Lay it On Me: Creating Layered Constraints

Bryan Morris (Ciena),
Andrei Tarnauceanu (BTA Design Services)
bmorris@ciena.com, atarnauceanu@btadesignservices.com

***Abstract*- SystemVerilog provides a rich syntax to create complex constraints-based configuration and stimulus. However, when a system has many inter-dependent constraints that are bi-directional or have specific ordering it can be difficult for an engineer to understand the entire constraint space. Errors in understanding how these inter-related constraints lead to difficult debugging of constraint failures, or incorrect constraints silently providing an invalid solution. This paper describes a pragmatic technique to create "layers" from your existing constraints. These constraint layers reduce the state space that an engineer must keep track of -- making it easier to comprehend.**

I. Introduction

Today's verification environments are complex. Creating constraints is one area of complexity for systems with complex configurations with many inter-dependent variables i.e., when variable A has a dependency on variable B which has a dependency on variable C.

SystemVerilog's constraint language manages this complexity, and a constraint solver identifies and resolves these dependencies, or complains when it cannot. Sometimes the user must carefully create their dependencies to ensure the correct order for the solver; while at the same time avoiding bi-directional constraints, which can introduce difficult to debug constraint solver issues. Engineers use the `solve A before B` construct, a call to a function, or for some simulators use `$void()` to enforce an order and/or resolve constraint solver issues.

As the dependencies get larger, the complexity of these inter-relationships become increasingly harder to control and understand. The solver can easily keep a very complex constraint *state space* in its memory – but this is impractical for an engineer. This paper discusses a method to solve constraints in a *layered* manner. This layering forces the engineer to add constraints into a specific layer and then controls the constraint solver to solve Layer N before it solves Layer N+1, repeating until all layers are solved.

The remainder of this paper discusses the following:

- The challenge of complex, inter-dependent constraints.
- A brief review of previous solutions that address complex constraint management.
- A description of the solution including details of the SystemVerilog macros that use this solution.
- A review of what limitations and benefits this system brought, and any new challenges that layered constraints introduces.
- Briefly describe future enhancements to address the limitations.

History

Today's ASICs are very large with a diverse and complex set of features supporting a wide range of customers' requirements. Configuring ASICs has become commensurately more complex resulting in a large and complex *state space*. A typical simulation has several hundred random variables with hundreds, perhaps thousands of constraints the solver must consider.

To manage this complexity, engineers create a hierarchy of random variables and constraints. The first level of hierarchy is constrained to select a general mode of operation e.g., run in Ethernet mode. The next level of random variables selects a sub-mode, and so on until the entire hierarchy is solved. The constraint solver must consider all layers to calculate the final answer. Traversing through this decision tree each on subsequent hierarchical level would ultimately configure the ASIC in one legal configuration from the hundreds of possible legal configurations.

As is typically done, an engineer must define all the constraints and define an ordering through any of the following means:

- Natural uni-directional ordering: where one variable has one inter-dependent variable.
- Use of `solve A before B` SystemVerilog construct to enforce an ordering between two variables.
- Calling a function to do a calculation. The constraint solver must call all functions in a constraint before considering that variable in the solution space. When variable `A` is passed to a function, and its dependent variable `B` is not, then variable `A` is considered before `B`. As well, any side-effect from the function (e.g., setting a variable) must be done prior to solving the current random state space.

- Or using a non-LRM compliant `$void()` system call. It exists to enforce the ordering in the same way as the function call – except this function has no side-effects i.e., does nothing.

Challenges

The challenges with this complex configuration and a hierarchical approach are as follows:

- It was difficult for an engineer to understand the hierarchy and ordering of the *entire* state space.
- Bi-directional constraints can affect the ordering of the constraints.
- Frequent solver constraint errors introduced by introducing too many **`solve before`** constructs (or use of `$void()`).
- When constraints are not ordered correctly it can lead to invalid solutions, that are not flagged by the constraint solver
- Even with the simulator's constraint debugging capabilities it is sometimes difficult to debug constraint solver failures due to the interconnected state space.

Pre-Existing Solutions

There are existing solutions to create a structured constraint hierarchy:

1. `uvm_factory` overrides to capturing constraints in a derived class and then using polymorphism to select the appropriate constraint classes at run-time. This is the recommended approach for the UVM library.
2. John Dickol's excellent paper from DVCon 2015 entitled "*SystemVerilog Constraint Layering via Reusable Randomization Policy Classes*" [1]
3. Tudor Timi's use of Mix-in design pattern to provide an AOP behavior from his Verification Gentleman blog "*Do You Want Sprinkles With That? Mixing-in Constraints*" [2]: and he expanded on this "goodness" in a related post in March of 2020 entitled: "*Favour Composition Over Inheritance – Even for constraints*" [3]

Using the factory to control which constraints to apply is a standard UVM technique. Briefly, if you have constraints in a `config` class registered with the UVM factory, any class derived from that `config` class can create constraints that override or add to the base class' set of constraints. At run time you can ask the UVM factory to create your `config`-derived class instead of the base `config` class. When you randomize that `config` object, the solver applies the constraints defined by the all the `config` class' derivation tree.

Dickol's paper [1] uses a *Policy* design pattern (also known as a *Strategy Pattern* [4]) to group and control which constraints the solver uses. Briefly (and perhaps doing the paper an injustice), you place your constraints into a set of `policy` classes. Each class logically groups together the constraints into a class e.g., a `policy` for controlling the properties of your class, or the protocol behaviors for your sequence item. The object being randomized holds a set of handles to the `policy` objects selected. Each `policy` class provides the appropriate constraints to apply e.g., a policy to create very big and very small packets.

These `policy` classes can be *dynamically* additive (added to a set of other `policy` classes), or can override another policy class (removing it from the set). At run-time you can select which policy classes to apply to your class. There is no need to re-compile when you want to change constraints! In our opinion, the key benefit of the `policy` class is that it forces the engineer to group logically related constraints. It is a pragmatic and effective solution to control your constraints and create a *layered* set of constraints.

We highly recommend reading John Dickol's paper [1].

Timi's solution [2] and [3] is another great example of using a software technique to dynamically control constraints. The first blog post uses the Mixin design pattern [5]. A mix-in adds an Aspect-Oriented Programming style to adding or replacing constraints. The second blog post extends Dickol's solution eliminating the need to store the constraints in the `uvm_config_db`.

These blog posts are worthwhile to read (and you should also subscribe to Timi's Verification Gentleman's blog post, too). The concepts Tudor Timi presents are beyond the scope of this paper to describe adequately.

There are undoubtedly other solutions that we were unaware of that the verification community has developed to help manage their constraint complexity.

If these are so great… Why didn't we use them?

In our opinion, these solutions are all *superior* to the layered constraints presented in this paper *if your project is starting from scratch*. The policy or mixin techniques can effectively and dynamically control constraints. They also *enforce by design* a structured approach to creating your constraints. When designing a new project with new constraints, we recommend that you seriously consider these object-oriented approaches first.

However, if you already have several hundred variables and constraints these solutions require a *significant* restructuring of your constraints into separate classes and adding all the framework needed to control which constraint class to use. Again, this would be worthwhile if you had sufficient time in your schedule to rework and test these structures. Many projects do not.

The *layered constraints* mechanism described in this paper is a *pragmatic*, non-OO solution. Layered constraints require minimal changes to your existing constraints (i.e., removing any ordering constructs used). The layered constraint mechanism controls the randomization by creating information that *overlays* the existing constraints. That is, it creates a control layer orthogonal to the constraints enforcing an order in which each constraint in the layer is selectively presented to the constraint solver.

It's arguable that it is simpler to understand and control these layered constraints than the OO techniques presented above. The key advantages of these *layered constraints* are:

- they concisely define the layering -- breaking the *state space* into understandable chunks— making it easier for the engineer to understand what is happening in each layer, and
- creates a *static* ordering that is easier to debug compared to determining which constraint was active when you add constraints dynamically through the OO techniques.

However, using static ordering requires a recompile when you change the ordering. It is also a maintenance issue when you add a layer in the middle. This issue is removed as discussed in the Named Layers section below.

II. LAYERED CONSTRAINTS

Operation

Layered constraints can be best explained with a simple example that picks two random variables that belong to layers 1 and 2 -- where the value of the variable in layer 2 is dependent on the value of the variable in layer 1.

Figure 1 shows a typical example of a bi-directional constraint. `m_device_mode` (line 6) is the mode of the device and `m_slice_mode` (line 7) is the mode of each slice in the device and depends on `m_device_mode`. Each of these variables have corresponding constraints `device_mode_c` (lines 9-11) and `slice_mode_c` (lines 13-20), respectively.

```
1  typedef enum {DEV_MODE1, DEV_MODE2, DEV_MODE3} e_device_mode;
2  typedef enum {SLICE_MODE_1_1, SLICE_MODE_1_2, SLICE_MODE_1_3,
3                SLICE_MODE_2_1,
4                SLICE_MODE_3_1, SLICE_MODE_3_2}  e_slice_mode;
5  class config_device;
6    rand e_device_mode m_device_mode;    //device mode
7    rand e_slice_mode  m_slice_mode[4];  //per slice mode
8
9    constraint device_mode_c {
10      m_device_mode inside {DEV_MODE_1, DEV_MODE_2 };
11   }
12
13   constraint slice_mode_c {
14     foreach (m_slice_mode[sl]) {
15       (m_device_mode == DEV_MODE_1) -> m_slice_mode[sl] inside
16             {SLICE_MODE_1_1, SLICE_MODE_1_2, SLICE_MODE_1_3};
17       (m_device_mode == DEV_MODE_2) -> m_slice_mode[sl] inside
18             {SLICE_MODE_2_1};
19     }
20   }
21  endclass : config_device
```

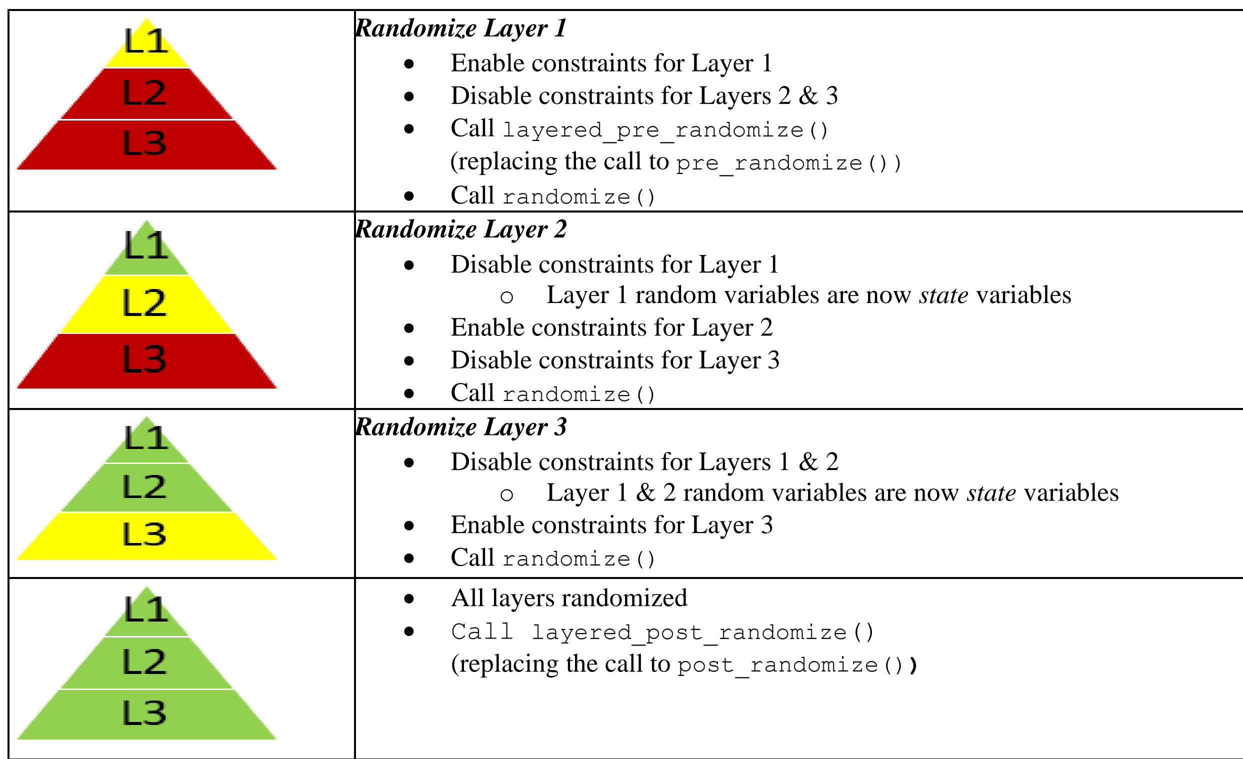
Figure 1 - Typical bi-directional constraint

Even this simple example illustrates the difficulty when creating an ordering of constraints that contain bi-directional constraints (i.e., the `slice_mode_c` implication constraint is bi-directional (lines 15 and 17)). If you don't define a randomization order, the constraint `slice_mode_c` picks `DEV_MODE_1` 75% of the time for `m_device_mode` as there are more options for the `m_slice_mode`.
To get an even distribution of the two device modes (`DEV_MODE_1` and `DEV_MODE_2`) you must break the bi-directionality of the constraint so `m_device_mode` is resolved first by the constraint solver. Before *layered constraints,* this was done using $**void(**`m_device_mode`**)** or **solve** `m_device_mode` **before** `m_slice_mode`. These solutions should be avoided: **$void** cannot be used across simulators, while **solve before** can create difficult to debug constraint issues for a large constraint space.

Layered constraints assign all random variables and their associated constraints into multiple randomization layers and then presents each layer's constraints to the constraint solver in a sequential manner -- solving Layer N before solving Layer N+1.

The flow is shown in the following diagram in Figure 2, where an engineer creates and randomizes 3 layers of constraints. Each section of the pyramid represents one layer of constraints:

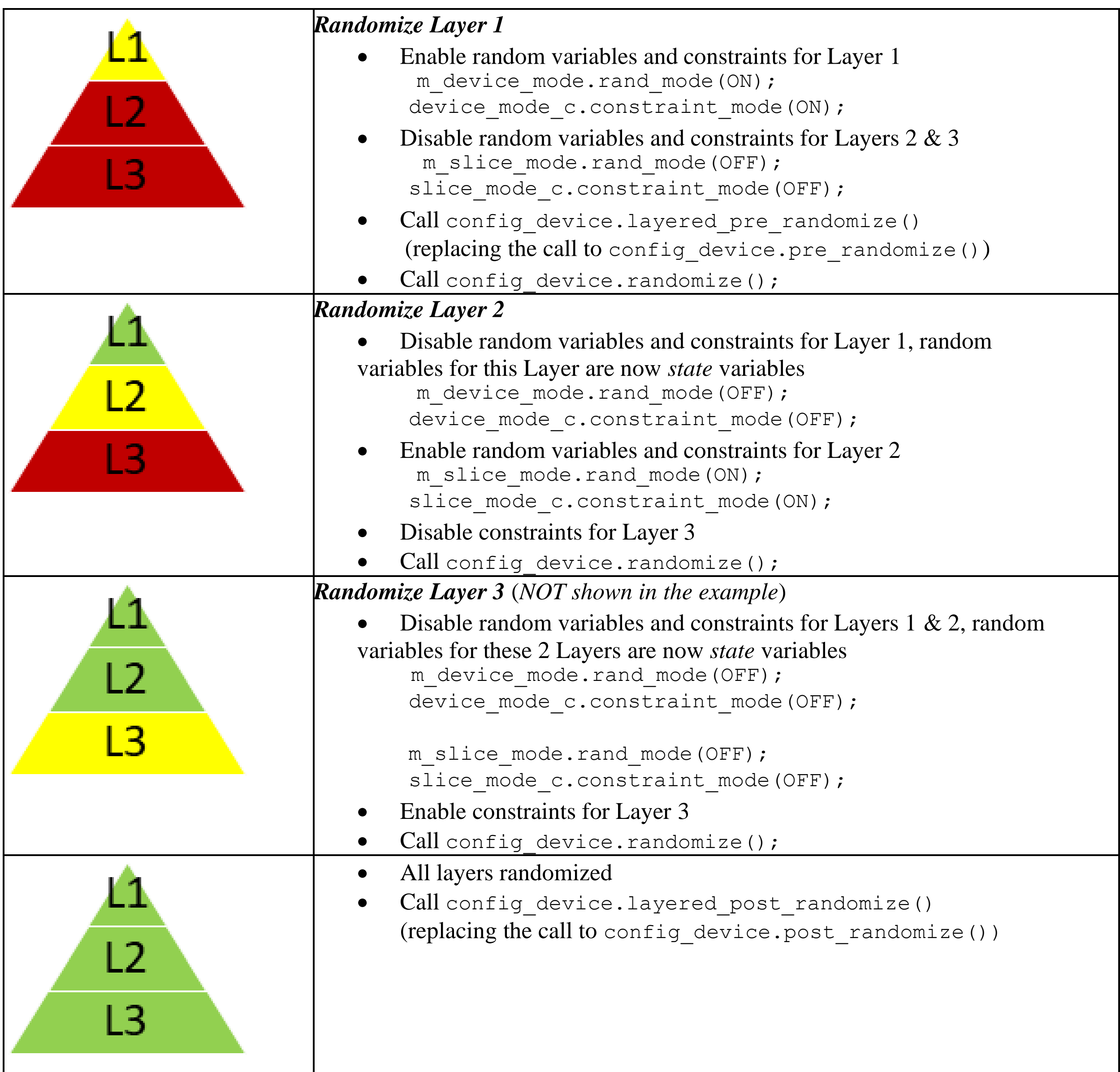
- Red Layers have their `.constraint_mode/.random_mode` disabled, and their random variables are not **yet** randomized.
- Yellow layers have their `.constraint_mode/.random_mode` enabled, and their random variables are **currently** being randomized.
- Green layers have their `.constraint_mode/.random_mode` disabled, and their random variables have been **previously** randomized i.e., they are now fixed values.



L1 L2 L3	***Randomize Layer 1*** • Enable constraints for Layer 1 • Disable constraints for Layers 2 & 3 • Call `layered_pre_randomize()` (replacing the call to `pre_randomize()`) • Call `randomize()`
L1 L2 L3	***Randomize Layer 2*** • Disable constraints for Layer 1 o Layer 1 random variables are now *state* variables • Enable constraints for Layer 2 • Disable constraints for Layer 3 • Call `randomize()`
L1 L2 L3	***Randomize Layer 3*** • Disable constraints for Layers 1 & 2 o Layer 1 & 2 random variables are now *state* variables • Enable constraints for Layer 3 • Call `randomize()`
L1 L2 L3	• All layers randomized • `Call layered_post_randomize()` (replacing the call to `post_randomize()`)

Figure 2 - Layered Randomization Flow

Using the example from Figure 1, `m_device_mode/device_mode_c` is assigned to Layer 1 and `m_slice_mode/slice_mode_c` is assigned to Layer 2 in order to decide the value of the device mode before the slice mode and thus giving a 50% chance for each one of the device modes to be picked by the constraint engine (see Figure 3 below).



L1
L2
L3
	Randomize Layer 1
	• Enable random variables and constraints for Layer 1
	`m_device_mode.rand_mode(ON);`
	`device_mode_c.constraint_mode(ON);`
	• Disable random variables and constraints for Layers 2 & 3
	`m_slice_mode.rand_mode(OFF);`
	`slice_mode_c.constraint_mode(OFF);`
	• Call `config_device.layered_pre_randomize()`
	(replacing the call to `config_device.pre_randomize()`)
	• Call `config_device.randomize();`

L1
L2
L3
	Randomize Layer 2
	• Disable random variables and constraints for Layer 1, random variables for this Layer are now *state* variables
	`m_device_mode.rand_mode(OFF);`
	`device_mode_c.constraint_mode(OFF);`
	• Enable random variables and constraints for Layer 2
	`m_slice_mode.rand_mode(ON);`
	`slice_mode_c.constraint_mode(ON);`
	• Disable constraints for Layer 3
	• Call `config_device.randomize();`

L1
L2
L3
	Randomize Layer 3 (*NOT shown in the example*)
	• Disable random variables and constraints for Layers 1 & 2, random variables for these 2 Layers are now *state* variables
	`m_device_mode.rand_mode(OFF);`
	`device_mode_c.constraint_mode(OFF);`
	
	`m_slice_mode.rand_mode(OFF);`
	`slice_mode_c.constraint_mode(OFF);`
	• Enable constraints for Layer 3
	• Call `config_device.randomize();`

L1 L2 L3	• All layers randomized • Call `config_device.layered_post_randomize()` (replacing the call to `config_device.post_randomize()`)

Figure 3 - Layered Constraint Randomization Example

There are a few rules that should be followed when using this method.

- The `post_randomize()` function must be empty since `.randomize()` is called N times (N=number of layers) and procedural code that might decide some final values of certain variables should be executed only once when the final `.randomize()` is called. This is done by implementing all the functionality that would usually be part of `post_randomize()` into a new function called `layered_post_randomize()`. This new function is called **after** the FINAL layer N has been randomized -- making it similar in behavior to the normal `post_randomize()` function.
- Similarly, a `layered_pre_randomize()` function is called ***before*** the FIRST layer is randomized. Again this is similar behavior to the existing `pre_randomize()` function.
- To keep the flow easy to understand and follow, each constraint should solve the value of one random variable. When using this method there should not be multiple random variables considered in the same constraint. This separation makes the control easier. (NOTE: This is a good practice, in general).

- This method allows 1 or more random variables to be part of the same layer, but the user *must* make sure that the functionality of the variables that are part of the same layer are mutually independent. Otherwise one might end up with bi-directional constraints that are part of the same layer defeating the purpose of the layering.

Figure 4 below presents a code snippet illustrating how to achieve layered randomization of the `config_device` class from Figure 1 above. First, create a `layered_config_device` derived class to control when each variable's `.rand_mode()` and each constraint's `.constraint_mode()` are toggled on and off (lines 6,10 for the variable control; lines 7 and12 for the constraints) in the `toggle_layered_constraints()` function. When the supplied `toggle_layer` argument is the same as the assigned layer e.g., hard-coded on line 6 as layer 1 and checked `(toggle_layer = 1)`) then the variable's `.rand_mode()` is enabled; otherwise it is disabled. Use a similar control mechanism for the constraint's `.constraint_mode()`.

The `layered_based_test` testcase creates the configuration (line 23) and then randomizes each layer by calling passing each layer to the `toggle_layered_constraints()`.

This code is tedious and error-prone 'boiler-plate' code. Final implementation is done using SystemVerilog macros (see next section) that replace all the procedural code from Figure 4 with declarative code.

```
1  class layered_config_device extends config_device;
2    // set the rand_mode,constraint_mode for variables based on the toggle_layer value
3    // - mode is ON if togle_layer is equal to layer assigned to the var/constraint
4    // OFF otherwise
5    virtual function void toggle_layered_constraints(int toggle_layer);
6      m_device_mode.rand_mode( (toggle_layer == 1 ? 1 : 0) );
7      m_device_mode_c.constraint_mode( (toggle_layer == 1 ? 1 : 0) );
8
9      foreach (m_slice_mode[sl]) begin
10       m_slice_mode[sl].rand_mode( (toggle_layer == 2 ? 1 : 0) );
11     end
12     slice_mode_c.constraint_mode( (toggle_layer == 2 ? 1 : 0) );
13
14     //… layer 3 code
15    endfunction : toggle_layered_constraints
16   endclass : layered_config_device
17
18  //testcase creates a handle to the config_device class and randomizes each layer
19  class layered_based_test extends uvm_test;
20    layered_config_device m_config_device;  //device configuration
21
22    virtual function void build_phase();
23      m_config_device = layered_config_device::type_id::create();
24
25      //for each one of the 3 layers:set the rand_mode/constraint_mode for
26      // variables depending on the layer they belong to using
27      // toggle_layered_constraints, call pre layered randomize for first layer,
28      // call randomize() function of the config class
29      for(int current_layer = 1; current_layer <= 3; current_layer++) begin
30        m_config_device.toggle_layered_constraints(current_layer);
31        if(current_layer == 1) m_config_device.layered_pre_randomize();
32        if(!m_config_device.randomize()) `uvm_fatal("randomization failed");
33        // …and call post layered randomize for last layer
34        if (current_layer == 3) m_config_device.layered_post_randomize();
35      end
36   endfunction : build_phase
37  endclass : layered_base_set
```

Figure 4 - Layering Constraint Toggling Code

SV Macros

This section describes the SystemVerilog macros replacing the boilerplate code controlling the layer randomization from Figure 4.

Declaring Layered Variables/Constraints

These macros are modelled on the `uvm_field_macros` with `begin` and `end` markers and assigning each variable and constraint to a layer. From the example above, the following snippet assigns the variable `m_device_mode` (from class `common_cfg`) to layer 1, with the associated default constraint assigned to the same layer.

```
1  `layered_constraints_begin(common_cfg )
2    `layered_variable(   1, m_device_mode )
3    `layered_constraint( 1, device_mode_c )
4  `layered_constraints_end
```

Expanding[1] the begin macro into the following code snippet. The `layered_constraints_begin` starts the definition of a class function `toggle_layered_constraints()`:

```
1  `define layered_constraints_begin(CLASS_NAME) \
2    `layered_constraints_decl__                 \
3     virtual function void toggle_layered_constraints(toggle_layer_t toggle_layer); \
```

On line 2 of the `layered_constraints_begin` macro declare class member variables to control the randomization (using the `layered_constraints_decl` macro):

- `m_current_layer` is the layer currently being randomized
- `m_ignore_these_constraints` holds the set of constraints that can be disabled (or ignored) when randomization happens.

```
1  `define __layered_constraints_decl__     \
2    toggle_layer_t m_current_layer;       \
3    static string m_ignore_these_constraints_q[$];
```

The macros between the begin and end define the functionality for this `toggle_layered_constraints()` function. For example, when declaring a `layered_constraint` it adds code to control the supplied constraint

- line 2 allows control when a constraint is *always* disabled.
- line 6 controls the enable/disable of the `.constraint_mode()` for the layer being randomized.

```
1  `define layered_constraint( LAYER, CONSTRAINT_NAME )  \
2    if ( is_constraint_ignored(`STRINGIFY(``CONSTRAINT_NAME``)) == TRUE )begin\
3      … debug code snipped.
4    end  \
5    else begin \
6      ``CONSTRAINT_NAME``.constraint_mode((toggle_layer == ``LAYER`` ? 1:0));\
7      … debug code snipped
8    end
```

The following declaration…

```
`layered_constraint( 1, m_device_mode_c )
```

… expands to the following code inside the `toggle_layered_constraints` function

```
1  if ( is_constraint_ignored("device_mode_c") == TRUE ) begin
2     ... debug code snipped
3  end
4  else begin
5    device_mode_c.constraint_mode((toggle_layer == 1 ? 1:0));
6    ... debug code stripped.
7  end
```

---

[1] All the exploded macros exclude the debug code for clarity.

On line 1, check whether this constraint was disabled i.e. to be ignored for this solution. Then either enable the constraint if the currently layer being randomized is layer 1 (the layer this constraint was assigned to) (line 6 in this snippet is identical to line 7 in Figure 4).

As each layer is randomized the class' `toggle_layered_constraints` is passed the layer number via the `toggle_layer` argument. The `layered_variable` macro is identical to the `layered_constraints` macros above, except it controls the variable's `rand_mode` i.e., when the variable's assigned layer is being randomized the variable's `rand_mode` is enabled; otherwise, it is disabled.

To close off the declarative macros, the `layered_constraints_end` macro:

```
1  `define layered_constraints_end \
2     endfunction : toggle_layered_constraints \
3  \
4     virtual function boolean_t is_constraint_ignored( input string constraint_name ); \
5       return( constraint_name inside {m_ignore_these_constraints_q} ? TRUE : FALSE ); \
6     endfunction : is_constraint_ignored
```

This macro closes the `toggle_layered_constraints` function (line 2). It adds the `is_constraint_ignored` (lines 4 to 6) function to return TRUE if the supplied constraint has been disabled for this solution i.e. ignored, or FALSE if the constraint should not be ignored.

We discuss above how to declare and assign a layered variable and constraint, and how the macros control each of these as each layer is randomized. We'll now discuss the macro that iterates through each layer and randomizes each: `` `RANDOMIZE_LAYERS ``

```
`define RANDOMIZE_LAYERS( OBJ_HANDLE, NUM_LAYERS, START_LAYER=1, DEBUG=TRUE )\
```

This macro accepts two mandatory arguments and two optional arguments:

`OBJ_HANDLE`

the handle to the object holding the layered constraints.

`NUM_LAYERS`

The number of layers you want to randomize. This provides the option do a partial randomization of all layers[2].

`START_LAYER`

Defines which layer you want to *start* randomizing. This in combination with the `NUM_LAYERS` could allow randomization of a sub-set of variables[3].

`DEBUG`

Enables the debug output (discussed in Debugging Layered Constraints below).
NOTE: the debug code for this macro is not shown for clarity

Similar to lines 29 to 32 in Figure 4, the macro iterates through each layer by calling the class` `toggle_layered_constraints` function followed by the call to the class' randomize function, repeatedly.

```
1   for ( ``OBJ_HANDLE``.m_current_layer=``START_LAYER``;
2    ``OBJ_HANDLE``.m_current_layer <= ``NUM_LAYERS``;
3    ``OBJ_HANDLE``.m_current_layer++ ) begin
4    ``OBJ_HANDLE``.toggle_layered_constraints(``OBJ_HANDLE``.m_current_layer );
5     if ( ``OBJ_HANDLE``.randomize() == 0 ) begin
6        ... uvm_fatal generated
7     end
8  end
```

The next section provides an example of how to use these macros.

---

[2] Never used this option -- an example of adding code that turned out be not as useful as originally thought.

[3] Never used this one either.

Use-Model

Figure 5 below shows the code in Figure 4 - Layering Constraint (lines 6-15) using the layered constraints macros:

- declare `m_device_mode, device_mode_c` in Layer 1,
- declare `m_slice_mode, slice_mode_c` in Layer 2,
- and Layer 3 variables that are outside the scope of this example.

```
1  class layered_config_device extends config_device;
2    `layered_constraints_begin( config_device )
3      `layered_variable(    1, m_device_mode)
4      `layered_constraint(  1, device_mode_c)
5
6      `layered_variable_array( 2, m_slice_mode)
7      `layered_constraint(     2, slice_mode_c)
8
9      `layered_variable(    3, m_foo_mode)
10     `layered_constraint(  3, foo_mode_c)
11   `layered_constraints_end
12
13  endclass : layered_config_device
```

Figure 5 - Declaring Layered Constraints Macros

Figure 6 below shows the base test code from Figure 4 - Layering Constraint (lines 29-35) using the `` `RANDOMIZE_LAYERS `` macro.

```
1  class layered_based_test extends uvm_test;
2    layered_config_device m_config_device;  //device configuration
3
4    virtual function void build_phase();
5      m_config_device = layered_config_device::type_id::create();
6
7      `RANDOMIZE_LAYERS( m_config_device, 3 )
8    endfunction : build_phase
9 endclass : layered_base_test
```

Figure 6 - SV Snippet RANDOMIZE_LAYERS

Thus far it has been shown how to implement the layered constraints in the environment base classes, following this use model.

- Create the normal set of random variables and associated constraints
- Allocate random variables and their constraints to relevant randomization layer.
- Use the `` `layered_variable_* `` and `` `layered_constraints_* `` macros to assign these variables and constraints into their appropriate layer.
- Randomize all layers using `` `RANDOMIZE_LAYERS `` macro

The next challenge comes when the user needs to write a testcase using derived classes with the layering approach where the test must supply the test-specific constraints. Instead of using the `` `layered_constraints_begin `` the user must use `` `layered_constraints_test `` macro. That is, the test-case uses `` `layered_constraints_test `` for all derived classes to add / remove test-specific constraints to the layers - building on the layered constraints in the base class declared by the `` `layered_constraints_begin `` macro. This is necessary since all derived classes must call a `super()` function in the base class, but when you're in the base class you can't call a `super()` of this function since it does not exist.

Figure 7 shows an example of how this can be achieved by creating a derived class from `layered_config_device` base class and adding a new constraint for `m_device_mode` in the testcase.

```
1  class test_layered_config_device extends layered_config_device;
2
3    constraint test_device_mode_c {
4      m_device_mode inside {DEV_MODE_1};
5    }
6
7    `layered_constraints_test( config_device )
8      `layered_constraint(1, test_device_mode_c)
9    `layered_constraints_end
10
11  endclass : test_layered_config_device
```

Figure 7 - Example Testcase Layered Constraint Control

Debugging Layered Constraints

Most of the code in the macros are to assist in the debug of the constraints. The debug code added into the macros is not shown so that we could focus on the functional behavior of the layered constraints mechanism. The main debug mechanism is a separate log file created for the layered constraints. Before randomizing each layer, print out the following information:

- All the variables defined along with an indication of whether its `.rand_mode()` is enabled or disabled for layer currently being randomized.
- All the constraints and their current value for `.constraint_mode()`.

After the layer has been randomized, print the current values of object being randomized (using the `uvm_field*` macros default print mechanism). This information has been found useful in watching the results of the randomization for each layer as the object changes value. Since the format is consistent for each layer, it is was effective to use a `diff` tool to show differences between the layers. This diffing made it easy to determine in which layer the incorrect randomization occurs.

Limitations

Once the layered constraints are declared and all random variables have been assigned to a layer, the verification team writes testcases by adding layered constraints in the derived classes. It is typical that new features are added, or features get modified late in the project. This may require adding new random variables inserted in a new layer between two existing layers. For example, if a new random variable needs to be inserted at level N, all variables that belong to layer N+1 will now be part of layer N+2, and the total number of layers will increase from M to M+1.

As the layers are assigned by hard coding the layer number into each macro, new layers require all the layers above the new one to be incremented by 1, and all testcases need to change to align with the new layers. The next section proposes an update to address this problem.

Future Work

Named Layers

To address the key limitation due to the *static* ordering creating maintenance issue i.e., when new layers needs to be added, we are considering creating "named layers".

Instead of grouping layers by number, group them by a unique name. Allowing the engineer to group all logically related constraints together i.e., those that must be solved together. Then separately hold the set of named layers that define the order each layer is randomized. For example, creating a simple ordered queue to hold the following layer names:

```
Common_Constraints
ModeSelection_Constraints
SliceSelection_Contraints
Remaining_Constraints
```

These *named layers* group together constraints, and the order in the queue defines which layer is presented to the solver during randomization i.e., `Common_Constraints`, then `ModeSelection_Constraints` and then `SliceSelection_Constraints`.

When a new layer is added, create a new name and insert it into the queue in the correct order:

```
Common_Constraints
ModeSelection_Constraints
SliceSelection_Contraints
NewLayer_Constraints
Remaining_Constraints
```

Now, present the `NewLayer_Constraints` to the solver after the `SliceSelection_Constraints`.

The reader may realize that by separating the grouping of constraints from the ordering enables the following actions during run-time:

- control which set of *named layers* are active by removing the *named layer* from the ordered list.
- inject new *named layers* (but they must be compiled), by adding a *named layer* into the queue from run-time information supplied.
- Re-order the named layers entirely (but we have not determined a use-case for this, yet).

Improve Debug

As discussed in Section Debugging Layered Constraints above, debugging layered constraints is a post-simulation analysis of a log file. While effective for regression analysis to determine which constraints were applied, we are considering the following improvements:

- Adding a meaningful description for each layer and each variable/constraint. This information will be logged when the constraint is presented to the solver. Assuming they convey the *intent* of the constraint, this information provides a complete picture of what each layer is intending to do for engineers who are not familiar with the *intent*.
- All the debug information is currently output to a single file. This made the `diff`'ing within the same file for value changes between layers a little more challenging. In the future, each layer will be output to a separate file – making the `diff`'ing between files simpler.

III. CONCLUSIONS

This paper offers a solution for complex random configurations when it is hard for the verification engineer to keep track of the dependencies between variables and the implications of bi-directional constraints that can be hard to debug and can reduce the randomization space.

Layered constraints try to break down and simplify the dependencies between variables and eliminate the use of bi-directional constraints. Layered constraints are easier for engineers to understand as it reduces the constraint state space to a more comprehensible size. The layering forces the engineer to carefully consider which layer they add constraints into; making it quicker to debug because they are explicitly aware of which layer it is assigned. Lastly, this concept reduces (or ideally eliminates) the need to use `$void()` and **`solve before`** that are the standard way of dealing with complex random configurations but have shortcomings as described in this paper.

This solution uses SystemVerilog macros to make it easier to declare each layer and to control the randomization in tests. These macros hide all the tedious boiler-plate code required for the layered randomization – making them clearer and less error-prone.

IV. REFERENCES

[1] J. Dickol, "System Verilog Constraint Lyaering via Reusable Randomization Policy Classes," 2015. [Online]. Available: http://events.dvcon.org/2015/proceedings/papers/04P_11.pdf.

[2] T. Timi, "Do You Want Sprinkles with That? - Mixing in Constraints," March 2015. [Online]. Available: https://blog.verificationgentleman.com/2015/03/mixing-in-constraints.html.

[3] T. Timi, "Favour Composition Over Inheritance - Even for Constraints," Marh 2020. [Online]. Available: https://blog.verificationgentleman.com/2020/03/composition-for-constraints.html.

[4] Wikipedia, "Srategy Pattern," 2020. [Online]. Available: https://en.wikipedia.org/wiki/Strategy_pattern.
[5] Wikipedia, "Mixin," 2020. [Online]. Available: https://en.wikipedia.org/wiki/Mixin.