简介，原则和目标

easier UVM 是一组编码指南，附带一个代码生成器，可以创建符合本指南的 UVM 代码。创建 easier UVM 是为了帮助个人和项目团队学习，然后尽快使用 UVM 提高工作效率，并减轻在公司或组织内支持 UVM 代码的负担。

SystemVerilog 是一种非常庞大且复杂的语言，UVM 是一个庞大而复杂的基类库。这对采用者提出了挑战，因为通常有很多方法可以做同样的事情。通过选择特定方法，用户可能会发现自己陷入困境，偏离行业最佳实践，或创建不可互操作的代码。 UVM 本身的存在是为了通过提供用于构建基于类的验证环境的标准基类库来解决此问题。但 UVM 为您提供了几种方法来做同样的事情，部分原因是需要向后兼容传统方法。向后兼容性本身就是一件好事，但是当你仍在学习复杂的方法时，选择太多可能是件坏事，并且拥有各种编码风格会增加维护和支持 UVM 代码库的负担。easier UVM 专门用于通过推荐一种方法来解决这个问题。

在设计 easier UVM 编码指南时，我们必须对如何做事做出具体选择。 该指南比官方 UVM 类参考或 UVM 用户指南更具说明性：本文档提供了一些非常具体的建议，说明要使用哪些 UVM 功能以及如何使用它们。 在某些情况下，我们已经能够推荐整个行业共同商定的最佳实践。 在其他情况下，我们不得不做出一个相当随意的选择，以支持一种做事方式而不是另一种方式。 通常认为提供明确的指导而不是提出替代方案更有用，但这并不是说替代方法不会同样有效。 在少数情况下，我们增加了一个附注，指出了替代方法，并解释了采取的方法的基本原理。

通过减少编码模式的数量，通过推荐特定的编码约定，以及通过自动生成验证环境的初始框架，Easier UVM 可以更轻松地创建可维护和可重用的 UVM 代码的代码库。 easier UVM 根据工业项目的经验总结了最佳实践。 在某些情况下，Easier UVM 已被证明可以在项目开始时保存 6 周的编码工作（取决于项目的细节），以帮助避免陷阱，使代码更可重用，并帮助统一 UVM 的方式，用于整个公司。 由于 Easier UVM 代码生成器本身在开源许可下可用，因此您甚至可以根据自己的需要自由修改代码生成器。

easier UVM 意味着被视为一组编码指南或偏好。单独 easier UVM 无法解决您在构建复杂验证环境时将面临的每个问题，因此，Easier UVM 中的每个规则都可能有例外。虽然从 Easier UVM 代码生成器生成的所有代码都符合指南，但您可以选择完全遵循 easier UVM 编码准则，或者您可以自由地调整指南以满足您自己的要求。因此，我们没有试图在准则列表中区分硬规则和软准则。 SystemVerilog 语言和 UVM 本身就是标准。

您将添加到代码生成器生成的基本框架的许多代码将是特定于项目的。例如，记分板通常会变得非常复杂，特定于应用程序并且难以编写，在某些情况下会使样板代码相形见绌。虽然 Easier UVM 旨在使事情变得更容易，但 SystemVerilog 和 UVM 仍然非常难以学习和使用，所以请不要认为 Easier UVM 将无需参加正式的培训课程或减少花时间在自我上的需要研究。您仍然需要对您正在做的事情有充分的了解，但是一旦您完成了正式的培训，Easier UVM 将为您提供一种方法，在您开始处理您的第一个真实项目时应用这些知识。

UVM 验证环境的结构

UVM 是一种在 SystemVerilog 中构建基于类的验证环境的方法，利用面向对象的编程技术来帮助代码重用。重用是 UVM 的核心：如果您不打算从一开始就重用验证码，那么您可能已经错过了 UVM 的重点。

UVM 验证环境的构建块是对象，即类的实例，而不是 Verilog 或 VHDL 用户熟悉的模块，进程和函数。使用对象的重要性在于它们可以在运行时替换，在重用验证组件和测试时不会篡改原始源代码，从而提供了极大的灵活性。您可以使用现有的 VIP（验证知识产权项目）并替换子组件，更改其生成的 transactionsequence，或扩展其行为，而无需触及（或复制）源代码，也无需 VIP 的原始作者。除了遵循良好的编码指南之外，以任何方式预测您的更改。这是使用一些 OOP（面向对象编程）技巧实现的。

easier UVM 识别三种主要类型的用户定义对象，以及一些其他不太重要的对象。三个主要对象类型是组件，sequence item 和 sequence（其中相应的类分别是 uvm_component，uvm_sequence_item 和 uvm_sequence），而不太重要的类包括配置和回调对象。组件是结构化的，也就是说，它们在模拟开始时（严格来说，在构建阶段）被实例化，而 sequence item 和 sequence 通常代表测试激励并且是动态的，即，它们在运行时实例化运行。只要遵循编码准则，任何这些对象都可以在创建时替换为修改或扩展版本。

easier UVM 定义了组件，sequence item 和 sequence 的编码模板。这些编码模板中的每一个都是简单且规则的，并且三种模板尽可能相互一致。在每个类中排序行的方式和用户定义名称的命名约定都有编码指南，所有这些都使您的代码看起来一致，并使其他人更容易找到自己的方式。

通过调用 factory 方法来实例化这三种对象中的每一种，这允许原始对象以原始作者不需要的方式替换替换。 以这种方式始终如一地使用 UVM factory 是能够在 UVM 中利用 OOP 的关键之一。

每个组件实例可以具有关联的配置对象，其包含特定于该组件实例的配置信息（即参数）。配置对象（和其他配置信息）以自上而下的方式插入配置数据库，通常来自测试或环境，然后可供层次结构中较低的组件访问。这种配置机制非常方便，因为配置对象可以使用单个调用随机化（具有内联约束），并且非常灵活，因为可以在适当的情况下从多个组件访问相同的配置对象。

除了使用正则化编码模板，factory 和配置之外，还存在验证环境的整体组织，将验证环境连接到 DUT，生成可重用激励，功能覆盖信息的收集，消息报告等问题，以及测试结束机制。easier UVM 为每个领域提供了具体指导。

词汇指南和命名约定

☐每行只有一个声明或定义。

除了帮助提高可读性之外，还有助于确保需要参考源代码的任何工具的平稳运行，例如编译器错误消息或代码覆盖率信息的注释以及源代码调试。 在单独的行上声明使得注释更容易。 在声明中使用以逗号分隔的相关名称列表可能没问题，但应避免在逗号分隔列表中包含变量初始化。

☐为 SystemVerilog 变量和类创建用户定义的名称时，请使用以下划线分隔的小写单词（与 camelBackStyle 相对）。

尽管此约定并不重要，但它确实有助于保持一致，并且此建议与 UVM 基类库本身一致。

☐为 SystemVerilog 枚举文字，常量和参数创建用户定义的名称时，请使用以下划线分隔的大写单词。

同样，这并不重要，但与 UVM 基类库一致。

☐将所有用户定义的 UVM 实例名称（即组件实例名称等字符串）限制为字符集 a-z，A-Z，

0-9 和_（下划线）。

请记住，使用其他标点字符或符号可能会使名称难以在软件用户界面或自动生成的报告的上下文中解释。 字符$和__（双下划线）因破坏下游工具而臭名昭着。

☐对局部变量使用较短的名称，对类名和包名等全局项使用更长，更具描述性的名称。

这只是一种很好的编程风格。

☐在用户定义的类成员变量的名称之前使用前缀 m_（在 SystemVerilog 中官方称为类属性）。

前缀 m_的动机是在从类的方法中引用时将类成员变量与函数参数和局部块作用域变量区分开来，并在从类外引用时区分类成员变量和方法。 前缀 m_仅用于类成员变量，而不是在块或方法内声明的变量，因为块变量的范围无论如何仅限于声明它们的块或方法。 不要将前缀 m_与 port，export 或虚拟接口一起使用，无论如何通过拥有自己的命名约定来区分它们。在 UVM 文档中还有许多名称没有 m_前缀的特殊变量，即 is_active，coverage_enable，checks_enable 和 regmodel。

☐在每个 agent 中分别使用名称 m_sequencer，m_driver 和 m_monitor 作为 sequencer，driver 和 monitor 的实例名称。

固定名称就足够了，因为每个 agent 只有一个 sequencer，driver 和 monitor。

☐分别在每个 env 和 agent 的实例名后面加上后缀_env 和_agent。

当在同一级别实例化多个环境或 agent 时，每个环境或 agent 都需要被赋予唯一的实例名称，例如， m_amba3_agent 与 m_pcie_agent。

☐使用名称 m_config 作为具有一个组件或 sequence 的配置对象的实例名称。

☐在用户定义的配置类名后使用后缀_config。

从配置数据库引用配置对象时，配置数据库字段名称应为“config”。

☐在用户定义的 port 名称后使用后缀_port。

☐在用户定义的 export 名称后使用后缀_export。

port 和 export 名称不需要前缀 m_，因为 port 和 export 总是类成员变量。

☐在用户定义的虚拟接口名称后使用后缀_vif。

如果组件中只有一个虚拟接口，则允许使用名称 vif，而不是将_vif 作为后缀添加到另一个名称。

☐使用关键字 typedef 引入的用户定义类型定义后，使用后缀_t。

类的 forward typedef 是一个例外，因为它总是引用类名。 后缀_t 的动机是，当名称用于在类中定义变量或函数参数（它告诉您从哪里开始查找类型定义）并将 typedef 与类成员变量和方法区分开来时，区分 typedef 和类名。 从课外访问时。 不要将后缀_t 添加到类名。通常可以将 UVM 类名称与通常使用它们的上下文中的其他名称区分开，例如， my_agent m_agent，其中前缀 m_将变量名称与类型名称区分开来

☐在用户定义的软件包名称后使用后缀_pkg。

见例子。

☐在任何可以为源代码增加价值的地方写评论，并帮助读者理解代码的用途。

不要写仅仅重复代码本身的注释，否则不必要，因为不必要的注释会增加维护代码的成本。

☐在有助于使代码更具可读性的任何地方包括空格（空行，缩进）。

具有很少或没有空白区域的代码可能难以扫描。

☐覆盖内置 UVM 虚拟方法时，请勿在重写方法定义的开头插入 virtual 关键字。

这样做对语义没有影响，但会使文本混乱。 这特别指的是 UVM 常用 phase 和 UVM run-time phase 方法（build_phase 等）以及 uvm_object（do_copy 等）的用户可定义挂钩。

一般准则

☐不要使用在 UVM 类参考或基类库中特别标记为已弃用的 UVM 功能。
☐请勿使用 UVM 类参考中未记录的 UVM 基类库代码的内部功能。

UVM 类参考是权威标准，而不是源代码。 例如，不要引用在 UVM 类库中声明的具有 m_前缀的任何变量，因为这些变量不是标准的一部分。

通用代码结构

☐在构建和编码验证环境时，主要考虑重用。

UVM 的主要目的之一是使验证组件，验证环境和测试激励可重复使用，因此请始终考虑关注点的分离。避免引入任何会妨碍后续重用的依赖项。每个 agent 都应该在重用时编写，以便可以在任何具有最小约束的验证环境中实例化。
☐始终使用一致的文件结构和一致的文件命名约定。

大多数文件应包含单个模块，接口，包或类，在这种情况下，文件名应与文件中定义的项的名称匹配，并且文件扩展名应为.sv。 （参见代码生成器）。
☐每个类都应该在一个包中定义（而不是在模块或文件范围内定义类）。

您可以拥有多个包，每个包中有多个类。可以在多个包中使用相同的类名，在这种情况下，每个匹配项都会定义一个不同的类。
☐在包中使用`include 指令允许将每个类放在一个单独的文件中

而不是将所有类放在一个非常大的包文件中。
☐使用条件编译保护措施避免多次编译同一个包含文件。
例:

```
`ifndef BUS_PKG_SV
`define BUS_PKG_SV
...
`endif // BUS_PKG_SV
```

☐不要在编译单元范围内使用通配符导入。

也就是说，不要编写导入声明，例如 `import my_package :: *;` 在任何模块或包之外，因为这样做会使任何导入的名称在文件中的所有模块和包中可见，从而破坏了使用包来限制名称范围的目的。 此建议适用于编译单元范围内的所有导入声明，而不仅仅是通配符导入，但通配符导入是最具破坏性的。
☐包含 `uvm_macros.svh` 并在引用 UVM 基类库的每个包或模块中导入 `uvm_pkg :: *`

而不是在文件范围包含/导入名称。

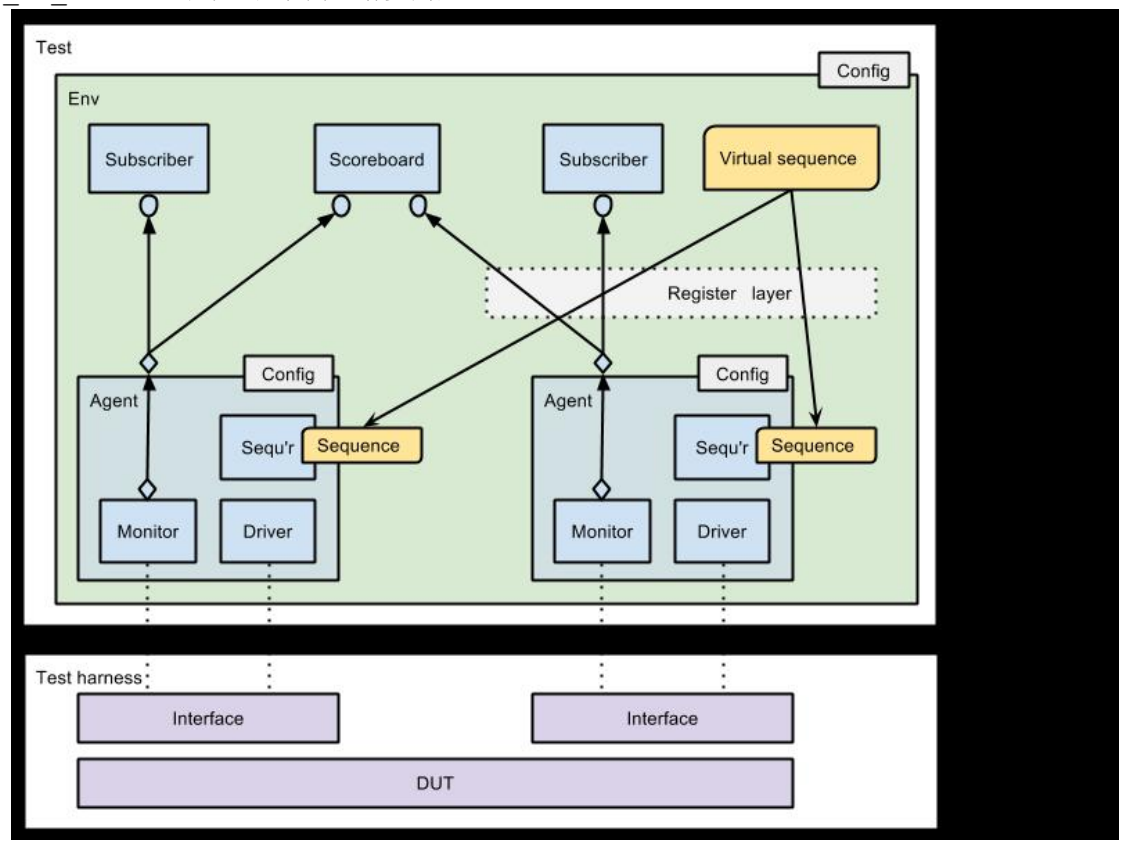
Example

```
`ifndef BUS_PKG_SV
`define BUS_PKG_SV
package bus_pkg;
`include "uvm_macros.svh"
import uvm_pkg::*;
`include "bus_tx.sv"
```

```
`include "bus_config.sv"
`include "bus_driver.sv"
`include "bus_monitor.sv"
`include "bus_sequencer.sv"
`include "bus_agent.sv"
`include "bus_coverage.sv"
`include "bus_seq_lib.sv"
endpackage
`endif // BUS_PKG_SV
```

☐每个接口使用一个 agent，带有 passive monitor 和可选的 sequence 和 driver，其存在性由类 uvm_agent 的 get_is_active 方法的值决定。

由于 DUT 通常具有多个接口，这意味着将多个 UVM agent 组织为并行结构。 （有关 get_is_active 的讨论和示例，请参阅配置。）



图：验证环境的组织

☐agent 不应实例化除一个 sequencer，一个 driver 和一个 monitor 的规范 agent 结构之外的组件。

agent 可以在应用程序要求的情况下实例化其他组件，但这通常应该避免。 每个 agent 通常都有一个关联的配置对象，但该对象应该由封闭的 env 实例化，而不是由 agent 本身实例化（请参阅配置）。

在 DUT 具有多个类似接口（例如，网络路由器上的多个端口）的情况下，相同 agent 的多个实例可以在另一个 UVM 组件内组合在一起以形成组件层次结构。 当验证环境从块级重用到系统级时，可能出现嵌套的验证环境层次结构。

☐使用虚拟 sequence 协调多个并行 agent 的激励生成活动

也就是说，使用虚拟 sequence 来启动属于多个 agent 的 sequence 上的 sequence。虚拟 sequence 可以在测试之间重用，这是激励重用的有用机制。避免使用“default_sequence”配置参数作为启动 sequence 的方法。通常，您应该通过调用其 start 方法来启动 sequence。如果要在阶段开始时启动 sequence，uvm-1.2 建议使用类 uvm_sequence_library。

☐检查和功能覆盖收集应在检查器，记分板，覆盖收集器和其他临时订户组件中执行，这些组件在任何 agent 外部实例化并连接到 monitor 的 analysis port。

通常，检查和功能覆盖率收集不应在 agent 本身中执行，agent 本身应保持协议特定但与 DUT 无关，并且不应按生成激励的顺序执行。在某些情况下，通过从配置数据库中获取某些参数，可以使 agent 更具可重用性。在任何情况下，agent 应该只包含协议固有的检查和覆盖集合代码，并且只要 agent 在测试之间或验证环境之间重用，就可以重用。为了最大限度地提高 agent 的可重用性，您应该尝试预测可能需要的配置更改，以支持未来版本的接口。

☐通常，使用 analysis port 和 export 连接 agent，检查器，记分板和 coverage 收集器。

monitor 或 agent 应仅使用传出 analysis port 将 transaction 发送到验证环境的其余部分。应限制 driver 与一个 sequencer 和一个 SystemVerilog 接口通信，并且不应该需要 analysis port。记分板可以合并多个并行 agent 的输出。 analysis port 减少了组件之间的耦合。通常，避免在组件之间进行直接对象引用。 在某些地方可以接受对另一个组件的直接引用，例如虚拟 sequence 直接引用 agent 中的顺控程序以启动子 sequence。

☐应编写 UVM 环境，使其可用作顶级环境，或在其他较大的验证环境中作为子环境重复使用。

IP 重用导致 VIP 重用，因此来自一个项目的 UVM env 可以作为下一个较大项目中的子组件重用，从而产生 UVM 组件的层次结构，包括嵌入式 agent，记分板和其他分析组件。

☐使用 factory 覆盖和/或配置数据库，使重新调整的 UVM 组件的行为适应新验证环境的需要。

避免修改重新使用的代码本身。 当首次编写组件并因此通过从配置数据库获取参数而构建到组件中时，可以预先使用组件的参数化要求来使用组态数据库。 factory 覆盖可用于进行意外的更改，但可能是一种钝器，因为它们会导致批量更换对象或方法。

☐使用 factory 覆盖和/或配置数据库，使重新调整的 UVM 组件的行为适应新验证环境的需要。

避免修改重新使用的代码本身。当首次编写组件并因此通过从配置数据库获取参数而构建到组件中时，可以预先使用组件的参数化要求来使用组态数据库。factory 覆盖可用于进行意外的更改，但可能是一种钝器，因为它们会导致批量更换对象或方法。

☐顶级模块应设置由测试检索的配置参数，测试应设置 env 检索的参数，并且 env 应设置由较低级别的 env 或 agent 检索的参数。

为了可重用性而将测试与验证组件分离，顶级模块或测试不应包含嵌入验证环境深处的组件的分层实例名称，也不应包含顶级模块或测试共享配置具有叶级 UVM 组件的参数名称。这是根据所需的重用级别进行的判断。例如，agent 不应该从顶级模块直接设置的配置数据库中获取虚拟接口，因为这样做会在顶级模块和 agent 之间引入可能阻碍重用的依赖关系。（参见配置。）

☐通过使用多个 sequencer 来表示分层协议，每个 sequencer 都有自己的 transaction 类型。

协议分层可能导致需要对现有的 UVM agent 进行分层。 创建 agent 的分层结构，其中可能在每个层之间传递不同的 transaction 类型。 使用 factory 覆盖和/或配置数据库来删除每个 agent 中的现有不需要的代码，并在 agent 中的 sequence 上运行新的协议转换 sequence 或直通 sequence，以在协议栈中上下传递 transaction。 （请参阅分层协议和分层 agent。）

Clocks,Timing,Synchronization,and Interfaces

☐在 SystemVerilog 的 module 中生成时钟和复位，而从不在基于 UVM class 的验证环境中，也从不在 SystemVerilog program 中生成。

否则，SystemVerilog scheduler 可能会给出不正确的行为。

☐优先使用 SystemVerilog module，而不是 SystemVerilog program。

program 扮演着将验证环境从 DUT 中 SV schedule 问题中隔离出来的角色，但是这同样能在 clocking block 中完成。使用 program 更方便的情况是处理异步断言。

☐使用 SystemVerilog 接口内的 clocking block 来检测和驱动一个同步 DUT 接口。

SV clocking block 能够很好地隔离开验证环境和门级 timing 以及 SV scheduler 中的不确定性。如果使用过程语句可以充分同步 DUT 引脚的驱动和采样，特别是如果使用简单的 RTL 代码建模 DUT，并且始终从 SystemVerilog 模块而不是程序生成激励，则不使用时钟模块可能是合理的。另注意，访问异步引脚时必须绕过时钟模块。

☐使用 modports 强制使用时钟块，通过 UVM 验证环境中的虚拟接口访问这些时钟块。

modport 还可以在通过 interface port 访问该接口的接口中强制使用时钟块。

☐使用将 clocking block 与异步信号组合在一起的 modports，以访问同时包含同步和异步信号的接口。

时钟块不能用于异步检测和驱动信号，但是可以将异步信号与 modport 中的时钟块组合在一起。

☐尽可能确保 DUT 接口信号与 driver 和 monitor 的精确延时。

driver 和 monitor 组件应与 DUT 接口同步，这意味着这些组件需要等待接口中的信号更改。因此，提供 transaction 给 driver 的 uvm 组件和 sequence 需要提供不阻塞 driver 执行的 transaction-level port 和 export，这样 driver 才总是能够即时响应 DUT 接口上的信号。请注意，sequence 可能仍然是阻塞的，但仅限于它们在等待 driver 时阻塞，而不是在等待外部事件时阻塞。

☐如果一个 driver 在驱动 DUT 接口时需要在 transaction 之间或者 transaction 内插入可变延迟，那么这个可变延迟应当存储在 transaction 中传递给 driver。

将延迟存储在 transaction 中可以使得 sequence 在生成 back2back 时能够控制 timing。

☐driver 应使用非阻塞 try_ *方法从 sequencer 中获取 transaction，以便在作者无法知道 sequence 是否会阻塞 driver 执行的情况下最大限度地提高可重用性。

假如一个运行在 sequencer 上的 sequence 阻塞了发送给 driver 的 transaction flow（比如 sequence 在等待环境中的其他事件），从 driver 调用阻塞性的 get / get_next_item 可能会导致 driver 错过接口上的关键信号。

```
class my_driver extends uvm_driver #(my_transaction);
   `uvm_component_utils(my_driver)
   virtual dut_if vif;
   ...
   task run_phase(uvm_phase phase);
       forever begin
           my_transaction tx
           =my_transaction::type_id::create("tx");
           seq_item_port.try_next_item(req);
           if (req != null) begin // Wiggle pins
```

```
                seq_item_port.item_done();
                @(posedge vif.clock);
                vif.en <= 1;
                vif.cmd <= req.m_cmd;
                vif.addr <= req.m_addr;
                vif.data <= req.m_data;
                end
            else begin
            // Insert an idle cycle
                @(posedge vif.clock);
                vif.en <= 0;
                vif.cmd <= 0;
                vif.addr <= 0;
                vif.data <= 0;
            end
        end
    endtask
endclass: my_driver
```

☐driver 只应在需要时从 sequencer 中提取 transaction。

driver 一次仅获取一个 transaction 而不是获取很多个 transaction，这样可以使得 sequence 能够根据验证环境实时控制 transaction 的随机化生成策略。

☐使用 uvm_event 或 uvm_barrier 在 sequence 和/或 analysis 组件（如记分板）之间进行临时同步。

当对分布在多个 UVM 组件上的并行进程，sequence 或 transaction stream 进行同步时，UVM 的 events 或 barriers 有时可能是比 port 和 export 更方便的通信机制。比如，将被发送到一个 DUT 接口的 transaction 正在等待被另外一个 DUT 接口检测的一个特定场景。另一个例子是几个 sequence 在锁定步骤中彼此运行但在不同的 agent 上运行。

☐monitor 不应在 SystemVerilog 接口中为变量或 wire 赋值。

monitor 应总是作为一个 passive 组件，负责检测接口数据和生成 transaction 并传递 transaction 到其他组件。monitor 的执行不能被其他组件或 sequence 阻塞。在接口中使用 SVA 和 coverage 指令进行协议检查和 coverage 收集。

☐在接口中使用并发断言和 cover property，以进行协议检查和相关的覆盖率收集。

Transactions

☐通过扩展类 uvm_sequence_item 来创建用户定义的 transaction 类。

这样才能作为 sequence item 使用。

☐尽量减少不同 transaction 类的数量。

在一个 agent 的 driver 和 monitor 中使用一个 transaction 类，这样易于维护。

☐在现在或将来可能需要随机化的类成员变量前面使用 rand 限定符。

☐在成员变量之后，定义一个构造函数，该构造函数包含一个字符串名称参数，其默认值可为空字符串，并调用 super.new。

☐推荐重写 convert2string，do_copy，do_compare，do_print 和 do_record 等方法（可使用相关宏`uvm_record_attribute 和`uvm_record_field 等）。
☐推荐重写 do_pack 和 do_unpack 方法（可使用相关宏`uvm_pack_int 等）。
☐使用 factory 实例化 transaction 对象。

```
var_name = transaction_type::type_id::create("var_name");
```

Sequence

☐通过扩展类 uvm_sequence 来创建用户定义的 sequence 类。
☐在现在或将来可能需要随机化的类成员变量前面使用 rand 限定符。
☐在成员变量之后，定义一个构造函数，该构造函数包含一个字符串名称参数，其默认值可为空字符串，并调用 super.new。
☐与 sequence 执行相关的任何 housekeeping 代码，例如提出和撤销 objection，都应放在 sequence 的 pre_start 和 post_start 方法中。
sequence 的 body 方法应该只执行 sequence 的原始功能行为。pre_start 和 post_start 被称为用户可定义的回调。
☐在 body 任务中使用如下通用模板：

```
req = tx_type::type_id::create("req");
start_item(req);
assert( req.randomize() with {…;} ) else `uvm_error( … );
finish_item(req);
```

☐不要使用`uvm_do 系列宏。
`uvm_do 系列宏隐藏了很多功能，如无必要，推荐使用上述的通用模板。
☐推荐在 sequence 中使用内建的 transaction 变量 req 和 rsp。
☐通过调用 start 方法在启动 sequence。
避免使用“default_sequence”配置参数作为启动 sequence 的方法。如果要在 phase 开始时启动 sequence，uvm-1.2 建议使用类 uvm_sequence_library，这是在 sequencer 上启动 background traffic 的有效方式，但调用 start 方法应该用作启动 sequence 的主要方法。
☐仅覆盖 sequence 类的 pre_do，mid_do 和/或 post_do 回调，以修改预先存在的“不可变”sequence 类的行为。
也就是说，只对那些你无权访问源代码或不希望修改源代码的 sequence 覆盖这些回调。不要覆盖 pre_do，mid_do 和/或 post_do 回调，来作为修改直接封闭的 sequence 类的 body 任务的行为的方法，而仅仅是作为修改你正在扩展的其他 sequence 类的行为的方法。 定义了一个或多个这些回调后，您需要使用 factory 覆盖来将原始 sequence 类替换为扩展 sequence 类。
例如：

```
// Original sequence class that we do not want to modify
class vip_seq extends uvm_sequence #(my_tx);
    `uvm_object_utils(vip_seq)
    function new (string name = "");
        super.new(name);
    endfunction
    task body;
```

```
            req = my_tx::type_id::create("req");
            start_item(req);
            if( !req.randomize() ) ...
            finish_item(req);
        ...
    // Sequence extended for a specific test
    class alt_seq extends vip_seq;
        `uvm_object_utils(alt_seq)
        ...
        int prev_addr = 0;
        function void mid_do(uvm_sequence_item this_item);
            my_tx tx;
            $cast(tx, this_item);
            tx.m_addr = prev_addr + $urandom_range(1, 7); // Overwrite the address field
        endfunction
        function void post_do(uvm_sequence_item this_item);
            my_tx tx;
            $cast(tx, this_item);
            prev_addr = tx.m_addr; // Store the address to constrain the next transaction
        endfunction
    endclass
    class my_test extends existing_test;
        `uvm_component_utils(my_test);
        ...
        function void start_of_simulation_phase(uvm_phase phase);
            // Factory override to replace the original sequence
            vip_seq::type_id::set_type_override( alt_seq::get_type() );
        endfunction
    endclass
```

□在 sequence 需要访问运行它的 sequencer 的情况下，使用宏 uvm_declare_p_sequencer 声明变量 p_sequencer。

使用 p_sequencer 变量访问运行 sequence 的 sequencer，有助于阐明 sequence 和 sequencer 之间的结构关系。 可以使用方法 uvm_sequence_item :: get_sequencer（）来返回 sequencer，但返回值的 base 类型为 uvm_sequencer_base。 宏 uvm_declare_p_sequence 允许您定义特定的 sequencer 类型。 不要使用内部变量 uvm_sequence_item :: m_sequencer。

例如：

```
    class my_sequence extends uvm_sequence #(my_tx);
        `uvm_object_utils(my_sequence)
        `uvm_declare_p_sequencer(the_sequencer_class_name)
        ...
        task pre_start;
            // Get the configuration object associated with the sequencer component
            // on which this sequence is currently running
            uvm_config_db #(my_config)::get(p_sequencer, "", "config", m_config);
```

```
        endtask
        task body;
            // Set the arbitration algorithm of the current sequencer
            p_sequencer.set_arbitration(SEQ_ARB_STRICT_RANDOM);
            begin
                sequence2 seq2;
                seq2 = sequence2::type_id::create("seq2");
                if ( !seq2.randomize() )
                    `uvm_error(get_type_name(), "Randomize failed")
                // Start a child sequence on the current sequencer
                seq2.start(p_sequencer, this);
            ...
        endtask
    endclass
```

□如果 sequence 需要访问自身运行的 sequencer 以外的 sequencer，请在 sequence 对象中添加成员变量，并在启动 sequence 之前分配该变量以引用到其他 sequencer。

这可能发生在虚拟 sequence 需要引用 sequencer 的位置以启动一个 sequence，或者 layering sequence 需要引用到 sequencer，才能从中获取 transaction。 在任何一种情况下，sequence 对象中的成员变量都应设置为在启动相关 sequence 之前引用另一个 sequencer。 在 virtual sequence 运行在空 sequencer 上的情况下，此方法无论如何都是必需的，因为如果 p_sequencer 变量的值为 null，则不能使用 p_sequencer 变量来获取对组件层次结构的访问。

例如：

```
    class my_sequence extends uvm_sequence #(my_tx);
        `uvm_object_utils(my_sequence)
        // Control knob idiom: a data member constrained when the sequence
        // is started
        rand int m_control_knob;
        function new (string name = "");
            super.new(name);
        endfunction
        task body;
            repeat (m_control_knob) // Number of transactions
            begin
                req = my_tx::type_id::create("req");
                start_item(req);
                if (!req.randomize())
                    `uvm_error(get_type_name(), "Randomize failed")
                finish_item(req);
            end
        endtask
    endclass
    class my_virtual_sequence extends uvm_sequence;
        `uvm_object_utils(my_virtual_sequence)
        my_sequencer m_child_sequencer; // Reference to child sequencer
```

```
    function new (string name = "");
        super.new(name);
    endfunction
    task body;
        // Start a non-virtual child sequence
        my_sequence seq;
        seq = my_sequence::type_id::create("seq");
        if ( !seq.randomize() with { m_control_knob < 8; } )
            `uvm_error(get_type_name(), "Randomize failed")
        seq.start(m_child_sequencer, this);
    endtask
endclass
class my_env extends uvm_env;
    ...
    function void run_phase(uvm_phase phase);
        // Create and start a virtual sequence
        my_virtual_sequence vseq;
        vseq = my_virtual_sequence::type_id::create("vseq");
        if (!vseq.randomize())
            `uvm_error(get_type_name(), "Randomize failed")
        // Set path within sequence object to sequencer for child sequence
        vseq.m_child_sequencer = m_agent.m_sequencer;
        phase.raise_objection(this, "Start of my_env");
        vseq.start(null, null);
        phase.drop_objection(this, "End of my_env");
    endfunction
    ...
endclass
```

Stimulus and Phasing

☐使用 virtual sequence 协调多个 agent 的行为。
不要过度约束 virtual sequence。 虚拟 sequence 应该用于协调多个并行 agent 的活动，但不要过度约束这些 agent 的活动。 对于约束随机验证，虚拟 sequence 不应被视为定向测试。避免使用生成非常特定场景的虚拟 sequence，除非绝对必要。 （当使用基于 graph 的激励方法时，可以自动生成顶级 sequence 以执行非常特定的场景，但这是另一回事。）
☐应在 null sequencer 上启动 virtual sequence
除非有特定原因来定义和实例化 sequencer，例如，访问存储在 sequencer 对象中的公共属性或访问 configuration 数据库。 （请注意，虚拟 sequence 不会与运行它的 sequencer 上的 sequence 队列有交互。）（参见示例。）
☐一个 top_level sequence 运行在每个 agent 上，在所有允许的子 sequence 中随机选择。
避免过度约束任何顶层 sequence。 对于约束随机验证，应以随机化基础。 在需要定向 sequence 的场景时，应将它们选为众多中的一个选择，而不是默认选择。 您可以添加特定

测试来约束顶层 sequence 以选择定向 sequence。 （当使用基于 graph 的激励方法时，可以自动生成顶层 sequence 以执行非常特定的场景，但这是另一回事。）

☐尽可能保持 sequence 通用。 避免编写定向 sequence，除非绝对必要。

基本 sequence 应该尽可能少地假设它们运行的上下文，以便它们可以生成任何场景。通常，避免使用旨在执行非常特定方案的 sequence。 此规则的一个例外是生成需要重现错误的特定条件的 sequence，在这种情况下，sequence 的名称和任何关联的覆盖点应该标志出 bug 名。

☐sequence 不应是 phase-aware 的，Sequence 应该在所有 run-time phase 中都能够被启动，这样有利于重用。

☐可以重写 run-time phase 的 reset_phase，configure_phase，main_phase，shutdown_phase 以生成激励，通常是通过启动 sequence，但绝不会在 driver，monitor，subscriber 或记分板等组件中生成。

driver，monitor，subscriber 或记分板等组件的运行不仅仅是在 run-phase 上。如果组件是 phase-aware 性质的话，就会不利于重用。可以使 run_phase 的行为依赖于某个变量，这样就能够让 run_phase 与其他 run-time phase 并行执行。但是，请覆盖 reset，configure，main，shutdown 等 phase。

☐在引入用户自定义的 run-time phase 时，phase 名不应该与预定义的 run-time 相重合。

比如某个 DUT 需要一个 training phase 和多个 main phase，那么合理使用规范命名更加有利于在多个环境中集成。

☐在集成多个环境时，如果每个 env 都覆盖了预定义或用户定义的 run-time phase，请注意通过引入 phase 域和跨域同步 phase 来正确排序 phase。

UVM 不会对每个内置 run-time phase 可以执行的操作施加任何明确的规则。在集成使用预定义或用户定义的 run-time phase 的组件时，可以将不同的组件放在不同的域中，并通过跨域同步 phase 明确定义不同域中的 phase 之间的关系。

Example

```
class   top_level_env extends   uvm_env;
    ...
    env m_env1;
    env m_env2;                     // Environments to be integrated
    function void build_phase(uvm_phase phase);
        uvm_domain domain1, domain2;
        m_env1 = env::type_id::create("m_env1", this);
        m_env2 = env::type_id::create("m_env2", this);
        domain1 = new("domain1");
        m_env1.set_domain(domain1);
        domain2 = new("domain2");
        m_env2.set_domain(domain2);          // Two new phase domains
                        // Synchronize specific run-time phases across domains
        domain1.sync(domain2, uvm_reset_phase::get(), uvm_configure_phase::get());
        ...
```

☐不要覆盖预定义的 pre-和 post-方法（例如 pre_reset_phase），而是保留这些 phase 以便在跨域同步 phase 时使用。

具有 pre_和 post_前缀的预定义 run-time phase 具有与其他预定义 phase 非常相似的非正式描述。为了避免对这些 phase 的含义做出不必要的假设，最好将自己限制在覆盖 reset，

configure，main 和 shutdown phase 以及在其他情况下添加用户定义的 phase。但是，在跨域排序预定义 phase 时，pre-和 post- phase 会产生非常有用的同步 hooks。甚至仅为此目的，定义用户定义 phase 的 pre-和 post- phase。

Example

```
domain1.sync(domain2, uvm_configure_phase::get(), uvm_post_configure_phase::get());
domain1.sync(domain2, my_post_training_phase::get(), uvm_pre_main_phase::get());
```

如果您选择执行 phase 跳转，则必须非常小心地在 phase 中止时正确清理。不要随便使用相位跳转，因为没有内置的安全措施。向后跳转应限制为跳转到其他 run-time phase。前向跳转应限制为跳转到 run-time phase 之后的常见 phase。

Objections

☐在任何 class 中都可以通过提起和撤销 objection 来控制验证平台的起始与结束。

通常，driver 从 sequencer 中获取 transaction 时都应该提起 objection，在处理完该 transaction 后撤销 objection。Monitor 在检测到新的 transaction 时应该提起 objection，在通过 analysis port 发生出该 transaction 后应该撤销 objection。当 Scoreboard 需要等待多个 item 时，应该提起 objection。

☐调用每个 objection（UVM 1.2 以后）的 set_propagate_mode（0）方法来禁用该 objection 的分层传播。

UVM 1.2 之前的行为（以及 UVM 1.2 中的默认值）是将每个 objection 传播到组件层次结构中，这会产生可测量的仿真速度损失但通常在功能上是多余的。

☐考虑在内部循环中提起和撤销 objection 的对仿真速度影响，例如：各个 transaction。如果仿真速度损失很大，则从内部循环中删除 objection。

在删除 objection 时，应该保证有其他的 objection 覆盖到此 objection，否则可能引起验证环境停止。

☐如果 sequence 要提出和撤销 objection，则应在其 pre_start 方法中调用 raise_objection，在其 post_start 方法中调用 drop_objection。

尽管可以在 sequence 的 body 任务中提出并删除 objection，但是将对 objection 限制在 pre / post_start 方法中更具有一致性。

☐在 sequence 中调用 raise_objection 与 drop_objection 方法时，将其置于 if (starting_phase ！= null)条件中。

在 uvm-1.2 之前，starting_phase 是类 uvm_sequence_base 的成员。从 uvm-1.2 开始，不推荐使用 starting_phase 变量，而必须使用 get_starting_phase 方法访问它。

```
task pre_start;
    uvm_phase starting_phase = get_starting_phase(); // uvm-1.2
    if (starting_phase != null)
        starting_phase.raise_objection(this, "Sequence started");
Endtask
```

☐在启动 sequence 之前，如果要提起 objection，则需要先设置 starting_phase 成员变量。从 uvm-1.2 开始，不推荐使用 starting_phase 变量，必须使用 set_starting_phase 方法设置：

```
task run_phase(uvm_phase phase);
    my_sequence seq;
    seq = my_sequence::type_id::create("seq");
```

```
        if ( !seq.randomize() )
            `uvm_error( ... )
        seq.set_starting_phase(phase); // uvm-1.2
        seq.start( ... );
    endtask
```

☐调用 raise_objection 或 drop_objection 时，总是传递一个字符串作为第二个参数来描述 objection 以帮助调试。

命令行参数+ UVM_OBJECTION_TRACE 打开 objection 跟踪，打印出每个被调用的 objection 的 description 参数。

☐如果调用 sequence 的 kill 方法并且 sequence 可以提出 objection，请确保覆盖 sequence 的 do_kill 方法以能够撤销 objection。

否则，objection 可能永远不会被撤销，这将阻止 phase 结束。如果 sequence 由于 phase 跳转而过早结束，则所有 objection 计数都会自动清除，因此不需要明确撤销 objection。在 phase 跳转时也不会自动调用 kill。例如：

```
    function void do_kill;
        if (starting_phase != null)
            starting_phase.drop_objection(this, "Sequence ended prematurely");
    endfunction
```

Components

☐通过扩展 uvm_component 的相应子类来创建用户定义的组件类，以实现需要的功能。

例如，monitor 应该扩展自 uvm_monitor，scoreboard 应该扩展自 uvm_scoreboard，等等。

☐在类中的第一行使用宏`uvm_component_utils 在 factory 中注册组件类。

在本指南中不建议使用字段宏，但如果使用了字段宏，则应在使用宏`uvm_component_utils_begin 声明任何的成员变量后立即注册 transaction 类。

☐在 factory 注册宏之后，使用规范命名方式的后缀声明 ports，exports 和 virtual interface。

☐声明了 ports，exports 和 virtual interface 后，再声明类的成员变量。

☐然后是构造函数 new，构造函数应包含 string name 和无默认值的 parent 参数，需要调用 super.new。

除了调用 super.new 之外，构造函数应该是空的，除非它需要实例化 covergroup 或初始化常量：

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

☐在 build_phase 中实例化组件类，而不应该在其他 phase 或者构造函数中实例化组件类。

☐总是使用 factory 实例化组件：

```
    var_name = component_type::type_id::create("var_name", this);
```

☐组件的 string 名称”var_name”应与变量名称 var_name 相同，除非有特定原因使字符串名称与变量名称不同，例如在使用相同变量的循环中创建多个组件对象时。

☐第二个参数 this 表示此组件的父类。

☐如果用户定义的组件类扩展了另一个用户定义的组件类，则应注意在适当的位置插入 super.<phase_name> _phase 形式的调用，即执行相应的基类 phase 方法。

如果用户定义的组件类是直接从 UVM 基类库进行扩展得到的，则内建 phase 方法不必调用 super.<phase_name> _phase，尽管这曾经在 OVM 中是一个推荐。

```
function void connect_phase(uvm_phase phase);
    super.connect_phase(phase); // Not necessary when extending uvm_component
...
Endfunction
```

□如果用户定义的组件类是直接从 UVM 基类库扩展得到的，在覆盖内建的 build_phase 方法时，请不要调用 super.build_phase。

相反，如果您确实需要调用 super.build_phase，则必须需要理解 uvm_component :: build_phase 方法调用了 apply_config_settings，它将在组件的字段名称和层次名称恰好与配置数据库中的名称和范围匹配的情况下，使用字段宏注册的字段的值设置为从配置数据库获取的相应值。

一个标准的写法如下：

```
class my_component extends uvm_env;
    `uvm_component_utils(my_component)
    // Transaction-level ports and exports
    uvm_analysis_port #(my_tx) a_port;
    // Virtual interfaces
    virtual dut_if vif;
    // Internal data members (variables)
    my_agent m_agent;
    // Constructor
    function new (string name, uvm_component parent);
        super.new(name, parent);
    endfunction
    // Standard phase methods
    function void build_phase(uvm_phase phase);
        a_port = new("a_port", this);
        m_agent = my_agent::type_id::create("m_agent", this);
    endfunction
    function void connect_phase(uvm_phase phase);
        ...
    endfunction
    task run_phase(uvm_phase phase);
        ...
    endfunction
endclass
```

Connection to the DUT

□每个 DUT 接口使用一个 SystemVerilog 接口实例。

例如，DUT 接口可以是总线接口，网络接口或串行接口。 SystemVerilog 接口用于在 UVM 验证环境和 DUT 之间传递信息。

☐在 UVM 验证环境使用虚拟接口访问 SystemVerilog 接口实例。

在基于类的验证环境中，UVM 验证环境内的 agent 通过引用基于 module 的实际接口实例的虚拟接口来监测和驱动连接到 DUT 的 SystemVerilog 接口上的线网和变量。作为一种替代的高级编码技术，如果您大量使用参数化接口，您可能需要通过从验证环境调用抽象基类中的方法来克服 SystemVerilog 中参数化虚拟接口的缺点。（请参阅如何从 UVM 访问参数化 SystemVerilog 接口。）

☐在配置数据库中的 configuration object 配置对象里将虚拟接口封装。

任何引用 SystemVerilog 接口实例的虚拟接口都应封装在配置对象中，并且该对象应设置到配置数据库中。对 uvm_config_db :: set 的调用应该来自可以访问相应接口实例的 SystemVerilog 模块的范围（可以使用层次名称）。 （参见配置。）

☐将顶层配置对象中的虚拟接口复制到顶层 env 的 build_phase 方法中的 agent 或较低级别 env 相关联的配置对象。

虚拟接口应从配置对象自上而下复制到配置对象，从 module 创建的顶层配置对象开始，并结束于与 agent 关联的配置对象。可能存在与嵌套 env 相关联的一个或多个中间配置对象。虚拟接口应该从顶层 env 中的顶层配置对象中提取，而不是在 testcase 中提取，这样 env 可以独立运行一个空的或普通的测试类。

```
class top_env extends uvm_env;
    ...
    top_config m_config;
    bus_config m_bus_config;
    function void build_phase(uvm_phase phase);
        if (!uvm_config_db #(top_config)::get(this, "", "config", m_config))
        `uvm_error(get_type_name(), "Unable to get top_config")
        m_bus_config = new("m_bus_config");
        m_bus_config.vif = m_config.bus_vif;
        ...
        uvm_config_db #(bus_config)::set(this, "m_bus_env", "config", m_bus_config);
    endfunction
    ...
endclass
```

☐agent 应检查其虚拟接口是否已被 set。

agent 应从其配置对象获取虚拟接口，并在其 driver 和 monitor 中分配虚拟接口变量。如果虚拟接口为空，则 agent 应报告 fatal_error，因为仿真没有必要继续。

```
class bus_agent extends uvm_agent;
    ...
    bus_config m_config;
    function void bus_agent::build_phase(uvm_phase phase);
        if (!uvm_config_db #(bus_config)::get(this, "", "config", m_config))
            `uvm_error(get_type_name(), "bus config not found"
        if (m_config.vif == null)
            `uvm_fatal(get_type_name(), "bus virtual interface not set")
        ...
    endfunction
```

endclass

TLM Connections

☐在 connect_phase 中建立 TLM port/export 连接，分配虚拟接口。
☐使用 port 和 export 在 UVM 组件之间进行通信，包括 analysis ports 和相应的 exports。
通常，首选 ports 和 exports 用于组件之间的通信，而不是使用点对点共享对象或其他 UVM 通信机制。在需要点对点共享对象进行同步的情况下，使用 uvm_event 或 uvm_barrier 来同步组件。 （请参阅时钟，时序，同步和接口）。
☐当 UVM 组件之间需要实现一对多的连接时，使用 analysis ports 和 analysis exports（或者是 uvm_subscriber 的对象）。
在许多情况下，analysis ports 和 analysis exports 优于常规的 ports 和 export，因为 analysis ports 支持向多个组件（所谓的 uvm_subscriber）广播 transaction，并允许 ports 可以是未连接状态。当然，使用常规 ports 和 export 实施一对一连接，有时也可能是您想要的。analysis ports 通常是将 transaction 传递出 agent，并将 transaction 传递到 scoreboard，checker 和覆盖率收集器组件或在组件之间传递的最佳选择。
☐在组件之间建立对等（peer-to-peer）连接时，将 ports（或 analysis ports）直接连接到 exports（或 analysis exports），而无需任何中间 FIFO。
ports 和 export 直接连接应被视为 UVM 中的标准，只在需要时才插入 FIFO。当需要 FIFO 时，它们应该插入组件内部而不是组件之间。
☐与 agent 进行通信的两种方式：将 agent 的 analysis ports 连接到 uvm_subscriber，或从外部的使用直接对象引用（direct object reference）来访问 agent 中的 sequencer。
由于 agent 具有已知的内部结构，因此允许使用层次对象引用直接从外部访问内部对象。建议将 monitor 的 analysis ports 连接到 agent 的 analysis ports，当然也可以直接从外部访问 monitor 的 analysis ports。

Configurations

本节中描述的 UVM 配置数据库与 UVM 的 factory 是相互独立的，后者将在下一节中介绍。
☐使用配置数据库 uvm_config_db 而不是资源数据库 uvm_resource_db。
使用 uvm_resource_db 的唯一情况是，在多次设置相同 item（即同名和相同范围）时，它们具有不同的规则。但是，不必学习两组规则，单独使用 uvm_config_db 就可以完成所需的一切。
☐将给定组件的配置参数分组到配置对象中，并将该配置对象设置到配置数据库中。
配置参数可以单独存储在配置数据库中（通过分别调用 set 和 get），或者可以在配置对象内分组。通常，最好使用配置对象，因为这样可以提供您希望找到配置参数的位置，并允许通过一次调用就能随机化所有配置参数。配置数据库实际上包含的是配置对象的引用，而不是对象本身。
☐顶层配置对象应包含对所有低层配置对象的引用。
顶层配置对象应该实例化自顶层模块，而低级配置对象应该从顶层配置对象的构造函数（new）中实例化。较低级别的配置对象通常与 agent 关联。此方法的好处是无需在顶层配

置对象和较低级别配置对象之间复制或复制信息，顶层配置对象可以包含对较低级别配置对象中变量的约束。可以从顶层模块或 test 类以及顶层 env 类对较低级别配置对象的变量赋值（见示例）。

☐通过扩展类 uvm_object 来创建用户定义的配置类。

扩展 uvm_object 为配置对象提供层次名称，并启用 UVM 种子以获得更好的随机稳定性。虽然配置对象不是 transaction 和组件，但随机化仍然有用。

Example

```
class my_agent_config extends uvm_object;
    virtual my_if vif;
    uvm_active_passive_enum is_active;
    bit coverage_enable; // From the UVM User Guide
    bit checks_enable;
    function new(string name = "");
        super.new(name);
    endfunction
endclass
```

☐将类名<component_class> _config 或<sequence_class> _config 分别用于与组件或 sequence 关联的配置类，其中<component_class>是组件的类名，<sequence_class>是 sequence 的类名。

☐在配置数据库中的配置对象使用字段名称“config”。

指向到从配置数据库中检索出的配置对象的变量名称应始终为 m_config。（参见命名约定。）

☐不要使用 factory 注册用户定义的配置类。

因此，配置类可以具有带有任意数量的用户自定义参数的构造函数。将配置对象视为一组参数值，而不是激励。

☐组件在 build_phase 中 get 和 set 配置参数（通常是配置的对象）。

这样，配置参数能够在 child 组件可见。组件应从其自己的配置对象中提取参数，然后为需要其自己的配置对象的任何 children 构造配置对象，然后将这些配置对象设置到配置数据库中，所有这些都在 build_phase 完成。在特殊情况下，可能会出现在后续 phase 从配置数据库中检索配置参数的情况。 UVM 有一个 wait_modified 方法，可以在 run_phase 调用，以便在设置配置参数时唤醒进程，但必须理解，就 CPU 仿真时间而言，设置和获取配置参数是一项相对昂贵的操作。

例：

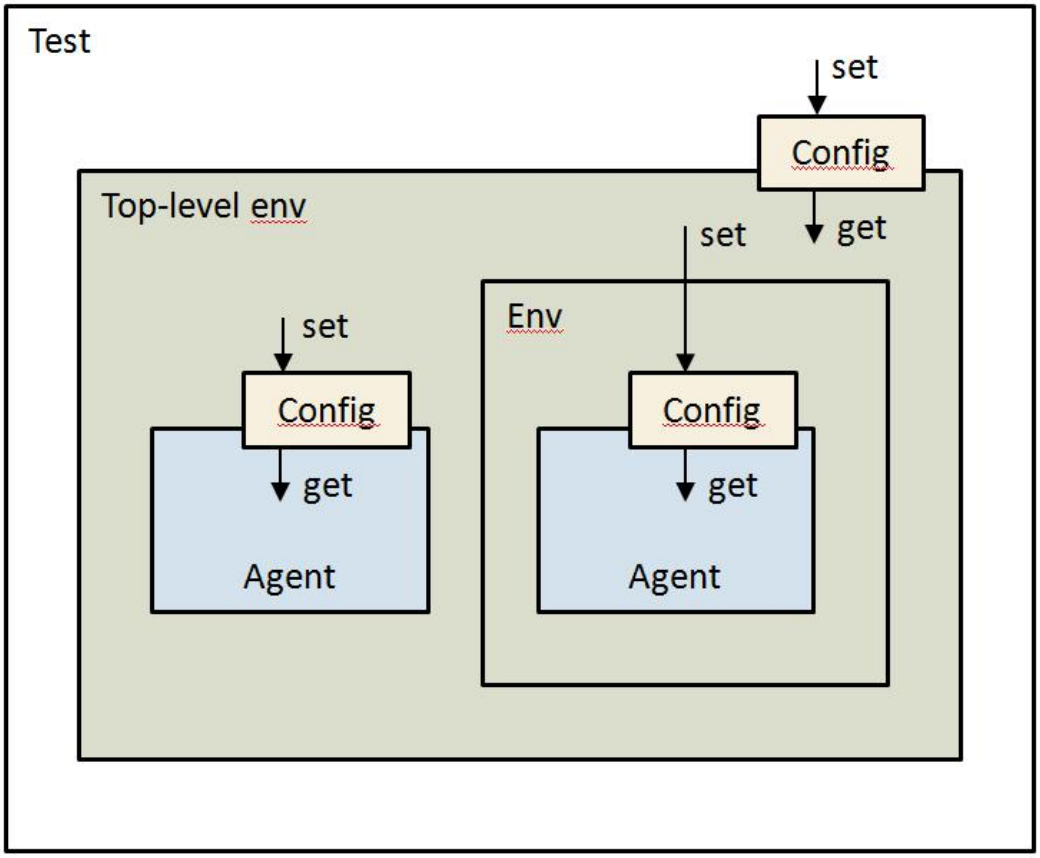
```
class my_agent_config extends uvm_object;
    virtual my_agent_if vif;
    uvm_active_passive_enum is_active = UVM_ACTIVE;
    bit coverage_enable;
    bit checks_enable;
    function new(string name = "");
        super.new(name);
    endfunction
    endclass
class top_config extends uvm_object;
    rand my_agent_config m_my_agent_config;
    function new(string name = "");
        super.new(name);
```

```
        m_my_agent_config = new("m_my_agent_config");
        m_my_agent_config.is_active = UVM_ACTIVE;
        m_my_agent_config.checks_enable = 1;
        m_my_agent_config.coverage_enable = 1;
    endfunction : new
endclass : top_config
module top_tb;
    ...
    top_config top_env_config;
    Initial begin
        top_env_config = new("top_env_config");
        if ( !top_env_config.randomize() )
            `uvm_error("top_tb", "Failed to randomize top-level configuration object" )
        top_env_config.m_my_agent_config.vif = th.my_agent_if_0;
        uvm_config_db#(top_config)::set(null,"uvm_test_top", "config",top_env_config);
        uvm_config_db#(top_config)::set(null,"uvm_test_top.m_env","config",top_env_c
    onfig);
        run_test();
    end
Endmodule
class top_env extends uvm_env;
    `uvm_component_utils(top_env)
    my_agent_config m_my_agent_config;
    my_agent_agent m_my_agent_agent;
    my_agent_coverage m_my_agent_coverage;
    top_config m_config;
    ...
    function void build_phase(uvm_phase phase);
        if (!uvm_config_db #(top_config)::get(this, "", "config", m_config))
        `uvm_error(get_type_name(), "Unable to get top_config")
        m_my_agent_config = m_config.m_my_agent_config;
        uvm_config_db  #(my_agent_config)::set(this,  "m_my_agent_agent",  "config",
    m_my_agent_config);
        if (m_my_agent_config.is_active == UVM_ACTIVE )
            uvm_config_db                                  #(my_agent_config)::set(this,
        "m_my_agent_agent.m_sequencer", "config", m_my_agent_config);
        uvm_config_db  #(my_agent_config)::set(this,  "m_my_agent_coverage",  "config",
    m_my_agent_config);
        m_my_agent_agent=my_agent_agent::type_id::create("m_my_agent_agent",this)
    ;
        m_my_agent_coverage=my_agent_coverage::type_id::create("m_my_agent_cover
    age",this);
    endfunction : build_phase
    ...
```

☐始终检查 uvm_config_db#(T):: get 的返回值（bit），以确保配置数据库中存在这个配置参数。此检查还可以帮助捕获配置参数名称的拼写错误。
☐如果 uvm_config_db#(T):: get 返回 0（即 get 失败），则应选择合理的默认值。
在未配置其配置参数的情况下，验证组件应具有合理的默认行为。通过不在配置数据库中设置配置对象或者不在配置对象中设置参数值（假设可以检测到此参数），可以取消设置配置参数。在任何一种情况下，组件都应检测到配置参数未被明确设置，并应选择适当的默认值。
☐每个组件都应该是仅仅 get 自己实例对应的配置对象，而不应该 get 到其他组件实例的配置对象。
也就是说，应该如下方式使用 get：

```
uvm_config_db#(T)::get(this, "", ...);
```

尽管一个组件不应当 get 到其他不相关组件的配置对象，但是一些联系比较紧密的组件也是允许共享同一个配置对象的。比如，driver，monitor 和 coverage collector 可以 get 到他们所在的 agent 的配置对象。当然，最好是多次使用 uvm_config_db#(T)::set.
☐组件实例相关联的配置对象应由其 parent 组件或该组件实例的其他直接祖先进行 set，而不应由任何其他组件实例设置。



图：使用配置对象

☐避免使用绝对层次路径名作为 uvm_config_db#T(T):: set 的第二个参数，并提供尽可能短的唯一路径名。
因为我们有一个与组件层次结构平行的配置对象层次结构，所以组件通常会 set 其直接子项的配置对象，并且当特定组件没有配置对象时，只需要深入到组件层次结构中即可。通常，如果确实需要从 test 或 env 深入到组件层次结构，请在路径名称的开头使用通配符，并提供

尽可能短的唯一名称。
□组件实例可以与一个配置对象关联，也可以不与配置对象关联，并且多个组件实例可以与同一配置对象关联。
如果在组件层次结构中的该位置没有要传递的配置参数，则组件没有必要 get 或 set 配置对象。
□对于 agent，其配置对象的 is_active 变量决定这个 agent 是 active 或者 passive 的。覆盖 virtual get_is_active 方法以返回此值。在 agent 中创建和连接 sequencer 和 driver 之前，请检查 get_is_active。
UVM 标准没有公开类 uvm_agent 的 is_active 成员，而是提供虚拟方法 get_is_active 来获取值，通过覆盖方法 build_phase，在 build_phase 中根据配置数据库中字段“is_active”的值设置 is_active 的值。虽然我们建议您应该在配置对象中封装 active 和 passive flag 并覆盖 get_is_active 以返回此标志，但我们还建议检查配置数据库中的“is_active”字段（如果存在），并对配置 objection 中 is_active 字段的值与配置数据库中的“is_active”字段之间的任何不一致给出警告。
例：

```
class my_agent extends uvm_agent;
    `uvm_component_utils(my_agent)
    uvm_analysis_port#(my_transaction) a_port;
    my_config       m_config;
    my_sequencer        m_sequencer;
    my_driver       m_driver;
    my_monitor    m_monitor;
    function new(string name, uvm_component parent);
        super.new(name, parent);
    endfunction
    function void build_phase(uvm_phase phase);
        if (!uvm_config_db #(my_config)::get(this, "", "config", m_config))
            `uvm_error(get_type_name(), "Agent config object is missing from
        config_db")
        if (get_is_active() == UVM_ACTIVE) begin
            m_sequencer = my_sequencer::type_id::create("m_sequencer", this);
            m_driver = my_driver ::type_id::create("m_driver", this);
        end
        m_monitor = my_monitor::type_id::create("m_monitor", this);
        a_port = new("a_port", this);
    endfunction
    function void connect_phase(uvm_phase phase);
        if (get_is_active() == UVM_ACTIVE)
            m_driver.seq_item_port.connect( m_sequencer.seq_item_export );
        m_monitor.a_port.connect( a_port );
    endfunction
    virtual function uvm_active_passive_enum get_is_active();
        return uvm_active_passive_enum'( m_config.is_active );
    endfunction
```

```
/*
// Alternative version that includes defensive programming to check for conflicts
// between the config object and the "is_active" field
local int m_is_active = -1;
virtual function uvm_active_passive_enum get_is_active();
    if (m_is_active == -1)begin
        if(uvm_config_db#(uvm_bitstream_t)::get(this,"","is_active",m_is_active))be
    gin
        if (m_is_active != m_config.is_active)
            `uvm_warning(get_type_name(), "is_active field in config_db conflicts
        with config object")
        end
        else
            m_is_active = m_config.is_active;
    end
    return uvm_active_passive_enum'(m_is_active);
endfunction
*/
endclass: my_agent
```

☐如果要对 sequence 进行参数化，则应将 sequence 的参数放入配置数据库中的配置对象中。配置对象应与 sequence 运行的 sequencer 相关联。

配置数据库中可能有多个配置对象与特定的 sequencer 相关联，但具有将在该 sequencer 上运行的不同 sequence。sequence 而不是 sequencer 组件本身将从配置数据库中获取这些对象。尽管 sequence 可以从配置数据库获取配置对象而不参考 sequencer 组件的层次路径，但这样做会使单独参数化该 sequence 的多个实例变得更加困难，这些实例可能在不同的 sequencer 运行或者没有 sequencer。换句话说，虽然 sequence 没有义务通过其 sequencer 获取其配置对象，但我们建议它这样做。（见例子）

☐与 sequence 关联的配置对象都应该从 sequence 的 start 中从配置数据库中获取，并且 sequence 对象中的变量指向该配置对象。

可以在 sequence 启动之前或 sequence 本身设置引用配置对象的变量。出于性能原因，最好在启动 sequence 时获取配置对象一次，不要在内部循环中。另一方面，为了支持后期随机化，sequence 应该在设置内联约束时使用最新的状态信息，因此有时适合从 sequence 的 body 任务中的配置对象获取信息，假设配置对象指向在 run-time phase 正在更新的信息。（见例子。）

☐如果一个组件在其子组件中对自己的变量（包括虚拟接口）进行连接，那么应该在 build_phase 中创建子组件并完成连接操作。

一个组件可能会通过在其子组件中对自己的变量进行连接以传递给子组件信息（尽管这样会损失一些灵活性），而不是使用配置数据库的方式。例如，agent 在其 driver 和 monitor 设置 virtual interface。必须记住，虽然 build_phase 方法是自上而下调用的，但 connect_phase 方法相对于组件层次结构被称为自底向上，因此 connect_phase 不能用于在层次结构中传播变量值（因为调用子进程的 connect_phase 在其父级调用 connect_phases 之前）。

The Factory

☐总是使用 factory 实例化 transaction，sequence，以及组件的对象。
不要简单使用 new 函数来实例化对象，而是应该使用如下方式：
实例化 transaction，sequence 等 object 对象：

```
var_name = type_name::type_id::create("var_name");
```

实例化组件的对象：

```
var_name = type_name::type_id::create("var_name", this);
```

这里的 var_name 是对象名称， type_name 是类名。以这种方式一致地使用 UVM factory 是能够在 UVM 中利用 OOP 的关键之一，因为它允许在运行时而不是在编译时确定对象的类型，因此可以在不需要修改源代码的情况下进行 override（调用 create）。
☐使用 factory 将 transaction，sequence，以及组件对象替换为其类的扩展类的另一个对象时，factory 覆盖应采用以下形式之一：

```
old_type_name::type_id::set_type_override(new_type_name::get_type());
old_type_name::type_id::set_inst_override(new_type_name::get_type() ... );
```

这与直接调用类 uvm_factory 的方法形成对比，只是为了保持一致性，并不鼓励这样做。
☐当您需要访问 factory 时，请调用静态方法 uvm_factory :: get( )。
不要使用全局变量 factory 来访问某个 factory（全局变量 factory 在 uvm_1.2 中被丢弃）。一个调用 factory 中的 print 方法例子：

```
uvm_factory factory = uvm_factory::get();   // 利用 get 函数得到
factory.print();
```

Tests

☐不要直接从测试中生成激励，而是使用测试来设置配置参数和 factory 覆盖。
通常最好从 env 而不是从测试中启动 sequence，并将测试限制为参数化或重配置环境。环境应该知道如何激励 DUT(见 Stimulus.)。
☐设置验证环境的默认配置，并在 env 类中生成默认激励，而不是测试类，以便即使使用空测试也能运行 env。
env 应该独立运行，并且应该使用有效激励来运行 DUT，即使是空的测试。在经典受约束随机验证适用的应用中，可以通过使用的不同种子重复测试以增加覆盖率。 （参见覆盖率驱动的验证方法。）
☐在适当的情况下，使用 text_base 类来定义一组测试中常见的结构和行为，并通过扩展这些 base 类来创建各个测试。
将公共代码移动到 base 类中是良好的面向对象编程实践的一个例子，但重要的是规划重用的原则。请记住每个标准 phase 方法都应该先调用 super.<phase_name>_phase 方法（请参阅组件）。
☐为了重复使用，请避免根据验证环境的具体细节进行测试。
例如，避免引入深入验证环境的层次对象引用。在引用 env 中的组件时，使用 uvm_top.find（“*.path”）来定位验证环境中的组件，而不是使用完整的“硬连线”对象引用(即层次化引用)，但请注意，find 方法在 CPU 时间方面可能很昂贵(所以不要从内部循环调用它)。

☐使用命令行参数修改测试行为，无需重新编译。
例如，使用+ UVM_TESTNAME 命令行参数选择 testcase，并通过调用不带参数的 run_test 来启动测试：uvm_top.run_test();您还可以使用命令行参数设置 factory 覆盖，设置配置数据库中的值以及设置打印阈值级别，所有这些都无需重新编译 SystemVerilog 源代码。

Messaging

☐要报告消息，请始终使用八个标准报告宏之一`uvm_info，`uvm_info_context 等，而不是 $display 及类似语句。
打印消息需要消耗一定的计算机仿真时间，因此，使用 UVM 的报告功能（例如消息详细程度）来控制生成的消息数非常重要。
☐将消息 ID 设置为静态 string 或 get_type_name( )。
消息 id 是报告宏的第一个参数，例如`uvm_info，`uvm_warning 等等。如果该字符串与当前 VIP 中的所有报告相同，则将其设置为静态字符串，从而有助于将报告标识为源自该 VIP。否则将其设置为 get_type_name（），它返回表示当前类类型的字符串。请注意，get_type_name( )在 VIP 的原始开发和调试期间可能很有用，但在要广泛应用的 VIP 中可能不太合适。如果没有其他明显更优的选择，get_type_name( )在任何情况下都是一个很好的 fallback。
☐在整个代码中仔细，有条理地和一致地设置消息详细级别，以避免日志文件中不必要的数据，并区分在验证环境开发调试期间使用的消息和在运行 testcase 时使用的消息。
☐默认情况下，将各个报告宏的详细级别设置为较大的数字，以便不太可能报告 message。
建议使用以下详细级别：

UVM_HIGH 和 UVM_DEBUG 用于调试验证环境或测试本身的消息，通常在验证环境或测试能够成功到达运行阶段的开始时应被抑制。

UVM_MEDIUM - 与 sequence 执行或 DUT 引脚级行为相关的消息，通常在 run phase 生成，用于帮助诊断 DUT 中的错误。

UVM_NONE - 消息不会被消息详细程度机制抑制，因此仅用于不经常用，或非常重要的信息，例如测试执行期间的主要阶段或状态更改

使用命令行参数+ UVM_VERBOSITY 从命令行设置特定 testcase 的消息详细级别非常有用。
☐仔细设置消息严重性级别，以区分纯信息性消息，可能代表错误的消息和肯定是错误的消息。
建议使用以下严重性级别：

UVM_INFO - 消息仅供参考，并不表示错误。

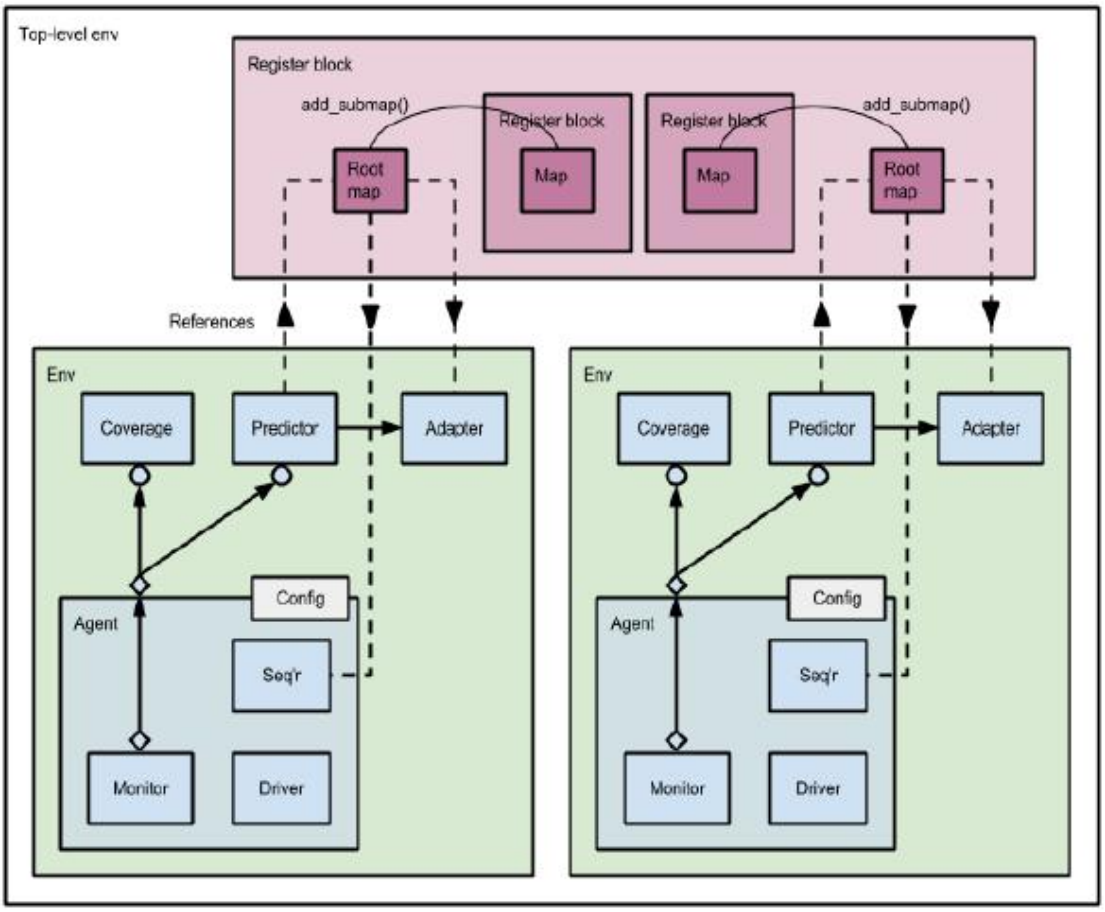
UVM_WARNING - 该消息指示需要进一步调查的潜在错误

UVM_ERROR - 该消息指示实际错误，但允许进行继续仿真。通过允许仿真持续一段时间，可以在有可能收集有关错误来源的更多信息时使用。默认情况下，UVM_ERROR 与退出计数关联，退出计数在达到最大错误计数时中止。

UVM_FATAL - 该消息指示严重性错误，不允许继续进行仿真。

Register Layer

☐如果使用生成器为寄存器模型创建 SystemVerilog 代码，请不要修改生成的代码。
您应该使用生成器来创建寄存器模型，因为这样做比手动输入寄存器模型更有效率。但是您需要注意，如果重新生成寄存器模型，您对生成的文件所做的任何编辑都将丢失，因此您应该避免进行任何此类编辑。
☐顶层 UVM 环境应使用 factory 实例化寄存器块，并应调用寄存器模型的 build 方法。
factor 方法调用 type_id :: create，并且应该在环境的 build_phase 中调用 build。
☐对于子环境使用寄存器模型的层次结构 UVM 环境，应该有一个顶层寄存器块，用于实例化与子环境关联的寄存器块，依此类推。



图：连接寄存器层
☐任何使用寄存器模型的 UVM env 都应该有一个名为 regmodel 的变量，该变量存储对该特定环境的寄存器块的引用。
☐具有寄存器模型的 UVM env 应设置所有子组件的 regmodel 变量，如果子组件也使用了寄存器模型到其寄存器块的相应子块。
子组件的 regmodel 变量应在 env 的 build_phase 中被设置。
☐如果 env 的 regmodel 变量的值为 null，则 UVM 环境应仅实例化一个寄存器块。
对于顶层环境，regmodel 的值将为 null，因此 env 应实例化寄存器块并设置 regmodel 的值。对于较低层次的 env，regmodel 的值不应为 null，因为它应该由更高层次的 env 设置。此机制允许将相同的 env 实例化为顶层 env（具有寄存器模型）或作为较低层次的 env（没有自己的寄存器模型）。例：

```
top_reg_block regmodel;
function void top_env::build_phase(uvm_phase phase);
    if (regmodel == null)begin
        // Instantiate register model for top-level env
        regmodel = top_reg_block::type_id::create("regmodel");
        regmodel = build();
    end
    // Set regmodel variable of lower-level env through config object
    m_bus_env_cfg = new("m_bus_env_cfg");
    m_bus_env_cfg.regmodel = regmodel.bus;
    ...
endfunction
```

□寄存器模型中每个子寄存器块的变量名称和 UVM 实例名称应与相应 agent 的名称相对应。例如，对于名为 bus 的 agent，与实例化该 agent 的 env（下面的 bus_reg_block）关联的寄存器块的实例应该在寄存器模型中具有变量名称 bus，并且应该具有 UVM 实例名称“bus”。顶层寄存器块（下面的 top_reg_block）需要有一个或多个地址映射（下面的 bus_map），它包含子寄存器块的地址映射（下面的 bus.bus_map）

```
// Top-level register block
class top_reg_block extends uvm_reg_block;
    `uvm_object_utils(top_reg_block)
    bus_reg_block bus; // Child register block for an agent named bus
    uvm_reg_map bus_map;
    function new(string name = "");
        super.new(name, UVM_NO_COVERAGE);
    endfunction
    virtual function void build;
    bus = bus_reg_block::type_id::create("bus");
    bus.configure(this);
    bus.build();
    bus_map = create_map("bus_map", 'h0, 1, UVM_LITTLE_ENDIAN);
    default_map = bus_map;
    bus_map.add_submap(bus.bus_map, 'h0);
    lock_model();
    endfunction
    ...
```

□寄存器块应仅模拟 DUT 寄存器，这些寄存器可由与 UVM 环境相关联的 UVM sequence 访问。

寄存器模型应该被构造为一组分层嵌套的寄存器块，这些寄存器块反映了 DUT 的层次结构，使得各个寄存器块可以在模块，子系统或全芯片级重用。（见图。）

□使用寄存器模型并实例化了 agent 的 UVM 环境应实例化并连接寄存器 adapter 和该 agent 的寄存器 predictor。

寄存器适配器和预测器应该在 build_phase 中实例化，并且应该在 env 的 connect_phase 方法中连接。适配器应连接到 sequencer，预测器应连接到 agent 的 monitor。（见图。）

□寄存器模型应使用显式预测，以使其镜像值与 DUT 中的寄存器值保持同步。

显式预测要求您将 agent 中 monitor 的 analysis port 连接到 UVM 寄存器预测器，从而允许每次在 DUT 接口上存在相关活动时更新寄存器模型中的镜像值，无论该活动是否由寄存器层启动的。预测器组件应在与 agent 相同的 env 中实例化，并且与预测器关联的地址映射应通过调用参数值为 0 的 set_auto_predict 方法关闭自动预测。

☐应分配每个子寄存器块中预测器的地址映射变量.map，以引用顶层寄存器块的相应地址映射。

这确保了使用系统地址映射中的全局地址而不是本地地址映射中的偏移来访问寄存器。例：

```
// To connect the register layer to an agent named bus
bus_agent m_bus_agent;
bus_reg_block regmodel;
reg2bus_adapter m_reg2bus;
uvm_reg_predictor #(bus_tx) m_bus2reg_predictor;
function void bus_env::build_phase(uvm_phase phase);
    ...
    m_bus_agent = bus_agent ::type_id::create("m_bus_agent", this);
    m_reg2bus = reg2bus_adapter::type_id::create("m_reg2bus", this);
    m_bus2reg_predictor =
    uvm_reg_predictor #(bus_tx)::type_id::create("m_bus2reg_predictor", this);
endfunction
function void top_env::connect_phase(uvm_phase phase);
    if (regmodel.get_parent() == null)
        regmodel.default_map.set_sequence(m_bus_agent.m_sequencer, m_reg2bus);
    m_bus2reg_predictor.map = regmodel.bus_map;
    m_bus2reg_predictor.adapter = m_reg2bus;
    regmodel.bus_map.set_auto_predict(0);
    m_bus_agent.m_monitor.ap.connect( m_bus2reg_predictor.bus_in );
Endfunction
```

在验证环境的开发过程中，为了调试，打印寄存器模型中的寄存器的详细信息可能会有所帮助。这应该在 end_of_elaboration_phase 方法中完成。例：

```
function void end_of_elaboration_phase(uvm_phase phase);
    uvm_reg regs[$];
    string name;
    regmodel.bus_map.get_registers(regs);
    `uvm_info(get_type_name(),
    $sformatf("Found %d registers", regs.size()), UVM_MEDIUM)
    for (int j = 0; j < regs.size(); j++)
        `uvm_info(get_type_name(),$sformatf("regs[%0d]:%s",j,regs[j].get_name()),UVM_
    HIGH)
Endfunction
```

☐在寄存器模型中读取或写入寄存器的寄存器 sequence 应扩展自 uvm_sequence，并且应具有名为 regmodel 的变量，该变量存储着对相应寄存器块的指针。例：

```
class my_reg_sequence extends uvm_sequence;
    `uvm_object_utils(my_reg_sequence)
    bus_reg_block regmodel;
```

```
    task body;
        uvm_reg_data_t data;
        uvm_status_e status;
        regmodel.reg0.write(status, .value('hab), .parent(this));
        assert (status == UVM_IS_OK);
        regmodel.reg0.read(status, .value(data), .parent(this));
        assert (status == UVM_IS_OK);
        assert (data === 'hab);
    endtask
endclass
```

☐在启动读取或写入寄存器的 sequence 之前，请先设置该 sequence 的 regmodel 变量。

例：

```
// Starting a register sequence
my_reg_sequence vseq;
vseq = my_reg_sequence::type_id::create("vseq");
vseq.randomize();
vseq.regmodel = regmodel;
vseq.set_starting_phase(phase);
vseq.start(null);
```

Functional Coverage

有关 UVM 中覆盖率驱动验证的一般性讨论，请参阅覆盖率驱动验证方法学。

☐使用 SystemVerilog covergroup 结构在 UVM 验证环境中收集功能覆盖率。

处理或转换来自 DUT 的值以创建为实际采样覆盖点的派生值，有时这是必要的或更方便的。例如，您可以计算在总线上连续出现的两个地址之间的差异，并将结果值用作覆盖点。该技术可以克服在覆盖组实例化时就进行了定义但覆盖点的定义不能动态地改变的基本限制。例：

```
class my_agent_coverage extends uvm_subscriber #(bus_tx);
    `uvm_component_utils(my_agent_coverage)
    bus_tx m_item;
    int m_address_delta;
    covergroup m_cov;
        cp_address_delta: coverpoint m_address_delta {
            bins zero = {0};
            bins one = {1};
            bins negative = { [-128:-1] };
            bins positive = { [1: 127] };
            option.at_least = 16;
        }
    endgroup
    function new(string name, uvm_component parent);
        super.new(name, parent);
        m_cov = new;
```

```
        endfunction : new
        function void write(input bus_tx t);
            m_item = t;
            m_address_delta = m_item.current_address - m_item.previous_address;
            m_cov.sample();
        endfunction : write
    endclass : my_agent_coverage
```

☐在适当的情况下，使用 cover property 在 interface 中收集功能覆盖信息。

使用 cover property 语句的基于 property 的 coverage 是收集接口协议中时间 sequence 的功能覆盖信息的好方法（与使用 covergroup 语句的基于样本的覆盖相反），但请注意，cover property 语句不能在类中使用。 （参见并发断言）

☐将 covergroup 作为嵌入式 covergroup 放置在类中，或者将 covergroup 放在 package 中并参数化 covergroup，以便可以从该 package 中的类进行实例化。

嵌入式 covergroup 是在类中使用 covergroup 的最直接的方法，但是几个类可以通过将 covergroup 声明放在任何类之外的 package 中，并在类中用适当的参数实例化 covergroup 来重用相同的 covergroup。

☐Covergroups 应该在 UVM 组件类中实例化，而不是在 sequence 或 transaction 中。

coverage 应该从在整个仿真过程中持续存在的准静态对象中收集，而不是从随时间动态变化的对象中收集。

☐Covergroups 应在 UVM subscribers 或 scoreboards 中实例化，并且 scoreboard 是在 UVM env 类中实例化并连接到 monitor/agent 的 analysis port。

使用 monitor 以通过其 analysis port 发送的 transaction 形式收集和组合信息，但不要将 covergroup 放在 monitor 中。monitor 的数据收集和 scoreboards 的数据分析之间的这种分离对于重用是重要的。

☐在 coverage 收集器类的构造函数中实例化 covergroup。

SystemVerilog 规定必须在构造函数中实例化嵌入的 covergroup。这违反了 UVM 中保持构造函数为空并从 build_phase 方法创建子对象的一般规则。

☐为了收集 DUT 内部信号的功能覆盖信息，在单个 SystemVerilog module（或接口）中封装对 DUT 层次路径的引用，然后使用虚拟接口和 interface 从 UVM 环境访问该模块。

可以使用层次路径或使用 bind 语句访问 DUT 中的内部信号。在单个模块（或接口）中封装所有层次路径可以使得验证环境保持干净。

☐如果 coverage 收集跨越多个 DUT 接口，因此需要从多个 agent 接收 analysis transaction，请使用`uvm_analysis_imp_decl 宏在 coverage collector 类中提供多个 analysis exports。

uvm_subscriber 类仅仅只有一个 analysis export，`uvm_analysis_imp_decl 宏提供了接受多个传入 transaction 流的最方便的方法，每个传入 transaction 流都有自己独立的 write 方法。例：

```
    `uvm_analysis_imp_decl(_expected)
    `uvm_analysis_imp_decl(_actual)
    class my_cov_collector extends uvm_scoreboard;
        `uvm_component_utils(my_cov_collector)
        uvm_analysis_imp_expected #(tx_t, my_cov_collector) expected_export;
        uvm_analysis_imp_actual #(tx_t, my_cov_collector) actual_export;
        ...
        function void build_phase(uvm_phase phase);
            expected_export = new("expected_export", this);
```

```
        actual_export = new("actual_export", this);
    endfunction
    ...
    function void write_expected(tx_t t);
        ...
    endfunction
    function void write_actual(tx_t t);
        ...
    endfunction
    ...
```

□将覆盖点分为多个覆盖组，以便将的 specification features 的 coverage 与 implementation features 的 coverage 分开。这将有助于重新使用 coverage 模型。

□在 coverage 收集器的配置对象中使用变量 coverage_enable 来启用或禁用 coverage 收集。coverage 收集会产生性能和内存消耗，而某些用例可能不需要收集 coverage。“UVM 用户指南”建议使用名为 coverage_enable 的变量来实现此目的。

□通过调用 sample 方法而不是为 covergroup 指定时钟事件来对进行采样。

这可以是内建 sample 方法或带有参数列表的 sample 方法，如 covergroup IDENTIFIER with function sample(...)。调用 sample 方法允许仅在 transaction 从 DUT 到达 coverage collector 组件时才对值进行采样。

□不要频繁地过度对覆盖组进行采样。考虑对每个覆盖点使用条件表达式 iff(...)来降低采样频率。

过于频繁的采样会不必要地增加需要存储和分析的覆盖数据量。每次对 covergroup 进行采样时，对每个覆盖点进行采样可能没有必要或没有意义。任何条件表达式都应该保持简单：复杂的 iff 条件很难调试。

□在 DUT 端口或者 DUT 内部进行采样，不要在激励上面进行采样。在采样 DUT 的寄存器的值时，应该等到 DUT 寄存器值发生变化后才采样，而不是在激励发生变化时就进行采样。

□考虑将每个覆盖组和覆盖点的 option.at_least 设置为默认值 1 以外的某个值。

option.at_least 的默认值仅确保每个状态被命中一次，这通常不足以测试状态是否已经锁定。

□不要设置覆盖组和覆盖点的 option.weight 或 option.goal。

有两个潜在的问题。首先，给某些状态赋予更大或更小权重的方法可能会扭曲覆盖率报告，其次，这些选项在不同的仿真工具上可能有不同实现方式。

□仔细设计 coverprint bin，以确保涵盖功能重要的 case。

由于 100％覆盖状态空间是不现实的，因此仔细设计 coverage bin 对于验证质量至关重要。一种解决方案的是为典型值，特殊值和边界条件创建单独的 bin。bin 的选择应与验证计划有关。

□设计覆盖点时，请指定非法值或不需要覆盖的值为 ignore_bins。不要使用 illegal_bins。

Covergroups 应限于收集功能覆盖率信息，而不是直接与错误报告相关联。应使用断言或使用 UVM 报告处理程序来捕获非法值。