UVM 入门和进阶实验 2

同学们进入了实验 2，也依然会遇到"代码量暴增”的情况。不过还好，我们已经经过了 SV 模块实验 3 到实验 4 代码暴增的同样情况，这点心理承受能力还是有的。而且，相比较于 SV 而言，UVM 在实验 2 的代码，你一定非常熟悉！为什么这样讲呢？因为它们几乎是全部从 SV 最后的实验 5 移植过来的，它们包括了:

- SV 验证环境结构
- SV 组件之间的通信管道
- SV 的激励产生和发送模式
- SV 的数据检查和报告
- SV 的测试开始与结束方式
- SV 的配置方式

如果你对上面的一些要素还不熟悉，请再通读 SV 模块的实验 5 代码，分别对照上述的元素。之所以要求大家做这样的回顾，是因为我们从实验 2 开始一直贯穿到 UVM 入门模块的实验 5，会将之前 SV 的实验代码逐渐利用 UVM 所学的知识点，完全按照 UVM 的特性，最终移植为纯粹的 UVM 测试。那么这么设计实验的原因在哪儿呢？对学员的帮助有什么呢？用处太大了！

- 在 SV 的实验部分，我们做的是“加法”，即代码量不断增加，逐一完善为 SV 的"纯软件”类型的灵活验证环境。
- 在 UVM 的实验部分，我们做的是“减法”，即将搬迁来的 SV 验证环境中 上述的验证环境元素，逐一替换为 UVM 的特性，继而让同学们在亲自动手实验的过程中，能够充分比较和理解 UVM 的特性，以及与 SV 对应特性相比，如此替换

的优势在什么地方。

- 由上述两点可以看出， 我们的实验代码部分在接下来的试验中并不会"暴增”， 而只是做 UVM 对应特性的替换。在替换过程中，我们除了需要按照实验要求完成代码， 还应该思考，SV 与 UVM 在验证环境上不同的实验方式，究竟孰优孰劣?

那么，我们接下来就进入到我们 UVM 入门的实验“正餐”，关于 SV 完整验证环境的移植。我们本次实验中，需要将之前学习到的以下知识充分应用和理解。它们包括:

- 各个验证组件的使用
- 验证组件之间的层次关系
- 工厂的“注册”和”创建”
- “域自动化”和 uvm object 的预定义方法
- “phase” 的自动执行和顺序关系
- 消息宏的简单使用
- 通过 config_db 对接口的传递
- 测试的选择和开始，以及对仿真结束的控制

看起来需要掌握的有点多，不过呢，路桑为了让大家能够抓住重点，顾大局而不拘小节，已经实现完成了初步的结构改造，鼓励大家对路桑的问题加以思考，稍后我们也会在实验代码回顾环节，帮助大家做详细的解答。那么，UVM 世界的"构建之门”就为大家敞开了，注意喽，这次你不再仅仅是“观光客”去感受 UVM 的世界观（实验 1），而是要真正在已经初步完成 SV 环境到 UVM 环境的迁移过程中，去认真审视，UVM 世界与 SV 世界的不同。

验证组件和层次构建

我们首先将各个 package 中的 SV 组件替换为 UVM 组件，在替换过程中，我们需要遵循以下基本规则：

- 实现组件对应原则：
  - 。 SV 的 Transaction 类对应 uvm sequence item。
  - 。SV 的 driver 类对应 uvm driver 。
  - 。SV 的 generator 类我们稍后会替换为 uvm sequence + uvm sequencer。
  - 。SV 的 monitor 对应 uvm monitor。
  - 。SV 的 agent 对应 uvm. agent.
  - 。SV 的 env 对应 uvm env.
  - 。SV 的 checker 对应 uvm scoreboard.
  - 。SV 的 reference model 和 coverage model 均对应 uvm component.
  - 。SV 的 test 对应 uvmtest
- 在遵循以上对应原则的过程中，我们在进行类的转换时，需要注意：
  - 。SV 的上述类均需**继承于其对应的 UVM 类**
  - 。在类定义过程中，一定需要使用`**uvm_object_utils()**或者`**uvm_component_utils()完成类的注册。**至于注册时，该使用哪一个类，需要清楚各个类分别是 uvm_object 类还是 uvm_component 类。因此，类的地图可以在红宝书的图 10.1 中找到，请大家将扫描放大，放在你学习工作环境最显眼的地方，路桑当年学习日语音阶的时候，也是遵照这个方法。UVM 的类确实错综复杂，有了路桑这张类库地图，闯荡验证江湖就不那么胆怯喽。

    所以这一点， 即指的是 UVM 类的注册需要遵循的规则。

。在使用上述工厂注册宏的时候，会伴随着“域声明自动化”，一般而言，我们建议将 sequence item 定义时，应当伴随其域声明，即利用 `uvm_object_utils_begin 和`uvm_object_utils_end 完成。这是由于对于 sequence item 对象的拷贝、比较和打印等操作比较多，**因此建议在定义 sequence item 类时，也完成域的自动化声明。**

。UVM 初学者，**一定要注意构建函数 new()的声明方式。**请不要试图在构建函数上“玩花样”。路桑指的是，uvm_object 的相关类，它的 new 函数参数只有一个即 string name，而 uvm_component 的相关类它的 new 函数参数只能有两个，即 string name 和 uvm_component parent，除此以外，增加或者减少参数都是非法的哦。**所以这一点，即 UVM 类需要遵循的构建函数定义的规则。**

。在组件之间的层次关系构建中，我们依然按照之前 SV 组件的层次关系。因此，**通过保留这些层次关系，只需要稍后在不同的 phase (阶段)完成组件的例化和连接即可。**这一点，我们将在稍后的 phase 阶段实验思考环节进一步说明。

因此，请同学们按照上述的转换方式，参考 SV 实验 5 的代码和路桑给出的代码，进行对应和参照，思考上述 SV 到 UVM 的代码迁移过程。

测试的开始和结束

从这一部分开始， 我们可以再次加深对测试的开始、环境构建的过程、连接以及结束的控制。请同学们就以下几点，完成对 SV 迁移到 UVM 环境的思考:

- 在 tb.sv 文件中，路桑已经注释了之前关于 SV 各个 test 声明、例化、外部参数传递、执行选择过程以及开始测试的过程。同时，路桑添加了与 UVM 相关的语句:

  。**通过 uvm_config_db 完成了各个接口从 TB (硬件-侧)到验证环境 mcdf_env(软件-侧)的传递。** 这也很好地实现了以往 SV 函数的剥离，即 UVM 用户不再需要深入到目标组件一侧， 调用其 set interface()即可完成传递。这种传递方式有赖于 config_db 的数据存储和层次传递特性。而在 mcdf_env 中，路桑暂时保留了 mcdf_env 的 set_interface()以及其各个子组件的 set_interface()函数。所以，可以看得出来，路桑改造的仅仅是在 TB 与 mcdf_env 之间的接口传递，然而理论上，我们可以移除所有的 set_interface()函数，完全使用 **uvm_config_db_set 和 get 方法，从而使得 mcdf env 与其各个子组件之间也实现“层次剥离”** ，这样也就进一步促进了组件之间的独立性。

  。**调用 run test()函数即完成了 test 的选择、例化和开始测试。**这也是红宝书 10.5.2 节如何开始 UVM 仿真中所阐述的部分，因此通过对照 SV 代码，大家可以进一步理解为何 run test()如此重要，它提供的便利包括：

  ■ 用户可以在代码中指定 UVM test,或者为了避免代码频繁修改，可以通过 **+ UVM_ TESTNAME= mytest 在仿真选项中灵活传递 test 名。**

  ■ **在 run. test()执行中，它会初始化 objection 机制**，即查看 objection 有

没有挂起的地方，因此，在 test 或者 generator 中必须至少有一处地方使用 phase.raise. objection(来挂起仿真，避免仿真退出，而在仿真需要结束时，使用 phase.drop. objection()来允许仿真可以退出。请在 MCDF test 中查找这两个函数的调用，理解对应 test 将在何时结束，并且试着注释掉这两句话，看看将会有什么惊喜等着你，试着去解释一下原因。

■ **创建 uvm test 组件，及其以下的各层组件群。**

■ **调用 phase 控制方法，按照所有 phase 按照顺序执行。**这一点，我们在之前的实验 1 已经做了代码练习，请大家再认真思考，uvm_component 类中的 build_phase 与 new 函数相比，它们之间的关系是什么？有先后顺序吗？有包含顺序吗？另外，new 函数中可以执行哪些语句，而 build_phase 函数又适合执行哪些语句?

■ 在 **build_phase** 中需要注意，凡是 UVM 类,均应该使用 **"T:type_ id:create()"**的方式完成创建，这也为后来工厂的类型覆盖(override)提供了方便。

■ 此外，关于 **connect_phase** 函数的练习，我们在本试验中做的不多，更多的代码实现将在实验 3 即 TLM 通信中实现。同学们可以先在 Mcdf_checker 以及 mcdf_env 中观察其 connect_phase 函数，**理解 Connect_phase 中需要做的动作是什么？**那么与之前 SV 的 new 函数相比，其既做了对象的例化，又做了对象的连接,然而在 UVM 中，将对象的例化放置在 build_phase 中，而将对象的连接放置在 connect_phase 中，这么做的好处在哪里呢?

■ 除了我们将 SV 中 new()函数的对象例化和连接分别移植到 UVM 的

build_phase 和 connect_phase 函数以外，我们也对 SV 组件的 run 任务做了小的改动，可以观察到的细微变化即是将其改为 run_phase 任务。除了名字上的细小差别，在学习了，上一节 phase 控制以后，同学们需要认识到，UVM 的顶层(uvm_root)会控制所有 phase 的执行顺序，而且各个 phase 不再需要手动执行了。这意味着，之前在 SV 中，我们需要从 test 到 env 再到各个 agent，一层层地调用 run 任务，继而“一盏盏地点亮”组件，与这种稍显笨拙的方式相比，UVM 做了什么？在整个 phase 调度方面，它就是一个贴心的管家有木有？完全不再需要你操心了呢！