Testbench Configuration Mantra

Stephen D'Onofrio
Paradigm Works
Paradigm Works, Inc., MA, USA
stephen.donofrio@paradigm-works.com

ABSTRACT
All testbenches, even the simplest testbenches, need some kind of configuration knobs (sometimes called configuration fields or configuration parameters) that are used to control setting up some feature in the verification environment. Ideally, the environment also includes some kind of mechanism that allows test writers a way to override a configuration knob's default value. Configuration knobs are typically setup in the testbench during the building phase and directly used for DUT (Design Under Test) initialization. There are various categories of configuration knobs including (but not limited to) testbench topology knobs, simulation specific knobs, verification component knobs, and testbench specific knobs.

More sophisticated random testbenches generally contain more configuration knobs. Properly placing all these configuration knobs in a testbench may get out of hand quickly. A testbench with a poorly set up configuration structure will most likely inhibit test controllability, perhaps degrades the overall testbench randomness, and may prohibit easy future reuse.

There are no practical guidelines available in VMM and OVM methodologies for structuring configuration knobs in a testbench. However, both these methodologies provide two sophisticated configuration techniques: (1) field automation/structural control configuration mechanism (i.e. registering configuration knobs and using set_config_* calls) and (2) configuration classes that are used in conjunction with the factory. Both these techniques have their own mechanisms that allow test writers means for overriding default configuration field values.

Testbench developers are now plagued with the question of what style configuration technique should they use. Should both these techniques coexist? In addition, the OVM and VMM methodologies do not include suggestions for classifying and compartmentalizing configuration knobs in a testbench that helps promote reuse.

Finally, we will show users templates for OVM and VMM code that contains the configuration techniques discuses above. The templates help enforces a consistent look and feel and enable rapid development and maintenance of the verification code across multiple-sites and cultural barriers.

General Terms

Verification component – In this paper this term is equivalent to both a VMM Transactor (see page 14 reference [2]) or VIP (Verification Intellectual Property) and an OVM OVC (see page 10 in reference [1]).

Configuration mechanism – In this paper this term is equivalent to the OVM configuration mechanism that uses get_config*/set_config* and VMM structural control get*/set*configuration mechanism

Configuration class – This is a class that encapsulates configuration fields

Horizontal reuse - reuse from project to project

Vertical reuse - unit level to system level reuse

1. Introduction
This paper takes a pragmatic view of the configuration techniques utilized by the VMM and OVM methodologies. With the release of VMM 1.2, it appears that the OVM and VMM configuration options are converging. Verification teams need to understand the advantages and limitations of the configuration options that are available in these methodologies so that they can make intelligent decisions.

Today's large verification efforts include tremendous amounts of configuration knobs. These verification efforts require using techniques that allow for reuse and extensibility. The methodology in which you use to declare configuration, layer configuration and the techniques used for your configuration can help or hinder your verification effort. This paper focuses on the techniques we successfully used in the past and steps through the evaluation process we went through.

This paper discusses configuration technique for:
(1) Structuring and programming testbenches
(2) Programming the DUT
(3) Controlling scenarios and tests

Both the field automation/structural control configuration mechanism utilized by OVM and VMM and configuration class configuration techniques is examined in this paper.

RTL configuration described in the VMM User Guide [2] and verilog libmap techniques that connect a testbench to RTL are beyond the scope of this paper.

The primary focus of the paper is on the testbench configuration that occurs once before a test (simulation) is executed. From our experience, this is generally the only time configuration is setup, randomized, and dispersed throughout the testbench. Occasionally, it may be a requirement to change the configuration dynamically during a simulation. This paper only touches on this subject by setting up a simple example of a single configuration field in the middle of a simulation.

We find that testbench configuration may be broken up into several categories that are described in the next section. As mentioned above, there are two configuration techniques – the configuration class technique and configuration mechanism technique. This paper will examine how each of the configuration categories size up against each configuration techniques.

2. Configuration Categories

There are four categories of configuration fields that we find in a testbench:

(1) Design configuration knobs
(2) Verification component configuration knobs
(3) Topology configuration knobs
(4) Test configuration knobs

(1) Design Configuration Knobs

All DUTs with any level of complexity have large number amounts of features. Often these features are controlled by registers that are accessible to software, verilog parameters, preprocessor defines, configuration pins, or core generators. For example, a PCIE core may include *link-width* field that contains the value of the size that is created by a core generator tool. Another example is a *foo* option for a MAC interface that is set or cleared by software writing to a register via a CPU interface.

Typically features are controlled by registers in a design that can be accessed via software. The verification environment normally includes an initialization sequence that is responsible for taking the design configuration fields (these fields are typically randomized) and driving its data into the register via a CPU interface.

Occasionally, the testbench will need to verify variations of DUT features that are controlled via compile time parameters or core generator switches. This requires advanced testbench DUT connection schemes that are beyond the scope of this paper.

Design configuration knobs are typically randomized in a random testbench. Occasionally, design configuration fields need to be constrained in order to close coverage holes or reproduce scenarios that may have occurred in the lab.

(2) Verification Component Configuration Knobs

Verification components typically include various knobs for setting up modes, controlling stimulus and responses sent into the DUT. For example, most verification components have knobs for controlling the size of the intra gap delay between packets. Another example for a PCIE Express verification component, the component may have a field that controls different acceptable link widths for link initialization.

Verification component configuration fields, similar to design configuration fields, are typically heavily randomized in order to stress the DUT but may also need to be constrained in order to hit specific corner cases.

(3) Topology Configuration Knobs

Both VMM and OVM describe testbench topology configuration knobs. These knobs control how the testbench structure is built. Robust VIP includes topology knobs for disabling/enabling monitors, drives, and agents. These knobs are significant for promoting vertical and horizontal verification component reuse.

For example, the OVM User Guide [1] describes knobs (or fields) for controlling whether an agent is passive or active, how many slave/master agents, and whether to build a "bus" monitor.

Similarly, the VMM User Guide [2] describes topology configuration knobs for controlling monitors and xactors are proactive, reactive, and passive (see section 8).

Topology configuration knobs are typically fixed (not dynamically randomized). For example, it does not make sense to randomly enable/disable a driver component on a per simulation basis. Its value is fixed based on the topology of the testbench.

Test Configuration Knobs

Testbench configuration knobs are also described in both VMM and OVM. These configuration knobs describe what kind and how much stimulus to drive into the design. Additional, other examples of test control fields are switches for enabling checkers, coverage, and assertions.

For example, the OVM User Guide [1] figure 5.3 describes standard configuration fields for enabling checker and coverage on page 69. Test/sequence configuration is described on pages 80-82.

Similarly, throughout the VMM User Guide [2] it describes test/scenario configuration fields for controlling and ending simulations.

What should NOT be included in Configuration?

Often teams will add non-configuration field variables that have nothing to do with configuration. Since a configuration object is referenced in multiple components that do not have visibility to each other, sometimes teams will use the configuration object reference to pass information to/from multiple components. For example, a monitor may capture the state of the bus and stuff the value in a property in the configuration object . A sequence then may read the state value from the configuration object and do something with it. This practice pollutes the configuration class and causes maintenance headaches. Additionally, it may prohibit easy reuse of the components because they rely on each other via the configuration object. VMM Channels or TLM interfaces should be used to communicate to/from multiple components.

3. **What does OVM and VMM offer for configuration?**

Both OVM and VMM have similar configuration capabilities.

(1) Configuration class technique

- Configuration descriptions (knobs) are encapsulated in a configuration class
- Configuration field descriptions may be random and have an associated default constraint
- Configuration is randomized before the simulation occurs
- Some steps need to be taken to pass down the configuration object to lower layer testbench components
- Test cases may customize (or override) the default configuration constraints using the factory

(2) OVM/VMM Configuration mechanism

This approach goes by different names. In the OVM Manual [1] it is called the “OVM Configuration Mechanism” and in VMM it is called “Hierarchical/Structural Configuration"

- Configuration field descriptions (knobs) are embedded in the verification components – i.e. inside the driver, sequencer, monitor
- Configuration field descriptions have a default value
- The configuration data is NOT randomized unless you do some extra work – this paper described a technique for accomplishing this task
- There are library calls “set_*” that allow higher layers such as the “test case” layer override the default values in the lower component layers.
    - Wild * card searches via the testbench hierarchy can be made to distinguish which downstream component to override
- VMM allows overriding configuration using command line options. OVM also has a contribution for command line option.

OVM and VMM provide advanced capabilities for controlling the configuration fields through the configuration mechanism. The primary purpose of the configuration mechanism is to control the field value setup during the build phase. The build phase occurs before any simulation time is advanced. The fields may also be changed during simulation time (or the run phase) but requires additional work and is beyond the scope of this paper.

The configuration mechanism gives test writers and higher layer testbench components (i.e. module/system OVCs or subenvs) the ability to override the default field settings of the components. A testbench hierarchy is established in top-down fashion where parent components are built before their child components. Higher-level testbench layers (test cases) and components (system/module OVCs or subenvs) can override default configuration settings. Increasing configuration override priority is from right to left in Figure 1.

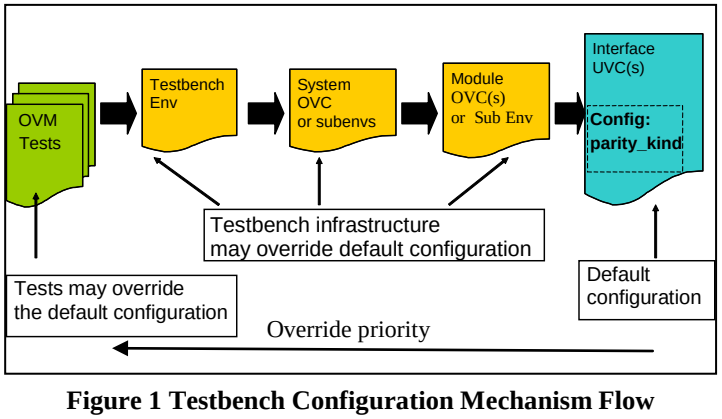

Figure 1 Testbench Configuration Mechanism Flow

4. Configuration and testbench architecture

Advanced testbench architectures are typically composed with an environment (or testbench layer) that encapsulates one or more verification components. The verification components model a specific protocol. The environment typically also includes scoreboards. Scoreboards and their associated transfer functions are connected to the verification components and verify the data integrity by comparing actual data against expected results. In addition, more advanced testbenches may include subenvs (or module OVCs as described in section 3.4 in Step-by-Step Functional Verification with SystemVerilog and OVM[3]). These components further enhance reuse by assisting in bringing unit level testbench components into the higher level testbenches (or perhaps a system level testbench).

All these testbench components need some access to configuration data. This paper is going to examine various configuration approaches using both OVM and VMM testbenches. The testbenches built for this paper include three verification components – a CPU interface, PCIE interface, and MAC interface. Additionally, the examples show how to connect configuration data to initialization sequences.

Below is a drawing of an OVM testbench architecture used for this paper.

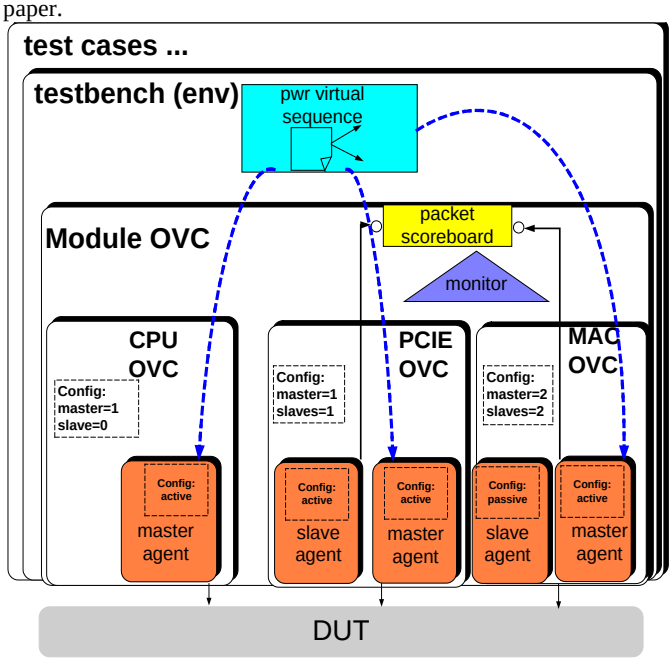

Figure 2 OVM Testbench

This testbench includes a DUT, three interface OVCs – the CPU, PCIE, and MAC OVCs, a Module OVC, a scoreboard and virtual sequencer.

Below is a drawing of a VMM testbench architecture used for this paper.

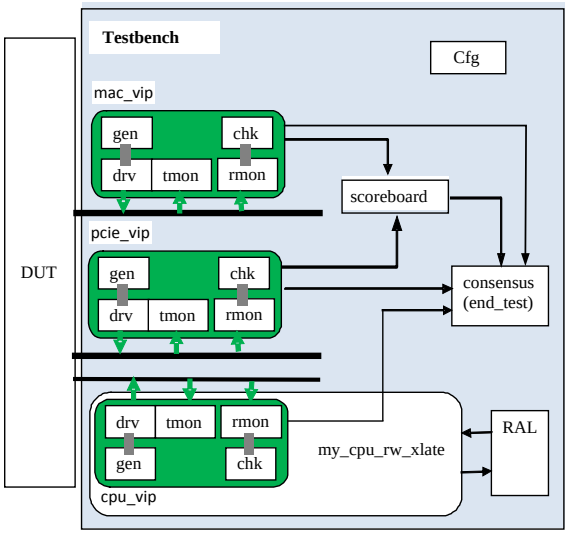

Figure 3 Typical VMM Testbench

This testbench includes a DUT, three VIPs – the CPU, PCIE, and MAC, RAL, consensus, and a scoreboard.

The following table lists the configuration knobs that are examined in this paper.

Location	*Configuration*	*Category*
MAC	foo	design/verification component
MAC	bar	design/verification component
MAC	num_masters	Topology
MAC	num_slaves	Topology
MAC	intf_checks_enable	Test
MAC	has_bus_monitor	Test
MAC	intf_checks_enable	Test
MAC	has_bus_monitor	Test
PCIE	lane_reversal_support	design/verification component
PCIE	supported_link_width	design/verification component
PCIE	max_payload_size	design/verification component
PCIE	bar_0..7_start	design/verification component
PCIE	bar_0..7_end	design/verification component
PCIE	num_masters	Topology
PCIE	num_slaves	Topology
CPU	Parity	design/verification component
CPU	num_masters	Topology
CPU	num_slaves	Topology
CPU	intf_checks_enable	Test
CPU	has_bus_monitor	Test
MOD	Mode	design/verification component
SEQ	num_mac_packets	Test
SEQ	num_pcie_packets	Test
SEQ	mac_seq_kind	Test
SEQ	pcie_seq_kind	Test

Table 1 Configuration Knobs

5. Managing Configuration

As described above there are two techniques for handling configuration in both VMM and OVM. In this section, methodology for configuration breakdown, configuration coordination, randomizing configuration, and overriding configuration is described showing both of these techniques. The examples in this section are all done with OVM.

5.1 Configuration breakdown

In this section we describe various ways for breaking down configuration fields using a configuration class and then using the configuration mechanism.

5.1.1 Configuration class breakdown

In this section the configuration class breakdown methodology is examined in detail. A testbench usually contains one or more verification components. Each of these verification components will likely need to access configuration. Advanced testbenches that use subenvs (module OVCs) may also include configuration. The testbench environment may additionally include its own configuration.

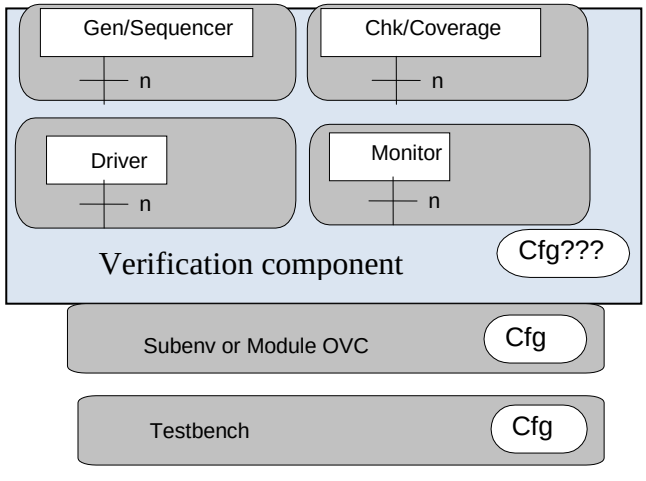

Figure 4 Configuration Class Example

It is relatively straight forward to model the configuration for the module OVC and testbench – usually one configuration class for the testbench and one configuration class for the module OVC.

The verification component is made up of multiple components. Verification teams need to make decisions on how to break up the configuration for the components. The remainder of this section show several options of how one may consider breaking down configuration classes inside a “verification component”.

Option A – One Environment Configuration

In this example the "testbench cfg" block maintains the configuration fields for the verification component(s). The verification component uses the testbench configuration.

- Pros
  - Only one configuration file to find all configuration knobs
- Cons

- Inhibits reuse because there is not a separation between the "Verification Components" and the Testbench Environment
- Difficult to maneuver through the configuration file because too many configuration knobs in one class

Option B – Each Component defines configuration

With this option each components inside the verification component includes its own configuration class.

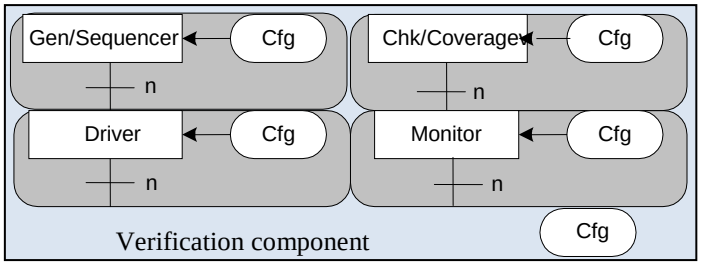

Figure 5 Configuration class for each component

The configuration classes for each of the components inside the verification component are instantiated and randomized inside the testbench. Each of the configuration class instances are referenced inside their associated component inside the verification component.

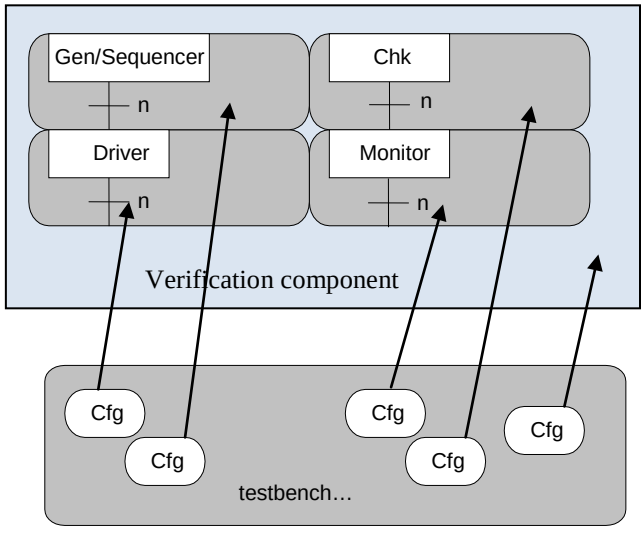

Figure 6 Testbench - configuration class for each component

Option B Pros/Cons

- Pros
  - Each component has its own configuration object
    - All the configuration knobs belong to the component. This is certainly not the case for option A.
    - Nice for reuse. Easy to take out individual components from the "verification component".
- Cons
  - Lots of configuration objects.
    - Extra work for the environment configuration to coordinate all the configurations.
  - Potentially many configuration knobs will be duplicated within the same verification component
    - How do you handle constraints for the duplicated configuration knobs?
    - Some extra coding and perhaps processing time (problem not an issue)
    - What happens if the configuration needs to change in mid simulation?
      - Need to re sync all the duplicate fields

Option C - Each Layer has a single Configuration

In this example, the gen/sequencer and checker/coverage are considered the top layer and the driver/monitor are considered the bottom layer. Each of these layers includes their own configuration class.

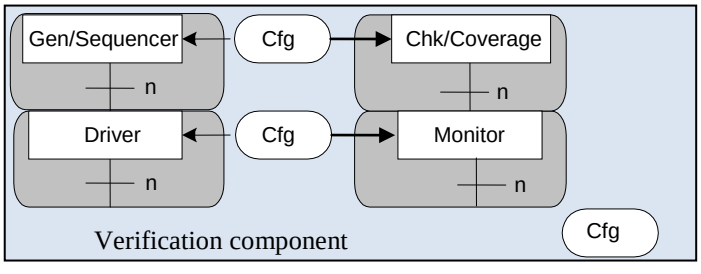

Figure 7 Configuration class for each layer

Again, the configuration classes for this verification component are instantiated and randomized inside the testbench. This time there are two fewer configuration classes that need to be instantiated. The configuration class instances are referenced inside the verification component.

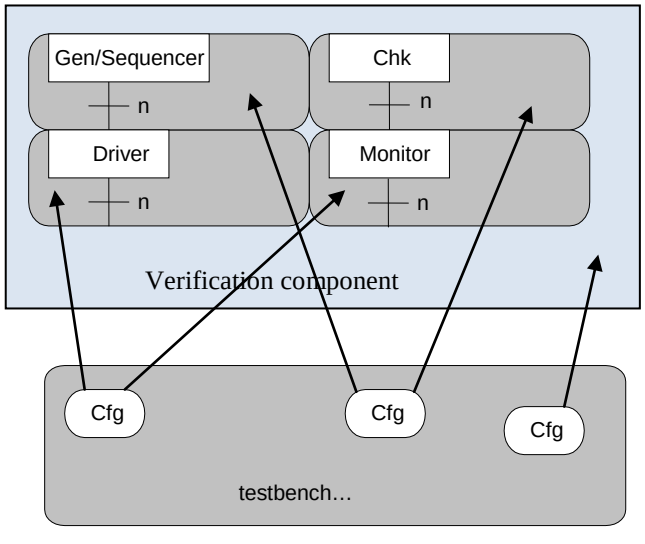

Figure 8 Testbench - configuration class for each layer

Option C Pros/Cons

- Pros
  - Each layer of components has its own configuration class
    - Most of the configuration knobs belong to the component. Although there is some potential for an unused field (i.e. the check_enable is only used in the monitor and not in the "gen/sequencer")
    - Somewhat easy for reuse. Easy to take out individual components from the "verification components"
  - Less configuration classes compared to option B
- Cons
  - Still two configuration classes for a single verification component needs to maintain.
  - Potentially there may be configuration knobs duplicated within the same verification component – Same as Option B

Option D - Verification Component has a single Configuration

In this example, each layer of components inside the verification component shares the same configuration class.

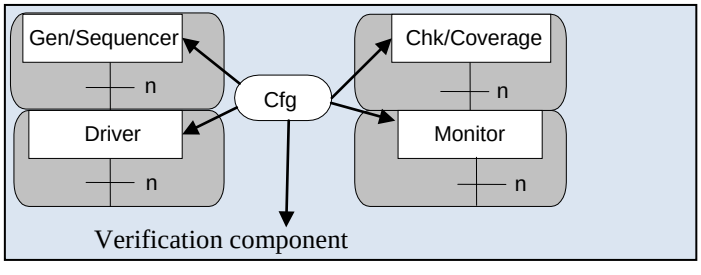

Figure 9 Single configuration class

Again, the configuration class for this verification component is instantiated and randomized inside the testbench. This time the testbench only needs to instantiates a single configuration classes. The configuration class instances need to be referenced by all the components in the verification component.

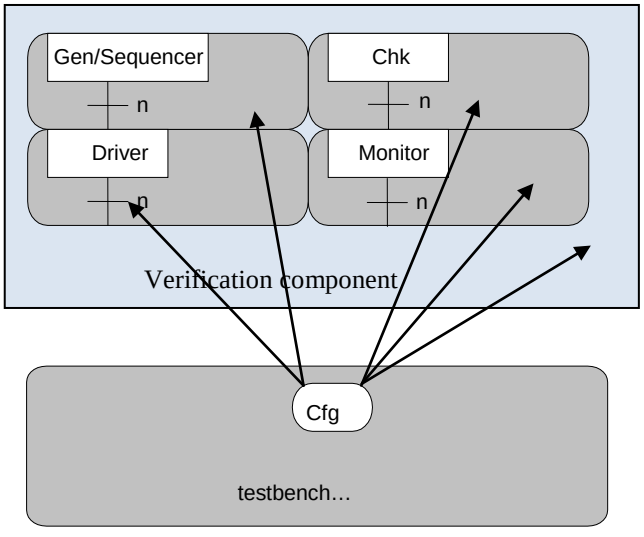

Figure 10 Testbench - single configuration class

Option D – Pros/Cons

- Pros
  - Each "verification component" has a single configuration class
    - This option provides us with single configuration class that may easily be ported for vertical and horizontal reuse
  - The testbench only needs to maintain a single configuration class for the verification component
- Cons
  - The individual components (monitor, driver, and generator) that make up the "verification component" most likely configuration do not use all the knobs included in the configuration class. For example, if there may be configuration field such as "drive_n_packets" in the configuration class that only the used by the driver component.
  - It is not as easily to reuse individual components (monitor, driver, and generator) because the configuration class is at the "verification component" level of abstraction.

Evaluating verification component options

- Option A has no reuse potential so it is out of the question
- Option B has the most reuse potential but has a large amount of testbench overhead and maintenance for keeping track of the configuration data. In our example, we have a cpu driver and monitor that include a CPU configuration field called "parity_kind". If we use this option then we always need to make sure the value of the "parity_kind" field need to stay in sync inside both the cpu driver and monitor. This is not a scalable methodology as the amount of configuration fields in a testbench grows.
- Option C has some of the same maintenance issues as option B. It is a hybrid option between options B and D. It does not seem valuable to break up configuration between these layers.
- Option D has the least amount of testbench maintenance overhead. Looking at our example of the "parity_kind"

configuration field, now the driver and monitor reference the same configuration object – there is no sync issues. Verification component reuse is achievable with this option. However, using this option make it more difficult to reuse individual components compared to option B. But our main objective is usually not to break apart verification components and reuse only the monitor, driver, and sequencer. This option seems the most viable solution for breaking down configuration for a verification component.

5.2.2 Configuration mechanism breakdown

When using the configuration mechanism, the configuration fields are embedded directly inside the component in which the configuration is needed rather than a separate configuration class. For example, the ***parity_kind*** field for the CPU verification component is included in both the CPU driver and the CPU monitor. The driver needs to calculate the proper parity when driving data into the Host interface on the DUT and the monitor needs to check if the DUT is properly transmitting the data.

5.3 Verification component configuration knobs

This section describes how to populate configuration fields inside a verification component using the configuration class technique and the configuration mechanism technique.

5.3.1 Configuration class knobs

A configuration class is implemented as a standalone file and included in the verification components distribution. This configuration class needs to be self-contained for easy vertical and horizontal reuse. In other words, it should only contain generic fields and constraints. When adding a verification component to a testbench, the verification component's configuration may be extended to allow for testbench specific constraints.

Below is the code snippet of the example MAC OVC configuration class using configuration breakdown option D.

The *mac_config* class inherits the *ovm_object* base class. The light weight ovm_object base class allows us to declare configuration fields and register the configuration class with the OVM factory. Furthermore, the fields are registered the fields with the OVM automation macros. This automates our fields for copy, compare, and print operations.

```
class mac_config extends ovm_object;
  rand bit foo;
  rand int unsigned bar;
  rand int unsigned num_masters = 1;
  …

   `ovm_object_utils_begin(mac_config)
    `ovm_field_int(foo, OVM_ALL_ON)
    `ovm_field_int(bar, OVM_ALL_ON)
    `ovm_field_int(num_masters, OVM_ALL_ON)
     …
  `ovm_object_utils_end

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


endclass
```

Alternatively, one could have inherited the configuration class from the *ovm_component* base class. This would have added the MAC configuration class into the testbench hierarchy and allow controls for the set_* configuration. However, when dynamically building the testbench components, we ran into race conditions between randomizing the configuration class and the dynamic build. This is due to the nature of the top-down builds methodology. The testbench level build (where the dynamic build occurs) took place before the child configuration build occurs. Therefore, it is safest to inherit the configuration class from the ovm_object class to ensure that test writers can only use the factory to override the configuration class.

5.3.2 Configuration mechanism knobs

This section describes embedding design/verification configuration fields inside a verification component. The following two code snippets show how the CPU ***parity*** design/verification configuration field is declared inside the driver and monitor. The monitor has additional test configuration fields to enable checking and coverage. The OVM macros are used to turn on field automation. Field automation not only automates our fields for copy, compare, and print operations but it additionally registers the field with the configuration mechanism.

```
class cpu_master_driver extends ovm_driver #(cpu_transfer);

  cpu_parity_t parity;

  `ovm_component_utils_begin(cpu_master_driver)
    `ovm_field_enum(cpu_parity_t, parity, OVM_ALL_ON)
  `ovm_component_utils_end
```

```
class cpu_master_monitor extends ovm_monitor;

 // This field controls if this monitor has its checkers enabled
 // (by default checkers are on)
 bit checks_enable = 1;

 // This field controls if this monitor has its coverage enabled
 // (by default coverage is on)
 bit coverage_enable = 1;

 cpu_parity_t parity;

 `ovm_component_utils_begin(cpu_master_monitor)
   `ovm_field_int(checks_enable, OVM_ALL_ON)
   `ovm_field_int(coverage_enable, OVM_ALL_ON)
   `ovm_field_enum(cpu_parity_t, parity, OVM_ALL_ON)
 `ovm_component_utils_end
```

Alternatively, the above code snippets may be implemented without the OVM macros by explicitly calling out the get_config* code. The following code snippet shows the extra code that is needed to avoid the OVM macro. It is recommended to avoid this extra code that will cost the verification team extra implementation, potential debugging, and extra code maintenance.

```
class cpu_master_driver extends ovm_driver #(cpu_transfer);

  cpu_parity_t parity;

  `ovm_component_utils_begin(cpu_master_driver)
  `ovm_component_utils_end

function void build();
   int unsigned loc_parity;
   super.build();

   if (!get_config_int("parity", loc_parity)) begin
      string msg;
      $sformat(msg, "\"parity\" is NOT in the configuration database, using
default value");
      ovm_report_warning("build", msg);
   end
   else begin
     string msg;
     $cast(parity, loc_parity);

      $sformat(msg, "\"parity\" is in the configuration database with value
%0d: enum value %s", loc_parity, parity.name() );
      ovm_report_info("build", msg);
   end

endfunction : build
```

At this point the ***parity*** field inside the cpu_master_driver and cpu_master_monitor can be set to ODD or EVEN by higher layer components using the set_config_int* function. However, it is requirement for the CPU verification component that both the *cpu_master*_driver and *cpu_master_monitor* contain the same ***parity*** value. So care must be taken when setting up the ***parity*** field value.

By default we want the configuration fields inside a verification component to operate with random values – i.e. random parity ODD or EVEN. To accomplish this, a parity configuration field is added at the top-level cpu_env.

The snippet of the cpu_env is shown below. The ***parity*** field in the *cpu_env* is declared using SystemVerilog '*rand'* keyword. Additionally, in the build phase, SystemVerilog *randomize ()* call is added. Without these two additions the ***parity*** configuration field would not randomize. Finally, the set_config_int* call synchronizes the ***parity*** field in the cpu_master_driver and cpu_master_monitor with the value set in the cpu_env.

```
class cpu_env extends ovm_env;

  rand cpu_parity_t parity;

`ovm_component_utils_begin(cpu_env)
    `ovm_field_enum(cpu_parity_t, parity, OVM_ALL_ON)
 `ovm_component_utils_end

function void cpu_env::build();

   if (this.randomize() == 0)
     ovm_report_fatal("build", "randomize failed");
   super.build();

  set_config_int("*", "parity", parity);
   …
endfunction : build
…
```

5.4 Coordinating multiple configurations

This section describes how to coordinate configuration data between a verification component and a subenv (module OVC). Coordinating using the configuration class technique is discussed first and followed by the configuration mechanism technique.

5.4.1 Coordinating multiple configurations with configuration class

A design may operate in multiple modes which affects the setup/randomization of the verification component's configuration. The design modes are DUT specific and modeled in the module configuration.

For the example testbenches shown in Figure 2 and 3, there are four operational modes at the module level - MODE_1, MODE_1_NO_MAC, MODE_2, and MODE_3. A module configuration class is added to model this behavior using a design configuration field called ***mode***.

```
class my_module_ovc_config extends ovm_object;

  rand mode_kind_t mode;

  // factory registration
  `ovm_object_utils_begin(my_module_ovc_config)
    `ovm_field_enum(mode_kind_t, mode, OVM_ALL_ON)
  `ovm_object_utils_end
….
endclass : my_module_ovc_config
```

These module level modes affect the MAC design configuration fields ***bar*** and ***foo***. The relationship between the modes and MAC design configuration fields is listed in Table 2.

Sub Env Module OVC Mode	*MAC bar*	*MAC foo*
MODE_1	9	1
MODE_1_NO_MAC	don't care	don't care
MODE_2	9	0
MODE_3	9	0

Table 2 Mode relationship to MAC configuration

Depending on the mode selected, constraints need to be applied to the MAC configuration fields to enforce the desired relationship. To achieve this, a new MAC configuration class is created. It inherits the *mac_config* class and adds a constraint based on the module's ***foo*** field. The new MAC configuration class includes a reference to the *module_ovc_cfg*. Show below is a snippet of the new MAC configuration class.

```
class my_module_ovc_mac_config extends mac_config;

 // NOTE: This is a reference to the module OVC Configuration
 my_module_ovc_config module_ovc_cfg;

  // factory registration
  `ovm_object_utils_begin(my_module_ovc_mac_config)
  `ovm_object_utils_end

  constraint bar_c {
    bar == 9;
  }

  constraint foo_c {
              if (module_ovc_cfg.mode == MODE_1)
          foo == 1;
              else
          foo == 0;
  }

 function void set_module_ovc_config_ref(my_module_ovc_config
                                                 in_module_ovc_cfg);
        module_ovc_cfg = in_module_ovc_cfg;
 endfunction : set_module_ovc_config_ref
```

5.4.2 Coordinating configurations with configuration mechanism

Recall that there are four operational modes that have an affect on the MAC configuration – see Table 2.

The coordination with the configuration mechanism is done through the procedural code since we are using set_config* calls as shown below. In the code, first, the ***mode*** field is declared as ***rand***. Next, in the build phase, the MAC configuration's ***foo*** field is setup based on the ***mode*** setting. Finally, the MAC verification component is built based on the mode setting.

```
class my_module_ovc_env extends ovm_env;

 rand mode_kind_t mode;

virtual function void build();

    this.randomize();
    super.build();

     if (mode == MODE_1)
      set_config_int("mac_inst", "foo",  1);
    else
      set_config_int("mac_inst", "foo",  0);

    set_config_int("mac_inst", "bar",  9);
     …
    if (module_ovc_cfg.mode != MODE_1_NO_MAC) begin
        mac_inst = mac_env::type_id::create("mac_inst",this);
    end
  endfunction : build
```

5.5 Randomizing configuration

This section shows the methodologies for randomizing configurations. Randomizing with both the configuration class technique and the configuration mechanism technique are discussed.

5.5.1 Randomizing using configuration class

Configuration classes are instantiated at the testbench layer. This makes it easy to push the configuration objects down to any level in the testbench hierarchy. Also at the testbench level the handle for the module OVC configuration object is passed into the MAC verification component's configuration class.

After instantiating and randomizing the configuration classes, a set_config_object* call is invoked to push the configuration object's handle down to the verification components and subcomponents. A code snippet for the example OVM testbench is shown below.

```
class my_testbench_tb extends ovm_env;
  …
  my_module_ovc_config module_ovc_cfg;
  my_module_ovc_mac_config mac_cfg;
  …
   // build
   virtual function void build();
      module_ovc_cfg =    my_module_ovc_config::type_id::create("module_ovc_cfg", this);
      if (module_ovc_cfg.randomize() == 0)
          ovm_report_fatal("build", "module_ovc_cfg.randomzie() failed!");

     mac_cfg = my_module_ovc_mac_config::type_id::create("mac_cfg", this);
     mac_cfg.set_module_ovc_config_ref(module_ovc_cfg);

     if (mac_cfg.randomize() == 0)
       ovm_report_fatal("build", "mac_cfg.randomzie() failed!");

    super.build();
    …
    set_config_object("v_sequencer", "mac_cfg", mac_cfg, 0);
    …
    if (module_ovc_cfg.mode != MODE_1_NO_MAC)
      set_config_object("my_module_ovc_env_inst.mac_inst", "mac_cfg", mac_cfg, 0);
   …
   endfunction : build
```

5.5.2 Randomizing using the configuration mechanism

Unlike using the configuration class where the configuration class needs to be randomized explicitly in the testbench layer, for the embedded configuration fields, a SystemVerilog *randomize ()* call can be added to the enclosing verification component or subcomponent. This ensures that the random behavior is self contained inside the verification component. It also makes the randomization call portable to help promote easy reuse.

```
class my_module_ovc_env extends ovm_env;

  rand mode_kind_t mode;

virtual function void build();

     // randomize this env
     this.randomize();

     super.build();
   endfunction : build
```

Care needs to be given when some fields need proper constraints. The constraints are added directly to the enclosing component. The code snippet below shows default constraints to control the upper and lower limits of the ***bar*** and ***foo*** fields.

Last thing to mention is the "+OVM_DEC" addition. When used with the OVM field automation, this flag causes the ***foo*** and ***bar*** fields to print out in decimal format rather than the default hexadecimal format.

```
class mac_env extends ovm_env;
…
  rand int unsigned foo;
  rand int unsigned bar;

  constraint bar_c {
                                bar >= 5;
                                bar <= 15;
                              }

  constraint foo_c {
                                 foo >= 100;
                                 foo <= 200;
                              }

   `ovm_component_utils_begin(mac_env)
    …
    `ovm_field_int(bar, OVM_ALL_ON+OVM_DEC)
    `ovm_field_int(foo, OVM_ALL_ON+OVM_DEC)
  `ovm_component_utils_end
```

5.6 Stimulus and configuration

This section describes how configuration helps control testbench stimulus and how to implement an initialization sequence. Both configuration class technique and the configuration mechanism technique are discussed.

5.6.1 Stimulus and configuration classes

In the sequence library for our example testbenches, a boiler plate sequence was developed as shown in Figure 11. It has the ability to be reused for a majority of the tests. This boiler plate sequence has a number of test configuration knobs. The boiler plate sequence starts the ***init_dut_seq***. When ***init_dut_seq*** finishes, the PCIE traffic, MAC traffic, and background sequences are invoked. When both the MAC and PCIE sequences finish, the test ends.

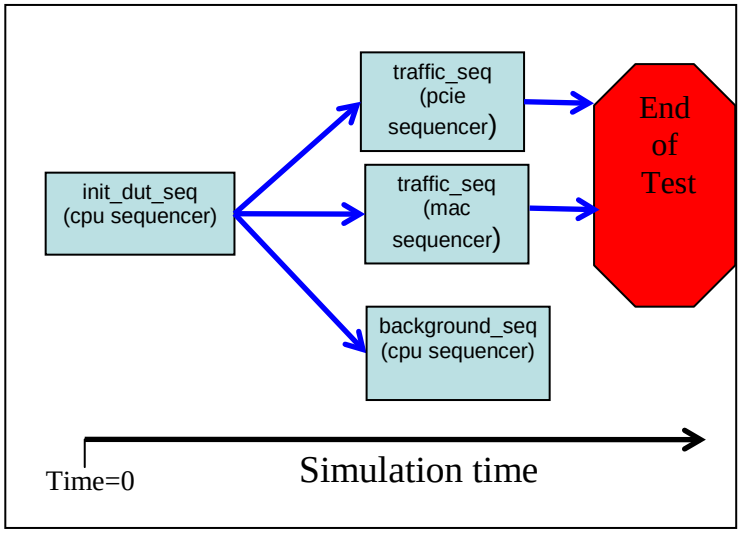

Figure 11 Boiler Plate Sequence

The virtual sequencer has a configuration class that includes the following test configuration fields to control the boiler plate sequence.

Sequence Test Configure Field	*Value*
num_pcie_packets	0..0xffffffff
pcie_seq_kind	PCIE_SMALL_SEQ, PCIE_LARGE_SEQ, PCIE_RAND_SEQ
num_mac_packets	0..0xffffffff
mac_seq_kind	MAC_SMALL_SEQ, MAC_LARGE_SEQ, MAC_RAND_SEQ
use_background_traffic	0..1

Table 3 Sequencer Configuration Field

A snippet of our sequencer configuration class is shown below.

```
class my_testbench_virtual_sequence_config extends ovm_env;

  rand int unsigned num_pcie_packets;
  rand int unsigned num_mac_packets;
  rand int unsigned use_background_traffic;
  rand pcie_seq_kind_t pcie_seq_kind ;
  rand mac_seq_kind_t mac_seq_kind ;

 constraint packets_c {
                          num_pcie_packets == 1;
                          num_mac_packets == 1;
                          use_background_traffic == 1;
                          pcie_seq_kind == PCIE_RAND_SEQ;
                           mac_seq_kind == MAC_RAND_SEQ;
 }

 `ovm_component_utils_begin(my_testbench_virtual_sequence_config)
  `ovm_field_int(num_pcie_packets, OVM_ALL_ON)
  `ovm_field_int(num_mac_packets, OVM_ALL_ON)
  `ovm_field_int(use_background_traffic, OVM_ALL_ON)
  `ovm_field_enum(pcie_seq_kind_t, pcie_seq_kind, OVM_ALL_ON)
  `ovm_field_enum(mac_seq_kind_t, mac_seq_kind, OVM_ALL_ON)
 `ovm_component_utils_end

endclass : my_testbench_virtual_sequence_config
```

The ***init_dut_seq*** is responsible for programming the DUT via the CPU interface. Recall that the MAC configuration object handle is passed into the virtual sequencer in the testbench's build phase. All sequences in the virtual sequence library have access to the MAC configuration via the built-in ***p_sequencer*** pointer. This allows the initialization sequence to properly program the ***foo*** MAC register. A snippet of the initialization sequence is shown below.

```
class testbench_init_seq extends ovm_sequence #(ovm_sequence_item);
 …
 write_cpu_master_seq write_cpu_master_seq_inst;

 virtual task body();
    // Setup Design
     `ovm_do_on_with(write_cpu_master_seq_inst,
                          p_sequencer.cpu_0_m_sequencer,
                          { write_cpu_master_seq_inst.addr == 32'h000;
                            write_cpu_master_seq_inst.data ==
                            p_sequencer.module_ovc_cfg.mode;} )

    // Program MAC
    if (p_sequencer.module_ovc_cfg.mode != MODE_1_NO_MAC) begin
        `ovm_do_on_with(write_cpu_master_seq_inst,
                          p_sequencer.cpu_0_m_sequencer,
                          { write_cpu_master_seq_inst.addr == 32'h100;
                            write_cpu_master_seq_inst.data ==
                            p_sequencer.mac_cfg.foo;} )
     end
…
```

A snippet of the MAC traffic sequence is shown below. The virtual_sequencer_config data is visible to the sequencer via the built-in ***p_sequencer*** pointer. The sequence sends out a specified number of packets. The number is controlled by the test configuration field ***num_mac_packets***. The kind of MAC packets sent is controlled by the test configuration field ***mac_seq_kind***.

```
class mac_seq extends ovm_sequence #(ovm_sequence_item);

  large_mac_master_seq large_mac_master_seq_inst;
  small_mac_master_seq small_mac_master_seq_inst;
  mac_seq_kind_t mac_seq_kind;

  virtual task body();

   repeat (p_sequencer.virtual_sequence_cfg.num_mac_packets) begin
      if (p_sequencer.virtual_sequence_cfg.mac_seq_kind ==
MAC_RAND_SEQ)
         assert(std::randomize(mac_seq_kind) with { mac_seq_kind !=
                                                    MAC_RAND_SEQ; } );

      else
              mac_seq_kind = p_sequencer.virtual_sequence_cfg.mac_seq_kind;

      case (mac_seq_kind)
        MAC_SMALL_SEQ : `ovm_do_on(small_mac_master_seq_inst,
                                   p_sequencer.mac_0_m_sequencer)
        MAC_LARGE_SEQ : `ovm_do_on(large_mac_master_seq_inst,
                                   p_sequencer.mac_0_m_sequencer)
      endcase
    end
 endtask : body
```

5.6.2 Stimulus and configuration mechanism

The test configuration class fields can be converted to configuration fields in the virtual sequencer component. The code snippet below shows the virtual sequencer with embedded test configuration fields and design configuration fields.

```
class my_testbench_virtual_sequencer extends ovm_sequencer;

  int unsigned num_pcie_packets = 1;
  int unsigned num_mac_packets = 1;
  int unsigned use_background_traffic = 1;
  pcie_seq_kind_t pcie_seq_kind = PCIE_RAND_SEQ;

  mode_kind_t mode;
  bit  foo;
  max_payload_t max_payload_size;
  cpu_parity_t parity;

  function new(input string name="", input ovm_component parent=null);
     super.new(name, parent);
     `ovm_update_sequence_lib
  endfunction

  // OVM automation macros for sequencers
  `ovm_sequencer_utils_begin(my_testbench_virtual_sequencer)
     `ovm_field_int(num_pcie_packets, OVM_ALL_ON)
     `ovm_field_int(num_mac_packets, OVM_ALL_ON)
     `ovm_field_int(use_background_traffic, OVM_ALL_ON)
     `ovm_field_enum(pcie_seq_kind_t, pcie_seq_kind, OVM_ALL_ON)
     `ovm_field_enum(mode_kind_t, mode, OVM_ALL_ON)
  `ovm_sequencer_utils_end


endclass: my_testbench_virtual_sequencer
```

When using the configuration mechanism approach, each design configuration field needs to be pushed into the initialization sequence and eventually into a verification component. With the configuration

class, the entire set of configuration fields are obtained via a configuration object. The following code snippet shows how to push the module's ***mode***, MAC's ***foo***, and PCIE's ***max_payload_size*** configuration fields into the sequencer. This methodology is not very scalable once the design grows.

```
class testbench_tb extends ovm_env;

  function void connect();
     …
    v_sequencer.mode = mod_ovc_inst.mode;
    v_sequencer.max_payload_size =
              mod_ovc_inst.pcie_inst.max_payload_size;
    if (my_module_ovc_env_inst.mode != MODE_1_NO_MAC)
      v_sequencer.foo  = mod_ovc_inst.mac_inst.foo;
    v_sequencer.parity =mod_ovc_inst.cpu_inst.parity;

  endfunction : connect

endclass : testbench_tb
```

The following is a snippet of the initialization sequence.

```
// Setup Design
    `ovm_do_on_with(write_cpu_master_seq_inst,
                            p_sequencer.cpu_0_m_sequencer,
                               { write_cpu_master_seq_inst.addr == 32'h000;
                                 write_cpu_master_seq_inst.data ==
                                                        p_sequencer.mode;} )

    // Program MAC
if (p_sequencer.mode != MODE_1_NO_MAC) begin
     `ovm_do_on_with(write_cpu_master_seq_inst,
                              p_sequencer.cpu_0_m_sequencer,
                                { write_cpu_master_seq_inst.addr == 32'h100;
                                 write_cpu_master_seq_inst.data ==
                                                    p_sequencer.foo;} )
  end

    // Program PCIE
    `ovm_do_on_with(write_cpu_master_seq_inst,
                            p_sequencer.cpu_0_m_sequencer,
                               { write_cpu_master_seq_inst.addr == 32'h200;
                                 write_cpu_master_seq_inst.data ==
                                                p_sequencer.max_payload_size;} )
```

5.7 Overriding configuration

Tests usually need to customize the configuration data in order to control test configuration fields, manipulate design configuration fields to hit corner cases or error conditions. This section describes override configuration using configuration class technique and the configuration mechanism technique.

5.7.1 Overriding configuration classes

The class factory allows test writers a means for overriding configuration class data without touching any of the testbench code. An example of overwriting a random field with a single value is shown below.

Simple Single Value Override Example

At the module layer, the MAC ***bar*** check is always set to 9. In the code snippet below, a new class inherited from the MAC configuration class forces the ***bar*** to an illegal value of 4.

```
class my_test_module_ovc_mac_config extends my_module_ovc_mac_config;

   // Provide implementations of virtual methods such as get_type_name and
create
   `ovm_object_utils_begin(my_test_module_ovc_mac_config)
   `ovm_object_utils_end

   constraint bar_c {
     bar == 4;
   }

endclass : my_test_module_ovc_mac_config
```

In the following code, a test called "test_mac_cfg_override" uses the factory to override the default MAC configuration type with the derived *my_test_module_ovc_mac_config* type, so ***bar*** is set to 5 rather than 9.

```
class test_mac_cfg_override extends my_testbench_base_test;

  `ovm_component_utils(test_mac_cfg_override)

  virtual function void build();

     factory.set_type_override_by_type(my_module_ovc_mac_config::get_type(),
                                       my_test_module_ovc_mac_config::get_type());

     factory.print();

    // Create the tb
    super.build();
  endfunction : build
endclass : test_mac_cfg_override
```

Complex Override Example

Using configuration class, it is easy to add more complex constraints such as ranges and distributions. In the code below, ***bar*** is constrained to be one of the 1, 2, 3, 4, 5 or 6 with a weighted ratio of 1/4-1/4-1/4-1/4-2-5.

```
class my_test_module_ovc_mac_config extends my_module_ovc_mac_config;

   // Provide implementations of virtual methods such as get_type_name and
create
   `ovm_object_utils_begin(my_test_module_ovc_mac_config)
   `ovm_object_utils_end


   constraint bar_c {
                              bar dist { [1:4] :/ 5, 3 := 2, 6 := 5};
   }

endclass : my_test_module_ovc_mac_config
```

5.7.2 Overriding with configuration mechanism

The configuration mechanism overrides configuration fields using a top-down approach. This allows tests to have complete control of the values driven into the configuration hierarchy.

Simple Single Value Override Example

At the subenv (module OVC) layer, the MAC ***bar*** check is always set to 9. In the code snippet below, the set_config_int* call forces the ***bar*** to be an illegal value of 3.

```
class test_mac_cfg_simple_override extends my_testbench_base_test;

  `ovm_component_utils(test_mac_cfg_simple_override)

  virtual function void build();

    set_config_int("*", "bar", 3);

    // Create the tb
    super.build();
  endfunction : build
endclass : test_mac_cfg_simple_override
```

Distribution Override Example 1

The following code shows how to create a complex distribution constraint. The first step is to create a new MAC env class *my_mac_env* that inherits the *mac_env* class. In the *my_mac_env* class, the *bar_c* constraint is overridden with the new distribution constraint.

```
class my_mac_env extends mac_env;

  `ovm_component_utils_begin(my_mac_env)
  `ovm_component_utils_end

  constraint bar_c {
                      bar dist {7 := 1, 8 := 2};
                    }

endclass // my_mac_env
```

Next, the factory is used to override the *mac_env* class with the *my_mac_env* class.

```
class test_mac_cfg_simple_override extends my_testbench_base_test;

  `ovm_component_utils(test_mac_cfg_simple_override)

  virtual function void build();

    factory.set_type_override_by_type(mac_env::get_type(),
                                      my_mac_env::get_type());

    // Create the tb
    super.build();
  endfunction : build
```

Distribution Override Example 2

Alternatively, the ***bar*** may be setup using procedural code rather than a constraint as shown in the following code.

```
class test_mac_cfg_simple_override extends my_testbench_base_test;

  `ovm_component_utils(test_mac_cfg_simple_override)

  virtual function void build();

   randcase
     1 : bar = 7;
     2 : bar = 8;
   endcase

    set_config_int("*", "bar", 5);

    // Create the tb
    super.build();
  endfunction : build
endclass : test_mac_cfg_simple_override
```

Range Override Example

Alternatively, in the MAC environment a range start and range end could have been introduced. The code snippet below shows two new range fields ***bar_start*** with a value of 3 and ***bar_end*** with a value of 5 added to the bar_c constraint.

```
class mac_env extends ovm_env;
…
  rand int unsigned bar;
  int unsigned bar_start= 3;
  int unsigned  bar_end =5;

  constraint bar_c {
                      bar >= bar_start;
                      bar <= bar_end;
                    }


    `ovm_component_utils_begin(mac_env)
    …
    `ovm_field_int(bar, OVM_ALL_ON+OVM_DEC)
    `ovm_field_int(bar_start, OVM_ALL_ON+OVM_DEC)
    `ovm_field_int(bar_end, OVM_ALL_ON+OVM_DEC)
  `ovm_component_utils_end
```

Now tests can easily change the range using simple set_config* calls as shown below. However, this technique is limited to ranges.

```
class test_mac_cfg_simple_override extends my_testbench_base_test;

  virtual function void build();

    set_config_int("*", "bar_start", 2);
    set_config_int("*", "bar_end", 6);

    // Create the tb
    super.build();
  endfunction : build
endclass : test_mac_cfg_simple_override
```

5.8 Dynamically changing configuration

This section describes dynamically changing configuration using the configuration class technique and the configuration mechanism technique.

5.8.1 Dynamically changing configuration classes

The following code shows a test that dynamically changes the value of the parity field in the CPU configuration. Since all the components reference the configuration object, a single assignment to the parity filed is all that is needed because the change is automatically visible to all the components.

```
task run();
  my_testbench_tb0.cpu_cfg.parity = EVEN_PARITY;
   #1000;
   my_testbench_tb0.cpu_cfg.parity = ODD_PARITY;
    #1000;
   global_stop_request();
 endtask // run
```

5.8.2 Dynamically changing with configuration mechanism

The configuration mechanism only updates get_config* fields in the build phase. If the configuration mechanism could easily be used in the run phase, then dynamic updates for configuration fields would be an elegant technique. There are ways to make the configuration mechanism work dynamically but they require a bit of work and are not part of the mainstream OVM methodology.

5.9 Evaluating configuration mechanism and configuration classes

The configuration mechanism is awkward to use with design/verification configuration knobs that are located inside verification components. Configuration fields that are located inside a verification component typically require randomization and may additionally require a default constraint. Standard verification components developed by the instructions in the OVM User Manual do not include these randomization capabilities. This requires verification teams to implement randomization enhancements for verification components. After these enhancements are made then it is possible for test writers to extend verification components (envs, agents, monitors, and drivers) and use the factory in order to override default random behavior. In addition, test writers may use a single set_config* calls to override a configuration field with a fixed value. Although randomization of configuration fields inside verification components is possible as shown in this paper, it may not seem like a natural fit for some users.

Additionally, there are extra complications with using the configuration mechanism with a verification component that includes duplicate design/verification configuration fields. In this paper we included an example of a parity field that is incorporated in both the driver and monitor of a CPU verification component. It is a requirement for this ***parity*** configuration field to maintain the same value, by default the parity needs to come up with a random value, and test writers may override the parity with a fixed value. In order to implement these requirements it cost additional code as shown in the paper.

In contrast, when design/verification configuration fields are placed inside a configuration class then we do not have the issues listed above. Therefore, it is less esoteric to use the configuration class technique for design/verification configuration knobs. The only limitation is the test writers can not override using a single set_config* call. Instead, test writers can override configuration fields in the configuration class by using inheritance and the factory as shown in this paper.

When it comes to test and topology configuration knobs where synchronizing and randomization may not be less of an issue it is advantageous to utilize the configuration mechanism technique. This way overrides for these fields is accomplished using a simple set_config_* call. The set_config* call using the configuration technique is simpler than extending configuration classes and using the factory.

6. VMM Considerations

The same basic principles for configuration methodology mentioned above apply to VMM 1.2[2].

A random configuration fields can be implemented using the VMM configuration mechanism (vmm_opt). The vmm methodology includes macros that help with randomization `vmm_unit_config_rand_*. The vmm_unit_config_rand_* includes get_* coding that is similar to the OVM macros and a rand_mode shut off randomization for the configuration field. VMM testbench developers need to add randomization and constraints. In the code snippet below the ***foo*** and ***bar*** configuration fields are declared in the MAC env and the randomization code is incorporated.

```
class mac extends vmm_timeline;
  `vmm_typename(mac)

  rand bit foo;
  rand  int unsigned bar;

  constraint bar_c {
                                bar >= 5;
                                bar <= 15;
                        }

   function void build_ph();

 `vmm_unit_config_rand_boolean(foo,
                        "foo mode enable bit", _verbosity,  mac)
  `vmm_unit_config_rand_int(bar, 9,
                        "bar", _verbosity, mac)

    if ((this.randomize()) == 0);
      `vmm_fatal(log, "Failed to randomize configuration");
```

The above code creates default ***foo*** configuration that is randomly 0 or 1 and ***bar*** configuration that is randomized with a value from 5 to 15.

A test case can overwrite the default values. The following is a snippet of a test that overrides the ***bar*** default random 5 to 15 range setting with a fixed illegal value of 35.

```
class test_test extends vmm_test;
  `vmm_typename(test_test)

  pwr_env env;

  function new(string name, pwr_env env);
    super.new(name, "test testcase");

        vmm_opts::set_int("%*:bar", 35);

    this.env = env;

  endfunction
…

endclass
```

The example below shows the configuration class technique we used. We choose to perform our randomization for the configuration classes occurs in the VMM *start_of_sim_ph* phase in the testbench layer. Users may choose other techniques.

```
function void pwr_env::start_of_sim_ph();

  // Setup CFG descriptor
  this.host_cfg = host_cfg::create_instance(this,
            {this.get_object_name(), "_CFG"}, `__FILE__,
`__LINE__);

  if (!is_subenv) begin
    host_cfg.set_log();

    if (host_cfg.randomize() == 0)
      `vmm_fatal(log, "Failed to randomize configuration");
  end
```

In order to override the default configuration, an extended configuration class needs to be created and then the factory can override with the new class type or an instance of the new class. Below is a snippet of the test_host_cfg class that is extended from host_cfg.

```
class test_host_cfg extends host_cfg;
    `vmm_typename(test1_host_cfg)

  constraint c_parity {
                        parity == ODD;
                      }

…
```

Finally, in the test's *configure_test_ph* the host_cfg is replaced by the host_cfg using the vmm factory as shown below.

```
virtual function void configure_test_ph();
     test_host_cfg = new;

    pwr_host_cfg::override_with_copy("@%*", test_host_cfg,
                                     log, `__FILE__, `__LINE__);
…
```

Please see both our VMM and OVM examples to see all the implementation details.

7. Automating testbench configuration

OVM and VMM are quite open ended when it comes to configuration. OVM and VMM also lack with recommendations for directory structure, file-naming conventions and coding styles. As shown in this paper, well-structured OVM and VMM configuration helps with reusability. It is a time-consuming task for an organization to decide on which approach is most suitable for their verification teams to utilize based on their verification charter. We found that just implementing what we believe is a "best-practice" OVM testbench framework is a time consuming task.

It is important that organizations uniformly deploy their "best practice" methodologies in order to reap the awards of reuse. However, it is normally difficult to achieve so. For example, an organization may decide to develop testbenches using an OVM or VMM approach as described in this paper. If one of the verification teams in the organization mistakenly not utilize agents in their verification components, then this may diminish the ability to reuse this particular component in future testbenches. Another example could be that one of the verification teams does not use analysis ports in their scoreboard and once again diminishes easy reuse in other testbenches.

To overcome these deployment obstacles, we developed a Template Generator (TG) tool that could automatically generate a testbench based on templates. Figure 12 shows the flow of the TG Tool. We created a complete set of generic OVM and VMM templates to feed into the TG. These templates were implemented using our "best-practice" techniques for configuration, monitors, sequencers, sequences, drivers, agents, virtual sequences. The template generator builds up an entire OVM framework or testbench (i.e. OVM_testbench 1_*) that includes a makefile and a dummy test that allows teams to compile all the code out of the box using Synopsys, Cadence or Mentor simulators. The TG allows teams to control the name and number of verification components they want to generate.

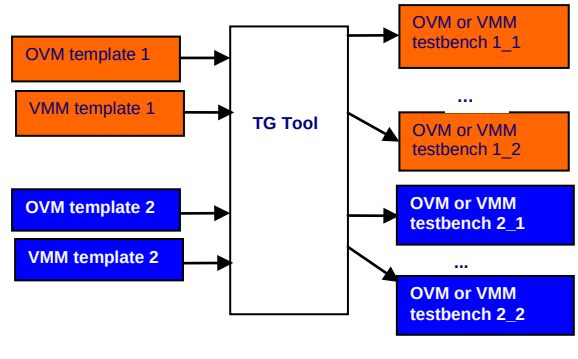

Figure 12 Template Generator Flow

Moreover, organizations may easily customize the templates for any

number of changes such as coding styles, naming conventions and copyright format in file headers. Using the TG truly deploys testbench code that has the same “look-and-feel” throughout the company. This significantly speeds up testbench development. This is especially true if the teams are attempting to learn a new methodology such as OVM or VMM. It helps bring the entire team up to speed using the new methodology. Last but not least, the TG is also capable of merging changes into previously generated code in the case where the teams decide to modify their “best practice” approaches.

6. CONCLUSION

There are two techniques for configuring a testbench: using the configuration mechanism and using the configuration class. Both choices are powerful and are supported by both OVM and VMM methodologies. However, as shown earlier, the configuration mechanism is more suited for configuring test and topology related parameters. In contrast, the configuration class is more suited for configuring design and verification related parameters. It is also recommended that a verification component should include only a single configuration class in order to ease maintenance and promote reuse.

The author also recommends that verification teams use the configuration choices in a consistent manner. As shown earlier, the use of templates to set up a testbench can help organizations quickly deploy "best-practice" code methodologies and reliably gets an entire team on the same page.

7. ACKNOWLEDGMENTS

A special thanks to my colleagues at Paradigm Works for their many technical debates on the best way to handle configuration. The ideas presented in this paper are based on numerous discussions, email threads, and conference calls I had with the following people.
Ambar Sarkar
Jeff Wilcox
Richard Musacchio
Ning Guo
Jack Collins

8. REFERENCES

[1] OVM User Guide Version 2.0.3 November 2009 OVM World http://ovmworld.org
[2] VMM Standard Library User Guide Version D-2009.12 December 2009 VMM Central http://www.vmmcentral.org
[3] Step-by-Step Functional Verification with SystemVerilog and OVM Sasan Iman Hassen
[4] System Verilog Template Generator Paradigm Works http://svf-tg.paradigm-works.com/svftg/