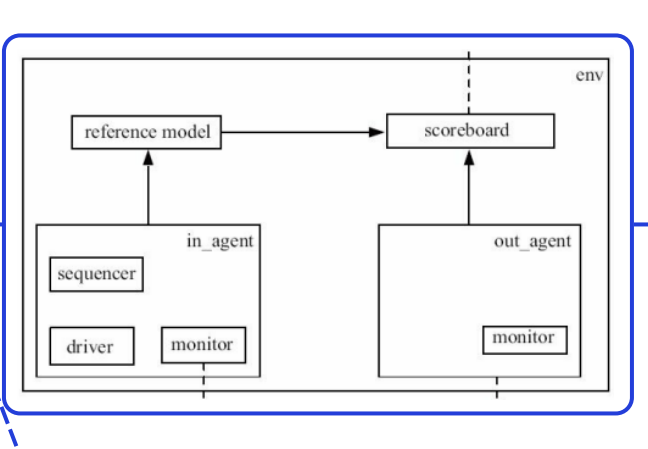
UVM框架

- **机制**
  - 类的定义、声明、实现
    - classs A;
...
endclass
    - A a_inst;
    - a_inst = new();
  - factory机制
    - 目的：自动创建一个类的实例并调用其中的函数（function）和任务（task）。
    - 方法：在类定义时声明uvm_component_utils宏，将该类注册进UVM列表。实例化该类后，这个类的main_phase就会被自动调用。
  - objection机制
    - 在main_phase中使用
    - raise_objection必须在main_phase中第一个消耗仿真时间的语句之前。
    - drop_objection语句是finish函数的替代
  - virtual interface
    - 目的：避免在验证平台中使用绝对路径。
      - 在SystemVerilog中使用interface来连接验证平台与DUT的端口。定义了interface后，在top_tb中实例化DUT时，就可以直接使用。
    - config_db机制
      - 分为set和get两步操作
        - 在top_tb中执行set操作：uvm_config_db#(virtual my_if)::set(null, "uvm_test_top", "vif", input_if);
        - 在my_driver中，执行get操作：uvm_config_db#(virtual my_if)::get(this, "", "vif", vif)
        - uvm_config_db#（virtual my_if）则是一个参数化的类，其参数就是要寄信的类型
        - 第一和第二个参数是路径索引；set函数的第四个参数表示要将哪个interface
通过config_db传递给my_driver，get函数的第四个参数表示把得到的interface传递给哪个my_driver的成员变量。
      - 引入build_phase
        - build_phase在new函数之后main_phase之前执行。
        - 在build_phase中主要通过config_db的set和get操作来传递一些数据，以及实例化成员变量等。
        - 需要加入super.build_phase语句
        - build_phase是不消耗仿真时间的，总是在仿真时间（$time函数打印出的时间）为0时执行。
  - TLM（Transaction Level Modeling）
    - 实现component之间transaction级别的通信。
      - 数据的发送，uvm_analysis_port
        - uvm_analysis_port是一个参数化的类，其参数就是这个analysis_port需要传递的数据的类型。uvm_analysis_port #(my_transaction) ap;
        - 声明了ap后，需要在monitor的build_phase中将其实例化。ap = new("ap", this);
        - 在main_phase中，当收集完一个transaction后，需要将其写入ap中：ap.write(tr);
      - transaction级别通信的数据接收方式，uvm_blocking_get_port
        - 是一个参数化的类，其参数是要在其中传递的transaction的类型。
        - 在参考模型reference model的build_phase中对其进行实例化，同样scoreboard中也有两个port分别从reference model和o_gent获取数据。
        - main_phase中，通过port.get任务来得到从i_agt的monitor中发出的transaction。
      - env中使用fifo将两个端口联系在一起。
        - env中定义一个fifo，并在build_phase中将其实例化
          - uvm_tlm_analysis_fifo #(my_transaction) agt_mdl_fifo;
          - agt_mdl_fifo = new("agt_mdl_fifo", this);
        - fifo的类型是uvm_tlm_analysis_fifo，它本身也是一个参数化的类，其参数是存储在其中的transaction的类型
        - connect_phase
          - 在env中将fifo分别与my_monitor中的analysis_port和my_model中的blocking_get_port相连
          - i_agt.ap.connect(agt_mdl_fifo.analysis_export);
mdl.ap.connect(mdl_scb_fifo.analysis_export);
  - field_automation
    - 直接调用copy、compare、print等函数，而无需自己定义。
    - 使用uvm_field系列宏实现

      ```
      `uvm_object_utils_begin(my_transaction)
        `uvm_field_int(dmac, UVM_ALL_ON)
        `uvm_field_int(smac, UVM_ALL_ON)
        `uvm_field_int(ether_type, UVM_ALL_ON)
        `uvm_field_array_int(pload, UVM_ALL_ON)
        `uvm_field_int(crc, UVM_ALL_ON)
      `uvm_object_utils_end
      ```

    - 里使用uvm_object_utils_begin和uvm_object_utils_end来实现my_transaction的factory注册
    - 使用uvm_field宏注册所有字段
- **组件**
  - transaction
    - 一笔transaction就是一个包。简单来说，一个transaction就是把具有某一特定功能的一组信息封装在一起而成为的一个类。
    - transaction的基类是uvm_sequence_item。
    - uvm_sequence_item的祖先就是uvm_object。UVM中具有这种特征的类都要使用uvm_object_utils宏来实现。
  - env
    - 引入一个容器类，在这个容器类中实例化driver、monitor、reference model和scoreboard等。

      - 1.创建3个fifo通道，i_agt->mdl,o_agt->scb,mdl->scb
2.将端口部署到fifo的两侧，因为有3个fifo，所有有6个端口。

        ```
        class my_env extends uvm_env;

          my_agent   i_agt;
          my_agent   o_agt;
          my_model   mdl;
          my_scoreboard scb;

          uvm_tlm_analysis_fifo #(my_transaction) agt_scb_fifo;
          uvm_tlm_analysis_fifo #(my_transaction) agt_mdl_fifo;
          uvm_tlm_analysis_fifo #(my_transaction) mdl_scb_fifo;

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

          virtual function void build_phase(uvm_phase phase);
             super.build_phase(phase);
             i_agt = my_agent::type_id::create("i_agt", this);
             o_agt = my_agent::type_id::create("o_agt", this);
             i_agt.is_active = UVM_ACTIVE;
             o_agt.is_active = UVM_PASSIVE;
             mdl = my_model::type_id::create("mdl", this);
             scb = my_scoreboard::type_id::create("scb", this);
             agt_scb_fifo = new("agt_scb_fifo", this);
             agt_mdl_fifo = new("agt_mdl_fifo", this);
             mdl_scb_fifo = new("mdl_scb_fifo", this);

          endfunction

          extern virtual function void connect_phase(uvm_phase phase);

          `uvm_component_utils(my_env)
        endclass

        function void my_env::connect_phase(uvm_phase phase);
          super.connect_phase(phase);
          i_agt.ap.connect(agt_mdl_fifo.analysis_export);
          mdl.port.connect(agt_mdl_fifo.blocking_get_export);
          mdl.ap.connect(mdl_scb_fifo.analysis_export);
          scb.exp_port.connect(mdl_scb_fifo.blocking_get_export);
          o_agt.ap.connect(agt_scb_fifo.analysis_export);
          scb.act_port.connect(agt_scb_fifo.blocking_get_export);
        endfunction
        ```

    - 验证平台中的组件在实例化时都应该使用type_name：：type_id：：create的方式。
  - agent
    - agent
      - driver和monitor处理的是同一种协议，UVM中通常将二者封装在一起，成为一个agent。
        - 根据is_active这个变量的值来决定是否创建driver的实例。创建一个用于发送transaction的port，该port的数据来自monitor.ap。

          ```
          class my_agent extends uvm_agent ;
            my_sequencer  sqr;
            my_driver     drv;
            my_monitor    mon;
            uvm_analysis_port #(my_transaction)  ap;

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

            extern virtual function void build_phase(uvm_phase phase);
            extern virtual function void connect_phase(uvm_phase phase);

            `uvm_component_utils(my_agent)
          endclass
          function void my_agent::build_phase(uvm_phase phase);
            super.build_phase(phase);
            if (is_active == UVM_ACTIVE) begin
               sqr = my_sequencer::type_id::create("sqr", this);
               drv = my_driver::type_id::create("drv", this);
            end
            mon = my_monitor::type_id::create("mon", this);
          endfunction

          function void my_agent::connect_phase(uvm_phase phase);
            super.connect_phase(phase);
            if (is_active == UVM_ACTIVE) begin
               drv.seq_item_port.connect(sqr.seq_item_export);
            end
            ap = mon.ap;
          endfunction
          ```

      - 所有的agent都要派生自uvm_agent类，且其本身是一个component，应该使用uvm_component_utils宏来实现factory注册。
      - sequencer也要加入
    - monitor
      - 验证平台中实现监测DUT行为的组件是monitor。driver负责把transaction级别的数据转变成DUT的端口级别，并驱动给DUT，monitor的行为与其相对，用于收集DUT的端口数据，并将其转换成transaction交给后续的组件如reference model、scoreboard等处理。
        - collect_one_pkt函数从vif中接收端口级别的数据然后组成transaction。发送给agent。

          ```
          class my_monitor extends uvm_monitor;
            virtual my_if vif;
            uvm_analysis_port #(my_transaction) ap;
            `uvm_component_utils(my_monitor)
            function new(string name = "my_monitor", uvm_component parent = null);
              super.new(name, parent);
            endfunction

            virtual function void build_phase(uvm_phase phase);
              super.build_phase(phase);
              if(!uvm_config_db#(virtual my_if)::get(this, "", "vif", vif))
                `uvm_fatal("my_monitor", "virtual interface must be set for vif!!!")
              ap = new("ap", this);
            endfunction

            extern task main_phase(uvm_phase phase);
            extern task collect_one_pkt(my_transaction tr);
          endclass
          task my_monitor::main_phase(uvm_phase phase);
            my_transaction tr;
            while(1) begin
              tr = new("tr");
              collect_one_pkt(tr);
              ap.write(tr);
            end
          endtask
          task my_monitor::collect_one_pkt(my_transaction tr);
          ```

      - 所有的monitor类应该派生自uvm_monitor
      - 与driver类似，在my_monitor中也需要有一个virtual my_if
      - uvm_monitor在整个仿真中是一直存在的，所以它是一个component，要使用uvm_component_utils宏注册
      - 由于monitor需要时刻收集数据，永不停歇，所以在main_phase中使用while（1）循环来实现这一目的。
  - reference model
    - reference model用于完成和DUT相同的功能，输出被scoreboard接收，用于和DUT的输出相比较。
      - 接受事务级别的信号然后通过高级语言处理。
        - 本例中mdl从i_agt.monitor中获取transaction，然后复制一份发送给scb。

          ```
          class my_model extends uvm_component;

            uvm_blocking_get_port #(my_transaction)  port;
            uvm_analysis_port #(my_transaction)  ap;

            extern function new(string name, uvm_component parent);
            extern function void build_phase(uvm_phase phase);
            extern virtual  task main_phase(uvm_phase phase);

            `uvm_component_utils(my_model)
          endclass

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

          function void my_model::build_phase(uvm_phase phase);
            super.build_phase(phase);
            port = new("port", this);
            ap = new("ap", this);
          endfunction

          task my_model::main_phase(uvm_phase phase);
            my_transaction tr;
            my_transaction new_tr;
            super.main_phase(phase);
            while(1) begin
              port.get(tr);
              new_tr = new("new_tr");
              new_tr.copy(tr);
              `uvm_info("my_model", "get one transaction, copy and print it:", UVM_LOW)
              new_tr.print();
              ap.write(new_tr);
            end
          endtask
          ```

  - scoreboard
    - scoreboard要比较的数据一是来源于reference model，二是来源于o_agt的monitor。

      ```
      super.main_phase(phase);
      fork
        while (1) begin
          exp_port.get(get_expect);
          expect_queue.push_back(get_expect);
        end
        while (1) begin
          act_port.get(get_actual);
          if(expect_queue.size() > 0) begin
            tmp_tran = expect_queue.pop_front();
            result = get_actual.compare(tmp_tran);
            if(result) begin
              `uvm_info("my_scoreboard", "Compare SUCCESSFULLY", UVM_LOW);
            end
      ```

    - 完成my_scoreboard的定义后，也需要在my_env中将其实例化。
  - sequence机制
    - sequence机制用于产生激励，它是UVM中最重要的机制之一。
    - sequencer
      - sequence产生transaction，sequencer将sequence发送给driver，而driver负责接收transaction。
      - sequencer的定义非常简单，派生自uvm_sequencer，并且使用uvm_component_utils宏来注册到factory中。
    - sequence
      - sequence就像是一个弹夹，里面的子弹是transaction，而sequencer是一把枪。
      - sequence是一个uvm_object。有其生命周期，使用uvm_object_utils宏注册到factory中。
      - 代码：

        ```
        virtual task body();
          if(starting_phase != null)
            starting_phase.raise_objection(this);
          repeat (10) begin
            `uvm_do(m_trans)
          end
          #100;
          if(starting_phase != null)
            starting_phase.drop_objection(this);
        endtask
        ```

        - uvm_do。这个宏是UVM中最常用的宏之一，它用于：①创建一个my_transaction的实例m_trans；②将其随机化；③最终将其送给
sequencer。
          - uvm_do宏产生了一个transaction并交给sequencer，driver取走这个transaction后，uvm_do并不会立刻返回执行下一次的uvm_do宏，而是等待在那里，直到driver返回item_done信号。此时，uvm_do宏才算是执行完毕，返回后开始执行下一个uvm_do，并产生新的transaction。
      - 一个sequence在向sequencer发送transaction前，要先向sequencer发送一个请求，sequencer把这个请求放在一个仲裁队列中。sequencer要做两件事。
        - 1.检测仲裁队列里是否有某个sequence发送transaction的请求
        - 2.检测driver是否申请transaction。
        - 如果仲裁队列中有发送请求，同时driver也在向sequencer申请新的transaction，那么将会同意发送请求，sequence产生transaction并交给sequencer，最终driver获得这个transaction。
          - 在uvm_driver中有成员变量seq_item_port，而在uvm_sequencer中有成员变量seq_item_export，这两者之间可以建立一个"通道"，通道中传递的transaction类型就是定义my_sequencer和my_driver时指定的transaction类型。
          - 在my_agent中，使用connect函数把两者联系在一起：drv.seq_item_port.connect(sqr.seq_item_export);
          - 二者连接好之后，就可以在driver中通过get_next_item任务向sequencer申请新的transaction：seq_item_port.get_next_item(req);
      - default_sequence的使用
        - 作用/目的：向sequencer中送出transaction
        - sequence可以在my_env的main_phase中通过seq.start()手工启动。
          - 一般不用这种方式

            ```
            task my_sequencer::main_phase(uvm_phase phase);
            my_sequence seq;
            phase.raise_objection(this);
            seq = my_sequence::type_id::create("seq");
            seq.start(this);
            phase.drop_objection(this);
            endtask
            ```

        - 使用最多的还是通过default_sequence的方式启动sequence。在某个component（如my_env，实际在case中）的build_phase中设置如下代码：

          ```
            uvm_config_db#(uvm_object_wrapper)::set(this,
             "env.i_agt.sqr.main_phase",
                                 "default_sequence",
            case0_sequence::type_id::get());
          ```

          - 在sequence中使用starting_phase进行提起和撤销objection：   virtual task body();

            ```
                if(starting_phase != null)
                   starting_phase.raise_objection(this);
                repeat (10) begin
                   `uvm_do(m_trans)
                end
                #100;
                if(starting_phase != null)
                   starting_phase.drop_objection(this);
             endtask
            ```

    - driver
      - driver只负责驱动transaction，而不负责产生，只要有transaction就驱动，所以必须做成一个无限循环的形式。
        - 通过get_next_item任务来得到一个新的req，并且驱动它，驱动完成后调用item_done通知sequencer。

          ```
          task my_driver::main_phase(uvm_phase phase);
            vif.data <= 8'b0;
            vif.valid <= 1'b0;
            while(!vif.rst_n)
               @(posedge vif.clk);
            while(1) begin
               seq_item_port.get_next_item(req);
               drive_one_pkt(req);
               seq_item_port.item_done();
            end
          endtask
          ```

      - 将transaction级别的信号拆解为signal级别的信号驱动给dut。
  - 测试用例
    - base_test
      - base_test派生自uvm_test，使用uvm_component_utils宏来注册到factory中。在build_phase中实例化my_env，并设置sequencer的default_sequence。需要注意的是，这里设置了default_sequence，其他地方就不需要再设置了。
        - 如果新增了case的话在各自新增的case中启动default_sequence。
      - 新增report_phase，在main_phase结束之后执行。
    - 独立的测试用例的启动
      - 定义：要测试一个DUT是否按照预期工作，需要对其施加不同的激励，这些激励被称为测试向量或pattern。一种激励作为一个测试用例，不同的激励就是不同的测试用例。
        - 在tb中使用run_test();
        - 在仿真命令中增加 +UVM_TEST_NAME=my_case0
        - 新case的类派生自base_test，同文件中需要定义自己的sequence以产生不同的transaction激励。

          ```
          class case0_sequence extends uvm_sequence #(my_transaction);
            my_transaction m_trans;

            function  new(string name= "case0_sequence");
               super.new(name);
            endfunction

            virtual task body();
               if(starting_phase != null)
                  starting_phase.raise_objection(this);
               repeat (10) begin
                  `uvm_do(m_trans)
               end
               #100;
               if(starting_phase != null)
                  starting_phase.drop_objection(this);
            endtask
            `uvm_object_utils(case0_sequence)
          endclass


          class my_case0 extends base_test;
            function new(string name = "my_case0", uvm_component parent = null);
               super.new(name,parent);
            endfunction
            extern virtual function void build_phase(uvm_phase phase);
            `uvm_component_utils(my_case0)
          endclass
          function void my_case0::build_phase(uvm_phase phase);
            super.build_phase(phase);
            uvm_config_db#(uvm_object_wrapper)::set(this,
                                 "env.i_agt.sqr.main_phase",
                                 "default_sequence",
                                 case0_sequence::type_id::get());
          endfunction
          ```