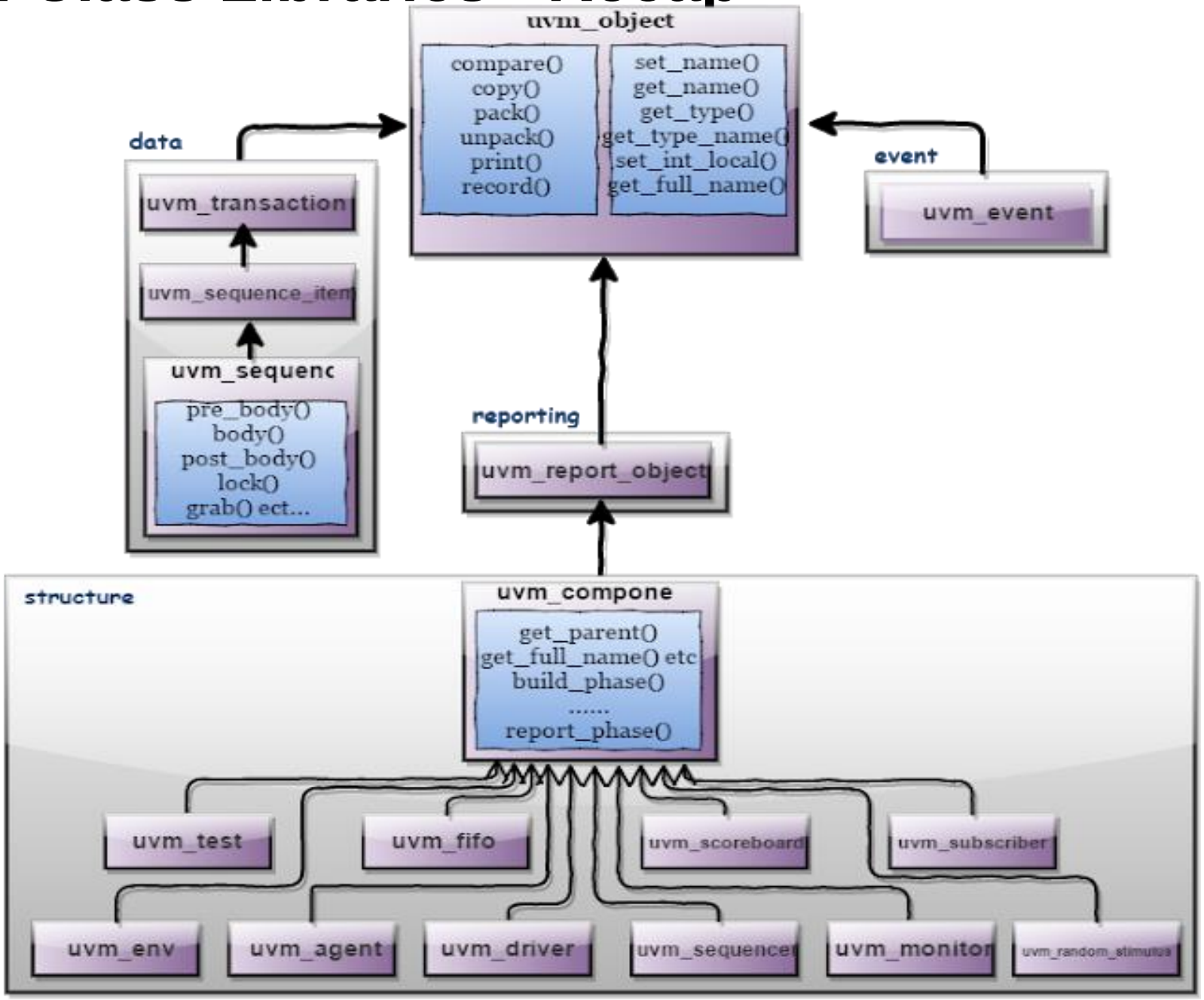
UVM sequence

- Reference
- Universal Verification Methodology UVM Cookbook by Siemens Digital Industries Software

UVM Class Libraries - Recap

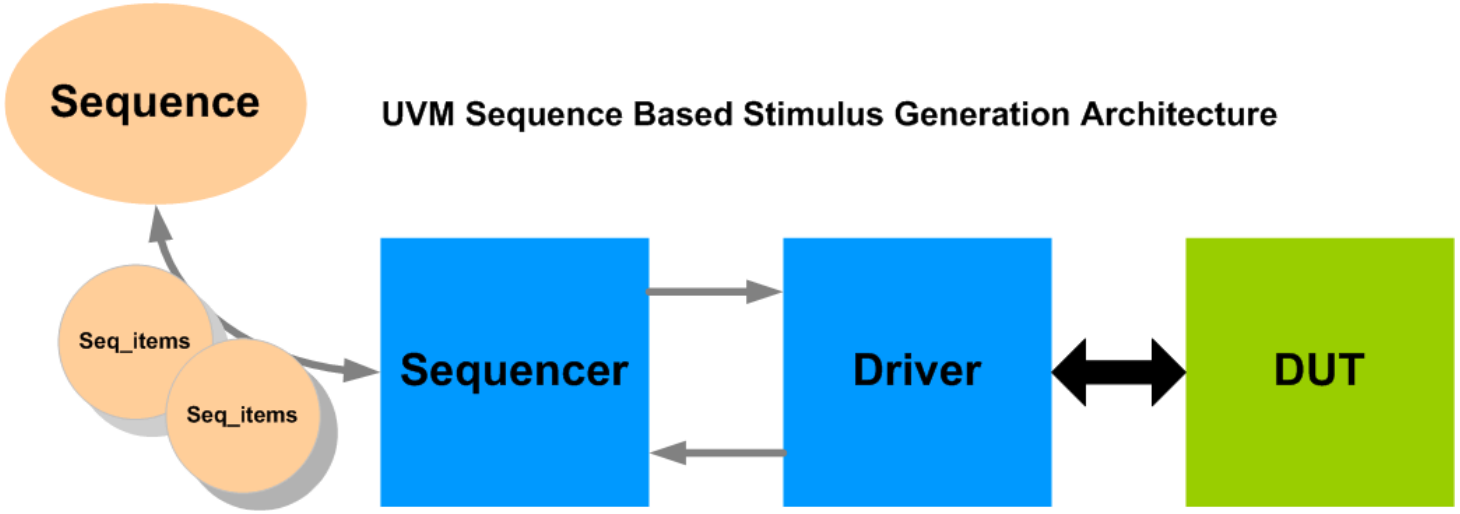


UVM sequence

- Made up of many data items put together in different ways to create interesting scenarios.
- Primary means of generating stimulus in the UVM
- Ex: Processor with instructions, ADD, load_A, ST_R, MUL, load_B. An addition would require the instructions as load_A, load_B, ADD, ST_R.
- A sequence generates a series of sequence_item's and sends it to the driver via sequencer.
- Advantages
  - Sequences can be reused.
  - Stimulus generation is independent of testbench.
  - Easy to control the generation of transaction.
  - Sequences can be combined sequentially and hierarchically.

UVM sequence

- A complete sequence generation requires following 4 classes.
  1- Sequence item
  2- Sequence
  3- Sequencer
  4- Driver



UVM sequence

- A uvm_sequence is derived from an uvm_sequence_item
- A sequence is parameterized with the type of sequence_item, this defines the type of the item sequence that will send/receive to/from the driver.

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

Ex:

```
class write_sequence extends uvm_sequence #(mem_seq_item);
....
....
endclass
```

- **request/req:**
  - A transaction that provides information to initiate the processing of a particular operation.
- **response/rsp:**
  - A transaction that provides information about the completion or status of a particular operation.

UVM sequence

- Create a user-defined class inherited from *uvm_sequence*, register with factory and call *new*
- Declare the default sequencer to execute this sequence
- Define the body method

```
class my_sequence extends uvm_sequence #(my_seq_item);
  `uvm_object_utils(my_sequence)

 `uvm_declare_p_sequencer(my_sequence)

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

   task body();
   ...
   endtask
endclass
```

UVM sequence

- Declare the default sequencer to execute this sequence
- p_sequencer
  - a variable, used as a handle to access sequencer properties
  - defined using the macro `uvm_declare_p_sequencer(SEQUENCER_NAME)

UVM sequence

- Body method
  - The operation which is intended to do by sequence is defined inside a body method.
  - Along with a body() method, pre_body, and post_body methods are called by default
  - These pre_body and post_body tasks are additional (which are useful to perform any operation before and after the execution of the body() method.
  - pre_body() and post_body() methods are optional.

```
task pre_body();
...
endtask
task body();
...
endtask
task post_body();
...
endtask
```

UVM sequence

- Steps to write a sequence
  - Create a seq_item, randomize it, and then send it to the driver.
  - To perform this operation any one of the following approaches is followed in the sequence.
    - Using macros like `uvm_do , `uvm_create, `uvm_send etc
    - Using existing methods from the base class

    a. Using wait_for_grant(), send_request(), wait_for_item_done() etc

    b. Using start_item/finish_item methods.

UVM sequence

Macros	Description
`uvm_do (seq/item)	Create, randomize and send to the driver will be executed
`uvm_do_with (seq/item, constraints)	`uvm_do + constraints can be defined while randomizing
`uvm_do_pri(seq/item, priority)	`uvm_do + mentioned priority is considered.
`uvm_do_pri_with(seq/item, constraints, priority).	Combination of `uvm_do_with and `uvm_do_pri
`uvm_create(seq/item)	Creates a sequence or item.

UVM sequence

Macros	Description
`uvm_send(seq/item)	Sends seq/item without creating and randomizing it. (So, make sure the seq/item is created and randomized first.)
`uvm_rand_send(seq/item)	Directly sends a randomized seq/item without creating it. So, make sure the seq/item is created first.
`uvm_rand_send_with(seq/item)	Directly sends a randomized seq/item with constraints but without creating it. So, make sure seq/item is created first
`uvm_send_pri(seq/item, Priority)	`uvm_send + priority is also considered.
`uvm_rand_send_pri(seq/item, Priority)	Combination of `uvm_rand_send and `uvm_send_pri
`uvm_rand_send_pri_with(seq/item, Priority)	Combination of `uvm_rand_send_with and `uvm_send_pri.

UVM sequence

- Note:

- `uvm_do macro call does not invoke pre_body and post_body methods
- A sequence macro call is not recommended to use because it takes more time to execute on the simulator which results in slow simulation.

UVM sequence Examples Using macros

```
class my_sequence extends uvm_sequencer #(seq_item);
  `uvm_object_utils(my_sequence)
   function new (string name = "my_sequence")
     super.new(name);
   endfunction

  task body();
    `uvm_do(req);
  endtask
endclass
```

```
class my_sequence extends uvm_sequencer #(seq_item);
  `uvm_object_utils(my_sequence)
   function new (string name = "my_sequence")
     super.new(name);
   endfunction

   task body();
     `uvm_do_with(req, {req.<variable> == 0;}); // any constraint
   endtask
endclass
```

UVM sequence Examples Using macros

```
class my_sequence extends uvm_sequencer #(seq_item);
  `uvm_object_utils(my_sequence)
   function new (string name = "my_sequence")
     super.new(name);
   endfunction
   task body();
     `uvm_create(req);
      assert(req.randomize());
      `uvm_send(req);
   endtask
endclass
```

```
class my_sequence extends uvm_sequencer #(seq_item);
  `uvm_object_utils(my_sequence)
   function new (string name = "my_sequence")
     super.new(name);
   endfunction

   task body();
     `uvm_create(req);
     `uvm_rand_send(req);
   endtask
endclass
```

UVM sequence Using base methods

- Using wait_for_grant(), send_request(), wait_for_item_done()
- Using start_item/finish_item methods

UVM sequence Using base methods

- Methods are defined in the uvm_sequence_base class which is derived from the uvm_sequence_item class

Type	Methods	Description
function	create_item	Creates and initializes request items or sequences and initializes using the factory.
task	wait_for_grant	Issues a request to the current sequencer and it returns when the sequencer has granted the sequence. Hence, it is a blocking method that waits for a grant from the sequencer.
function	send_request(uvm_sequence_item request, bit rerandomize = 0)	Send request items to the driver via the sequencer. If rerandomize = 1, the item will be randomized before sent to the driver.
task	wait_for_item_done()	This task will block until the driver calls put or item_done. It is an optional call.
	get_response (RSP)	This is optional in case the driver sends back any response

UVM sequence Using base methods

- Steps:

1. Create a seq_tem using create() method.
2. wait_for_grant
3. Randomize seq_item
4. send_request(req)
5. wait_for_item_done()
6. get_response (rsp)

```
class my_sequence extends uvm_sequencer
                                    #(seq_item);
  `uvm_object_utils(my_sequence)

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

  task body();
    req = seq_item::type_id::create("req");
    wait_for_grant();
    assert(req.randomize());
    send_request(req);
    wait_for_item_done();
    get_respose(rsp);
  endtask

endclass
```

UVM sequence Using base methods

- Using start_item/finish_item methods
- The start_item and finish_item tasks are also defined in the uvm_sequence_base class.
- Both methods initiate the operation of sequence items.
- Note: There should be no simulation time consumed between start_item and finish_item call.

Methods	Description
start_item(req)	It blocks until the sequencer grants the sequence and the sequence_item access to the driver.
finish_item(req)	It blocks the driver until it finishes the transfer protocol for the sequence item.

UVM sequence Using base methods

- Steps:

1. Create a seq_tem using create() method.
2. start_item(req)
3. Randomize seq_item
4. finish_item(req)

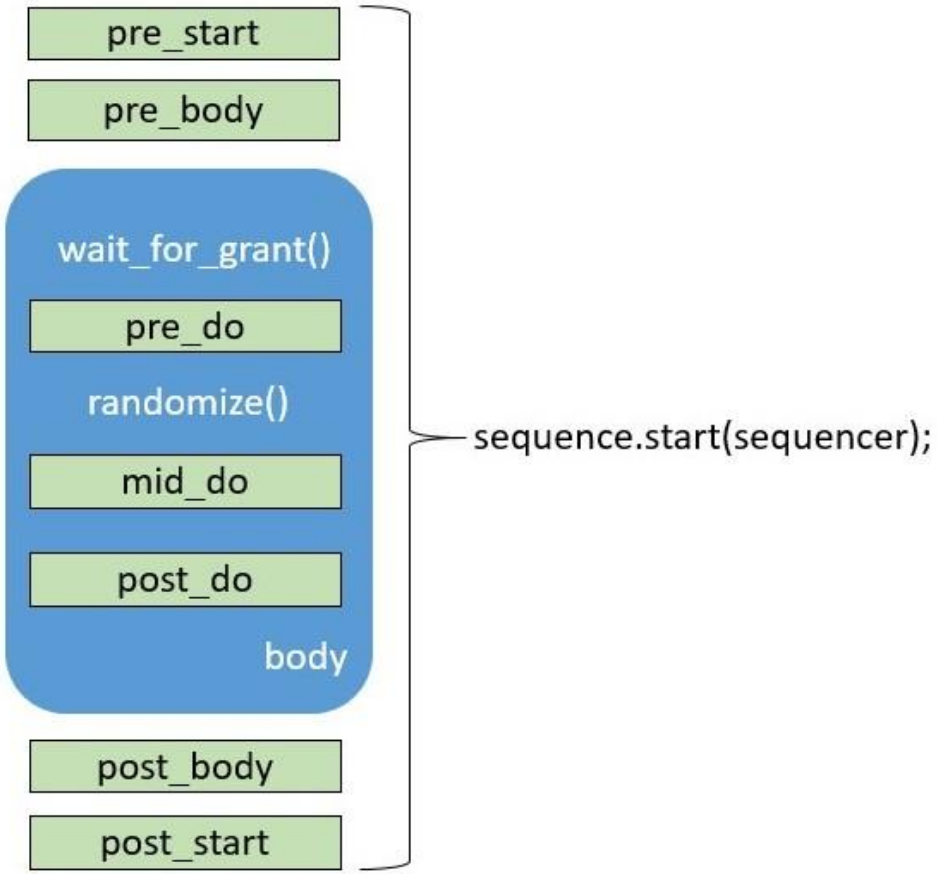
```
class my_sequence extends uvm_sequencer #(seq_item);
  `uvm_object_utils(my_sequence)
   function new (string name = "my_sequence")
     super.new(name);
   endfunction
   task body();
     req = seq_item::type_id::create("req");
     start_item(req);
     assert(req.randomize());
     finish_item(req);
   endtask
endclass
```

UVM sequence

- Start a Sequence
- A sequence is started by calling the start method that accepts a pointer to the sequencer through which sequence_items are sent to the driver.
- A pointer to the sequencer is also commonly known as m_sequencer.
- The start method assigns a sequencer pointer to the m_sequencer and then calls the body() task.
- On completing the body task with the interaction with the driver, the start() method returns.
- As it requires interaction with the driver, the start is a blocking method.

UVM sequence

- Following methods are called during sequence execution via the start method



UVM sequence

- Following methods are called during sequence execution via the start method

Method type	Methods	Description
task	pre_start	It is a user-definable callback that is called before the optional execution of the pre_body task.
task	pre_body	It is a user-definable callback that is called before the execution of body only when the sequence is started with start.
task	pre_do	It is a user-definable callback task that is called on parent sequence (if any) before the item is randomized and after sequence has issued wait_for_grant() call.
function	mid_do	It is a user-definable callback function that is called after the sequence item is randomized, and just before the item is sent to the driver.

UVM sequence

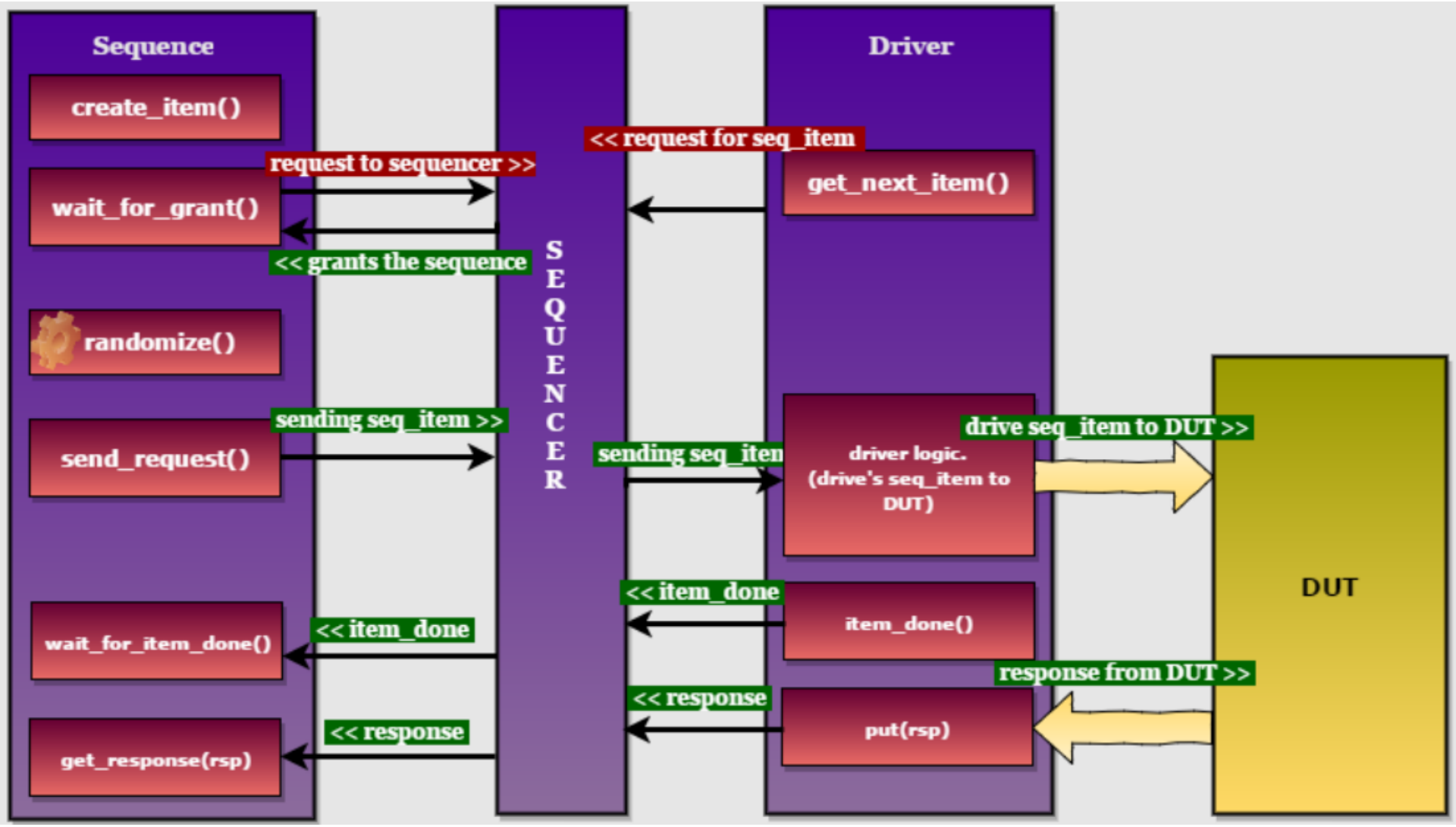
- Following methods are called during sequence execution via the start method

Method type	Methods	Description
task	body	It is a user-defined task to write main sequence code.
function	post_do	It is a user-definable callback function that is called after completing the item using either put or item_done methods.
task	post_body	It is a user-definable callback task that is called after the execution of the body only when the sequence is started with the start method.
task	post_start	It is a user-definable callback that is called after the optional execution of the post_body task.

UVM sequence

- mid_do and post_do are functions and other methods are tasks.
- The pre_start and post_start methods are always called.

UVM sequence - handshake between the sequence, sequencer and driver



UVM sequence - Example

- Assume a processor instructions - PUSH_A,PUSH_B,ADD,SUB,MUL,DIV and POP_C
- sequence Item

```
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
```

UVM sequence - Example

- sequence
  - Define an operation addition using uvm_sequence
  - The instruction sequence should be "PUSH A PUSH B ADD POP C"

```
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)

  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
```

Difference between m_sequencer and p_sequencer

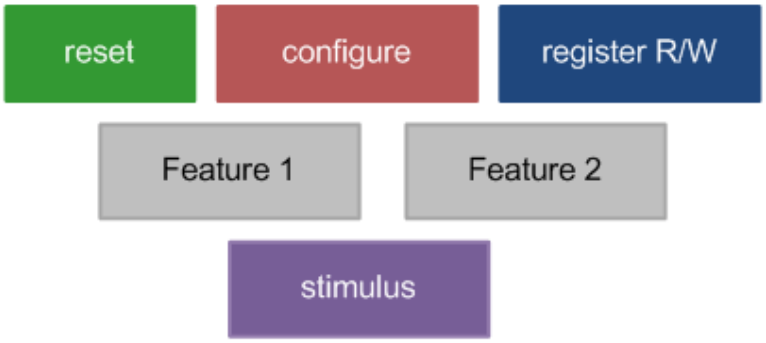
m_sequencer

- Contains reference to the sequencer(default sequencer) on which the sequence is running.
- Determined by:
  - sequencer handle provided in the start method
  - sequencer used by the parent sequence
  - sequencer that was set using the set_sequencer method

p_sequencer

- a variable, used as a handle to access sequencer properties
- defined using the macro `uvm_declare_p_sequencer(SEQUENCER_NAME)

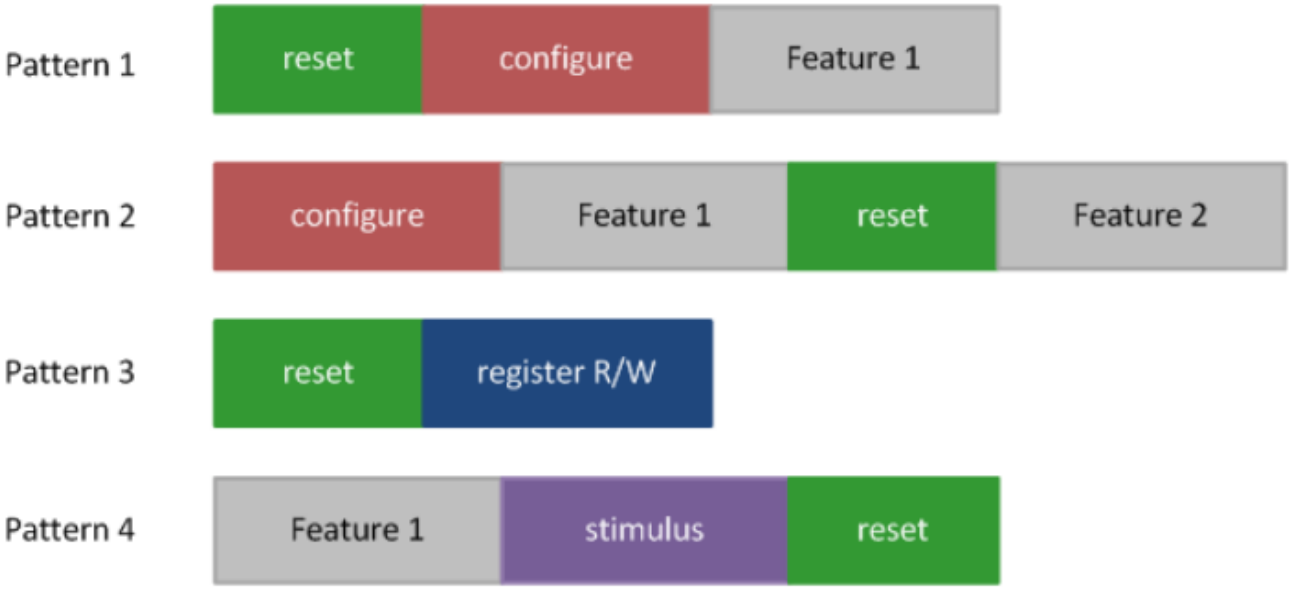
Creating and using a sequence



Sequence Library

Some options for sequences are:

- Use existing sequences to drive stimulus to the DUT individually
- Combine existing sequences to create new ones
- Pull random sequences from sequence library and execute them on DUT

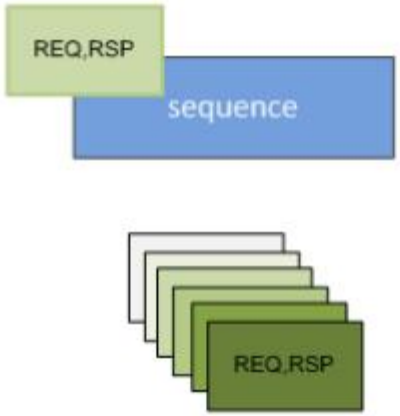


Creating and using a sequence

Sequences can do operations on sequence items, or start new sub-sequences.

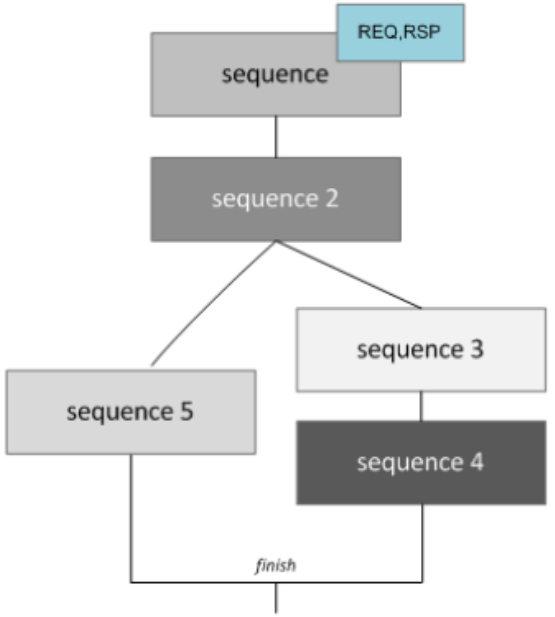
- Execute using the start() method of a sequence or `uvm_do macros
- Execute sequence items via start_item/finish_item or `uvm_do macros

Sequence can run multiple items of same type



Use start_item/finish_item
or `uvm_do_* macros

Sequence can spawn multiple sub-sequences



Use start () method or
`uvm_do_* macros

Creating and using a sequence

1. Derive from uvm_sequence base class with a specified data object type.
2. Register the sequence with the factory using `uvm_object utils.
3. Set the default sequencer that should execute this sequence.
4. Code the main stimulus inside the body() task.

Sequence execution

- Most important properties of a sequence are:
  - Body method
  - m_sequencer handle

- The sequence gets executed on calling *start of the sequence* from the test.

  Sequence_name.start(sequencer_name);

- sequencer_name specifies on which sequencer the sequence has to run
  - There are methods, macros and pre-defined callbacks associated with uvm_sequence.
  - Users can define the methods(task or function) to pre-defined callbacks, these methods will get executed automatically upon calling start of the sequence.
  - These methods should not be called directly by the user.