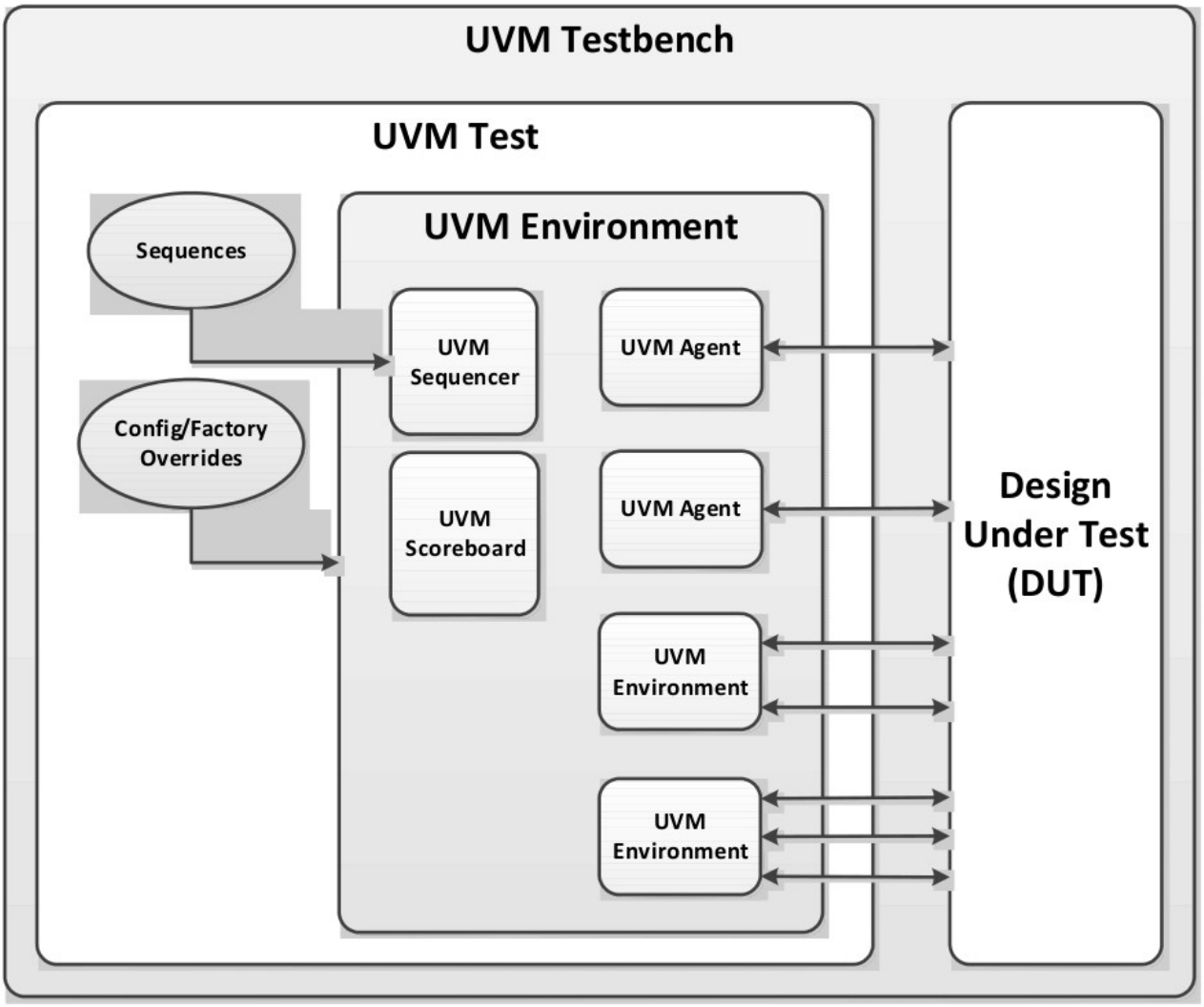
1. UVM Testbench Component



- **UVM Testbench**
  - UVM Testbench bao gồm 2 thành phần chính là UVM Test & DUT (và các config giữa chúng)
- **UVM Test**
  - UVM Test là top-level của các UVM components, thực hiện 3 chức năng chính:
    - Instantiate top-level environment
    - config environment (thông qua factory override hoặc configuration database)
    - apply stimulus cho DUT bằng cách gọi UVM sequence thông qua environment
- **UVM Environment**
  - UVM Environment là lớp chứa các thành phần gồm UVM agent, scoreboard, hoặc các UVM environment khác(top environment chứa các environment khác của DUT. Ví dụ 1 SoC design environment chứa PCIe environment, USE environment, Mem Controller environment).
- **UVM Scoreboard**
  - UVM Scoreboard có chức năng kiểm tra hành vi của DUT, UVM Scoareboard nhận transaction chứa các input và output của DUT thông qua Agent's Analysis Port (bên dưới), đưa input qua Reference Model (kết quả tin cậy) và so sánh kết quả tin cậy với kết quả của DUT

- **UVM Agent**
  - UVM Agent chứa các lớp tương tác trực tiếp với interface. Agent chứa:
    - UVM Sequence: kiểm soát stimulus flow
    - UVM Driver: apply các stimulus vào interface
    - UVM Scoreboard: monitor các thay đổi của interface
    - Agent có thể chứa các thành phần khác như coverage collectors, protocol checker, etc
- **UVM Sequencer**
  - UVM Sequencer có chức năng kiểm soát các stimulus được đưa vào DUT, các stimulus này được sinh ra bởi một hoặc các UVM Sequence khác nhau
- **UVM Sequence**
  - UVM Sequence có chức năng tạo ra các stimulus khác nhau (Sequence không nằm trong cây kế thừa component).
- **UVM Driver**
  - UVM Driver nhận các UVM Sequence Item transaction từ UVM Sequencer và lái chúng vào DUT Interface. Driver cũng có một TLM port để nhận các transaction từ Sequencer và có quyền truy xuất vào Interface để lái các tín hiệu
- **UVM Monitor**
  - UVM Monitor lấy mẫu từ DUT Interface, đóng gói các thông tin thành transaction để chuyển các transaction đó tới các thành phần khác. Tương tự như Driver, Monitor cũng cần có quyền truy xuất trực tiếp tới DUT Interface để lấy mẫu và có một TLM analysis port để broadcast các transaction được khởi tạo từ Monitor.

TRANSACTION-LEVEL MODELING (TLM)

2. TLM 1.0

- port: thành phần của producer để xuất transaction (hình vuông)
- export: thành phần của consumer để nhận transaction (hình tròn)
- uvm_analysis_port: có chức năng tựa như broadcast transaction từ 1 port sang nhiều export (hình thoi)

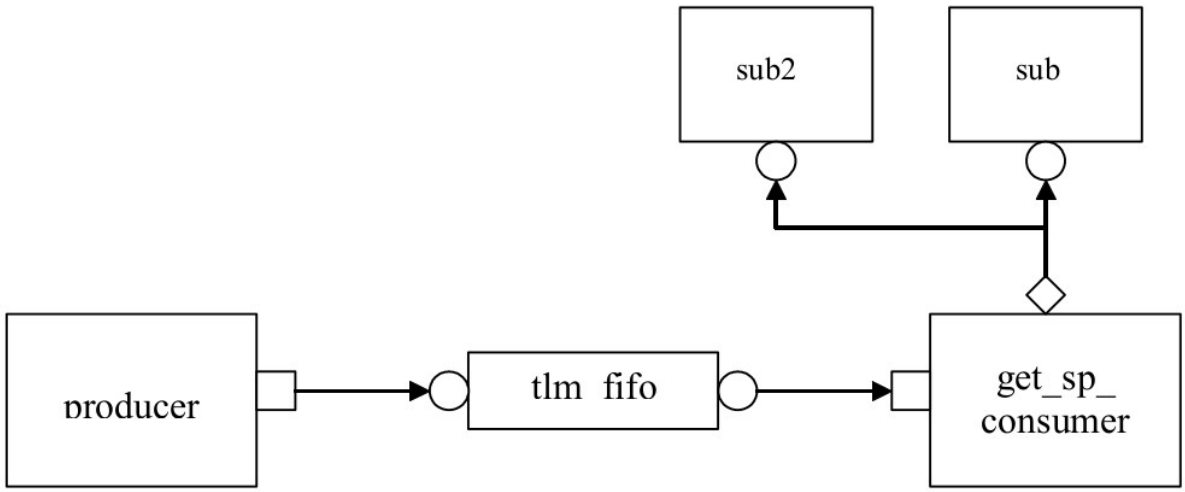


Figure 8—Analysis Communication

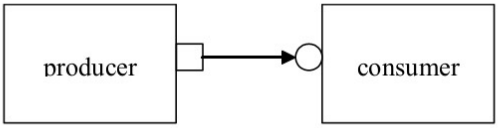
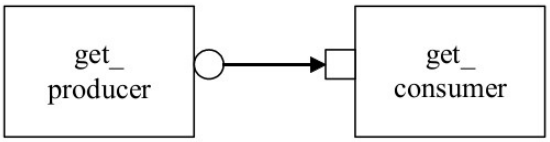


Figure 4—Single Producer/Consumer

- Hàm put()
  - put() được khởi tạo bởi consumer và được gọi bởi producer



- Hàm get()
  - get() được khởi tạo bởi producer và được gọi bởi consumer

3. TLM 1.0 - Giao tiếp giữa các tiến trình:

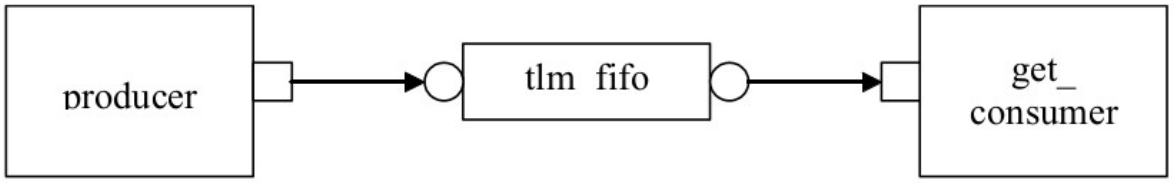


Figure 6—Using a uvm_tlm_fifo

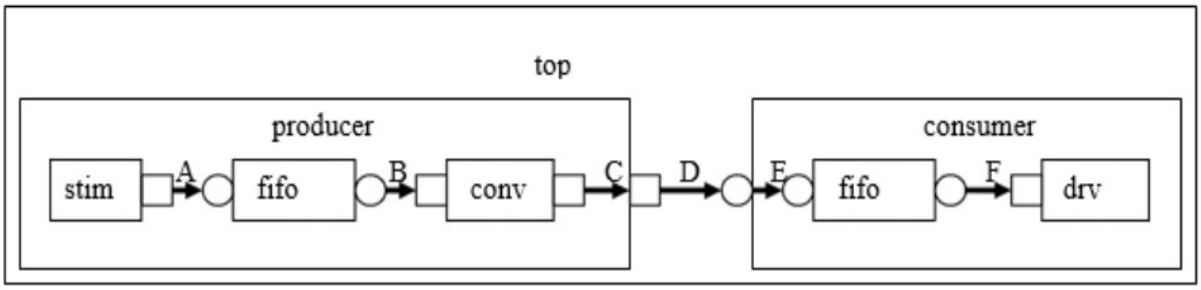
- tlm fifo nằm giữa producer và get_consumer. Transaction được tạo từ producer được đưa vào fifo và sẽ được consumber get(). Việc này giúp cho producer và consumber hoạt động độc lập không cần phải chờ đợi lẫn nhau. (producer sẽ ngừng put() vào fifo nếu fifo đã đầy)

4. TLM 1.0 - Blocking và Non-blocking

- Blocking: put() và get() là blocking, sẽ block cho tới khi thỏa yêu cầu được thỏa
- Non-Blocking: try_put(), try_get(), try_peek() trả về TRUE nếu có transaction, FALSE nếu không có transaction, được thực thi ngay lập tức trong mô phỏng

5. TLM 1.0 - Peer-to-peer connection

- Khi nằm chung 1 tầng kế thừa, port luôn kết nối với export. Ngược lại nếu khác tầng kế thừa, ta có thể khởi tạo liên kết port-to-port hoặc export-to-export



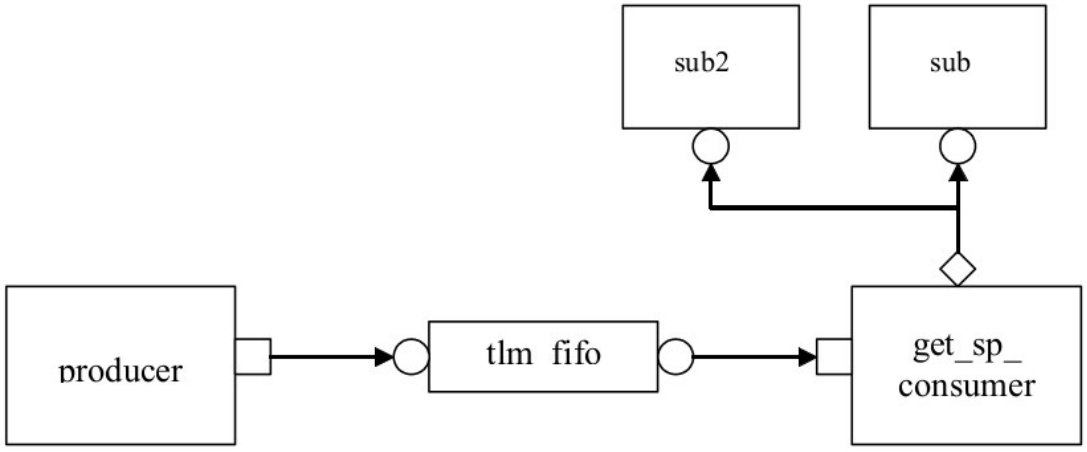
- Peer-to-peer: liên kết A, B, D, F
- Port-to-port: C
- Export-to-export: E

Table 1—TLM Connection Types

Connection type	connect() form
port-to-export	`comp1.port.connect(comp2.export);`
port-to-port	`subcomponent.port.connect(port);`
export-to-export	`export.connect(subcomponent.export);`

NOTE—The argument to the `port.connect()` method may be either an export or a port, depending on the nature of the connection (that is, peer-to-peer or hierarchical). The argument to `export.connect()` is always an export of a child component.

6. TLM 1.0 - Analysis port



- Analysis port (hình thoi): là một port TLM có một chức năng duy nhất là write().
- Một analysis port liên kết với một hoặc nhiều export() để broadcast transaction tới các export khác nhau, do đó tương tự như put() và get() của producer và consumer, hàm write() được khởi tạo bởi các export và được gọi bởi analysis port.
- Các hàm write() của từng export sẽ được analysis port gọi lần lượt.

DEVELOPING REUSABLE VERIFICATION COMPONENTS

7. Modeling data Item for Generation

- Data Item là:
  - Là các stimulus được đưa vào DUT
  - Các transaction đưa đi xung quanh môi trường verification
  - Khởi tạo của các class được định nghĩa bởi người dùng
  - coverage và checking
- NOTE: uvm_sequence_item là base class của UVM, các data item nên được khởi tạo từ uvm_sequence_item
- Các bước tạo một data item:
  - Xác định transaction specification (properties, constraint, task, function)
  - Khởi tạo từ uvm_sequence_item
  - Khởi tạo constructor
  - Thêm các “knob”

- Gọi các UVM macros để kích hoạt các hàm (printing, copying, comparing, etc)
- Xác định các hàm do_* để dùng trong khởi tạo, so sánh, printing, packing và unpacking cho các transaction

- UVM có các hàm built-in cho các transaction (print(), copy(), compare(), get_transaction_id() trong uvm_sequence_item)
- UVM macros thực thi các lệnh utilities (copy, compare, print), `uvm_object_utils macro

```
1  class simple_item extends uvm_sequence_item;
2    rand int unsigned addr;
3    rand int unsigned data;
4    rand int unsigned delay;
5    constraint c1 { addr < 16'h2000; }
6    constraint c2 { data < 16'h1000; }
7    // UVM automation macros for general objects
8    `uvm_object_utils_begin(simple_item)
9      `uvm_field_int(addr, UVM_ALL_ON)
10     `uvm_field_int(data, UVM_ALL_ON)
11     `uvm_field_int(delay, UVM_ALL_ON)
12   `uvm_object_utils_end
13   // Constructor
14   function new (string name = "simple_item");
15     super.new(name);
16   endfunction : new
17 endclass : simple_item
```

  - line 1: Khởi tạo data item từ uvm_sequence_item để tạo ra sequence item
  - line 5 & line 6: constraint cho các phần tử rand của transaction
  - line 7 tới line 12: UVM macros để cấp các utilities cho transaction (copy, compare, print, pack)
- Transaction kế thừa từ transaction

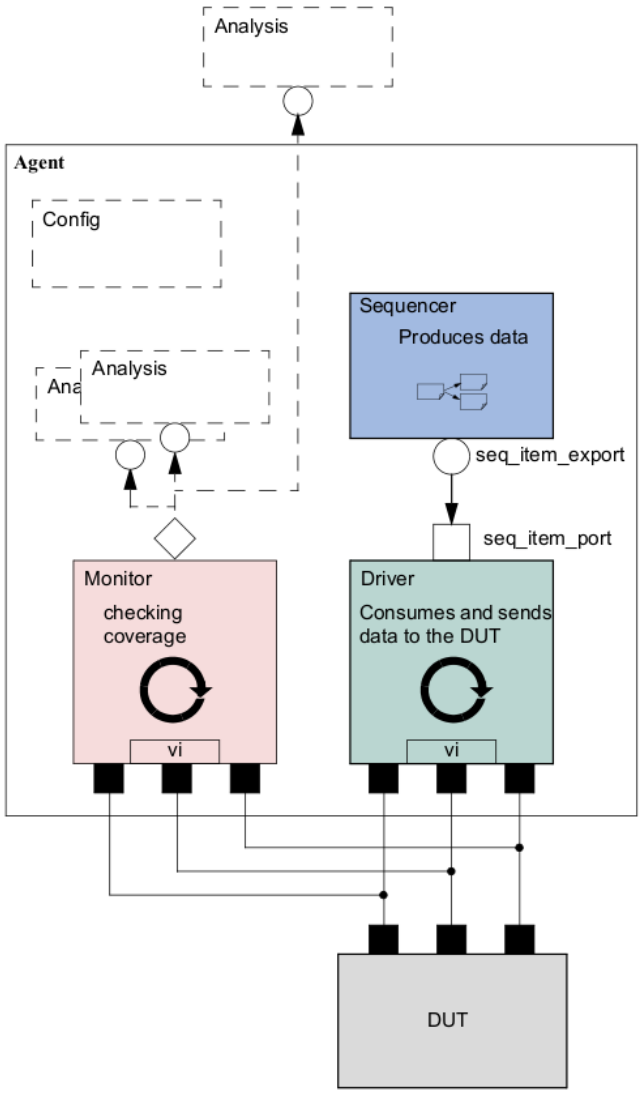
```
class word_aligned_item extends simple_item;
     constraint word_aligned_addr { addr[1:0] == 2'b00; }
     `uvm_object_utils(word_aligned_item)
     // Constructor
     function new (string name = "word_aligned_item");
```



```
       super.new(name);
     endfunction : new
   endclass : word_aligned_item
```

  - word_aligned_item được kế thừa từ simple_item có các constraint cũ của simple item và thêm một constraint mới là word_aligned_addr
  - NOTE: để sử dụng chức năng kế thừa thì lớp cha (base class) nên sử dụng các virtual method để cho phép lớp con có thể override các function

- Các thành phần của một transaction-level verification enviromnent bao gồm:
  - generator (sequencer): tạo ra và điều hướng các sequence cho DUT
  - driver: chuyển các transaction thành signal để đưa vào DUT (transactions => signals)
  - monitor: theo dõi các hoạt động signal-level của DUT Interface và biến chúng thành các transaction (signals => transactions)
  - analysis components: coverage collector hoặc scoreboard
- UVM cho phép tái sử dụng các thành phần:



8. Khởi tạo Driver

- Nhiệm vụ của driver là lái các data item vào bus thông qua giao thức Interface. Driver lấy data từ sequencer. UVM có built-in class là uvm_driver, lớp driver do người dùng khởi tạo nên được gọi từ lớp này. Lớp driver có một TLM port để giao tiếp với sequencer. Lớp driver có thể được thực hiện nhiều hơn 1 lần ở run_phase
- Các bước để khởi tạo một lớp driver
  - Khởi tạo từ lớp nền uvm_driver
  - (optional) thêm các UVM macros để sử dụng các tiện ích của UVM (print, copy, compare, etc)
  - Lấy data item tiếp theo từ sequencer và thực hiện việc lái
  - Xác định virtual interface trong driver để kết nối driver với DUT
- ví dụ:

```
1  class simple_driver extends uvm_driver #(simple_item);
2    simple_item s_item;
3    virtual dut_if vif;
4    // UVM automation macros for general components
5    `uvm_component_utils(simple_driver)
6    // Constructor
7    function new (string name = "simple_driver", uvm_component parent);
8      super.new(name, parent);
9     endfunction : new
10   function void build_phase(uvm_phase phase);
11     string inst_name;
12     super.build_phase(phase);
13       if(!uvm_config_db#(virtual dut_if)::get(this,
14                                               "","vif",vif))
15        `uvm_fatal("NOVIF",
16                   {"virtual interface must be set for: ",
17                     get_full_name(),".vif"});
18   endfunction : build_phase
19   task run_phase(uvm_phase phase);
20     forever begin
21       // Get the next data item from sequencer (may block).
22       seq_item_port.get_next_item(s_item);
23       // Execute the item.
24       drive_item(s_item);
25       seq_item_port.item_done(); // Consume the request.
26     end
27   endtask : run
28
29   task drive_item (input simple_item item);
30     ... // Add your logic here.
31   endtask : drive_item
32 endclass : simple_driver
```

- Line 1: Gọi driver từ uvm_driver
- Line 5: Thêm các UVM infrastructure macros
- Line 13: Lấy resource để define virtual interface
- Line 22: gọi get_next_item() để lấy data item kế tiếp từ sequencer để execute
- Line 25: Báo cho sequencer việc execute data hiện tại đã hoàn thành
- Line 30: thêm các application-specific logic vào việc lái data

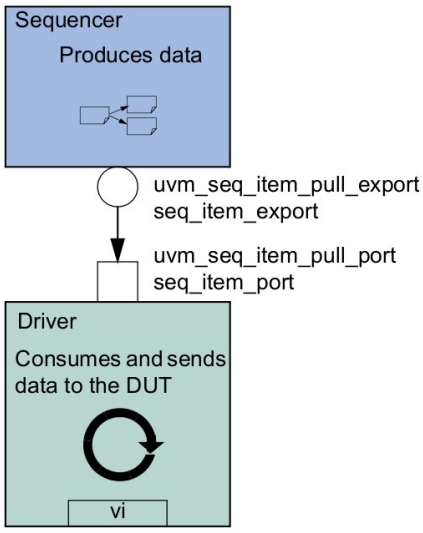
9. Khởi tạo sequencer

- UVM có cung cấp base class để khởi tạo sequencer là lớp uvm_sequencer (được khởi tạo với parameter gồm request và respone item). uvm_sequencer chứa các base functionality cho phép một sequence có thể giao tiếp với driver

```
uvm_sequencer #(simple_item, simple_rsp) sequencer;
```

10. Kết nối driver với sequencer

- driver và sequencer được liên kết với nhau thông qua TLM, seq_item_port của driver được kết nối với sequencer thông qua seq_item_export.



10.1. Tương tác cơ bản của driver và sequencer:

- Tương tác cơ bản của driver và sequencer là sử dụng các task get_next_item() và item_done().

```
forever begin
    get_next_item(req);
    // Send item following the protocol.
    item_done();
end
```

- get_next_item() bị chặn cho tới khi item được cấp bởi sequence đang chạy trong sequencer

10.2. Querrying for the randomized item

- Thêm vào get_next_item(), lớp uvm_seq_pull_port cung cấp task try_next_item(), dùng hàm này để driver execute một vài idle transactions (khi DUT cần được stimulate nhưng không có data để truyền).
- Ví dụ sử dụng try_next_item() thay vì get_next_item() ở task run_phase() (ví dụ ở driver) để truyền các idle transaction trong lúc chưa có data để truyền

```
task run_phase(uvm_phase phase);
    forever begin
      // Try the next data item from sequencer (does not block).
      seq_item_port.try_next_item(s_item);
      if (s_item == null) begin
        // No data item to execute, send an idle transaction.
        ...
      end
      else begin
        // Got a valid item from the sequencer, execute it.
        ...
        // Signal the sequencer; we are done.
        seq_item_port.item_done();
      end
    end
  endtask: run
```

10.3. Fetching Consecutive Randomized items

Kết nối sequencer-driver là một single item handshake, ta có thể dùng lệnh item_done() mà ko cần có respone và cung cấp respone bằng cách gọi một subsequent để put_response(r)

10.4. Sending processed data back to the sequencer

Ở một vài sequences, một giá trị được khởi tạo bị phụ thuộc vào data được khởi tạo trước đó. Mặc định, data item giữa driver và sequencer được copy bằng tham chiếu, nghĩa là những thay đổi của data bởi driver sẽ được hiện trong sequencer, ở trương hợp này driver phải gửi tín hiệu thông báo về sequencer sử dụng item_done() hoặc put_response(

```
seq_item_port.item_done(rsp);
```

or using the `put_response()` method,

```
seq_item_port.put_response(rsp);
```

or using the built-in analysis port in `uvm_driver`.

```
rsp_port.write(rsp);
```

NOTE: put_response() là blocking method, vì vậy sequence phải trả lời bằng get_response(

10.5. Using TLM-Based Drivers

Seq_item_port có sẵn rong uvm_driver là một port 2 chiều (bidirectional port), có tích hợp hàm get() và peek() để request item từ sequencer, và put() để provide a response. Vì vậy các components có thể giao tiếp với sequencer.

```
// Pause sequencer operation while the driver operates on the transaction.
   peek(req);
// Process req operation.
   get(req);
// Allow sequencer to proceed immediately upon driver receiving transaction.
   get(req);
// Process req operation.
```

NOTE: peek() là một blocking method

NOTE: get() báo cho sequencer thực hiện transaction kế tiếp. trả về transaction như peek(

Để provide a respone sử dụng blocking_slave_port, driver có thể dùng: seq_item_port.put(response)

response cũng có thể được gửi sử dụng analysis_port

11. Creating the Monitor

Monitor lấy tín hiệu từ bus và chuyển nó thành các events, data và status information. Các thông tin này có thể được truy xuất bởi các components khác thông qua TLM interfaces và channels.

Nên tách Scoreboard và Monitor ra 2 class riêng biệt, Monitor collect tín hiệu và gửi đến Scoreboard để kiểm tra.

Ví dụ bên dưới là một Monitor đơn giản chứa các function sau:

- Thu thập thông tin từ bus thông qua virtual interface (xmi trong ví dụ)

- Thu thập thông tin được sử dụng trong coverage và checking

- Thu thập data được exported tới analysis_port (item_collected_port trong ví dụ)

```
class master_monitor extends uvm_monitor;
     virtual bus_if xmi; // SystemVerilog virtual interface
     bit checks_enable = 1; // Control checking in monitor and interface.
     bit coverage_enable = 1; // Control coverage in monitor and interface.
  uvm_analysis_port #(simple_item) item_collected_port;
     event cov_transaction; // Events needed to trigger covergroups
   protected simple_item trans_collected;
   `uvm_component_utils_begin(master_monitor)
        `uvm_field_int(checks_enable, UVM_ALL_ON)
        `uvm_field_int(coverage_enable, UVM_ALL_ON)
     `uvm_component_utils_end
  covergroup cov_trans @cov_transaction;
        option.per_instance = 1;
        ... // Coverage bins definition
     endgroup : cov_trans
   function new (string name, uvm_component parent);
        super.new(name, parent);
        cov_trans = new();
        cov_trans.set_inst_name({get_full_name(), ".cov_trans"});
        trans_collected = new();
        item_collected_port = new("item_collected_port", this);
     endfunction : new
   virtual task run_phase(uvm_phase phase);
        collect_transactions(); // collector task.
     endtask : run
    virtual protected task collect_transactions();
      forever begin
        @(posedge xmi.sig_clock);
        ...// Collect the data from the bus into trans_collected.
        if (checks_enable)
          perform_transfer_checks();
        if (coverage_enable)
          perform_transfer_coverage();
        item_collected_port.write(trans_collected);
      end
     endtask : collect_transactions
   virtual protected function void perform_transfer_coverage();
      -> cov_transaction;
     endfunction : perform_transfer_coverage
   virtual protected function void perform_transfer_checks();
        ... // Perform data checks on trans_collected.
     endfunction : perform_transfer_checks
endclass : master_monitor
```

collect_transaction là protected virtual task được khởi tạo phần đầu của run() phase. Hàm này chạy một vòng lặp forever và thu thập data ngay khi có tín hiệu đã có data ở bus. Tín hiệu này sau đó được gửi đến item_collected_port để broadcast tới các component khác.
Coverage collection và checking ảnh hưởng đến run-time performance nên chỉ turn-on khi cần và turn-off khi không cần, do đó chúng phải có thể được bật tắt sử dụng setting coverage_enable hoặc checks_enable (set giá trị 0). Ví dụ:

```
uvm_config_int::set(this,"*.master0.monitor", "checks_enable", 0);
```

Bên trên, hàm perform_transfer_checks được gọi nếu check_enable = 1 và cov_transaction event được gọi nếu coverage_enable =1.

12. Instantiating components

Ta có thể khởi tạo component bằng cách gọi constructor như bên dưới

```
class my_component extends uvm_component;
      my_driver driver;
      ...
  virtual function void build_phase(uvm_phase phase);
      super.build_phase(phase);
        driver = new("driver",this);
        ...
      endfunction
endclass
```

Nhưng ta nên khởi tạo chúng bằng create() method sử dụng build() phase

```
class my_component extends uvm_component;
     my_driver driver;
       ...
  virtual function void
   build_phase(uvm_phase phase);
     super.build_phase(phase);
       driver = my_driver::type_id::create("driver",this);
         ...
     endfunction
endclass
```

type_id::create() method là type-specific static method dùng để trả về instance của desired type từ factory (trường hợp này là my_driver). Việc sử dụng factory cho phép người tester có thể derive một class mới từ my_driver và làm cho factory trả về class mới thế chỗ cho my_driver. Vì vậy, các parent component có thể dùng class mới mà ko cần phải chỉnh sửa parent class

a) Declare a new driver extended from the base component and add or modify functionality as desired.

```
class new_driver extends my_driver;
  ... // Add more functionality here.
endclass: new_driver
```

b) In your test, environment, or testbench, override the type to be returned by the factory.

```
virtual function void build_phase(uvm_phase phase);
```



```
  set_type_override_by_type(my_driver::get_type());
  super.build_phase(phase);
    new_driver::get_type());
endfunction
```

13. Creating the Agent

Lớp Agent khởi tạo và kết nối các lớp driver, monitor và sequencer sử dụng TLM connection. Agent cũng chứa các configuration information.

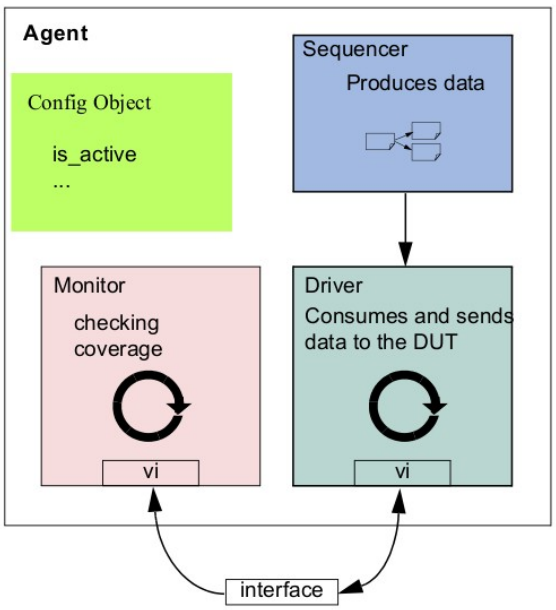


Figure 13—Agent

13.1. Operating Modes

Agent có 2 mode:

- Active mode: agent khởi động và lái tín hiệu tới DUT. mode này cần khởi tạo driver và sequencer. monitor cũng được khởi tạo để kiểm tra và coverage
- Passive mode: ở mode này driver và sequencer không cần hoạt động, chỉ cần monitor được khởi tạo. mode này được dùng ở bước checking và coverage collect

Xét ví dụ đơn giản simple_agent class bên dưới. ***thay vì sử dụng constructor thì ta sử dụng UVM build phase để configure và construct các subcomponents của agent*** - sử dụng type_id::create() để khởi tạo các subcomponents.

```
1  class simple_agent extends uvm_agent;
2    ... // Constructor and UVM automation macros
3    uvm_sequencer #(simple_item) sequencer;
4    simple_driver driver;
5    simple_monitor monitor;
6    // Use build_phase to create agents's subcomponents.
7    virtual function void build_phase(uvm_phase phase);
8      super.build_phase(phase)
9     monitor = simple_monitor::type_id::create("monitor",this);
10     if (is_active == UVM_ACTIVE) begin
11       // Build the sequencer and driver.
12       sequencer =
13         uvm_sequencer#(simple_item)::type_id::create("sequencer",this);
14       driver = simple_driver::type_id::create("driver",this);
15     end
16   endfunction : build_phase
17   virtual function void connect_phase(uvm_phase phase);
18     if(is_active == UVM_ACTIVE) begin
19       driver.seq_item_port.connect(sequencer.seq_item_export);
20     end
21   endfunction : connect_phase
22 endclass : simple agent
```

**NOTE: super.build_phase() enables the automatic configuration for UVM fields declared via the uvm_field_* macros during the build phase

Line 9: monitor được khởi tạo sử dụng create()

Line 10 - Line 15: điều kiện if() được gọi để quyết định driver và sequencer có được tạo trong agent này hay không. Nếu thỏa điều kiện (is_active = UVM_ACTIVE) thì 2 lớp tương ứng sẽ được khởi tạo.

Ở ví dụ này, configuration information chính là flag is_active. Ở tầng environment, tester có thể tạo ra các cờ như num_masters kiểu int, num_slaves kiểu int hoặc has_bus_monitor để configure.

**NOTE: hàm create() nên luôn được gọi ở build_phase() method.

Line 18 - Line 20: kiểm tra cờ is_active để kiểm tra agent có ở active mode hay không, nếu có thì kết nối giữa sequencer và driver được thực hiện thông qua connect_phase()

13.2. Connecting components

connect_phase() xảy ra sau khi build phase hoàn thành, connect_phase() nên được dùng để kết nối các components bên trong agent

14. Creating the Environment

Tester cần thiết kế một Environment có tính tái sử dụng.

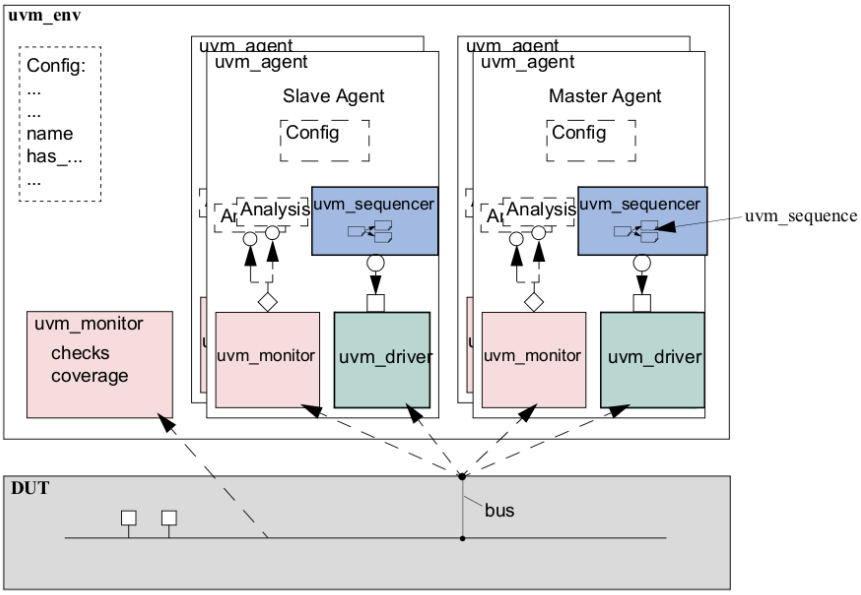


Figure 14—Typical UVM Environment Architecture

14.1. The Environment class

Environment class là top container chứa tất cả các component có thể tái sử dụng. Env khởi tạo và config tất cả các subcomponents. Hầu hết việc tái sử dụng verification xảy ra ở tầng enviromnent, việc tái sử dụng thông qua việc config lớp env và các lớp agent bên trong để có được các test cases mong muốn.

Ví dụ, tester có thể thay đổi số master và slave trong một environment mới như bên dưới:

```
class ahb_env extends uvm_env;
      int num_masters;
      ahb_master_agent masters[];
   `uvm_component_utils_begin(ahb_env)
        `uvm_field_int(num_masters, UVM_ALL_ON)
      `uvm_component_utils_end
      virtual function void build_phase(phase);
        string inst_name;
        super.build_phase(phase);
         if(num_masters ==0))
            `uvm_fatal("NONUM",{"'num_masters' must be set";
    masters = new[num_masters];
        for(int i = 0; i < num_masters; i++) begin
          $sformat(inst_name, "masters[%0d]", i);
          masters[i] = ahb_master_agent::type_id::create(inst_name,this);
        end
        // Build slaves and other components.
      endfunction
  function new(string name, uvm_component parent);
        super.new(name, parent);
      endfunction : new
endclass
```

NOTE: tương tự như agent, create được dùng để khởi tạo các subcomponents của env.

15. Enable Scenario Creation

Environment user điều khiển các enviromnent-generated patterns để config các sequencers của env. Tester có thể:

- Xác định các sequences mới để generate các transaciton mới
- Xác định các sequences mới để invoke các sequences đã tồn tại
- Override các default knob của các data item để modify driver và behavior của enviromnent
- “Enable” any new behavior/sequences

15.1. Declaring User-Defined Sequences

Sequences chứa các data item tạo nên một scenario cụ thể hoặc một patterns nào đó. Verification component có thể chứa một thư viện chứa các basic sequences. Việc này hỗ trợ cho tái sử dụng và giảm độ dài của test. Một sequence cũng có thể gọi một sequence khác để tạo ra các scenario phức tạp hơn.

**NOTE: UVM class Library cung cấp uvm_sequence base class. các sequences class nên được derived trực tiếp hoặc gián tiếp từ base class này.

Để khởi tạo một user-defined sequence:

a) tạo sequence từ uvm_sequence base class và specify request + response item type parameter.

b) Dùng `uvm_object_utils macro để register sequence type vào factory

c) nếu sequence cần access tới các derived type-specific functionality hoặc sequencer của nó. Thêm code hoặc dùng `uvm_declare_p_sequencer macro.

d) Hiện thực task body() của sequence với trường hợp cụ thể muốn test. ở body task, ta có thể execute data items và các sequences khác.

Ở ví dụ bên dưới, lớp simple_seq_do được derived từ uvm_sequence và sử dụng `uvm_object_utils macro. một sequencer đơn giản cũng được khởi tạo để chạy simple_seq_do.

```
class simple_seq_do extends uvm_sequence #(simple_item);
      rand int count;
      constraint c1 { count >0; count <50; }
      // Constructor
      function new(string name="simple_seq_do");
        super.new(name);
      endfunction
      //Register with the factory
      `uvm_object_utils(simple_seq_do)
      // The body() task is the actual logic of the sequence.
      virtual task body();
        repeat(count)
        // Example of using convenience macro to execute the item
          `uvm_do(req)
      endtask : body
   endclass : simple_seq_do

class simple_sequencer extends uvm_sequencer #(simple_item) ;
   // same parameter as simple_seq_do
   `uvm_component_utils(simple_sequencer)
   function new (string name="simple_sequencer", uvm_component parent) ;
      super.new(name,parent) ;
   endfunction
endclass
```

15.2. Sending Subsequences and Sequence Items

Sequence cho phép người dùng xác định:

- Một chuỗi data items để gửi tới DUT
- Một chuỗi các hành động được thực hiện ở DUT interfaces

Có thể dùng các sequences để generate static lists of data items mà không kết nối DUT interface.

15.2.1 Basic Flow for Sequences and Sequence Items

Để gửi một sequence item, body() của một sequence cần create() item (sử dụng factory), gọi start_item() của item đó, có thể randomize nó, và cuối cùng gọi finish_item(). Để gửi một subsequence, body của parent sequence cần tạo subsequence, có thể randomize nó, và gọi start() cho subsequence đó. (subsequence có thể gọi get_response() nếu cần giao tiếp với parent sequence)

2 hình bên dưới cho thấy flow cho sequence item và các sequences được hiện thực bơi uvm_do macros. có sử dụng các method như pre_do(), mid_do(), post_do() cho sequence.
NOTE: pre_body() và post_body() method không được gọi cho subsequences.

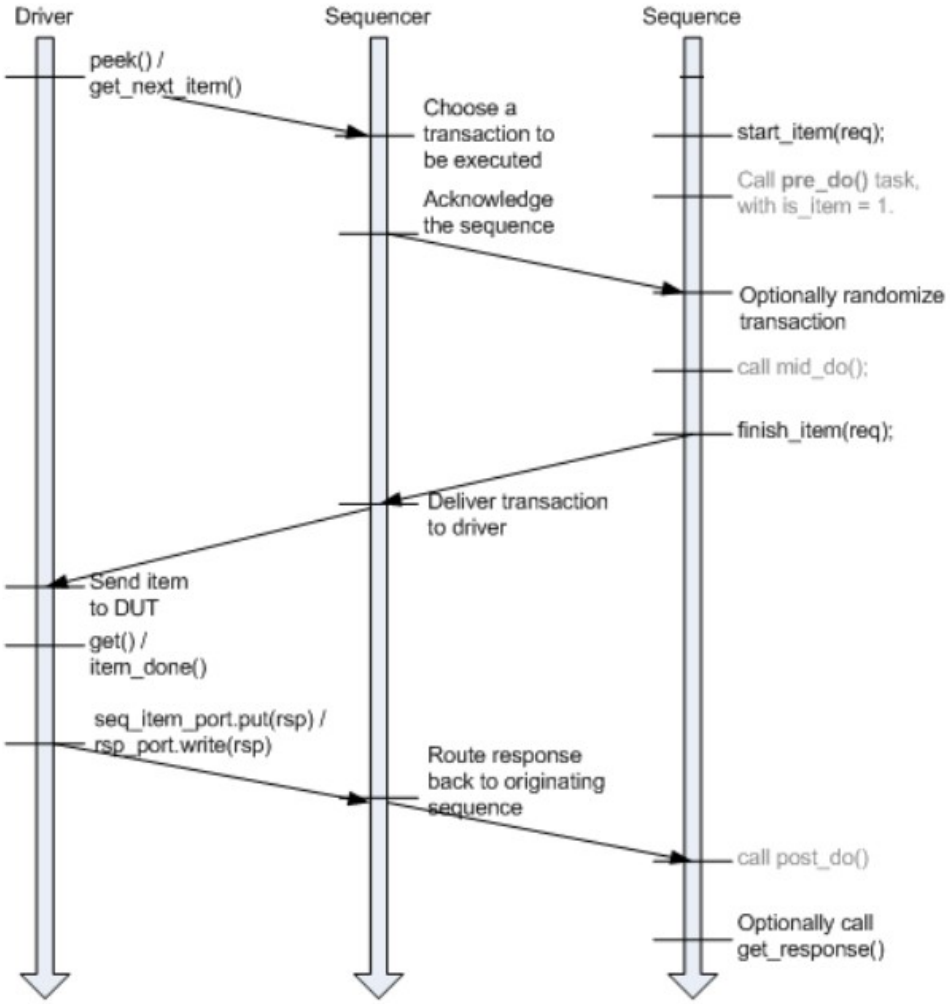
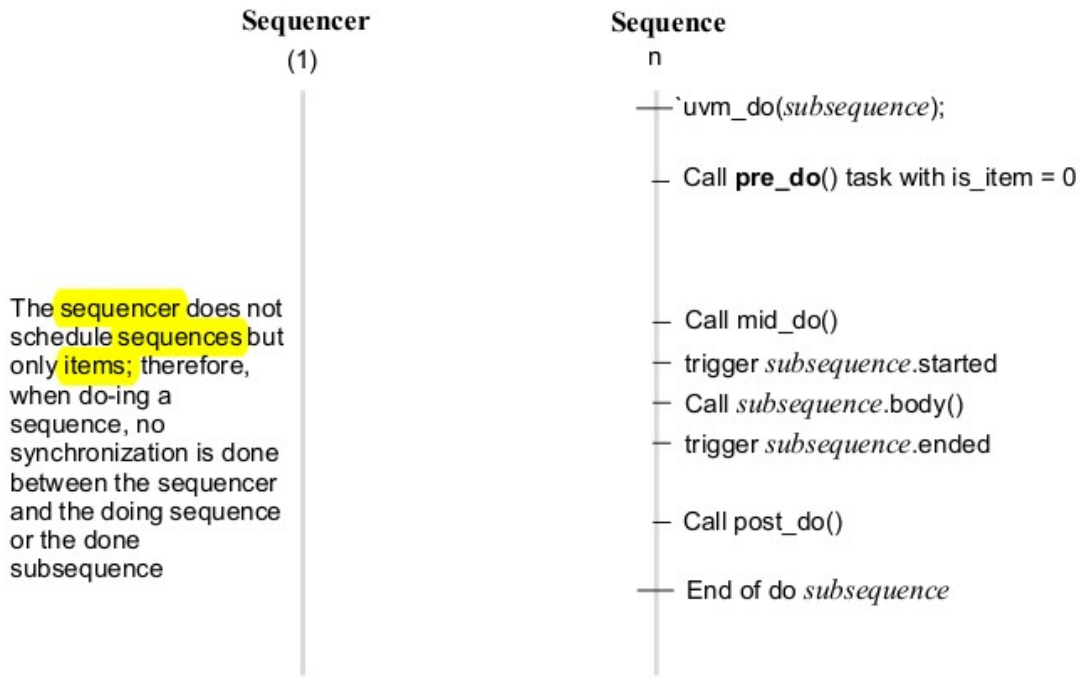


Figure 15—Sequence Item Flow in Pull Mode



Note This flow does not depend on the driver interaction mode.

Figure 16—Subsequence Flow

15.2.2 Sequence and Sequence Item Macros

`uvm_do: macro này và các biến thể provide các hàm có thể gọi để create, randomize, và gửi transaction item tới một sequence.

`uvm_do macro delay việc randomize của item cho tới khi driver gửi tín hiệu sẵn sàng để nhận sequence mới và pre_do method đã được executed.

`uvm_do_with cho phép constraint được đưa vào randomize hoặc bypass randomization

15..2.2.1 `uvm_do

macro này nhận một tham số là uvm_sequence hoặc uvm_sequence_item. Ở hình bên dưới, khi driver request một item từ sequencer, item này được randomize và đưa tới driver

```
class simple_seq_do extends uvm_sequence #(simple_item);
      ... // Constructor and UVM automation macros
      // See Section 4.7.2
      virtual task body();
        `uvm_do(req)
      endtask : body
    endclass : simple_seq_do
```

Tương tự, một biến sequence có thể được xử lý như ở figure 16. Ở ví dụ bên dưới, một sequence khác (simple_seq_sub_seqs) được khởi tạo, sử dụng `uvm_do để execute một sequence thuộc lớp simple_seq_do đã được khởi tạo phía trên

```
class simple_seq_sub_seqs extends uvm_sequence #(simple_item);
      ... // Constructor and UVM automation macros
      // See Section 4.7.2
      simple_seq_do seq_do;
      virtual task body();
        `uvm_do(seq_do)
      endtask : body
    endclass : simple_seq_sub_seqs
```

15.2.2.2 `uvm_do_with

macro này tương tự như `uvm-do. tham số đầu tiên là một biến khởi tạo từ uvm_sequence_item (bao gồm items và sequences). tham số thứ hai có thể là bất kỳ inline constraint hợp lệ nào nếu dùng trong arg1.randomize(). Việc này cho phép ta thêm các inline constraints khác nhau trong khi sử dụng item hoặc biến sequence ban đầu.

Ở ví dụ bên dưới, sequence produces 2 data items với specific constraint ở addr và data.

```
class simple_seq_do_with extends uvm_sequence #(simple_item);
      ... // Constructor and UVM automation macros
      // See Section 4.7.2
      virtual task body();
        `uvm_do_with(req, { req.addr == 16'h0120; req.data == 16'h0444; } )
        `uvm_do_with(req, { req.addr == 16'h0124; req.data == 16'h0666; } )
      endtask : body
    endclass : simple_seq_do_with
```

macro có thể được thay thế với một user-defined task

```
class simple_seq_do_with extends uvm_sequence #(simple_item);
  task do_rw(int addr, int data);
    item= simple_item::type_id::create("item",,get_full_name());
    item.addr.rand_mode(0);
    item.data.rand_mode(0);
    item.addr = addr;
    item.data = data;
    start_item(item);
    randomize(item);
    finish_item(item);
  endtask
virtual task body();
  repeat (num_trans)
    do_rw($urandom(),$urandom());
endtask
...
endclass : simple_seq_do_with
```

15.3. Starting a Sequence on a sequencer

Mặc định, Sequencer không execute bất kỳ một sequences nào. start() method cần được gọi để một hoặc nhiều sequences. Gọi start() có thể được viết trực tiếp trong code của tester. Còn một cách khác, đó là tester có thể specify một sequence có thể bắt đầu tự động ở một phase nào đó thông uvm_config_db.

15.3.1. Manual Starting

Tester có thể khởi tạo và randomize một sequence và gọi start() cho sequence đó bất kỳ lúc nào

15.3.2 Using the automated Phase-Based Starting

Mỗi khi run-time phase bắt đầu, sequencer sẽ kiểm tra xem có bất kì resource nào liên quan tới phase để xác định có sequence cho việc chạy tự động hay không. Resource này có thể được định nghĩa ở user code, cụ thể là ở test. Ví dụ, resource setting này làm cho sequencer đã được xác định bắt đầu bằng cách khởi động main_phase và khởi tạo loop_read_modify_write_seq sequence.

```
uvm_config_db#(uvm_object_wrapper)::set(this,
                ".ubus_example_tb0.ubus0.masters[0].sequencer.main_phase",
                "default_sequence",
              loop_read_modify_write_seq::type_id::get());
```

Ta cũng có thể bắt đầu một instance của sequence:

```
lrmw_seq = loop_read_modify_write_seq::type_id::create("lrmw",,
                                                       get_full_name());
// set parameters in lrmw_seq, if desired
uvm_config_db#(uvm_sequence_base)::set(this,
                  ".ubus_example_tb0.ubus0.masters[0].sequencer.main_phase",
                  "default_sequence", lrmw_seq);
```

Bằng việc khởi tạo một instance của sequence, instance đó có thể randomize với specific parameters được đặt một cách rõ ràng hoặc constraints.

15.4. Overriding Sequence Items và Sequences (Tương tự $cast I think)

Tester có thể configure sim env bằng cách chỉnh sửa sequence hoặc sequence item sẵn có bằng cách dùng factory. Các bước để override một sequence hoặc sequence item:

a) declare một lớp sequence hoặc sequence item từ base class hợp lệ. Ở ví dụ bên dưới: lớp word_aligned_item được derive từ base class là simple_item

b) gọi method override của uvm_factory (built-in method),

```
// Affect all factory requests for type simple_item.
   set_type_override_by_type(simple_item::get_type(),
      word_aligned_item::get_type());
   // Affect requests for type simple_item only on a given sequencer.
   set_inst_override_by_type("env0.agent0.sequencer.*",
```



```
      simple_item::get_type(), world_aligned_item::get_type());
   // Alternatively, affect requests for type simple_item for all
   // sequencers of a specific env.
   set_inst_override_by_type("env0.*.sequencer.*",
      simple_item::get_type(), word_aligned_item::get_type());
```

Ở hình trên:

+ Dòng đầu thực hiện affect cho cả factory, override class mới lên class cũ - set_type_override_by_type

+ Dòng thứ hai thực hiện affect cho một sequencer cụ thể - set_inst_override_by_type

+ Dòng thứ ba thực hiện affect cho tất cả sequencer trong một env cụ thể – set_inst_override_by_type

16. Managing End of Test

UVM support cơ chế objection mechanism cho phép các components giao tiếp với nhau về trạng thái hoạt động. Mỗi phase đều có một built-in objection, giúp synchronize hoạt động test và thông báo đã an toàn để kết thúc phase hay chưa.

Một process sẽ raise a phase objection khi bắt đầu hoạt động của nó, và hoạt động này cần hoàn thành trước khi nó drop objection thông báo hoạt động kết thúc, khi đó phase sẽ bị terminated.

Ví dụ: Một master agent cần phải hoàn thành các việc read và write trước khi của nó trước khi run phase được ngừng

Với các sequence, có 3 cách phase objection được handle:

a) Non-Phase Aware Sequences

1. Caller sẽ handle phase objection
2. sequence itself is not phase aware

```
class test extends ovm_test;
    task run_phase(uvm_phase phase);
        phase.raise_objection(this);
        seq.start(seqr);
        phase.drop_objection(this);
    endtask
endclass
```

b) Phase Aware Sequence (Explicit Objection)

1. caller sẽ pass the starting phase
2. sequence will explicitly call raise/drop to control objection
3. Where exactly the raise/drop is called is up to the user design.

```
class seq extends uvm_sequence #(data_item);
    task body();
        uvm_phase p = get_starting_phase();
        if(p) p.raise_objection(this);
```



```
        //some critical logic
        If(p) p.drop_objection(this);
    endtask
```

c) Phase Aware Sequences (Implicit Objection)

1. caller sẽ pass the starting phase
2. Bên trong sequence (thường là trong seq::new), tester sẽ gọi set_automatic_phase_objection(1);

3. uvm_sequence_base sẽ tự động handle phase raise/drop

```
class test extends ovm_test;
    task run_phase (uvm_phase phase);
        seq.set_starting_phase(phase);
        seq.start(seqr);
    endtask
endclass

class seq extends uvm_sequence #(data_item);
    function new(string name = "seq");
        super.new(name);
        set_automatic_phase_objection(1);
    endfunction
    task body();
        // Sequence logic with no objection
        // as it is already handled in the base class
    endtask
endclass
```

Nếu tester đang dùng cơ chế UVM task-based phases default_sequence, "caller" sẽ chính là UVM sequencer, vì thế option a ko thể dùng được trực tiếp, do đó mặc định phải sử dụng option b hoặc option c.

Khi tất cả các objection được drop, phase đang chạy sẽ kết thúc. Tuy nhiên có vài trường hợp các process chạy song song cần thêm thời một vài chu kỳ để xử lý và chuyển transaction cuối tới scoreboard. Để giải quyết vấn đề này, ta có thể dùng phase_ready_to_end() method để re-raise phase objection nếu một transaction vẫn đang di chuyển. Hoặc ta có thể dùng cách là thêm delay time cycles vào.

17. Implementing Checks and Coverage

Table 5—SystemVerilog Checks and Coverage Construct Usage Overview

	class	interface	package	module	initial	always	generate	program
assert	no	yes	no	yes	yes	yes	yes	yes
cover	no	yes	yes	yes	yes	yes	yes	yes
covergroup	yes	yes	yes	yes	no	no	yes	yes

17.1. Implementing Checks and Coverage in Classes

Class Check và coverage nên được hiện thực bên trong các classes derived từ uvm_monitor.

Bus monitor mặc định được tạo bên trong env và nếu check và coverage được kích hoạt, bus monitor sẽ thực hiện các chức năng đó.

Class check có thể được viết bằng procedural code hoặc SV immediate assertion

TIPS: dùng assertion cho simple check (ít dòng code) và dùng functions cho các complex check (nhiều dòng code)

Ví dụ bên dưới thực hiện assertion check, assertion này verifies kích thước của transfer là 1, 2, 4, 8. Nếu không assertion fail.

```
function void ubus_master_monitor::check_transfer_size();
     check_transfer_size : assert(trans_collected.size == 1 ||
        trans_collected.size == 2 || trans_collected.size == 4 ||
        trans_collected.size == 8) else begin
          // Call DUT error: Invalid transfer size!
        end
   endfunction : check_transfer_size
```

Ví dụ bên dưới là ví dụ thực hiện function check. function kiểm tra size của value có khớp với size của data dynamic array hay không.

```
function void ubus_master_monitor::check_transfer_data_size();
     if (trans_collected.size != trans_collected.data.size())
        // Call DUT error: Transfer size field / data size mismatch.
  endfunction : check_transfer_data_size
```

Ở 2 ví dụ trên, việc kiểm tra nên được thực hiện ngay sau khi transfer được thu thập bởi monitor. Do cả 2 đều xuất hiện cùng một lúc, một fuction mang tính bao bọc được tạo để chủ một function được thực hiện.

```
function void ubus_master_monitor::perform_transfer_checks();
     check_transfer_size();
     check_transfer_data_size();
   endfunction : perform_transfer_checks
```

Hàm wrapper perform_transfer_checks được thực hiện tuần tự sau khi item đã được thu thập bởi monitor

Ví dụ bên dưới là covergroup nằm trong class dược derive từ uvm_monitor

```
// Transfer collected beat covergroup.
   covergroup cov_trans_beat @cov_transaction_beat;
     option.per_instance = 1;
     beat_addr : coverpoint addr {
       option.auto_bin_max = 16; }
     beat_dir : coverpoint trans_collected.read_write;
     beat_data : coverpoint data {
       option.auto_bin_max = 8; }
     beat_wait : coverpoint wait_state {
       bins waits[] = { [0:9] };
       bins others = { [10:$] }; }
     beat_addrXdir : cross beat_addr, beat_dir;
     beat_addrXdata : cross beat_addr, beat_data;
   endgroup : cov_trans_beat
```

17.2. Implementing Checks and Coverage in Interface

Interface check được thực hiện bằng các assertion. Assertiong được thêm vào để kiểm tra hoạt động của tín hiệu trong một giao thức. Ví dụ: một assertion có thể kiểm tra một địa chỉ không bao giờ xuất hiện X hoặc Y trong một valid transfer.

17.3. Controlling Checks and Coverage

Nên có một cách để kiểm soát việc checks được thực hiện và coverage được thu thập. Ta có thể dùng UVM bit field cho mục đích này. field có thể được kiểm soát bằng uvm_config_db

interface (xem uvm_config_db trong class reference để hiểu thêm). Bên dưới là một ví dụ sử dụng checks_enable bit để kiểm soát việc check

```
if (checks_enable)
    perform_transfer_checks();
```

Nếu checks_enable được set = 0 thì checking function ko được thực hiện. Ví dụ bên dưới cho thấy cách để tắt checks cho master0.monitor

```
uvm_config_db#(int)::set(this,"masters[0].monitor", "checks_enable", 0);
```

USING VERIFICATION COMPONENTS

1. Target: Các bước cần có để xây dựng một testbench từ các reusable verification components

Người thiết kế cần phân biệt giữa:

+ Testbench intergrator: chịu trách nhiệm cho constructor và configuration testbench

+ test writer: sử dụng các thành phần để tạo ra các tests (test writer có thể skip phần configuration testbench và tới trực tiếp phần tạo ra test)

Các bước để tạo một testbench từ các verification component là:

a) Review the reusable verification component configuration parameters.

b) Instantiate and configure reusable verification components.

c) Create reusable sequences for interface verification components (optional).

d) Add a virtual sequencer (optional).

e) Add checking and functional coverage extensions.

f) Create tests to achieve coverage goals.

1. Creating a Top-Level Environment

Top-level env là một container xác định các components nằm trong UVM tests.

Top-level env khởi tạo và config các reusable verification IP và xác định các configuration mặc định của IP đó

Nhiều tests khác nhau có thể khởi tạo bằng cách sử dụng top-level env class

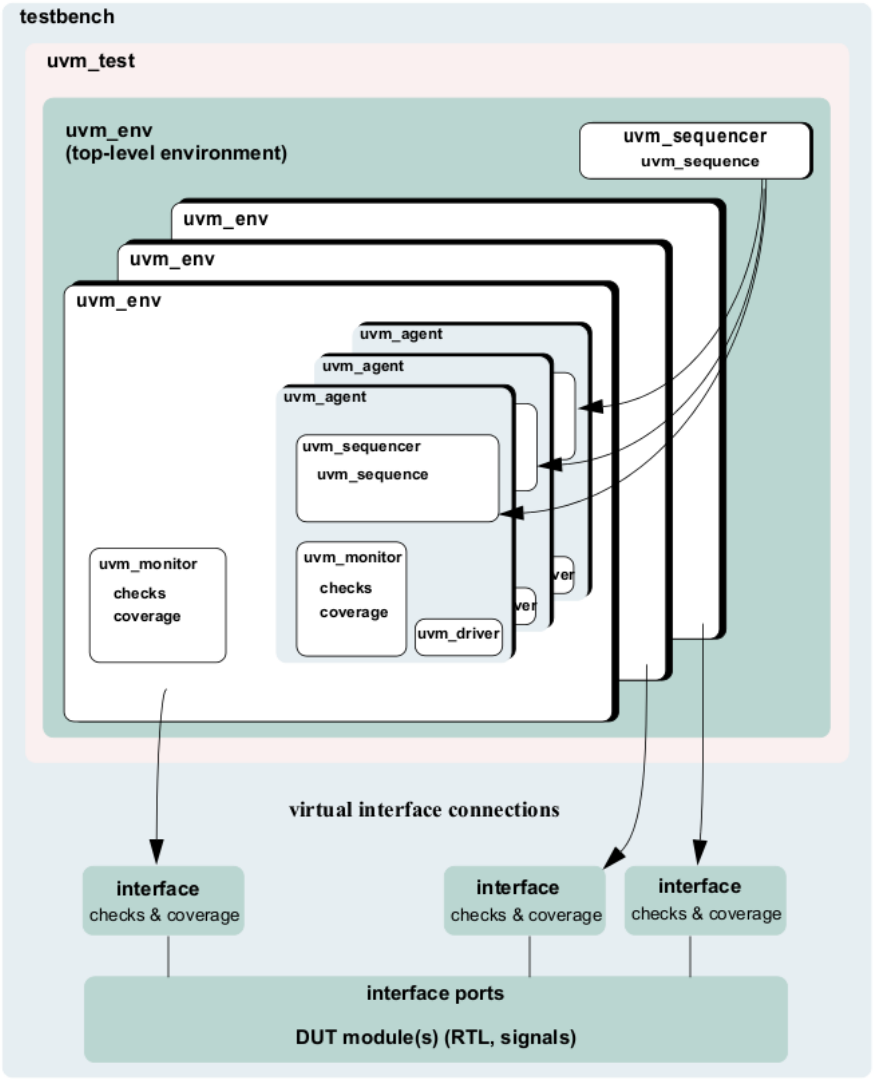


Figure 17—Verification Environment Class Diagram

2. Instantiating Verification Components

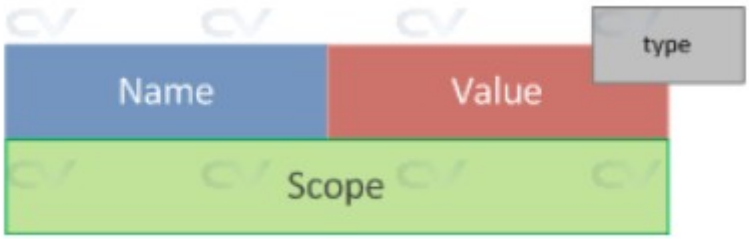
Side note about uvm_config_db

UVM config database
Nguồn: https://www.chipverify.com/uvm/uvm-config-db
UVM có một internal database cho phép lưu các biến vào dưới dạng tên và có thể được truy xuất sau này bởi các thành phần trong Testbench. Lớp **uvm_config_db** có các static function, do đó các function này phải được sử dụng :: scope để có thể gọi

- static function: https://www.chipverify.com/systemverilog/systemverilog-static-variables-functions
- :: scope operator: https://verificationguide.com/systemverilog/systemverilog-scope-resolution-operator/

database này cho phép ta lưu các configuration settings dưới các tên khác nhau được dùng để thay đổi config các thành phần của Testbench khi cần mà không cần phải chỉnh sửa lại testbench code.

Ví dụ: Để bật chức năng functional coverage cho một agent class, ta chỉ cần lấy đường dẫn của agent đó và set bên trong configuration database là 1. agent đó có thể kiểm tra giá trị của biến này và sau đó thực hiện việc coverage collecting nếu biến được bật.



1. Set()

```
static function void set (  uvm_component cntxt,
                            string        inst_name,
                            string        field_name,
                            T             value);
```

Sử dụng static function này của **uvm_config_db** để đặt giá trị cho một biến trong configuration database. Ví dụ bên dưới, set() function sẽ đặt giá trị 1 cho biến tên **cov_enable** ở đường dẫn **uvm_test_top.m_env.m_apb_agent**

```
virtual function void build_phase (uvm_phase phase);
    ...
    uvm_config_db #(int) :: set (null, "uvm_test_top.m_env.m_apb_agent", "cov_enable", 1);
    ...
endfunction
```

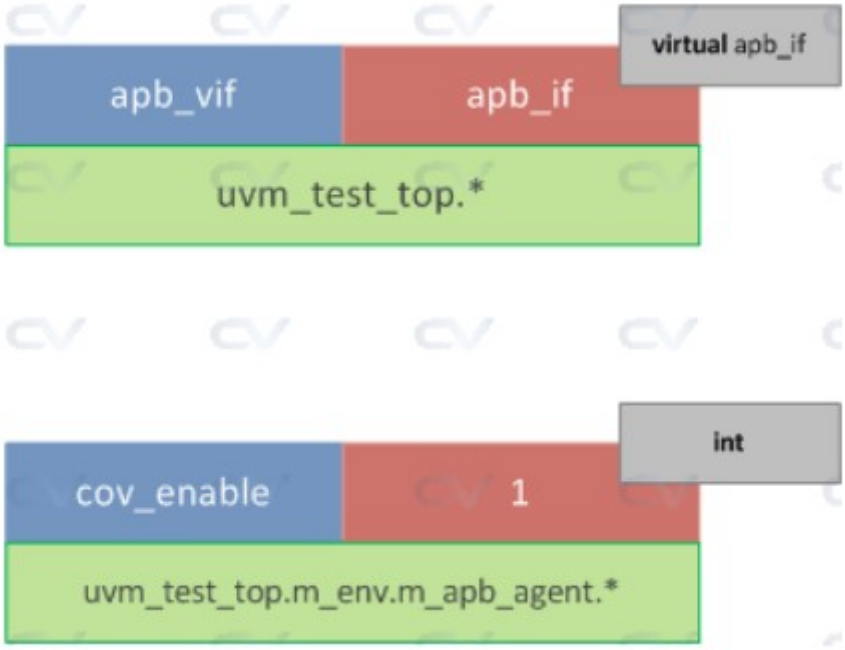
Sử dụng hàm set() sẽ tạo mới hoặc cập nhật một configuration setting đã tồn tại của “field_name”(tên biến) tại “inst_name”(đường dẫn) từ cntxt. Setting sẽ được tạo cho full scope nếu ta đặt **{cntxt, “.”, inst_name}**. Nếu cntxt được đặt là null, thì scope sẽ lấy từ inst_name. Xem ví dụ bên dưới.

```
// Set virtual interface handle under name "apb_vif" available to all components below uvm_te
uvm_config_db #(virtual apb_if) :: set (null, "uvm_test_top.*", "apb_vif", apb_if);

// Set an int variable to turn on coverage collection for all components under m_apb_agent
uvm_config_db #(int) :: set (null, "uvm_test_top.m_env.m_apb_agent.*", "cov_enable", 1);

// Consider you are in agent's build_phase then you may achieve the same effect by
uvm_config_db #(int) :: set (this, "*", "cov_enable", 1);
```

Ở ví dụ trên, dòng cuối cùng, **this,** nếu lớp hiện tại đang là uvm_test_top.m_env.m_apb_agent thì full scope chính là uvm_test_top.m_env.m_apb_agent.



2. get()

```
static function bit get (  uvm_component cntxt,
                           string        inst_name,
                           string        field_name,
                  inout    T             value);
```

Sử dụng static function get() để lấy giá trị của biến đã được set() ở file_name từ configuration database. Cần phải nhớ rằng giá trị chỉ được trả về nếu field_name scope chính xác. Ví dụ: Nếu trước đó đã set() một biến bằng tên field_name là **m_config** tại scope **"uvm_test_top.m_env.m_func_cov"**, vậy thì hàm get() cần phải đưa cùng scope đó, để có thể lấy giá trị ở trường field_name. cntxt là starting point và được nối với trường inst_name để có được scope hoàn chỉnh. Kiểm tra ví dụ bên dưới.

```
// Get virtual interface handle under name "apb_vif" into local virtual interface handle at m_e
uvm_config_db #(virtual apb_if) :: get (this, "*", "apb_vif", apb_if);

// Get int variable fails because no int variable found in given scope
uvm_config_db #(int) :: get (null, "uvm_test_top", "cov_enable", cov_var);
```

3. exist()

```
static function bit exists (  uvm_component cntxt,
                              string        inst_name,
                              string        field_name,
                              bit           spell_chk);
```

Kiểm tra giá trị của field_name có tồn tại trong inst_name hay không, sử dụng cntxt như starting point. Nếu field_name không tồn tại ở scope đã cung cấp, hàm exist() sẽ trả về 0. tham số spell_chk có thể được đặt = 1 để có thể kích hoạt spell checking nếu người dùng nghĩ là field_name đã tồn tại trong database.

```
// Check if interface handle exists at the given scope
if (! uvm_config_db #(virtual apb_if) :: exists (this, "*", "apb_vif"))
    `uvm_error ("VIF", "Could not find an interface handle", UVM_MEDIUM)
```

4. wait_modified()

```
static task wait_modified ( uvm_component cntxt,
                            string        inst_name,
                            string        field_name);
```

Sử dụng task này để block đoạn code đang được thực thi cho tới khi configuration setting tên field_name tại scope {cntxt.inst_name}được set vào uvm_config_db

```
class my_agent extends uvm_agent;

    virtual task run_phase (uvm_phase phase);
        ...
        // Waits until loopCount variable gets a new value
        uvm_config_db #(int) :: wait_modified (this, "", "loopCount");
    endtask
endclass

class my_env extends uvm_env;

    my_agent m_apb_agent;

    virtual task main_phase (uvm_phase phase);
        ...
        // Update loopCount variable in database
        for (int i = 0; i < N; i++) begin
            ...
            uvm_config_db #(int) :: set (this, "m_apb_agent", "loopCount", i);
        end
    endtask
endclass
```

5. Convinient task

There are a few `typedef` aliases for the following parameterized versions of `uvm_config_db`.

```
typedef  uvm_config_db #(uvm_bitstream_t)     uvm_config_int;
typedef  uvm_config_db #(string)              uvm_config_string;
typedef  uvm_config_db #(uvm_object)          uvm_config_object;
typedef  uvm_config_db #(uvm_object_wrappet)  uvm_config_wrapper;
```

```
class ubus_example_env extends uvm_env;
    // Provide implementations of virtual methods such as get_type_name().
      `uvm_component_utils(ubus_example_env)
  // UBus reusable environment
      ubus_env ubus0;
  // Scoreboard to check the memory operation of the slave
      ubus_example_scoreboard scoreboard0;
  // new()
      function new(string name, uvm_component parent);
        super.new(name, parent);
      endfunction : new
  // build_phase()
      virtual function void build_phase(uvm_phase phase);
        super.build_phase(phase); // Configure before creating the
                                  // subcomponents.
        uvm_config_db#(int)::set(this,"ubus0",
                                 "num_masters", 1);
    uvm_config_db#(int)::set(this,".ubus0",
                             "num_slaves", 1);
     ubus0 = ubus_env::type_id::create("ubus0", this);
        scoreboard0 =
    ubus_example_scoreboard::type_id::create("scoreboard0",
          this);
        endfunction : build_phase
  virtual function connect_phase();
        // Connect slave0 monitor to scoreboard.
        ubus0.slaves[0].monitor.item_collected_port.connect(
        scoreboard0.item_collected_export);
      endfunction : connect
  virtual function void end_of_elaboration_phase(uvm_phase phase);
        // Set up slave address map for ubus0 (basic default).
        ubus0.set_slave_address_map("slaves[0]", 0, 16'hffff);
      endfunction : end_of_elaboration_phase
endclass : ubus_example_env
```

Một vài ví dụ configuration khác:

— Set the `masters[0]` agent to be active:

```
uvm_config_db#(uvm_active_passive_enum)::set(this,"ubus0.masters[0]",
    "is_active", UVM_ACTIVE);
```

— Do not collect coverage for `masters[0]` agent:

```
uvm_config_db#(int)::set(this, "ubus0.masters[0].monitor",
    "coverage_enable", 0);
```

— Set all slaves (using a wildcard) to be passive:

```
uvm_config_db#(uvm_active_passive_enum)::set(this,"ubus0.slaves*",
    "is_active", UVM_PASSIVE);
```

Ở ví dụ trên, ubus_example_env new() constructor không được dùng để tạo top-level

component (vì giới hạn của ngôn ngữ Systemverilog). thay vào đó build_phase() function được dùng, la built-in phase của UVM.

2 dòng uvm_config_db::set cài đặt số lượng master và slave đều là 1. Những setting này được dùng bơi ubus0 env ở build_phase().

create() được dùng để khởi tạo các subcomponents (thay vì gọi trực tiếp new() constructor), vì vậy ubus_env hoặc các lớp scoreboard có thể được thay thế với các lớp con mà ko thay đổi top-level env file.

super.build_phase() được gọi ở dòng đầu tiên trong build() function.

connect_phase() được dùng để tạo liên kết giữa slave monitor và scoreboard. Slave monitor chứa một TLM analysis port được kết nối với TLM analysis export của scoreboard.

Sau build_phase() và connect_phase(), user có thể điều chỉnh thông số run-time (run-time properties) vì env đã được build và connect. Ví dụ, **end_of_elaboration_phase()** ở ví dụ trên.

3. Creating Test classes

Test classes chứa các test scenarios và goals. Nhiệm vụ của nó là lựa chọn top-level env sẽ được sử dụng cũng như kích hoạt các configuration của env đó và các subcomponents của env đó.

Một test cung cấp data và sequence generation với inline constraint.

Test trong UVM là các lớp được derived từ uvm_test class. Thông thường, một base test class sẽ khởi tạo và configure top-level env, sau đó base class này sẽ được extended thành các lớp con để test các trường hợp cụ thể khác nhau (different test scenarios)

4. Verification Component Configuration

4.1 Verification Component Configurable Parameters

Dựa vào giao thức được sử dụng trong một thiết bị, người thiết kế khởi tạo các verification components cần thiết và config chúng cho một operation mode mong muốn.

Example of **standard** configuration parameters:

- Một **agent** có 2 mode là **active mode** và **passive mode**. Ở active mode, agent lái giao thông tới DUT. Ở passive mode, agent kiểm tra và thu thập coverage.
- **Mặc định, Monitor** thu thập coverage và kiểm tra DUT interface. User có thể tắt các hoạt động này bằng cách thay đổi standard parameter **check_enable** và **coverage_enable.**

Example of **user-defined** parameters:

- số lượng master agents và slave agent trong một AHB verification component.
- operation modes hoặc tốc độ của một bus

Một verification component cần support cả standard configuration parameter và provide user-defined parameter nếu cần.

4.2 Verification Component Configuration Mechanism - uvm_config_db

** Tiếp theo của bảng về uvm_config_db phía trên **

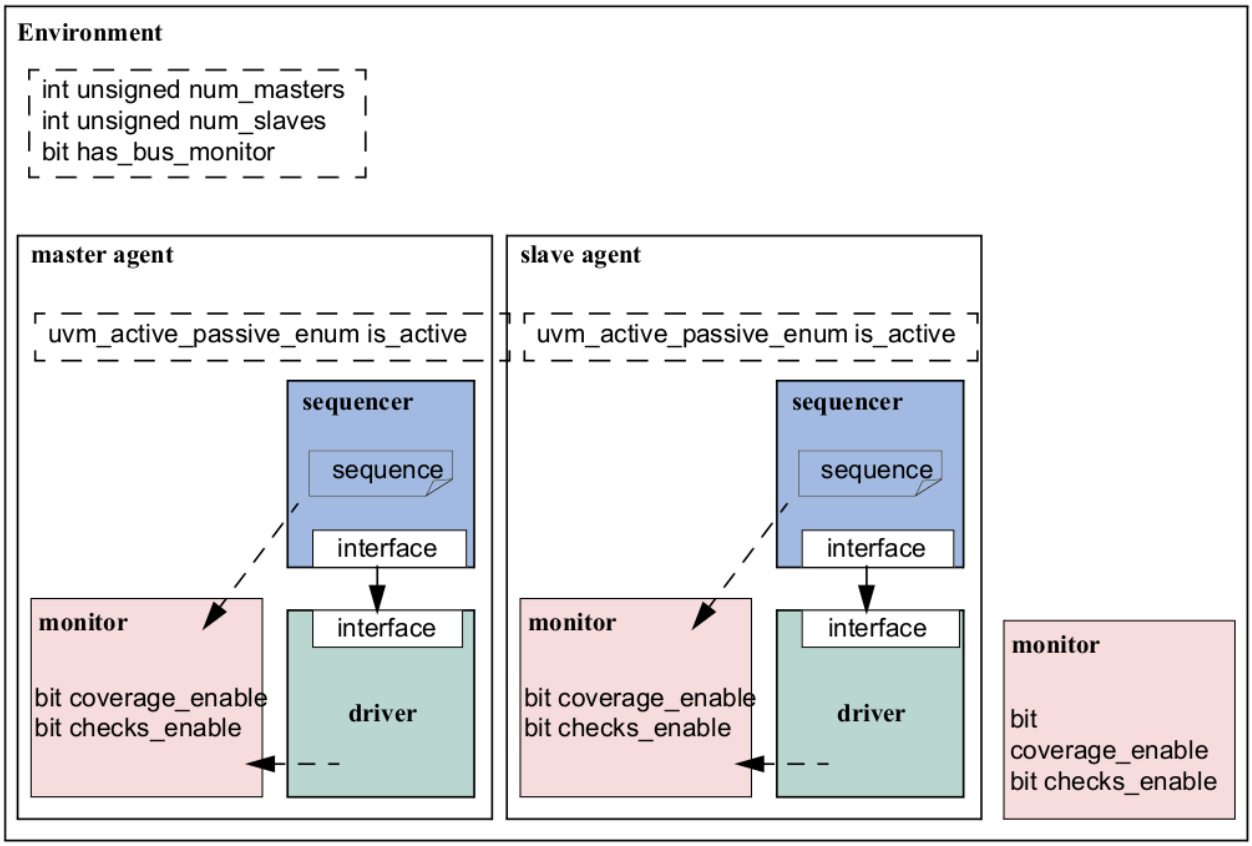


Figure 18—Standard Configuration Fields and Locations

Ví dụ về uvm_config_db:

```
uvm_config_db#(int)::set(this,"*.masters[0]", "master_id", 0);
uvm_config_db#(uvm_object_wrapper)::
          set(this, "*.ubus0.masters[0].sequencer.main_phase",
                "default_sequence", read_modify_write_seq::type_id::get());
uvm_config_db#(virtual ubus_if)::set(this,"ubus_example_env0.*","vif",vif);
uvm_resource_db#(myobject)::set("anyobject", "shared_config", data, this);
```

- Ví dụ 1: set giá trị cho master_id field của tất cả master component có tên khởi tạo kết thúc bằng masters[0].
- Ví dụ 2: gọi masters[0].sequencer thực hiện execute một sequence thuộc loại read_modify_write_seq khi tiến vào main phase.
- Ví dụ 3: cho thấy cách để define một loại virtual interface mà tất cả các components thuộc ubus_example_env0 nên sử dụng để set biến vif của chúng.
- Ví dụ 4: cho thấy cách store một vài shared resource tới vị trí mà bất kì object ở bất cứ đâu trong cây phả hệ verification có thể access.
- NOTE: Khi uvm_resource_db::set() được thực hiện từ một class, parameter cuối nên là "this" để cho phép debugging messages hiển thị nơi mà setting được tạo.

4.3 Lựa chọn giữa uvm_resource_db và uvm_config_db

uvm_config_db và uvm_resource_db chia sẻ chung database gốc. Do đó ta có thể ghi vào database sử dụng uvm_config_db::set() và lấy dữ liệu từ database sử dụng uvm_resource_db::read_by_name()

uvm_config_db được sử dụng khi cần ghi configuration setting vào một component cụ thể trong cây phả hệ của testbench (ví dụ coverage_enable cho tất cả component của một agent)

uvm_resource_db được sử dụng khi không quan tâm đến cây phả hệ (setting này được share với tất cả components trong testbench)

5. Creating and Selecting a User-Defined Test

Trong UVM testbench, test là một class có chức năng đóng gói các yêu cầu kiểm tra đặc thù được viết bởi test writer. Phần này mô tả cách tạo và lựa chọn một test. Đồng thời cũng mô tả cách tạo ra một **test family base class** để kiểm tra một topology configuration

5.1. Creating the Base Test

Xét ví dụ về ubus_example_env ở phần 2 bên trên:

```
class ubus_example_env extends uvm_env;
    // Provide implementations of virtual methods such as get_type_name().
      `uvm_component_utils(ubus_example_env)
  // UBus reusable environment
      ubus_env ubus0;
  // Scoreboard to check the memory operation of the slave
      ubus_example_scoreboard scoreboard0;
  // new()
      function new(string name, uvm_component parent);
        super.new(name, parent);
      endfunction : new
  // build_phase()
      virtual function void build_phase(uvm_phase phase);
        super.build_phase(phase); // Configure before creating the
                                  // subcomponents.
        uvm_config_db#(int)::set(this,"ubus0",
                                       "num_masters", 1);
    uvm_config_db#(int)::set(this,".ubus0",
                                   "num_slaves", 1);
     ubus0 = ubus_env::type_id::create("ubus0", this);
        scoreboard0 =
    ubus_example_scoreboard::type_id::create("scoreboard0",
          this);;
      endfunction : build_phase
  virtual function connect_phase();
        // Connect slave0 monitor to scoreboard.
        ubus0.slaves[0].monitor.item_collected_port.connect(
        scoreboard0.item_collected_export);
      endfunction : connect
  virtual function void end_of_elaboration_phase(uvm_phase phase);
        // Set up slave address map for ubus0 (basic default).
        ubus0.set_slave_address_map("slaves[0]", 0, 16'hffff);
      endfunction : end_of_elaboration_phase
endclass : ubus_example_env
```

test class hoàn chỉnh của ví dụ trên:

```
class ubus_example_base_test extends uvm_test;
      `uvm_component_utils(ubus_example_base_test)
      ubus_example_env ubus_example_env0;
      // The test's constructor
      function new (string name = "ubus_example_base_test",
        uvm_component parent = null);
        super.new(name, parent);
      endfunction
  // Update this component's properties and create the ubus_example_tb
    component.
      virtual function build_phase(); // Create the top-level environment.
        super.build_phase(phase);
        ubus_example_env0 =
    ubus_example_tb::type_id::create("ubus_example_env0", this);
      endfunction
endclass
```

build_phase() function của base test khởi tạo **ubus_example_env**. Thư viện UVM sẽ execute build_phase() function của ubux_example_base_test khi đi qua từng phase của các components. Bằng cách này, top-level env cũng như các sub-component sẽ tạo ra các children components của chúng khi build_phase() function được thực thi.

NOTE: Tất cả định nghĩa nằm trong base test sẽ được kế thừa bất kì lớp test nào là con của base test (lớp con của ubus_example_base_test). Nghĩa là, không có lớp con nào phải build top-level env một lần nữa nếu lớp con gọi **super.build_phase()**. Tương tự, run_phase() task cũng có thể được kế thừa, cũng như các phase khác

5.2 Creating tests from a Test-Family Base Class

Từ base test phía trên (ubus_example_base_test), ta có thể tạo các lớp tests là lớp con và kế thừa các đặc thù từ base class.

Bởi vì top-level env được tạo từ build_phase() function của base test, và run_phase() task định nghĩa run phase, lớp con có thể có một vài điều chỉnh nhỏ khác với lớp cha (vd: thay đổi sequence mặc định được executed bởi agent trong env)

Ví dụ sau là một lớp test con đơn giản kế thừa từ ubus_example_base_test

```
class test_read_modify_write extends ubus_example_base_test;
  `uvm_component_utils(test_read_modify_write)
  // The test's constructor
      function new (string name = "test_read_modify_write",
         uvm_component parent = null);
        super.new(name, parent);
      endfunction
  // Register configurations to control which
      // sequence is executed by the sequencers.
      virtual function void build_phase(uvm_phase phase);
        // Substitute the default sequence.
              uvm_config_db#(uvm_object_wrapper)::
            set(this, "ubus0.masters[0].sequencer.main_phase",
                "default_sequence", read_modify_write_seq::type_id::get());
      uvm_config_db#(uvm_object_wrapper)::
            set(this, "ubus0.slaves[0].sequencer.main_phase",
                "default_sequence",slave_memory_seq::type_id::get());
        super.build_phase(phase);
      endfunction
endclass
```

Lớp con trong ví dụ trên thay đổi sequence mặc định được execute bởi masters[0] agent và slaves[0] agent. ****Quan trọng**** Ta cần phải hiểu được super.build_phase(), thông qua base class, sẽ tạo ra top-level env (ubus_example_env0) và tất cả các subcomponents của nó. **Vì vậy, bất kì configuration nào ảnh hưởng đến quá trình building các component (such as how many masters to create) phải được set trước khi gọi super.build_phase().**

5.3 Test Selection

Sau khi xác định user-defined test ở ví dụ bên trên, **uvm_pkg::run_test()** task cần phải được invoked để chọn một test để simulate. Prototype của nó là:

```
task run_test(string test_name="");
```

=> Cần đưa trường String test_name vào làm parameter để chạy run_test

UVM hỗ trợ việ xác định test nào sẽ được chạy thông qua 2 cơ chế. **testname** (tên được dùng để register test với factory) có thể được truyền trực tiếp vào **run_test()** task hoặc ta có thể xác định sử dụng command line **+UVM_TESTNAME.** Nếu cả 2 được dùng, command được ưu tiên. Một khi test name được chọn, run_test() task gọi factory để tạo một instance của test đó với một instance name của **uvm_test_top**. Cuối cùng, run_test() bắt đầu chạy test qua việc đi qua từng sim phases.

6. Creating Meaningful Tests

Để đạt được mục tiêu của verification một cách hệ thống, user sẽ cần phải kiểm soát test generation để cover các khu vực cụ thể.

User có thể kiểm soát việc tạo ra test sử dụng những phương pháp sau:

- Add constraints to control individual data items. This method provides basic functionality

- Use UVM sequences to control the order of multiples data items. Thí provides more flexibility and control

6.1 Constraining Data Items

Mặc định, sequencers sẽ liên tục tạo ra các data items ngẫu nhiên. Test writer có thể kiểm soát số lượng data items được gen và thêm constraints vào các data items để kiểm soát miền giá trị của chúng.

To constraint data items:

a) Xác định lớp của data items và generated fields của chúng trong verification component

b) Tạo lớp con từ data item class và thêm hoặc ghi đè các constraints mặc định

c) Trong một test, điều chỉnh enviromnent để sử dụng data items mới được xác định

d) chạy mô phỏng sử dụng command line option để xác định test name

Ví dụ về data item:

```
typedef enum bit {BAD_PARITY, GOOD_PARITY} parity_e;
class uart_frame extends uvm_sequence_item;
      rand int unsigned transmit_delay;
      rand bit start_bit;
      rand bit [7:0] payload;
      rand bit [1:0] stop_bits;
      rand bit [3:0] error_bits;
      bit parity;
      // Control fields
      rand parity_e parity_type;
  function new(input string name);
        super.new(name);
      endfunction
  // Optional field declarations and automation flags
      `uvm_object_utils_begin(uart_frame)
        `uvm_field_int(start_bit, UVM_ALL_ON)
        `uvm_field_int(payload, UVM_ALL_ON)
        `uvm_field_int(parity, UVM_ALL_ON)
        `uvm_field_enum(parity_e, parity_type, UVM_ALL_ON + UVM_NOCOMPARE)
        `uvm_field_int(xmit_delay,  UVM_ALL_ON + UVM_DEC + UVM_NOCOMPARE)
      `uvm_object_utils_end
  // Specification section 1.2: the error bits value should be
      // different than zero.
      constraint error_bits_c {error_bits != 4'h0;}
  // Default distribution constraints
      constraint default_parity_type {parity_type dist {
        GOOD_PARITY:=90, BAD_PARITY:=10};}
  // Utility functions
      extern function bit calc_parity ( );
      ...
      endfunction
endclass: uart_frame
```

Tạo constraint tên error_bits_c: error_bits ko được có giá trị 4’b0000

Tạo constraint tên default_paruty_type để phân bố tỉ lệ phần trăm randomization

6.2 Data Item Definition

Một vài field của lớp con có từ device specification (vd: 1 frame nên có payload được gửi tới DUT)

Các fields khác có nhiệm vụ hỗ trợ test writer kiểm soát việc genaration. (vd: field parity_type ko dc gửi tới DUT, nhưng nó cho phép test writer có thể dễ dàng specify và control parity distribution). Những control field này được gọi là “knob”.

verification component documentation nên liệt kê các data item’s knob, nhiệm vụ của chúng và legal range của chúng.

Data items có các specification constraints. Những constraints này có thể xuất phát từ DUT specification để tạo ra legal data item (vd: một legal frame phải có error_bits_c không được là 4’b0000).

Một loại constraints khác chính là các data item dùng để kiểm soát việc generation. (vd: trong constraint block default_parity_type phía trên, parity bit được ràng buộc 90% legal (good parity) và 10% illegal (bad parity))

6.3 Creating a Test-Specific Frame

Trong quá trình test, user có thể muốn thay đổi cách data items được generated (vd: test writer muốn có delay ngắn hơn). Điều này có thể được thực hiện bằng cách tạo lớp con từ lớp data item muốn thay đổi và thêm constraints và các thành phần khác vào lớp con.

Ví dụ: từ lớp uart_frame ở ví dụ trên, tạo lớp con của nó tên short_delay_frame

```
// A derived data item example
    // Test code
    class short_delay_frame extends uart_frame;
      // This constraint further limits the delay values.
      constraint test1_txmit_delay {transmit_delay  < 10;}
  `uvm_object_utils(short_delay_frame)
  function new(input string name="short_delay_frame");
       super.new(name);
      endfunction
endclass: short_delay_frame
```

Sau khi khởi tạo lớp con mới, **ta còn cần phải** configure lớp environment để sử dụng lớp con mới (short_delay_frame). Cơ chế factory của UVM Class Lib có thể được dùng để giới thiệu lớp con mới vào environment.

```
class short_delay_test extends uvm_test;
      `uvm_component_utils(short_delay_test)
      uart_tb uart_tb0;
      function new (string name = "short_delay_test",uvm_component parent
    null);
        super.new(name, parent);
      endfunction
  virtual function build_phase(uvm_phase phase);
        super.build_phase(phase);
        // Use short_delay_frame throughout the environment.
        factory.set_type_override_by_type(uart_frame::get_type(),
           short_delay_frame::get_type());
        uart_tb0 = uart_tb::type_id::create("uart_tb0", this);
      endfunction
  task run_phase(uvm_phase phase);
        uvm_top.print_topology();
      endtask
endclass
```

Việc gọi factory function tên **set_type_override_by_type()** sẽ chỉ dẫn cho enviromnent sử dụng short-delay frames.

Ở nhiều trường hợp, user muốn gửi special trafic (special data item) tới một interface nhưng vẫn gửi regular traffic (regular data item) tới các interface còn lại. Việc này có thể được thực hiện bằng các sử dụng **set_inst_override_by_type()** bên trong UVM component.

```
set_inst_override_by_type("uart_env0.master.sequencer.*",
      uart_frame::get_type(), short_delay_frame::get_type());
```

wildcards cũng có thể được dùng để override instantiation của một vài components

```
set_inst_override_by_type("uart_env*.master.sequencer.*",
      uart_frame::get_type(), short_delay_frame::get_type());
```

7. Virtual Sequences **Xem thêm ở UVM cookbook**

Ví dụ trên cho thấy cách kiểm soát một cách hiệu quả một sing-interface generation pattern. Tuy nhiên trong một môi trường mang tính hệ thống phân lớp, nhiều component có thể generate stimuli song song với nhau.

Do đó user se cần phải điều phối timing và data giữa đa kênh. Đồng thời user cũng cần define một system-level scenario có tính tái sử dụng.

các virtual sequence hợp tác với một virtual sequencer để điều phối stimulus generation trong testbench hierachy.

Thông thường, một virutal sequencer sẽ chứa tham chiếu tới các subsequencer của nó.

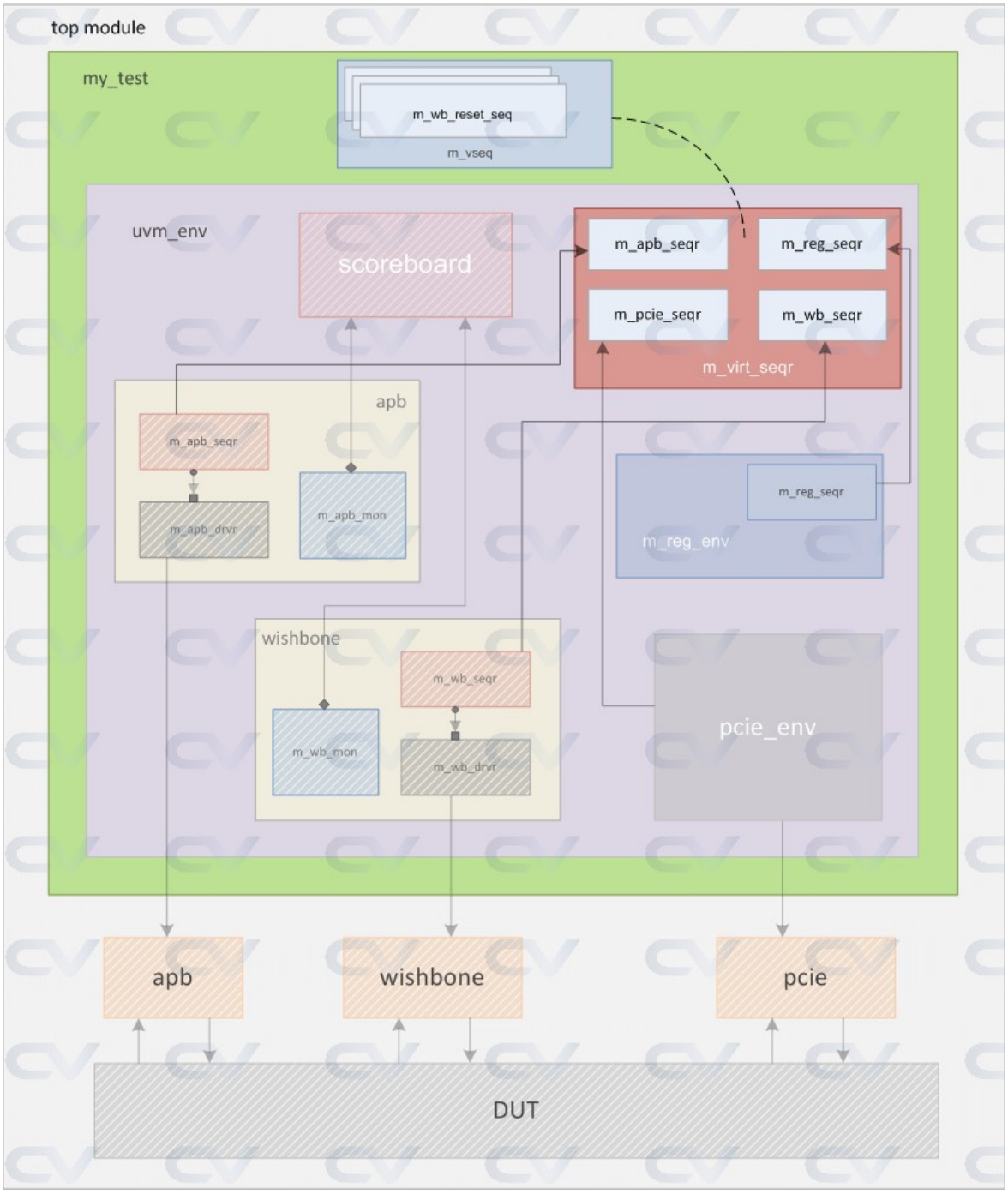
virtual sequence có thể gọi sequence khác có liên quan tới sequencer của nó. Cũng như các sequence của mỗi subsequencer đó.

Tuy nhiên, virtual sequencers không có data item riêng, do đó ko thể tự mình execute data item

virtual sequences có thể execute items trong sequencers có khả năng execute item (nghĩa là loại trừ virtual sequencer)

“Ta có thể gọi virtual sequence là một MASTER SEQUENCE hoặc COORDINATOR SEQUENCE (sequence điều phối)” - Seimen UVM cookbook pg 172

Sự cần thiết của virtual sequence tăng lên khi test writer cần các sequences khác nhau chạy trên các môi trường khác nhau. VD: một thiết kế SoC có thể có nhiều interfaces khác nhau và cần phải lái bởi một set of sequences khác nhau trên mỗi sequencer. Do đó cách tốt nhất để bắt đầu và kiểm soát các sequences khác nhau là sử dụng virtual sequence



Virtual sequencer

Nguồn: https://www.chipverify.com/uvm/uvm-virtual-sequencer

Định nghĩa: virtual sequencer là một UVM sequencer chứa các handles của tất cả sequencers. Tại sao ta lại cần virtual sequencer? Bởi vì ta sẽ dùng virtual sequence (ta cũng có thể sử dụng virtual sequence mà ko cần dùng virtual sequencer) và control tất cả các sequencers ở một nơi trung tâm.

Ở hình trên, có thể thấy được env chứa một APB agent, Wishbone agent, PCIE env và một register layer env. Mỗi component chứa sequences của riêng nó và sequencers tương ứng mà các sequences này sử dụng. Một virtual sequencer tên **m_virt_seqr** được khởi tạo và giữ tham chiêu tới mỗi sequencer. Do đó một virtual sequence executing trên virtual sequencer này sẽ có access tới tất cả các sequencers trong testbench

Có 3 cách để virtual sequencer có thể tương tác với các subsequencers của nó:

1. "Business as usual" - virtual subsequencer và subsequencer gửi các transsaction đồng thời.
2. Disable subsequencers - chỉ virtual sequencer được chạy
3. sử dụng **grab()** và **ungrab()** - virtual sequencer giành quyền điều khiển của driver cơ bản trong một khoảng thời gian giới hạn.

Khi sử dụng virtual sequences, hầu hết user disable subsequencers và gọi chỉ sequences từ virtual sequence. Để gọi sequences, ta dùng một trong 2 cách sau:

- Sử dụng **`uvm_do** macro phù hợp
- Sử dụng **start()** method của sequence

7.1. Creating a Virtual Sequencer

Để control nhiều sequencers từ một sequencer cấp cao, ta sử dụng một sequencer **không có liên kết với một driver** và **không thể tự mình xử lý item**. Sequencer có nhiệm vụ này chính là một **virtual sequencer**.

Để tạo ra một virtual sequencer có thể control nhiều subsequencers:

a) Tạo một lớp con từ uvm_sequencer class, lớp con này chính là virtual sequencer

b) Thêm các tham chiểu tới các sequencers, nơi mà các virtual sequences sẽ điều phối hoạt động. Những tham chiếu này sẽ được assigned bởi một component cấp cao hơn (thường là top-level environment)

Ví dụ bên dưới xác định một virutal sequencer với 2 subsequencers. 2 interfaces tên eth và cpu được tạo trong build function, sau đó sẽ được gắn liền với các subsequeners.

```
class simple_virtual_sequencer extends uvm_sequencer;
      eth_sequencer eth_seqr;
      cpu_sequencer cpu_seqr;
  // Constructor
      function new(input string name="simple_virtual_sequencer",
        input uvm_component parent=null);
        super.new(name, parent);
      endfunction
  // UVM automation macros for sequencers
      `uvm_component_utils(simple_virtual_sequencer)
endclass: simple_virtual_sequencer
```

subsequencers có thể là driver sequencers hoặc các virtual sequencers khác. Kết nối giữa subsequencers instaces thông qua tham chiếu được thực hiện ở ví dụ sau (phần 4.7.4 textbook)

7.2 Creating a virtual sequence

Tương tự như tạo một driver sequencer, với sự khác biệt như sau:

- Một virtual sequence sử dung **`uvm_do_on hoặc `uvm_do_on_with** để execute sequences trên bất kỳ subsequencer nào kết nối với virtual sequencer hiện tại.
- Một virtual sequence sử dụng **`uvm_do hoặc `uvm_do_with** để execute các virtual sequences khác của sequencer này. Một virtual sequence **không thể** sử dụng **`uvm_do hoặc `uvm_do_with** để execute item. Virtual sequencer không có items bên trong chúng, chúng chỉ chứa các sequences

Để tạo một virtual sequence:

- Declare một lớp con sequence từ base class uvm_sequence (tương tự driver sequence).
- Xác định một **body() method** để hiện thực luận lý mong muốn của sequence đó.
- Sử dụng **`uvm_do_on (hoặc `uvm_do_on_with)** macro để gọi sequences nằm trong subsequencers tương ứng.
- Sử dụng **`uvm_do (hoặc `uvm_do_with)** macro để gọi virtual sequences trong virtual sequencer hiện tại.

Ví dụ sau cho thấy một virtual sequence control 2 subsequencer (một cpu sequencer và một ethernet sequencer). Ta assume rằng cpu sequencer có một **cpu_config_seq** sequence và ethernet sequencer cung cấp một **eth_large_payload_seq** sequence bên trong thư viện của chúng. Ví dụ sau gọi 2 sequencers, lần lượt từng cái:

```
class simple_virt_seq extends uvm_sequence;
      ... // Constructor and UVM automation macros
      // A sequence from the cpu sequencer library
      cpu_config_seq conf_seq;
      // A sequence from the ethernet subsequencer library
      eth_large_payload_seq frame_seq;
      // A virtual sequence from this sequencer's library
      random_traffic_virt_seq rand_virt_seq;
```

ɔber 8, 2015

```
  virtual task body();
        // Invoke a sequence in the cpu subsequencer.
        `uvm_do_on(conf_seq, p_sequencer.cpu_seqr)
        // Invoke a sequence in the ethernet subsequencer.
        `uvm_do_on(frame_seq, p_sequencer.eth_seqr)
        // Invoke another virtual sequence in this sequencer.
        `uvm_do(rand_virt_seq)
      endtask : body
endclass : simple_virt_seq
```

Quan sát body() task:

`uvm_do_on macro đầu tiên gọi sequence bên trong cpu subsequencer.

`uvm_do_on macro thứ hai gọi sequence bên trong ethernet subsequencer.

`uvm_do macro cuối cùng gọi một virtual sequencer bên trong sequencer hiện tại (rand_virt_seq là instance của random_traffic_virt_seq - nằm trong thư viện của sequencer hiện tại).

7.3. Controlling Other Sequencers

Khi sử dụng một virtual sequencer, ta cần xem xét mối quan hệ hành vi giữa subsequencers và virtual sequence. Có 3 trường hợp cơ bản:

a) Business as usual - virtual sequencer và subsequencer generate traffic cùng lúc, đây là hành vi mặc định (ko cần phải setting)

b) Disable the subsequencers - sử dụng **uvm_config_db::set, default_sequence** property của subsequencers được set thành **null.** Bên dưới là ví dụ về disable default equences trên subsequencers của ví dụ 4.7.4

```
// Configuration: Disable subsequencer sequences.
    uvm_config_db#(uvm_sequence_base)::set(this, "*.cpu_seqr.*_phase",
       "default_sequence", null);;
    uvm_config_db#(uvm_sequence_base)::set(this, "*.eth_seqr.*_phase",
       "default_sequence", null);
```

c) Sử dụng grab() / lock() và ungrab() / unlock() - Ở trường hợp này, một virtual sequence có thể có được full control các subsequencers của nó trong một khoảng thời gian nhất định, sau đó sequences gốc sẽ tiếp tục hoạt động.

Sử dụng grab và lock APIs một cách hiệu quả sẽ ngăn chặn các items được executed trên một sequencer bị khóa, trừ khi các items này được tạo bởi sequence thực hiện việc khóa

grab() method đặt một lock request tại đầu của queue phân xử của sequencer, cho phép caller ngăn items hiện đang chờ không được xử lý

lock() đặt một request tại cuối hàng đợi, cho phép các items được phép xử lý trước khi lock được tiến hành

Ví dụ bên dưới mô tả việc sử dụng grab() và ungrab() trong sequence consumer interface:

```
virtual task body();
    // Grab the cpu sequencer if not virtual.
    if (p_sequencer.cpu_seqr != null)
      grab(p_sequencer.cpu_seqr);
    // Execute a sequence.
    `uvm_do_on(conf_seq, p_sequencer.cpu_seqr)
    // Ungrab.
    if (p_sequencer.cpu_seqr != null)
      ungrab(p_sequencer.cpu_seqr);
  endtask
```

**NOTE: khi grab nhiều sequencers, cần phải chắc chắn quy tắc để tránh deadlocks. Ví dụ: always grab in a standard order.

7.4. Connecting a Virtual Sequencer to Subsequencers

Để connect một virtual sequencer với subsequencers của nó:

a) Gán các tham chiếu của các sequencer được specified bên trong virtual sequencer tới các instances của chúng. Việc gán này nên được thực hiện sau khi tất cả các components được tạo.

```
v_sequencer.cpu_seqr = cpu_seqr;
v_sequencer.eth_seqr = eth_seqr;
```

b) Tiến hành connect ở connect() phase của virification env tại vị trí phù hợp trong verification enviromnent hierachy

Một cách khác là pointer của sequencer có thể được set như resource trong quá trình build (ở ví dụ bên dưới với **eth_seqr ở dòng cuối cùng - uvm_config_db::set...**)

Bên dưới là một ví dụ phức tạp hơn hiển thị một top-level enviromnent, có tác vụ khởi tạo ethernet và cpu components và virtual sequencer để control ethernet và cpu. Trong top-level enviromnent, đường dẫn tới các sequencers bên trong các components khác nhau được xem là đường dẫn được dùng để lấy một handle tới chúng và connect chúng với virtual sequencer.

```
class simple_tb extends uvm_env;
      cpu_env_c cpu0; // Reuse a cpu verification component.
      eth_env_c eth0; // Reuse an ethernet verification component.
      simple_virtual_sequencer v_sequencer;
      ... // Constructor and UVM automation macros
      virtual function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        // Configuration: Set the default sequence for the virtual sequencer.
        uvm_config_db#(uvm_object_wrapper)::set(this,
                                                "v_sequencer.run_phase",
                                                "default_sequence",
                                                simple_virt_seq.type_id::get());
        // Build envs with subsequencers.
        cpu0 = cpu_env_c::type_id::create("cpu0", this);
        eth0 = eth_env_c::type_id::create("eth0", this);

        // Build the virtual sequencer.
        v_sequencer =
    simple_virtual_sequencer::type_id::create("v_sequencer",
            this);
        endfunction : build_phase
  // Connect virtual sequencer to subsequencers.
      function void connect_phase();
        v_sequencer.cpu_seqr = cpu0.master[0].sequencer;
        uvm_config_db#(uvm_sequencer)::set(this,"v_sequencer",
                                           "eth_seqr",eth0.tx_rx_agent.sequencer);
      endfunction : connect_phase
endclass: simple_tb
```

8. Checking for DUT Correctness