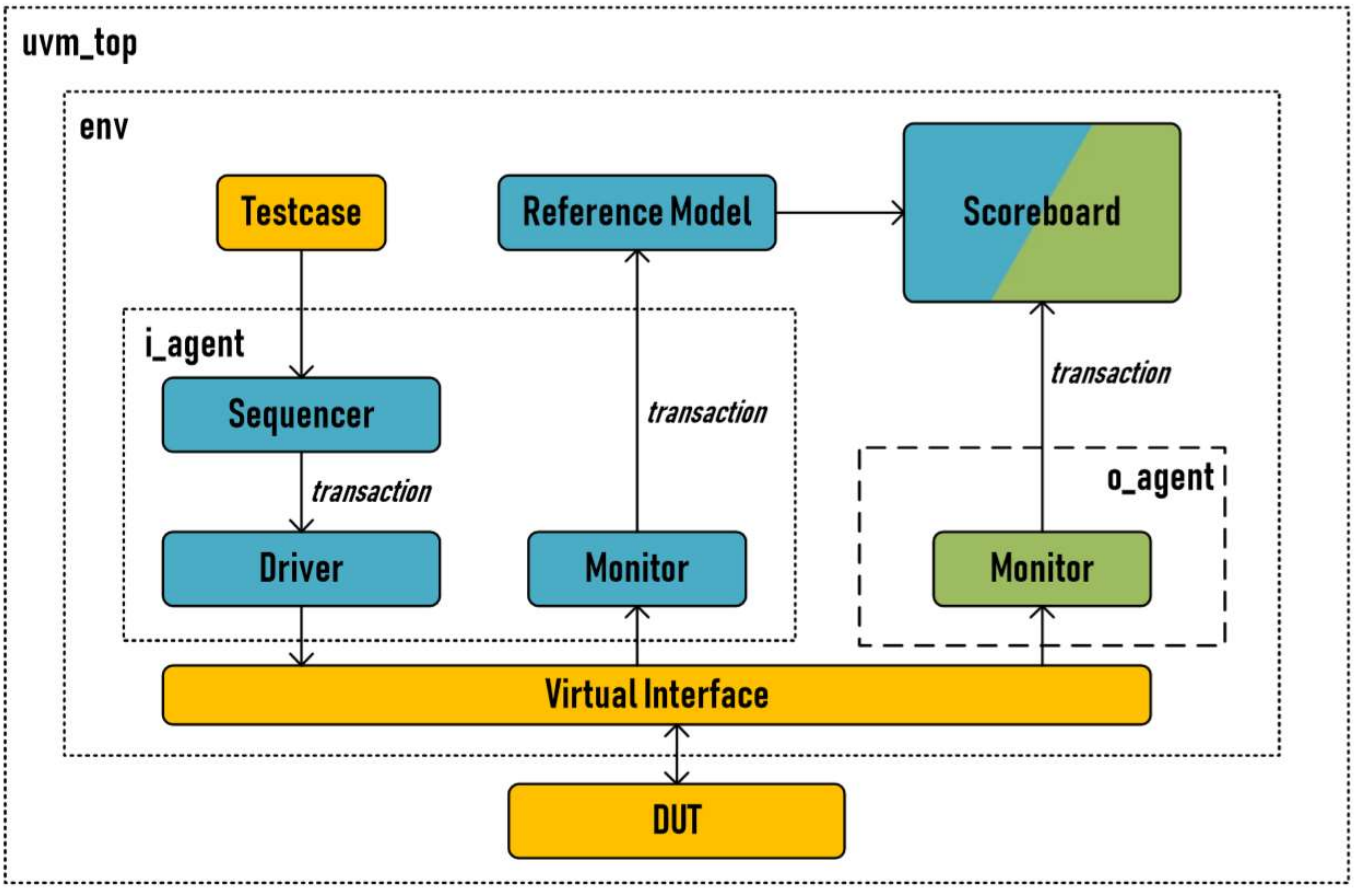
UVM文档

Author	Version	Changelog
Geralt	1.0	Initial Version

I. UVM测试平台框架和组件



其中DUT是待验证的设计。所有UVM模块都在UVM `env` 下生成， `testcase` 通过随机的方法（UVM自带随机）例化测试激励， `sequencer` 负责将得到的 `testcase` 以包的形式发送出去，包的形式定义在 `transaction` 中。包是UVM验证的特点，将底层信号抽象化就是包的概念。

> 例如验证一个AXI总线，向 `0xDEADBEEF` 写 `0xBABEBABE` 就是抽象的包，而对应的 `addr` ， `data` ， `ready` ， `valid` 等等就是底层信号。

`driver` 的任务就是将从 `sequencer` 收到的包“翻译”成底层的信号，并发送到 `interface` 上（一般为 `virtual interface` ，这里可以不必深究）。 `interface` 与DUT上的输入输出信号相连，在基础的UVM测试平台上， `interface` 上没有输入输出的概念，所有的信号线都可以用 `logic` 来表示。

`monitor` 对 `interface` 上的信号进行采集，其中 `i_mon` 输入信号（这里的输入信号并不是指DUT的输入，而是既可以是输入也可以是输出。例如AXI master到slave的握手信号，具体 `i_mon` 监测哪些子信号由用户代码决定）， `o_mon` 对输出信号进行收集。 `i_agent` 可以看作是 `sequencer` 、 `driver` 和 `i_mon` 的wrapper，会在内部例化这三个模块，并会被UVM `env` 例化。 `o_agent` 类似。

`i_mon` 收集到的信号会以相同的 `transaction` 形式打包给 `reference Model`， `reference model` 根据监测到的输入信号预测输出的结果； `o_mon` 收集到的信号也会以 `transaction` 形式打包给 `scoreboard` （一般不对 `o_mon` 收集到的信号进行处理而是直接进入 `scoreboard` ）。 `Scoreboard` 对预测的输出结果和实际的输出结果进行比较。

II. UVM模板

UVM模板中的文件如下：

- `base_test.sv` 测试基类，会实现一些测试框架的基础用法，例如定义验证平台最长运行时间（ `timeout` ），定义测试结束后打印的结果等等， `top_testcase.sv` 中实现的子类会继承 `base_test` 中的基本方法。
- `run_top.tcl` 测试平台运行入口，定义在irun中运行UVM的库、输入文件、配置等等，这里也可以加入用户自定义参数，如：

  ```
  +config=$1
  ```

  用户自定义参数可以以字符串的形式传入UVM验证平台中。
- `shm.tcl` 定义仿真过程中需要dump的波形，默认为所有波形，实际使用过程中不会修改这个文件。
- `top_agent.sv` 定义UVM `agent` ， `i_agent` 和 `o_agent` 都会定义在这个文件中， `i_agent` 和 `o_agent` 的区别可以通过 `uvm_agent` 自带的方法判断：

  ```
  is_active == UVM_ACTIVE
  ```

  来定义， `i_agent` 为 `ACTIVE` 模式，内部会例化 `sequencer` ， `driver` 和 `monitor` ； `o_agent` 为 `PASSIVE` 模式，内部仅会例化 `monitor` 。
- `top_driver.sv` 定义 `i_agent` 内部的 `driver` ，将 `transaction` 翻译为DUT-level信号的行为。
- `top_env.sv` 整个UVM环境，会在内部实例化 `agent` ， `reference model` 和 `scoreboard` ，并定义互相之间通信的 `analysis port` （用于传输 `transaction` ）
- `top_filelist.f` 该文件被 `run_top.tcl` 调用，作为寻找UVM库、UVM平台顶层文件 `top_tb.sv` 、DUT文件等等的入口
- `top_if.sv` 定义DUT和验证平台的接口
- `top_model.sv` 定义验证平台的 `reference model`
- `top_monitor.sv` 定义验证平台的 `monitor` ，较为简单的做法是同时定义两个 `monitor` ，一个为 `i_mon` ，另一个为 `o_mon` ，再分别为两个 `monitor` 声明端口
- `top_scoreboard.sv` 定义记分牌，较为简单的做法是直接调用 `compare` 方法比较预测包和实际包是否完全一样，同时可以在这里记录命中数和不匹配数。
- `top_sequence.sv` 定义 `sequence` ，即 `sequencer` 中例化的单个测试激励。在这里可以定义不同类型的 `sequence` ，来应对不同种类的测试激励（如*静态生成*10个随机包，或根据外部用户输入*动态生成指定个*随机包， `sequence` 可以看作是和外部的输入接口，一般较复杂的验证会从 `sequence` 侧接入Python进行测试用例管理）
- `top_sequencer.sv` 定义 `sequencer` ，此文件一般不用修改。
- `top_tb.sv` 例化测试激励，并定义与激励相连的 `interface` ，比较简单的做法是分别为 `i_agent` 的 `driver` 、 `monitor` 和 `o_agent` 的 `monitor` 定义三个 `interface` ，这样三组信号与DUT输入输出的关系可以互不干扰。（高级的做法是采用 `clocking port` ）在文件内按照dependency来include所有其他的SystemVerilog文件。标准的顺序如下：

```
`include "top_if.sv"
`include "top_transaction.sv"
`include "top_sequence.sv"
`include "top_sequencer.sv"
`include "top_driver.sv"
`include "top_monitor.sv"
`include "top_agent.sv"
`include "top_model.sv"
`include "top_scoreboard.sv"
`include "top_env.sv"
`include "base_test.sv"
`include "top_testcase.sv"
```

- `top_testcase.sv` 会被 `top_tb.sv` 调用，一般不用修改。
- `top_transaction.sv` 定义测试平台中用到的包，定义对包中抽象对象的约束（如范围等）

III. UVM测试平台运行过程及Debug方法

一般可以在 `base_test` 的 `build_phase` 中 `set_timeout` ：

```
uvm_top.set_timeout(800000000ns, 0);
```

认为没有问题后可以将这句话注释，这样测试平台会持续运行到所有 `phase` 结束为止。

UVM的执行过程是以UVM树和phase作为查找和运行顺序的，关于UVM树和 `phase` 可以参考：

UVM中的phase执行顺序

Debug思想是收敛。即预测的模型输出、Design DUT的输出和计划的目标输出（设计目标）要一致。相当于double check，预测的模型输出和计划输出相比较，同时和designer的DUT相比较。随着测试用例的增多，发生不收敛的可能性也越大，因此不同测试用例通过的个数越大，Design越可靠。

IV. SystemVerilog和UVM的一些语法

```
@(negedge vif.clk);
@(posedge vif.clk);
等待信号边沿
```

```
wait(vif.rst_n == 1'b1);
等待信号电平
```

```
repeat(10) @ (posedge vif.clk);
等待信号边沿多次
```

```
`uvm_info("top_driver", "end drive initial", UVM_LOW);
输出log，UVM_LOW/UVM_MEDIUM/UVM_HIGH是log的冗余度（verbosity），可以在testbench中设置，如果testbench对log的verbosity为UVM_MEDUIM，则代码中为UVM_MEDIUM或UVM_LOW的log都会被打出来，而UVM_HIGH会被屏蔽。
```

```
function ...
```
不耗时函数，一般用在main_phase之前的phase或者reference model及scoreboard中

```
task ...
```
耗时函数，即对应实际的仿真时间，一般都是main_phase中的函数

```
fork ... join
```
并行执行，并且等待内部并行块全部执行完后再执行下一句话。不同的并行块用begin...end区分。

```
fork ... join_none
```
并行执行，不等待并行块执行完，直接执行下一句话。

```
fork ... join_any
```
并行执行，等待内部任一个并行块执行完后执行下一句话。