);
  endfunction

  virtual function void post_do(uvm_sequence_item this_item);
    uvm_report_info(get_full_name(),
    "Seuqnce Action Macro Phase : POST_DO ",UVM_LOW);
  endfunction

  virtual task body();
    uvm_report_info(get_full_name(),
    "Seuqnce Action Macro Phase : Before uvm_do macro ",UVM_LOW);
    `uvm_do(req);
    uvm_report_info(get_full_name(),
    "Seuqnce Action Macro Phase : After uvm_do macro ",UVM_LOW);
  endtask
endclass
```

Download the example

uvm_sequence_2.tar
Browse the code in uvm_sequence_2.tar

Command to run the simulation

VCS Users : make vcs
Questa Users: make questa

Log file report:

```
UVM_INFO@0:reporter[sequencer.demo_uvm_do]
Seuqnce Action Macro Phase : Before uvm_do macro
UVM_INFO@0:reporter[sequencer.demo_uvm_do]
Seuqnce Action Macro Phase : PRE_DO
UVM_INFO@0:reporter[sequencer.demo_uvm_do]
Seuqnce Action Macro Phase : MID_DO

0: Driving Instruction MUL

UVM_INFO@10:reporter[sequencer.demo_uvm_do]
Seuqnce Action Macro Phase : POST_DO
UVM_INFO@10:reporter[sequencer.demo_uvm_do]
Seuqnce Action Macro Phase : After uvm_do macro
```

The above log file shows the messages from pre_do,mid_do and post_do methods.

List Of Sequence Action Macros:

These macros are used to start sequences and sequence items that were either registered with a <`uvm-sequence_utils> macro or whose associated sequencer was already set using the <set_sequencer> method.

`uvm_create(item/sequence)

This action creates the item or sequence using the factory. Only the create phase will be executed.

`uvm_do(item/sequence)

This macro takes as an argument a uvm_sequence_item variable or sequence . All the above defined 7 phases will be executed.

`uvm_do_with(item/sequence, Constraint block)

This is the same as `uvm_do except that the constraint block in the 2nd argument is applied to the item or sequence in a randomize with statement before execution.

`uvm_send(item/sequence)

Create phase and randomize phases are skipped, rest all the phases will be executed. Using `uvm_create, create phase can be executed. Essentially, an `uvm_do without the create or randomization.

`uvm_rand_send(item/sequence)

Only create phase is skipped. rest of all the phases will be executed. User should use `uvm_create to create the sequence or item.

`uvm_rand_send_with(item/sequence , Constraint block)

Only create phase is skipped. rest of all the phases will be executed. User should use `uvm_create to create the sequence or item. Constraint block will be applied which randomization.

`uvm_do_pri(item/sequence, priority)

This is the same as `uvm_do except that the sequence item or sequence is executed with the priority specified in the argument.

`uvm_do_pri_with(item/sequence , constraint block , priority)

This is the same as `uvm_do_pri except that the given constraint block is applied to the item or sequence in a randomize with statement before execution.

`uvm_send_pri(item/sequence,priority)

This is the same as `uvm_send except that the sequence item or sequence is

executed with the priority specified in the argument.

`uvm_rand_send_pri(item/sequence,priority)

This is the same as `uvm_rand_send except that the sequence item or sequence is executed with the priority specified in the argument.

`uvm_rand_send_pri_with(item/sequence,priority,constraint block)

This is the same as `uvm_rand_send_pri except that the given constraint block is applied to the item or sequence in a randomize with statement before execution.

Following macros are used on sequence or sequence items on a different sequencer.

`uvm_create_on(item/sequence,sequencer)

This is the same as `uvm_create except that it also sets the parent sequence to the sequence in which the macro is invoked, and it sets the sequencer to the specified sequencer argument.

`uvm_do_on(item/sequence,sequencer)

This is the same as `uvm_do except that it also sets the parent sequence to the sequence in which the macro is invoked, and it sets the sequencer to the specified sequencer argument.

`uvm_do_on_pri(item/sequence,sequencer, priority)

This is the same as `uvm_do_pri except that it also sets the parent sequence to the sequence in which the macro is invoked, and it sets the sequencer to the specified sequencer argument.

`uvm_do_on_with(item/sequence,sequencer, constraint block)

This is the same as `uvm_do_with except that it also sets the parent sequence to the sequence in which the macro is invoked, and it sets the sequencer to the specified sequencer argument. The user must supply brackets around the constraints.

`uvm_do_on_pri_with(item/sequence,sequencer,priority,constraint block)

This is the same as `uvm_do_pri_with except that it also sets the parent sequence to the sequence in which the macro is invoked, and it sets the sequencer to the specified sequencer argument.

Examples With Sequence Action Macros:

```
virtual task body();
   uvm_report_info(get_full_name(),
   "Executing Sequence Action Macro uvm_do",UVM_LOW);
   `uvm_do(req)
endtask

virtual task body();
   uvm_report_info(get_full_name(),
         "Executing Sequence Action Macro uvm_do_with ",UVM_LOW);
   `uvm_do_with(req,{ inst == ADD; })
endtask

virtual task body();
   uvm_report_info(get_full_name(),
       "Executing Sequence Action Macro uvm_create and uvm_send",UVM_LOW);
   `uvm_create(req)
   req.inst = instruction::PUSH_B;
   `uvm_send(req)
endtask

virtual task body();
   uvm_report_info(get_full_name(),
       "Executing Sequence Action Macro uvm_create and uvm_rand_send",UVM_LOW);
   `uvm_create(req)
   `uvm_rand_send(req)
endtask
```

Download the example

uvm_sequence_3.tar
Browse the code in uvm_sequence_3.tar

Command to sun the simulation

VCS Users : make vcs
Questa Users: make questa

Log file report

0: Driving Instruction PUSH_B
UVM_INFO@10:reporter[***]Executing Sequence Action Macro uvm_do_with
10: Driving Instruction ADD
UVM_INFO@20:reporter[***]Executing Sequence Action Macro uvm_create and uvm_send
20: Driving Instruction PUSH_B
UVM_INFO@30:reporter[***]Executing Sequence Action Macro uvm_do
30: Driving Instruction DIV
UVM_INFO@40:reporter[***]Executing Sequence Action Macro uvm_create and uvm_rand_send
40: Driving Instruction MUL

UVM SEQUENCE 3

Body Callbacks:

uvm sequences has two callback methods pre_body() and post_body(), which are executed before and after the sequence body() method execution. These callbacks are called only when start_sequence() of sequencer or start() method of the sequence is called. User should not call these methods.

```
virtual task pre_body()
virtual task post_body()
```

Example

In this example, I just printed messages from pre_body() and post_body() methods. These methods can be used for initialization, synchronization with some events or cleanup.

```
class demo_pre_body_post_body extends uvm_sequence #(instruction);

  instruction req;

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

  `uvm_sequence_utils(demo_pre_body_post_body, instruction_sequencer)

  virtual task pre_body();
    uvm_report_info(get_full_name()," pre_body() callback ",UVM_LOW);
  endtask

  virtual task post_body();
    uvm_report_info(get_full_name()," post_body() callback ",UVM_LOW);
  endtask

  virtual task body();
    uvm_report_info(get_full_name(),"body() method: Before uvm_do macro ",UVM_LOW);
    `uvm_do(req);
    uvm_report_info(get_full_name(),"body() method: After uvm_do macro ",UVM_LOW);
endtask

endclass
```

Download the example

uvm_sequence_4.tar
Browse the code in uvm_sequence_4.tar

Command to sun the simulation

VCS Users : make vcs
Questa Users: make questa

Log file report

```
UVM_INFO @ 0 [RNTST] Running test ...
UVM_INFO @ 0: reporter [***] pre_body() callback
UVM_INFO @ 0: reporter [***] body() method: Before uvm_do macro
0: Driving Instruction SUB
UVM_INFO @ 10: reporter [***] body() method: After uvm_do macro
UVM_INFO @ 10: reporter [***] post_body() callback
```

Hierarchical Sequences

One main advantage of sequences is smaller sequences can be used to create sequences to generate stimulus required for todays complex protocol.

To create a sequence using another sequence, following steps has to be done
1)Extend the uvm_sequence class and define a new class.
2)Declare instances of child sequences which will be used to create new sequence.
3)Start the child sequence using <instance>.start() method in body() method.

Sequential Sequences

To executes child sequences sequentially, child sequence start() method should be called sequentially in body method.
In the below example you can see all the 3 steps mentioned above.
In this example, I have defined 2 child sequences. These child sequences can be used as normal sequences.

Sequence 1 code:

This sequence generates 4 PUSH_A instructions.

```
  virtual task body();
    repeat(4) begin
      `uvm_do_with(req, { inst == PUSH_A; });
    end
  endtask
```

Sequence 2 code:

This sequence generates 4 PUSH_B instructions.

```
  virtual task body();
    repeat(4) begin
      `uvm_do_with(req, { inst == PUSH_B; });
    end
  endtask
```

Sequential Sequence code:

This sequence first calls sequence 1 and then calls sequence 2.

```
class sequential_sequence extends uvm_sequence #(instruction);

  seq_a s_a;
  seq_b s_b;

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

  `uvm_sequence_utils(sequential_sequence, instruction_sequencer)

  virtual task body();
    `uvm_do(s_a);
    `uvm_do(s_b);
  endtask

endclass
```

From the testcase, "sequential_sequence" is selected as "default_sequence".

Download the example

uvm_sequence_5.tar
Browse the code in uvm_sequence_5.tar

Command to sun the simulation

VCS Users : make vcs
Questa Users: make questa

Log file report

```
0: Driving Instruction PUSH_A
10: Driving Instruction PUSH_A
20: Driving Instruction PUSH_A
30: Driving Instruction PUSH_A
40: Driving Instruction PUSH_B
50: Driving Instruction PUSH_B
60: Driving Instruction PUSH_B
70: Driving Instruction PUSH_B
```

If you observe the above log, you can see sequence seq_a is executed first and then sequene seq_b is executed.

Parallelsequences

To executes child sequences Parallel, child sequence start() method should be called parallel using fork/join in body method.

Parallel Sequence code:

```
class parallel_sequence extends uvm_sequence #(instruction);

  seq_a s_a;
  seq_b s_b;

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

```
  `uvm_sequence_utils(parallel_sequence, instruction_sequencer)

  virtual task body();
    fork
      `uvm_do(s_a)
      `uvm_do(s_b)
    join
  endtask

endclass
```

Download the example

uvm_sequence_6.tar
Browse the code in uvm_sequence_6.tar

Command to sun the simulation

VCS Users : make vcs
Questa Users: make questa

Log file report

```
UVM_INFO @ 0 [RNTST] Running test ...
0: Driving Instruction PUSH_A
10: Driving Instruction PUSH_B
20: Driving Instruction PUSH_A
30: Driving Instruction PUSH_B
40: Driving Instruction PUSH_A
50: Driving Instruction PUSH_B
60: Driving Instruction PUSH_A
70: Driving Instruction PUSH_B
```

UVM SEQUENCE 4

Sequencer Arbitration:

When sequences are executed parallel, sequencer will arbitrate among the parallel sequence. When all the parallel sequences are waiting for a grant from sequencer using wait_for_grant() method, then the sequencer, using the arbitration mechanism, sequencer grants to one of the sequencer.

There are 6 different arbitration algorithms, they are :

Macro	Algorithm
SEQ_ARB_FIFO	Requests are granted in FIFO style. Priorities will not be considered.
SEQ_ARB_RANDOM	Requests are granted randomly. Priorities will not be considered.
SEQ_ARB_WEIGHTED	Based on the priority value, randomly grants the requests.
SEQ_ARB_STRICT_FIFO	Sequences with high priority value will be granted in FIFO style.
SEQ_ARB_STRICT_RANDOM	Sequences with high priority value will be granted randomly.
SEQ_ARB_USER	Grants are done based on user defined algorithm.

To set the arbitaration, use the set_arbitration() method of the sequencer. By default , the arbitration algorithms is set to SEQ_ARB_FIFO.

```
function void set_arbitration(SEQ_ARB_TYPE val)
```

Lets look at a example:

In this example, I have 3 child sequences seq_mul seq_add and seq_sub each of them generates 3 transactions.

Sequence code 1:

```
virtual task body();
  repeat(3) begin
    `uvm_do_with(req, { inst == MUL; });
  end
endtask
```

Sequence code 2:

```
virtual task body();
  repeat(3) begin
    `uvm_do_with(req, { inst == ADD; });
  end
endtask
```

Sequence code 3:

```
virtual task body();
  repeat(3) begin
    `uvm_do_with(req, { inst == SUB; });
  end
endtask
```

Parallel sequence code:

In the body method, before starting child sequences, set the arbitration using set_arbitration(). In this code, im setting it to SEQ_ARB_RANDOM.

```
class parallel_sequence extends uvm_sequence #(instruction);
  seq_add add;
  seq_sub sub;
  seq_mul mul;

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

  `uvm_sequence_utils(parallel_sequence, instruction_sequencer)

  virtual task body();
    m_sequencer.set_arbitration(SEQ_ARB_RANDOM);
    fork
      `uvm_do(add)
      `uvm_do(sub)
      `uvm_do(mul)
    join
  endtask
endclass
```

Download the example

uvm_sequence_7.tar
Browse the code in uvm_sequence_7.tar

Command to sun the simulation

VCS Users : make vcs
Questa Users: make questa

Log file report for when SEQ_ARB_RANDOM is set.

0: Driving Instruction MUL
10: Driving Instruction SUB
20: Driving Instruction MUL
30: Driving Instruction SUB
40: Driving Instruction MUL
50: Driving Instruction ADD
60: Driving Instruction ADD
70: Driving Instruction SUB
80: Driving Instruction ADD

Log file report for when SEQ_ARB_FIFO is set.

0: Driving Instruction ADD
10: Driving Instruction SUB
20: Driving Instruction MUL
30: Driving Instruction ADD
40: Driving Instruction SUB
50: Driving Instruction MUL
60: Driving Instruction ADD
70: Driving Instruction SUB
80: Driving Instruction MUL

If you observe the first log report, all the transaction of the sequences are generated in random order. In the second log file, the transactions are given equal priority and are in fifo order.

Setting The Sequence Priority:

There are two ways to set the priority of a sequence. One is using the start method of the sequence and other using the set_priority() method of the sequence. By default, the priority of a sequence is 100. Higher numbers indicate higher priority.

```
virtual task start (uvm_sequencer_base sequencer,
      uvm_sequence_base parent_sequence = null,
      integer this_priority = 100,
      bit call_pre_post = 1)

function void set_priority (int value)
```

Lets look a example with SEQ_ARB_WEIGHTED.

For sequence seq_mul set the weight to 200.
For sequence seq_add set the weight to 300.
For sequence seq_sub set the weight to 400.

In the below example, start() method is used to override the default priority value.

Code :

```
class parallel_sequence extends uvm_sequence #(instruction);

   seq_add add;
   seq_sub sub;
   seq_mul mul;

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

   `uvm_sequence_utils(parallel_sequence, instruction_sequencer)

   virtual task body();
      m_sequencer.set_arbitration(SEQ_ARB_WEIGHTED);
      add = new("add");
      sub = new("sub");
      mul = new("mul");
      fork
         sub.start(m_sequencer,this,400);
         add.start(m_sequencer,this,300);
         mul.start(m_sequencer,this,200);
      join
   endtask

endclass
```

Download the example

uvm_sequence_8.tar
Browse the code in uvm_sequence_8.tar

Command to sun the simulation

VCS Users : make vcs
Questa Users: make questa

Log file report

0: Driving Instruction MUL
10: Driving Instruction ADD
20: Driving Instruction SUB
30: Driving Instruction SUB
40: Driving Instruction ADD
50: Driving Instruction ADD
60: Driving Instruction ADD
70: Driving Instruction MUL
80: Driving Instruction SUB

UVM SEQUENCE 5

Sequencer Registration Macros

Sequence Registration Macros does the following
1) Implements get_type_name method.
2) Implements create() method.
3) Registers with the factory.
4) Implements the static get_type() method.
5) Implements the virtual get_object_type() method.
6) Registers the sequence type with the sequencer type.
7) Defines p_sequencer variable. p_sequencer is a handle to its sequencer.
8) Implements m_set_p_sequencer() method.

If there are no local variables, then use following macro

```
`uvm_sequence_utils(TYPE_NAME,SQR_TYPE_NAME)
```

If there are local variables in sequence, then use macro

```
`uvm_sequence_utils_begin(TYPE_NAME,SQR_TYPE_NAME)
`uvm_field_* macro invocations here
`uvm_sequence_utils_end
```

Macros `uvm_field_* are used for define utility methods.
These `uvm_field_* macros are discussed in
UVM_TRANSACTION

Example to demonstrate the usage of the above macros:

```
class seq_mul extends uvm_sequence #(instruction);

  rand integer num_inst ;
  instruction req;

  constraint num_c { num_inst inside { 3,5,7 }; };

  `uvm_sequence_utils_begin(seq_mul,instruction_sequencer)
  `uvm_field_int(num_inst, UVM_ALL_ON)
  `uvm_sequence_utils_end
```

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

    virtual task body();
      uvm_report_info(get_full_name(),
      $psprintf("Num of transactions %d",num_inst),UVM_LOW);
      repeat(num_inst) begin
        `uvm_do_with(req, { inst == MUL; });
      end
    endtask

  endclass
```

Download the example

uvm_sequence_9.tar
Browse the code in uvm_sequence_9.tar

Command to sun the simulation

VCS Users : make vcs
Questa Users: make questa

Log

```
UVM_INFO @ 0: reporter [RNTST] Running test ...
UVM_INFO @ 0: reporter [sequencer.seq_mul] Num of transactions 5
0: Driving Instruction MUL
10: Driving Instruction MUL
20: Driving Instruction MUL
30: Driving Instruction MUL
40: Driving Instruction MUL
```

Setting Sequence Members:

set_config_* can be used only for the components not for the sequences.
By using configuration you can change the variables inside components only not in sequences.

But there is a workaround to this problem.

Sequence has handle name called p_sequencer which is pointing the Sequencer on which it is running.
Sequencer is a component , so get_config_* methods are implemented for it.
So from the sequence, using the sequencer get_config_* methods, sequence members can be updated if the variable is configured.

When using set_config_* , path to the variable should be sequencer name, as we are using the sequencer get_config_* method.

Following method demonstrates how this can be done:

Sequence:

1) num_inst is a integer variables which can be updated.
2) In the body method, call the get_config_int() method to get the integer value if num_inst is configured from testcase.

```
class seq_mul extends uvm_sequence #(instruction);

  integer num_inst = 4;
  instruction req;

  `uvm_sequence_utils_begin(seq_mul,instruction_sequencer)
  `uvm_field_int(num_inst, UVM_ALL_ON)
  `uvm_sequence_utils_end

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


  virtual task body();
    void'(p_sequencer.get_config_int("num_inst",num_inst));

    uvm_report_info(get_full_name(),
    $psprintf("Num of transactions %d",num_inst),UVM_LOW);
    repeat(num_inst) begin
      `uvm_do_with(req, { inst == MUL; });
    end
  endtask

endclass
```

Testcase:

From the testcase, using the set_config_int() method, configure the num_inst to 3. The instance path argument should be the sequencer path name.

```
module test;

  instruction_sequencer sequencer;
  instruction_driver driver;

  initial begin
    set_config_string("sequencer", "default_sequence", "seq_mul");
    set_config_int("sequencer", "num_inst",3);
    sequencer = new("sequencer", null);
    sequencer.build();
    driver = new("driver", null);
    driver.build();

    driver.seq_item_port.connect(sequencer.seq_item_export);
    sequencer.print();
    fork
     begin
        run_test();
        sequencer.start_default_sequence();
     end
     #3000 global_stop_request();
    join
  end

endmodule
```

Download the example

uvm_sequence_10.tar
Browse the code in uvm_sequence_10.tar

Command to sun the simulation

VCS Users : make vcs
Questa Users: make questa

Log

```
UVM_INFO @ 0: reporter [RNTST] Running test ...
UVM_INFO @ 0: reporter [sequencer.seq_mul] Num of transactions 3
0: Driving Instruction MUL
10: Driving Instruction MUL
20: Driving Instruction MUL
```

From the above log we can see that seq_mul.num_inst value is 3.

UVM SEQUENCE 6

Exclusive Access

A sequence may need exclusive access to the driver which sequencer is arbitrating among multiple sequence. Some operations require that a series of transaction needs to be driven without any other transaction in between them. Then a exclusive access to the driver will allow to a sequence to complete its operation with out any other sequence operations in between them.

There are 2 mechanisms to get exclusive access:

- Lock-unlcok
- Grab-ungrab

Lock-Unlock

```
task lock(uvm_sequencer_base sequencer = Null)
function void unlock(uvm_sequencer_base sequencer = Null)
```

Using lock() method , a sequence can requests for exclusive access. A lock request will be arbitrated the same as any other request. A lock is granted after all earlier requests are completed and no other locks or grabs are blocking this sequence. A lock() is blocking task and when access is granted, it will unblock.

Using unlock(), removes any locks or grabs obtained by this sequence on the specified sequencer.

If sequencer is null, the lock/unlock will be applied on the current default sequencer.

Lets see an example:

In this example there are 3 sequences with each sequence generating 4 transactions. All these 3 sequences will be called in parallel in another sequence.

Sequence 1 code:

```
virtual task body();
  repeat(4) begin
    `uvm_do_with(req, { inst == PUSH_A; });
  end
endtask
```

Sequence 2 code:

```
virtual task body();
  repeat(4) begin
    `uvm_do_with(req, { inst == POP_C; });
  end
endtask
```

Sequence 3 code:

In this sequence , call the lock() method to get the exclusive access to driver. After completing all the transaction driving, then call the unclock() method.

```
virtual task body();
  lock();
  repeat(4) begin
    `uvm_do_with(req, { inst == PUSH_B; });
  end
  unlock();
endtask
```

Parallel sequence code:

```
virtual task body();
  fork
    `uvm_do(s_a)
    `uvm_do(s_b)
    `uvm_do(s_c)
  join
endtask
```

Download the example

uvm_sequence_11.tar
Browse the code in uvm_sequence_11.tar

Command to sun the simulation

VCS Users : make vcs
Questa Users: make questa

Log file:

```
0: Driving Instruction PUSH_A
10: Driving Instruction POP_C
20: Driving Instruction PUSH_A
30: Driving Instruction PUSH_B
40: Driving Instruction PUSH_B
50: Driving Instruction PUSH_B
60: Driving Instruction PUSH_B
70: Driving Instruction POP_C
80: Driving Instruction PUSH_A
90: Driving Instruction POP_C
100: Driving Instruction PUSH_A
110: Driving Instruction POP_C
```

From the above log file, we can observe that , when seq_b sequence got the access, then transactions from seq_a and seq_c are not generated.

Lock() will be arbitrated before giving the access. To get the exclusive access without arbitration, grab() method should be used.

Grab-Ungrab

```
task grab(uvm_sequencer_base sequencer = null)
function void ungrab(uvm_sequencer_base sequencer = null)
```

grab() method requests a lock on the specified sequencer. A grab() request is put in front of the arbitration queue. It will be arbitrated before any other requests. A grab() is granted when no other grabs or locks are blocking this sequence.

A grab() is blocking task and when access is granted, it will unblock.

Ungrab() method removes any locks or grabs obtained by this sequence on the specified sequencer.

If no argument is supplied, then current default sequencer is chosen.

Example:

```
virtual task body();
#25;
grab();
repeat(4) begin
```

```
`uvm_do_with(req, { inst == PUSH_B; });
end
ungrab();
endtask
```

Download the example

uvm_sequence_12.tar
Browse the code in uvm_sequence_12.tar

Command to sun the simulation

VCS Users : make vcs
Questa Users: make questa

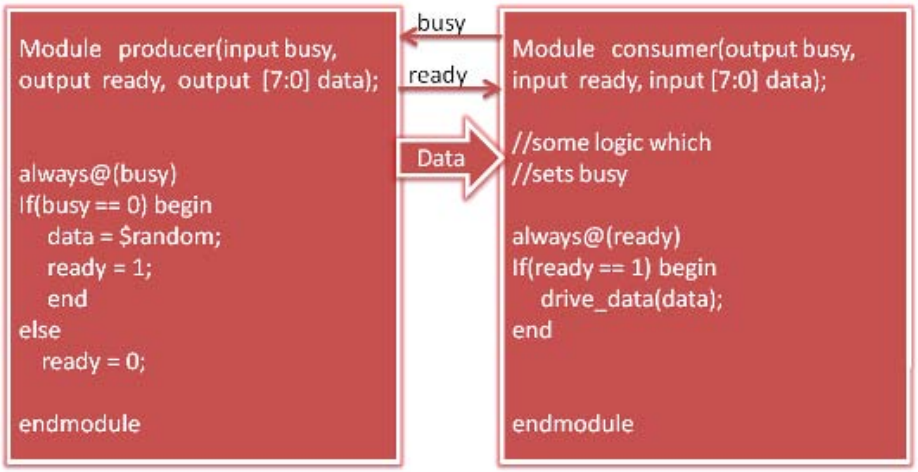
```
0: Driving Instruction PUSH_A
10: Driving Instruction POP_C
20: Driving Instruction PUSH_A
30: Driving Instruction PUSH_B
40: Driving Instruction PUSH_B
50: Driving Instruction PUSH_B
60: Driving Instruction PUSH_B
70: Driving Instruction POP_C
80: Driving Instruction PUSH_A
90: Driving Instruction POP_C
100: Driving Instruction PUSH_A
110: Driving Instruction POP_C
```

UVM TLM 1

Before going into the TLM interface concepts, lets see why we need TLM interface.

Port Based Data Transfer:

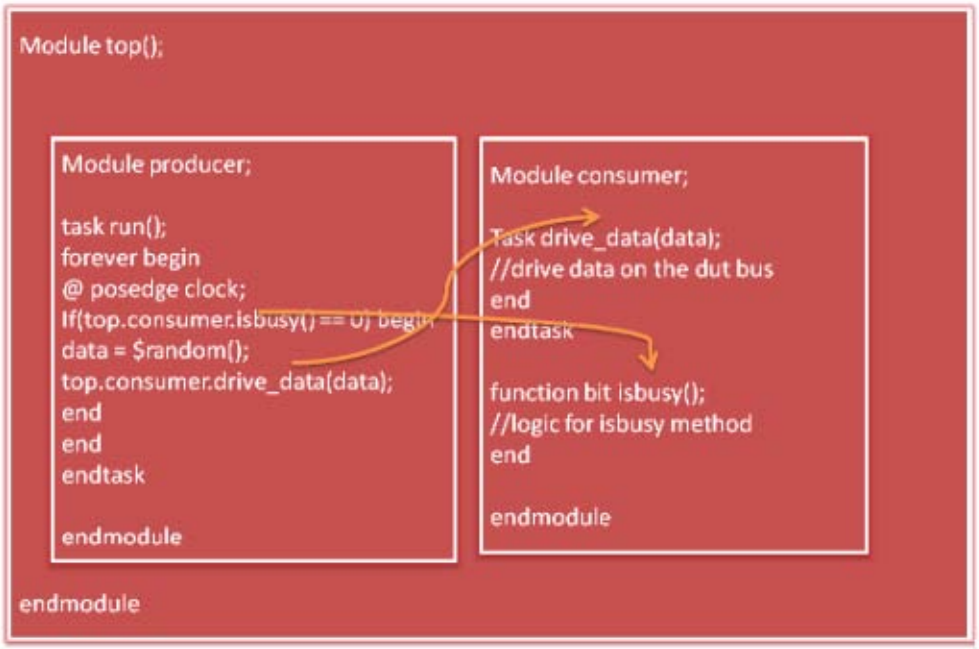
Following is a simple verification environment.



Components generator and driver are implemented as modules. These modules are connected using module ports or SV interfaces. The advantage of this methodology is, the two above mentioned components are independent. Instead of consumer module, any other component which can understand producer interface can be connected, which gives a great reusability.

The disadvantage of this methodology is , data transfer is done at lower lever abstraction.

Task Based Data Transfer:



In the above environment, methods are used to transfer the data between components. So, this gives a better control and data transfer is done at high level. The disadvantage is, components are using hierarchal paths which do not allow the reusability.

TLM interfaces:

UVM has TLM interfaces which provide the advantages which we saw in the above two data transfer styles.

Data is transferred at high level. Transactions which are developed by extending the uvm_sequence_item can be transferred between components using method calls.

These methods are not hierarchal fixed, so that components can be reused.

The advantages of TLM interfaces are

- 1) Higher level abstraction
- 2) Reusable. Plug and play connections.
- 3) Maintainability
- 4) Less code.
- 5) Easy to implement.
- 6) Faster simulation.
- 7) Connect to Systemc.
- 8) Can be used for reference model development.

Operation Supported By Tlm Interface:

- Putting:

Producer transfers a value to Consumer.

- Getting:

Consumer requires a data value from producer.

- Peeking:

Copies data from a producer without consuming the data.

- Broadcasting:

Transaction is broadcasted to none or one or multiple consumers.

Methods

BLOCKING:

```
virtual task put(input T1 t)
virtual task get(output T2 t)
virtual task peek(output T2 t)
```

NON-BLOCKIN:

```
virtual function bit try_put(input T1 t)
virtual function bit can_put()
virtual function bit try_get(output T2 t)
virtual function bit can_get()
virtual function bit try_peek(output T2 t)
virtual function bit can_peek()
```

BLOCKING TRANSPORT:

```
virtual task transport(input T1 req,output T2 rsp)
```

NON-BLOCKING TRANSPORT:

```
virtual function bit nb_transport(input T1 req,output T2 rsp)
```

ANALYSIS:

```
virtual function void write(input T1 t)
```

TIm Terminology :

- Producer:

A component which generates a transaction.

- Consumer:

A component which consumes the transaction.

- Initiator:

A component which initiates process.

- Target:

A component which responded to initiator.

TIm Interface Compilation Models:

- Blocking:

A blocking interface conveys transactions in blocking fashion; its methods do not return until the transaction has been successfully sent or retrieved. Its methods are defined as tasks.

- Non-blocking:

A non-blocking interface attempts to convey a transaction without consuming simulation time. Its methods are declared as functions. Because delivery may fail (e.g. the target component is busy and can not accept the request), the methods may return with failed status.

- Combined:

A combination interface contains both the blocking and non-blocking variants.

Interfaces:

The UVM provides ports, exports and implementation and analysis ports for connecting your components via the TLM interfaces. Port, Export, implementation terminology applies to control flow not to data flow.

- Port:
  Interface that requires an implementation is port.
- Import:
  Interface that provides an implementation is import ot implementation port.
- Export:
  Interface used to route transaction interfaces to other layers of the hierarchy.
- Analysis:
  Interface used to distribute transactions to passive components.

Direction:

- Unidirectional:
  Data transfer is done in a single direction and flow of control is in either or both direction.
- Bidirectional:
  Data transfer is done in both directions and flow of control is in either or both directions.

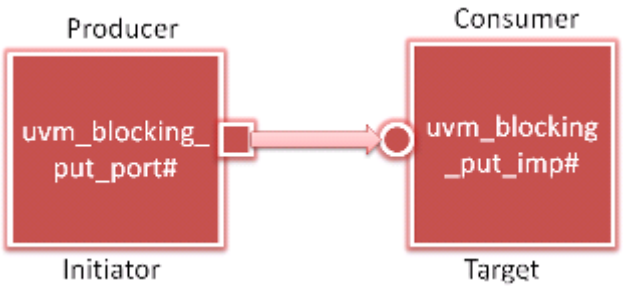
Examples:

A read operation is a bidirectional.

A write operation is unidirectional.

Lets look at a example:

In this example, we will use *_put_* interface.



There are 2 components, producer and consumer.

Producer generates the transaction and Consumers consumes it.

In this example, producer calls the put() method to send transaction to consumer i.e producer is initiator and consumer is target.

When the put() method in the producer is called, it actually executes the put() method which is defined in consumer component.

Transaction

We will use the below transaction in this example.

```
class instruction extends uvm_sequence_item;
  typedef enum {PUSH_A,PUSH_B,ADD,SUB,MUL,DIV,POP_C} inst_t;
  rand inst_t inst;

  `uvm_object_utils_begin(instruction)
  `uvm_field_enum(inst_t,inst, UVM_ALL_ON)
  `uvm_object_utils_end

  function new (string name = "instruction");
    super.new(name);
  endfunction
endclass
```

Producer:

1) Define producer component by extending uvm_component.

```
class producer extends uvm_component;

endclass : producer
```

2) Declare uvm_blocking_put_port port.

```
uvm_blocking_put_port#(instruction) put_port;
```

3) In the constructor, construct the port.

```
function new(string name, uvm_component p = null);
  super.new(name,p);
  put_port = new("put_port", this);
endfunction
```

4) Define the run() method. In this method, randomize the transaction.
Then call the put() of the put_port and pass the randomized transaction.

```
  task run;
    for(int i = 0; i < 10; i++)
    begin
      instruction ints;
      #10;
      ints = new();
      if(ints.randomize()) begin
        `uvm_info("producer", $sformatf("sending %s",ints.inst.name()), UVM_MEDIUM)
        put_port.put(ints);
      end
    end
  endtask
```

Producer source code

```
  class producer extends uvm_component;

    uvm_blocking_put_port#(instruction) put_port;

    function new(string name, uvm_component p = null);
      super.new(name,p);
      put_port = new("put_port", this);
    endfunction

    task run;
      for(int i = 0; i < 10; i++)
      begin
        instruction ints;
        #10;
        ints = new();
        if(ints.randomize()) begin
          `uvm_info("producer", $sformatf("sending %s",ints.inst.name()), UVM_MEDIUM)
          put_port.put(ints);
        end
      end
    endtask

  endclass : producer
```

Consumer:

1) Define a consumer component by extending uvm_component.

```
  class consumer extends uvm_component;

  endclass : consumer
```

2) Declare uvm_blocking_put_imp import. The parameters to this port are transaction and the consumer component itself.

```
uvm_blocking_put_imp#(instruction,consumer) put_port;
```

3) In the construct , construct the port.

```
function new(string name, uvm_component p = null);
  super.new(name,p);
  put_port = new("put_port", this);
endfunction
```

4) Define put() method. When the producer calls "put_port.put(ints);", then this method will be called. Arguments to this method is transaction type "instruction". In this method, we will just print the transaction.

```
task put(instruction t);
  `uvm_info("consumer", $sformatf("receiving %s",t.inst.name()), UVM_MEDIUM)
endtask
```

Consumer source code

```
class consumer extends uvm_component;

  uvm_blocking_put_imp#(instruction,consumer) put_port;

  function new(string name, uvm_component p = null);
    super.new(name,p);
    put_port = new("put_port", this);
  endfunction

  task put(instruction t);
    `uvm_info("consumer", $sformatf("receiving %s",t.inst.name()), UVM_MEDIUM)
    //push the transaction into queue or array
    //or drive the transaction to next level
    //or drive to interface
  endtask

endclass : consumer
```

Connecting producer and consumer

In the env class, take the instance of producer and consumer components.

In the connect method, connect the producer put_port to consumer put_port using p.put_port.connect(c.put_port);

Env Source code

```
class env extends uvm_env;
  producer p;
  consumer c;

  function new(string name = "env");
    super.new(name);
    p = new("producer", this);
    c = new("consumer", this);
  endfunction

  function void connect();
    p.put_port.connect(c.put_port);
  endfunction

  task run();
    #1000;
    global_stop_request();
  endtask

endclass
```

Testcase

```
module test;
  env e;

  initial begin
    e = new();
    run_test();
  end

endmodule
```

Download the example

uvm_tlm_1.tar
Browse the code in uvm_tlm_1.tar

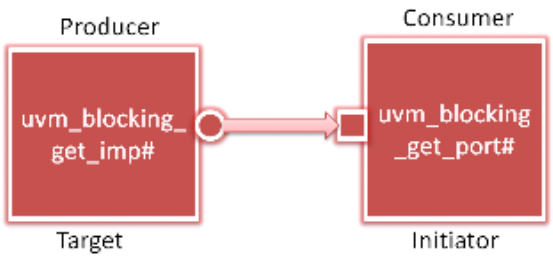
Command to sun the simulation

VCS Users : make vcs
Questa Users: make questa

Log

```
UVM_INFO producer.sv(26) @ 10:
env.producer [producer] sending PUSH_A
UVM_INFO consumer.sv(20) @ 10:
env.consumer [consumer] receiving PUSH_A
UVM_INFO producer.sv(26) @ 20:
env.producer [producer] sending PUSH_B
UVM_INFO consumer.sv(20) @ 20:
env.consumer [consumer] receiving PUSH_B
```

One more example using *_get_* interface as per the below topology.



Download the example

uvm_tlm_2.tar
Browse the code in uvm_tlm_2.tar

Command to sun the simulation

VCS Users : make vcs
Questa Users: make questa

All Interfaces In Uvm:

```
uvm_blocking_put_port #(T)
uvm_nonblocking_put_port #(T)
uvm_put_port #(T)
uvm_blocking_get_port #(T)
uvm_nonblocking_get_port #(T)
uvm_get_port #(T)
uvm_blocking_peek_port #(T)
uvm_nonblocking_peek_port #(T)
uvm_peek_port #(T)
uvm_blocking_get_peek_port #(T)
uvm_nonblocking_get_peek_port #(T)
uvm_get_peek_port #(T)
uvm_analysis_port #(T)
```

```
uvm_transport_port #(REQ,RSP)
uvm_blocking_transport_port #(REQ,RSP)
uvm_nonblocking_transport_port #(REQ,RSP)
uvm_master_port #(REQ,RSP)
uvm_blocking_master_port #(REQ,RSP)
uvm_nonblocking_master_port #(REQ,RSP)
uvm_slave_port #(REQ,RSP)
uvm_blocking_slave_port #(REQ,RSP)
uvm_nonblocking_slave_port #(REQ,RSP)
uvm_put_export #(T)
uvm_blocking_put_export #(T)
uvm_nonblocking_put_export #(T)
uvm_get_export #(T)
uvm_blocking_get_export #(T)
uvm_nonblocking_get_export #(T)
uvm_peek_export #(T)
uvm_blocking_peek_export #(T)
uvm_nonblocking_peek_export #(T)
uvm_get_peek_export #(T)
uvm_blocking_get_peek_export #(T)
uvm_nonblocking_get_peek_export #(T)
uvm_analysis_export #(T)
uvm_transport_export #(REQ,RSP)
uvm_nonblocking_transport_export #(REQ,RSP)
uvm_master_export #(REQ,RSP)
uvm_blocking_master_export #(REQ,RSP)
uvm_nonblocking_master_export #(REQ,RSP)
uvm_slave_export #(REQ,RSP)
uvm_blocking_slave_export #(REQ,RSP)
uvm_nonblocking_slave_export #(REQ,RSP)
uvm_put_imp #(T,IMP)
uvm_blocking_put_imp #(T,IMP)
uvm_nonblocking_put_imp #(T,IMP)
uvm_get_imp #(T,IMP)
uvm_blocking_get_imp #(T,IMP)
uvm_nonblocking_get_imp #(T,IMP)
uvm_peek_imp #(T,IMP)
uvm_blocking_peek_imp #(T,IMP)
uvm_nonblocking_peek_imp #(T,IMP)
uvm_get_peek_imp #(T,IMP)
uvm_blocking_get_peek_imp #(T,IMP)
uvm_nonblocking_get_peek_imp #(T,IMP)
uvm_analysis_imp #(T,IMP)
uvm_transport_imp#(REQ,RSP,IMP,REQ_IMP,RSP_IMP)
uvm_blocking_transport_imp#(REQ,RSP,IMP,REQ_IMP,RSP_IMP)
uvm_nonblocking_transport_imp#(REQ,RSP,IMP,REQ_IMP,RSP_IMP)
uvm_master_imp #(REQ,RSP,IMP,REQ_IMP,RSP_IMP)
uvm_blocking_master_imp#(REQ,RSP,IMP,REQ_IMP,RSP_IMP)
uvm_nonblocking_master_imp#(REQ,RSP,IMP,REQ_IMP,RSP_IMP)
uvm_slave_imp#(REQ,RSP,IMP,REQ_IMP,RSP_IMP)
uvm_blocking_slave_imp#(REQ,RSP,IMP,REQ_IMP,RSP_IMP)
uvm_nonblocking_slave_imp#(REQ,RSP,IMP,REQ_IMP,RSP_IMP)
```

UVM TLM 2

Analysis

The analysis port is used to perform non-blocking broadcasts of transactions. It is by components like monitors/drivers to publish transactions to its subscribers, which are typically scoreboards and response/coverage collectors. For each port, more than one component can be connected. Even if a component is not connected to the port, simulation can continue, unlike put/get ports where simulation is not continued.

The uvm_analysis_port consists of a single function, write(). Subscriber component should provide an implementation of write()method. UVM provides the uvm_subscriber base component to simplify this operation, so a typical analysis component would extend uvm_subscriber and its export is analysis_export.

Lets write a example.

In the example, we will define a monitor component and a subscriber.

Monitor source code:

In monitor, call the function write() pass the transaction.

```
class monitor extends uvm_monitor;
  uvm_analysis_port #(instruction) anls_port;

  function new(string name, uvm_component p = null);
    super.new(name,p);
    anls_port = new("anls_port", this);
  endfunction

  task run;
    instruction inst;
    inst = new();
    #10ns;
    inst.inst = instruction::MUL;
    anls_port.write(inst);
    #10ns;
    inst.inst = instruction::ADD;
    anls_port.write(inst);
    #10ns;
    inst.inst = instruction::SUB;
    anls_port.write(inst);
  endtask

endclass
```

Subscriber source code:

In Subscriber, define the write() method.

```
class subscriber extends uvm_subscriber#(instruction);
  function new(string name, uvm_component p = null);
    super.new(name,p);
  endfunction

  function void write(instruction t);
    `uvm_info(get_full_name(),
    $sformatf("receiving %s",t.inst.name()), UVM_MEDIUM)
  endfunction

endclass : subscriber
```

Env source code:

```
class env extends uvm_env;
  monitor mon;
  subscriber sb,cov;

  function new(string name = "env");
    super.new(name);
    mon = new("mon", this);
    sb = new("sb", this);
    cov = new("cov", this);
  endfunction

  function void connect();
    mon.anls_port.connect(sb.analysis_export);
    mon.anls_port.connect(cov.analysis_export);
  endfunction

  task run();
    #1000;
    global_stop_request();
  endtask
endclass

module test;
  env e;
  initial begin
    e = new();
    run_test();
  end
endmodule
```

Download the example

uvm_tlm_3.tar
Browse the code in uvm_tlm_3.tar

Command to sun the simulation

VCS Users : make vcs
Questa Users: make questa

From the below log, you see that transaction is sent to both the components cov and sb.

Log

```
UVM_INFO subscriber.sv(18) @ 0: env.cov [env.cov] receiving MUL
UVM_INFO subscriber.sv(18) @ 0: env.sb [env.sb] receiving MUL
UVM_INFO subscriber.sv(18) @ 0: env.cov [env.cov] receiving ADD
UVM_INFO subscriber.sv(18) @ 0: env.sb [env.sb] receiving ADD
UVM_INFO subscriber.sv(18) @ 0: env.cov [env.cov] receiving SUB
UVM_INFO subscriber.sv(18) @ 0: env.sb [env.sb] receiving SUB
```

Tlm Fifo

Tlm_fifo provides storage of transactions between two independently running processes just like mailbox. Transactions are put into the FIFO via the put_export and fetched from the get_export.

Methods

Following are the methods defined for tlm fifo.

```
function new(string name,
    uvm_component parent = null,
    int size = 1)
```

The size indicates the maximum size of the FIFO; a value of zero indicates no upper bound.

```
virtual function int size()
```

Returns the capacity of the FIFO. 0 indicates the FIFO capacity has no limit.

virtual function int used()
Returns the number of entries put into the FIFO.

virtual function bit is_empty()
Returns 1 when there are no entries in the FIFO, 0 otherwise.

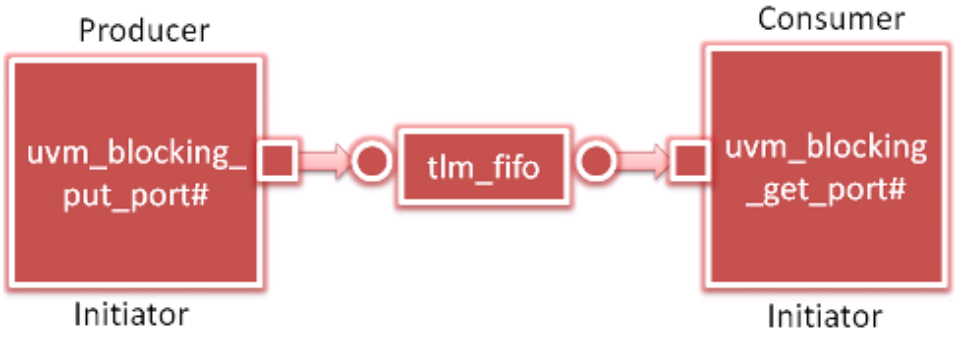
virtual function bit is_full()
Returns 1 when the number of entries in the FIFO is equal to its size, 0 otherwise.

virtual function void flush()
Removes all entries from the FIFO, after which used returns 0 and is_empty returns 1.

Example

Lets implement a example.

In this example, we will use a tlm_fifo to connect producer and consumer. The producer component generates the transaction and using its put_port pot() method, sends transaction out. The consumer component, to get the transaction from outside, uses get() method of get_port. These two ports are connected to tlm_fifo in the env class. In this example, producer and consumer are initiators as both components are calling the methods.



Producer source code:

```
class producer extends uvm_component;
  uvm_blocking_put_port#(int) put_port;

  function new(string name, uvm_component p = null);
    super.new(name,p);
    put_port = new("put_port", this);
  endfunction
```

```
    task run;
      int randval;
      for(int i = 0; i < 10; i++)
      begin
        #10;
        randval = $urandom_range(4,10);
        `uvm_info("producer", $sformatf("sending %d",randval), UVM_MEDIUM)
        put_port.put(randval);
      end
    endtask
  endclass : producer
```

Consumer source code:

```
  class consumer extends uvm_component;
    uvm_blocking_get_port#(int) get_port;

    function new(string name, uvm_component p = null);
      super.new(name,p);
      get_port = new("get_port", this);
    endfunction

    task run;
      int val;
      forever   begin
        get_port.get(val);
        `uvm_info("consumer", $sformatf("receiving %d", val), UVM_MEDIUM)
      end
    endtask
  endclass : consumer
```

Env source code:

```
  class env extends uvm_env;
    producer p;
    consumer c;
    tlm_fifo #(int) f;

    function new(string name = "env");
      super.new(name);
      p = new("producer", this);
      c = new("consumer", this);
      f = new("fifo", this);
    endfunction
```

```
    function void connect();
      p.put_port.connect(f.put_export);
      c.get_port.connect(f.get_export);
    endfunction

    task run();
      #1000 global_stop_request();
    endtask
  endclass
```

Download the example

uvm_tlm_4.tar
Browse the code in uvm_tlm_4.tar

Command to sun the simulation

VCS Users : make vcs
Questa Users: make questa

Log

```
UVM_INFO producer.sv(28) @ 10: env.producer [producer] sending 7
UVM_INFO consumer.sv(26) @ 10: env.consumer [consumer] receiving 7
UVM_INFO producer.sv(28) @ 20: env.producer [producer] sending 4
UVM_INFO consumer.sv(26) @ 20: env.consumer [consumer] receiving 4
```

UVM CALLBACK

Callback mechanism is used for altering the behavior of the transactor without modifying the transactor. One of the many promises of Object-Oriented programming is that it will allow for plug-and-play re-usable verification components. Verification Designers will hook the transactors together to make a verification environment. In SystemVerilog, this hooking together of transactors can be tricky. Callbacks provide a mechanism whereby independently developed objects may be connected together in simple steps.

This article describes uvm callbacks. uvm callback might be used for simple notification, two-way communication, or to distribute work in a process. Some requirements are often unpredictable when the transactor is first written. So a transactor should provide some kind of hooks for executing the code which is defined afterwards. In uvm, these hooks are created using callback methods. For instance, a driver is developed and an empty method is called before driving the transaction to the DUT. Initially this empty method does nothing. As the implementation goes, user may realize that he needs to print the state of the transaction or to delay the transaction driving to DUT or inject an error into transaction. Callback mechanism allows executing the user defined code in place of the empty callback method. Other example of callback usage is in monitor. Callbacks can be used in a monitor for collecting coverage information or for hooking up to scoreboard to pass transactions for self checking. With this, user is able to control the behavior of the transactor in verification environment and individual testcases without doing any modifications to the transactor itself.

Following are the steps to be followed to create a transactor with callbacks. We will see simple example of creating a Driver transactor to support callback mechanism.

Stpe 1) Define a facade class.

1) Extend the uvm_callback class to create a faced class.

```
class Driver_callback extends uvm_callback;

endclass : Driver_callback
```

2)Define required callback methods. All the callback methods must be virtual.
In this example, we will create callback methods which will be called before driving the packet and after driving the packet to DUT.

```
virtual task pre_send(); endtask
virtual task post_send(); endtask
```

3)Define the constructor and get_type_name methods and define type_name.

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

static string type_name = "Driver_callback";

virtual function string get_type_name();
  return type_name;
endfunction
```

Step 2) Register the facade class with driver and call the callback methods.

1)In the driver class, using `uvm_register_cb() macro, register the facade class.

```
class Driver extends uvm_component;

  `uvm_component_utils(Driver)
  `uvm_register_cb(Driver,Driver_callback)

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

endclass
```

2)Calling callback method.

Inside the transactor, callback methods should be called whenever something interesting happens.

We will call the callback method before driving the packet and after driving the packet. We defined 2 methods in facade class. We will call pre_send() method before sending the packet and post_send() method after sending the packet.

Using a `uvm_do_callbacks() macro, callback methods are called.
There are 3 argumentd to `uvm_do_callbacks(,) macro.
First argument must be the driver class and second argument is facade class.
Third argument must be the callback method in the facade class.

To call pre_send() method , use macro
`uvm_do_callbacks(Driver,Driver_callback,pre_send());
and similarly to call post_send() method,
`uvm_do_callbacks(Driver,Driver_callback,post_send());

Place the above macros before and after driving the packet.

```
virtual task run();
  repeat(2) begin
    `uvm_do_callbacks(Driver,Driver_callback,pre_send())
    $display(" Driver: Started Driving the packet ..... %d",$time);
    // Logic to drive the packet goes hear
    // let's consider that it takes 40 time units to drive a packet.
    #40;
    $display(" Driver: Finished Driving the packet ..... %d",$time);
    `uvm_do_callbacks(Driver,Driver_callback,post_send())
  end
endtask
```

With this, the Driver implementation is completed with callback support.

Driver And Driver Callback Class Source Code

```
class Driver_callback extends uvm_callback;
  function new (string name = "Driver_callback");
    super.new(name);
  endfunction

  static string type_name = "Driver_callback";

  virtual function string get_type_name();
    return type_name;
  endfunction

  virtual task pre_send(); endtask
  virtual task post_send(); endtask
endclass : Driver_callback


class Driver extends uvm_component;
  `uvm_component_utils(Driver)
  `uvm_register_cb(Driver,Driver_callback)

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

```
    virtual task run();
      repeat(2) begin
        `uvm_do_callbacks(Driver,Driver_callback,pre_send())
        $display(" Driver: Started Driving the packet ..... %d",$time);
        // Logic to drive the packet goes hear
        // let's consider that it takes 40 time units to drive a packet.
        #40;
        $display(" Driver: Finished Driving the packet ..... %d",$time);
        `uvm_do_callbacks(Driver,Driver_callback,post_send())
      end
    endtask
  endclass
```

Let's run the driver in simple testcase. In this testcase, we are not changing any callback methods definitions.

Testcase Source Code

```
  module test;
    Driver drvr;

    initial begin
      drvr = new("drvr");
      run_test();
    end
  endmodule
```

Download files

uvm_callback_1.tar

Browse the code in uvm_callback_1.tar

Command to run the simulation

VCS Users : make vcs

Questa Users: make questa

Log report

```
UVM_INFO @ 0: reporter [RNTST] Running test ...
Driver: Started Driving the packet ..... 0
Driver: Finished Driving the packet ..... 40
Driver: Started Driving the packet ..... 40
Driver: Finished Driving the packet ..... 80
UVM_ERROR @ 9200: reporter [TIMOUT] Watchdog timeout of '9200' expired.
```

Following steps are to be performed for using callback mechanism to do required functionality.

We will see how to use the callbacks which are implemented in the above defined driver in a testcase.

1) Implement the user defined callback method by extending facade class of the driver class.

We will delay the driving of packet be 20 time units using the pre_send() call back method.

We will just print a message from post_send() callback method.

```
class Custom_Driver_callbacks_1 extends Driver_callback;

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

  virtual task pre_send();
    $display("CB_1:pre_send: Delaying the packet driving by 20 time units. %d",$time);
    #20;
  endtask

  virtual task post_send();
    $display("CB_1:post_send: Just a message from post send callback method \n");
  endtask

endclass
```

2) Construct the user defined facade class object.

```
Custom_Driver_callbacks_1 cb_1;
cb_1 = new("cb_1");
```

3) Register the callback method with the driver component. uvm_callback class has static method add() which is used to register the callback.

```
uvm_callbacks #(Driver,Driver_callback)::add(drvr,cb_1);
```

Testcase 2 Source Code

```
class Custom_Driver_callbacks_1 extends Driver_callback;
  function new (string name = "Driver_callback");
    super.new(name);
  endfunction

  virtual task pre_send();
    $display("CB_1:pre_send: Delaying the packet driving by 20 time units. %d",$time);
    #20;
  endtask

  virtual task post_send();
    $display("CB_1:post_send: Just a message from post send callback method \n");
  endtask
endclass


module test;
  initial begin
    Driver drvr;
    Custom_Driver_callbacks_1 cb_1;
    drvr = new("drvr");
    cb_1 = new("cb_1");
    uvm_callbacks #(Driver,Driver_callback)::add(drvr,cb_1);
    uvm_callbacks #(Driver,Driver_callback)::display();
    run_test();
  end
endmodule
```

Download the example

uvm_callback_2.tar

Browse the code in uvm_callback_2.tar

Simulation Command

VCS Users : make vcs

Questa Users: make questa

Run the testcase. See the log results; We delayed the driving of packet by 20 time units using callback mechanism. See the difference between the previous testcase log and this log.

Log report

```
cb_1 on drvr ON
UVM_INFO @ 0: reporter [RNTST] Running test ...
CB_1:pre_send: Delaying the packet driving by 20 time units. 0
Driver: Started Driving the packet ..... 20
Driver: Finished Driving the packet ..... 60
CB_1:post_send: Just a message from post send callback method

CB_1:pre_send: Delaying the packet driving by 20 time units. 60
Driver: Started Driving the packet ..... 80
Driver: Finished Driving the packet ..... 120
CB_1:post_send: Just a message from post send callback method

UVM_ERROR @ 9200: reporter [TIMOUT] Watchdog timeout of '9200' expired.
```

Now we will see registering 2 callback methods.

1) Define another user defined callback methods by extending facade class.

```
class Custom_Driver_callbacks_2 extends Driver_callback;

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

    virtual task pre_send();
      $display("CB_2:pre_send: Hai .... this is from Second callback %d",$time);
    endtask

  endclass
```

2) Construct the user defined facade class object.

```
    Custom_Driver_callbacks_2 cb_2;
    cb_2 = new("cb_2");
```

3) Register the object

```
    uvm_callbacks #(Driver,Driver_callback)::add(drvr,cb_2);
```

Testcase 3 Source Code

```
class Custom_Driver_callbacks_1 extends Driver_callback;
  function new (string name = "Driver_callback");
    super.new(name);
  endfunction

  virtual task pre_send();
    $display("CB_1:pre_send: Delaying the packet driving by 20 time units. %d",$time);
    #20;
  endtask

  virtual task post_send();
    $display("CB_1:post_send: Just a message from post send callback method \n");
  endtask
endclass

class Custom_Driver_callbacks_2 extends Driver_callback;
  function new (string name = "Driver_callback");
  super.new(name);
  endfunction

  virtual task pre_send();
    $display("CB_2:pre_send: Hai .... this is from Second callback %d",$time);
  endtask
endclass


module test;
  initial begin
    Driver drvr;
    Custom_Driver_callbacks_1 cb_1;
    Custom_Driver_callbacks_2 cb_2;
    drvr = new("drvr");
    cb_1 = new("cb_1");
    cb_2 = new("cb_2");
    uvm_callbacks #(Driver,Driver_callback)::add(drvr,cb_1);
    uvm_callbacks #(Driver,Driver_callback)::add(drvr,cb_2);
    uvm_callbacks #(Driver,Driver_callback)::display();
    run_test();
  end
endmodule
```

Download source code
uvm_callback_3.tar
Browse the code in uvm_callback_3.tar

Command to run the simulation
VCS Users : make vcs
Questa Users: make questa

Run the testcase and analyze the result.

Log report

```
UVM_INFO @ 0: reporter [RNTST] Running test ...
CB_1:pre_send: Delaying the packet driving by 20 time units. 0
CB_2:pre_send: Hai .... this is from Second callback 20
Driver: Started Driving the packet ..... 20
Driver: Finished Driving the packet ..... 60
CB_1:post_send: Just a message from post send callback method

CB_1:pre_send: Delaying the packet driving by 20 time units. 60
CB_2:pre_send: Hai .... this is from Second callback 80
Driver: Started Driving the packet ..... 80
Driver: Finished Driving the packet ..... 120
CB_1:post_send: Just a message from post send callback method

UVM_ERROR @ 9200: reporter [TIMOUT] Watchdog timeout of '9200' expired.
```

The log results show that pre_send() method of CDc_1 is called first and then pre_send() method of Cdc_2. This is because of the order of the registering callbacks.

```
uvm_callbacks #(Driver,Driver_callback)::add(drvr,cb_1);
uvm_callbacks #(Driver,Driver_callback)::add(drvr,cb_2);
```

Now we will see how to change the order of the callback method calls.
By changing the sequence of calls to add() method, order of callback method calling can be changed.

Testcase 4 Source Code

```
module test;
  initial begin
    Driver drvr;
    Custom_Driver_callbacks_1 cb_1;
    Custom_Driver_callbacks_2 cb_2;
    drvr = new("drvr");
    cb_1 = new("cb_1");
    cb_2 = new("cb_2");
    uvm_callbacks #(Driver,Driver_callback)::add(drvr,cb_2);
    uvm_callbacks #(Driver,Driver_callback)::add(drvr,cb_1);
    uvm_callbacks #(Driver,Driver_callback)::display();
    run_test();
  end
endmodule
```

Download the source code

uvm_callback_4.tar

Browse the code in uvm_callback_4.tar

Command to run the simulation

VCS Users : make vcs

Questa Users: make questa

Run and analyze the results.

Log results show that, pre_send() method of CDs_1 is called after calling CDs_2 pre_send() method.

Log file report

```
UVM_INFO @ 0: reporter [RNTST] Running test ...
CB_2:pre_send: Hai .... this is from Second callback 0
CB_1:pre_send: Delaying the packet driving by 20 time units. 0
Driver: Started Driving the packet ..... 20
Driver: Finished Driving the packet ..... 60
CB_1:post_send: Just a message from post send callback method

CB_2:pre_send: Hai .... this is from Second callback 60
CB_1:pre_send: Delaying the packet driving by 20 time units. 60
Driver: Started Driving the packet ..... 80
Driver: Finished Driving the packet ..... 120
CB_1:post_send: Just a message from post send callback method

UVM_ERROR @ 9200: reporter [TIMOUT] Watchdog timeout of '9200' expired.
```

Methods:

add_by_name:

We have seen, the usage of add() method which requires object.
Using add_by_name() method, callback can be registered with object name.

```
static function void add_by_name(string name,
        uvm_callback cb,
        uvm_component root,
        uvm_apprepend ordering = UVM_APPEND)
```

delete:

uvm also provides uvm_callbacks::delete() method to remove the callback methods which are registered.

Similar to delete, delete_by_name() method is used to remove the callback using the object name.

```
static function void delete_by_name(string name,
        uvm_callback cb,
        uvm_component root )
```

Macros:

`uvm_register_cb

Registers the given CB callback type with the given T object type.

`uvm_set_super_type

Defines the super type of T to be ST.

`uvm_do_callbacks

Calls the given METHOD of all callbacks of type CB registered with the calling object

`uvm_do_obj_callbacks

Calls the given METHOD of all callbacks based on type CB registered with the given object, OBJ, which is or is based on type T.

`uvm_do_callbacks_exit_on

Calls the given METHOD of all callbacks of type CB registered with the calling object

`uvm_do_obj_callbacks_exit_on

Calls the given METHOD of all callbacks of type CB registered with the given object OBJ, which must be or be based on type T, and returns upon the first callback that returns the bit value given by VAL.