Configuration Conundrum: Managing Test Configuration with a Bite Sized Solution

Kevin Vasconcellos, kevin.vasconcellos@verilab.com
Jeff McNeal, jeff.mcneal@verilab.com

Verilab, Inc.
2121 Lohmans Crossing Road - Suite 504
Lakeway, TX 78734 USA

Abstract: The authors will show the reader how to take advantage of a design pattern called the Policy Pattern to make configuration of a UVM testbench easier for the user to modify. Using this pattern will allow test writers to quickly and easily modify the configuration of the DUT or testbench while also providing at-a-glance information about what configuration a test is using. This technique eliminates the need to know some testbench details: how the components of the testbench work; how components are built; and how to write complex constraint blocks This isolation also allows the testbench writers to modify how the internals of the testbench components are coded without disrupting tests or the test writers.

I. Introduction

Many projects have a dizzying array of constraints and/or overrides littering the tests. The tests have become victims of cut-and-paste code reuse and its inevitable pitfalls. New team members need to learn a confusing system to write new tests and copying the wrong exemplar test can lead one down a lengthy debugging process.

In some cases, configuration objects have many options that need to be accounted for in order to properly configure a DUT and testbench. These options are necessary to give users the ability to control configurations from their tests. When the assorted options are inter-dependent, cases can arise where there are many helper variables to manage constraints. Some options will limit the available choices of other options. Values for some options may be influenced by one or more other options.

Attempting to build a complete set of children of the configuration object to allow for all the possible combinations of the configuration options leads to a situation with many classes that are substantially the same. One would have to make any constraint modifications to all the classes that interact with those variables. This would also present the user with a dizzying array of classes to choose from, and they would have to carefully select the desired class from those that are implemented already or build their own classes.

When faced with this type of configuration conundrum, it would be nice to be able to add small blocks of constraints, or, even more interesting, subtract a block of constraints. Basic OOP and the way that constraints are implemented in SystemVerilog don't allow us to subtract constraints.

Wouldn’t it be great if a test could configure the testbench like this?

```
// Policies used in this test
env.test_cfg.policy_list.rm(env.test_cfg.german);   // don’t select for German cars
env.test_cfg.policy_list.add(env.test_cfg.american);
env.test_cfg.policy_list.add(env.test_cfg.luxury);
env.test_cfg.policy_list.add(env.test_cfg.manual)
env.test_cfg.policy_list.add(env.test_cfg.midwest);
```

Figure 1: The Test

Well, it can. Here are the steps.

II. PLANNING

Premise

Policies have been written about before. Both Dickol [1][2][3] and Nelson [4] introduce the policy concepts and apply them to sequence items. While not explicitly stated by Dickol or Nelson, a lightbulb moment occurred when the authors realized that one can use policies ***anywhere*** an object is randomized.

This technique is a way to package constraints into reusable "randomization policy" objects and to use one or more of them while randomizing an object [1]. Policies take advantage of what the SystemVerilog LRM calls "global constraints" which Dickol points out can be more intuitively called "hierarchical constraints" [2][3]. Policies can be reused anywhere the object they are associated with is used. If that object can be reused vertically in a testbench or in a new testbench, then the policies are also reusable there.

This paper will go beyond sequence items and sequences and apply policies to handle test configuration.

Choosing Configurations

The first step is to decide what configuration options to implement as policies. Not all options need to be policies, and not every option has to be converted to use the policy pattern. It is suggested to start with configuration features that are often enabled when debugging tests, since those will tend to be the things that users want to turn on for debugging, then turn off again for regressions. For example, a mode that shortens simulation time by reducing timeouts or counters is often invoked for speeding up debug. Our projects' testbench had a scrambling feature that made debugging waveforms vastly more complicated and frustrating, but rarely was the source of issues, so it was often disabled for new development and debug. Other good candidates are features that are simple conceptually but have effects on several places in the configuration hierarchy, such as selecting the number of lanes a protocol will use.

Picking Policies

Armed with a set of configuration options to be turned into policies, you'll next have to figure out what the user options will be. Since each policy class will hold a single constraint or set of constraints, a policy class will be written for each configuration option selected. Some configurations are binary and are just enabled or disabled. Some configurations control how a choice is made from a set. Still others constrain continuous variables to a range.

For many configurations, the user will only get the option to enable, disable, or randomize the configuration. In this case, write a policy class to constrain the feature to be enabled and write a policy class to constrain the feature to be disabled. If the user does not select either of those policies, then the option would be randomized by default. For example, a car's headlights may be either on or off, so we could say enabled or disabled.

Policies can be especially useful when a user has a group of things to choose among, and it is possible to select zero or more choices. Consider, for example, choosing from a set of car makes [Chevy, Ford, Cadillac, Honda, Lexus, VW, Mercedes]. The user may want to constrain the set to only include European makes, or they may want to exclude American cars. The testbench writers don't need to supply an exhaustive set of policies but may choose to implement them for the most common choices. For example, a policy to select domestic brands only, or to exclude luxury brands may be sufficient for most needs. Other policies may be added to the feature later. Not using any of the feature policies will leave the set unconstrained and completely randomized.

Some configuration controls will be setting thresholds on a range for a variable. An example is temperature control which can be set to keep the temperature above or below a specific setpoint, or between two limits.

Now armed with a list of policies, it is time to start implementing them.

Remember, the use of policies implies a shift in mindset: Legality constraints are still in the object being randomizing. Any added constraints that further constrain the object inside the legal space are now placed into policies. In other words, policies impose more direction to our object randomization.

III. PROCEDURE

The Policy Base Class

Before implementing the policy classes for each configuration option, a bit of common policy boilerplate code is needed. One piece is a base class for the policy classes themselves to extend, as shown in Figure 1, and a policy list class to organize the policies in Figure 2.

```
class policy_base #(type ITEM=uvm_object) extends uvm_object;
  rand ITEM item;

  function new(string name = "policy base"); <...>

  virtual function void set_item(ITEM item);
    this.item = item;
  endfunction : set_item
endclass : policy_base
```

Figure 1: Policy Base Class

```
class policy_list #(type ITEM=uvm_object) extends policy_base #(ITEM);
  rand policy_base #(ITEM) plist[$];

  function new(string name = "policy list"); <...>

  function void set_item(ITEM item);
    foreach(plist[i]) plist[i].set_item(item);
  endfunction : set_item
endclass : policy_list
```

Figure 2: Policy List Base Class

Now, use the templates to typedef the policy list and policy base for the configuration class, test_config. The code in Figures 1 and 2 is generic and could come from a project or common code library package. The typedefs allow specialization specifically for the test_config class. The typedef name also provides a reminder of what class a policy is associated with, especially when multiple objects have their own policies.

```
typedef class test_config;
typedef policy_base#(test_config) test_cfg_policy_base;
typedef policy_list#(test_config) test_cfg_policy_list;
```

Figure 3: Policy Typedefs for Config

The Policy Classes

Next, implement a set of policy classes. It is recommended to completely implement a set of policies for a single configuration option first, to make sure all the kinks are worked out. A pair of enable/disable policies is an excellent way to start.

The policy class itself is quite simple in most cases. It is just a container for a set of constraints on the object that is referenced by item in the policy base class. When coding the policy, remember to remove the corresponding constraints from the configuration object itself, so there are not conflicts when trying to randomize.

The example in Figure 4 below implements two policies; one to enable a car's headlights, and one to disable them. The constraints in this case are fairly trivial. In our project, a similar set of policies replaced a constraint in the configuration object that was conditional upon a helper variable. Since these policies are now mutually exclusive, the helper variable and conditional constraints become trivial constraints.

```
class disable_headlights_pcy extends test_cfg_policy_base;
  function new(string name = " disable_headlights_pcy"); <...>

  constraint disable_headlights_c { item.headlights_on == 0; }
endclass

class enable_headlights_pcy extends test_cfg_policy_base;
  function new(string name = "enable_headlights_pcy"); <...>

  constraint enable_headlights_c { item.headlights_on == 1; }
endclass
```

Figure 4: Disable and Enable Policies

The second example in Figure 5 illustrates a more complex set of constraints like a situation the authors had on a recent project. For that case there were three random variables that were related to each other, link speed, link encryption, and the type of the encryption. This policy is used to enable encryption on a protocol link. The style of encryption depended on what speed had been selected which was controlled elsewhere.

```
class enable_encryption_policy extends test_cfg_policy_base;
  <...>
  constraint enable_encryption_c {
    item.encrypt == 1;
    (item.selected_speed == GEN1) -> (item.encrypt_type1 == 1);
    (item.selected_speed == GEN2) -> (item.encrypt_type2 == 1);
  }
endclass
```

Figure 5: Enable Encryption Policy

The Enhanced Policy Classes

While working on our project, several enhancements were developed for the policy classes as detailed in Figure 6. For full code implementations, see the code on GitHub [7].

```
class policy_base #(type ITEM=uvm_object) extends uvm_object;
  string name;
  rand ITEM item;
  function new(string name = "policy base");...
    this.name = name;
  endfunction : new
  ...
  virtual function string get_name();
    return this.name;
  endfunction : get_name
endclass : policy_base

class policy_list #(type ITEM=uvm_object) extends policy_base #(ITEM);
  ...
  // Enhancements
  function void add(policy_base #(ITEM) pcy);
    plist.push_back(pcy);
  endfunction : add

  function string get_policy_list(); // Get the list of policies

  function void print_policy_list(); // prints all policies in list

  function bit has_policy(string policy_name); // true if in plist

  // true if one pcy is in plist
  function bit has_any_policy(string policy_names[$]);
```

```
  function void rm(string policy_name); // removes single policy

  function void flush(); // empties policy list

  function string convert2string(); // string of all policies in plist

  function int get_policy_index(string policy_name); // index in plist of pcy
endclass : policy_list
```

Figure 6: Enhanced Policy Classes

Putting it all together

Now that all these policies are coded, they need to be put somewhere so the test can use them. Since the policies are configuration policies, they are added to the configuration object as shown in Figure 7. Add a random instance of the policy list class created above. Don't apply any constraints to the policy_list instance as that would cause it to be trashed by randomization! As a convenience, instantiate a single instance of all the policies. This is not necessary, but it makes things simpler for the test writers, as they do not have to declare and instantiate any of the policies that they want to make use of. Note that the constraints are not applied to the test_config randomization unless they have been added to the policy_list, so they are not applied just by creating an instance. Figure 7 also shows the instantiation of the policy_list and the instance creation, via new(), of the various policies. The plist string is used as a convenience variable to show a user in the run log which policies are available for the test_config object. The final, crucial step is to add the call to policy_list.set_item(this) in pre_randomize(). The use of an enhanced method is shown in the post_randomize() method below.

```
class test_config extends uvm_object;
  rand car_make_t     my_car_make;
  rand bit            headlights_on;
  rand int            temp_setting;
  <...>
  string plist;
  rand test_cfg_policy_list policy_list;

  enable_headlights_pcy  lights_on;
  disable_headlights_pcy lights_off;
  automatic_trans_pcy    automatic_trans;
  manual_trans_pcy       manual_trans;
  american_car_pcy       american;
  german_car_pcy         german;
  luxury_car_pcy         luxury;
  japanese_car_pcy       japanese;
  midwest_mix_pcy        midwest;
  <...>

  // Legality constraints
  constraint temp_setting_c {
    temp_setting < 999;
    temp_setting > 0;
  };
  ...

  function new(string name = "");
    super.new(name);
    // Create the queue to hold policies
    policy_list = new();

    // Create instances of all policy objects
    lights_on = new("lights_on");
    lights_off = new("lights_off");
```

```
    automatic_trans = new("automatic_trans");
    manual_trans = new("manual_trans");
    american = new("american");
    german = new("german");
    japanese = new("japanese");
    luxury = new("luxury");
    midwest = new("midwest");

    // build a string to print the policy names to the log
    plist = $sformatf("%0s  %0s\n", plist, lights_on.get_name());
    plist = $sformatf("%0s  %0s\n", plist, lights_off.get_name());
    plist = $sformatf("%0s  %0s\n", plist, automatic_trans.get_name());
    plist = $sformatf("%0s  %0s\n", plist, manual_trans.get_name());
    plist = $sformatf("%0s  %0s\n", plist, american.get_name());
    plist = $sformatf("%0s  %0s\n", plist, german.get_name());
    plist = $sformatf("%0s  %0s\n", plist, japanese.get_name());
    plist = $sformatf("%0s  %0s\n", plist, luxury.get_name());
    plist = $sformatf("%0s  %0s\n", plist, midwest.get_name());
  endfunction : new

  function void pre_randomize();
    policy_list.set_item(this);
    `uvm_info(get_type_name(),
              $sformatf("TEST CFG : The policy_list contains %0d policies",
                        policy_list.plist.size()), UVM_LOW)
  endfunction : pre_randomize

  function void post_randomize();
    policy_list.print_policy_list();
  endfunction : post_randomize
```

Figure 7: Configuration Object

The Test

Now to the payoff. Test writers are now able to select configuration options merely by adding policies to the policy list. Since the base configuration object is printing all the available policies to the log, the user does not even have far to go to find what policies are available. With a judicious naming convention many of the policies can be self-explanatory and discoverable. For example, enable_*, disable_* for basic enable/disable/don't care groups could be used.

Policies are added to the policy list before the configuration object is randomized in the top-level test. This adds the benefit of making the tests much simpler to audit and understand. There is a compact list of policies in the same place in every test that shows at-a-glance how the test is configured (see Figure 8, below).

Policy classes allow test writers to concentrate on what mode they want and not on having to learn the details of the internal workings of the configuration objects. Policies that would enable or disable each of the various modes can be provided, which allows the test writers to select whether they wanted to turn things on, off or allow them to be randomized.

```
function build_phase(uvm_phase phase);
  <...>
  // Policies used in this test
  env.test_cfg.policy_list.rm(env.test_cfg.german);   // don't select for German cars
  env.test_cfg.policy_list.add(env.test_cfg.american);
  env.test_cfg.policy_list.add(env.test_cfg.luxury);
  env.test_cfg.policy_list.add(env.test_cfg.manual)
  env.test_cfg.policy_list.add(env.test_cfg.midwest);
endfunction : build_phase
```

Figure 8: Example Test Configuration

Note that there are no add() order dependencies since each policy merely contributes to the overall set of constraints.

IV. Final Notes

The authors used this methodology for both the test configuration and for error injection. In the authors experience, most of the configuration policies were quite simple but a few were not.

Error Injection

As another expansion of this concept, policies were also created to enable simple control of error injection. Policies were used to configure things like how many bit errors, where the bit errors go, or complex error patterns, etc. This is closer to the traditional usage of the policy pattern, where the implementation algorithm is selected at runtime instead of compile time. In this case the implementation is whether and how errors should be injected. For example, the test writer could control whether single bits in a transaction were bad, or whether bursts of bad bits were sent.

The error injection scheme provides some food-for-thought on the potential applicability of these policy techniques to other testbench areas where randomization, or perhaps even factory override, is used.

See the GitHub repository [7] for an example of how the authors used policies in conjunction with an error injection methodology.

V. Conclusion

Using the policy pattern methodology for the configuration controls allowed us to do several things simultaneously:

1. Provide a simple configuration control mechanism for our test writers
2. Shrink the learning curve for new test writers
3. Encapsulate the configuration behavior and details to hide it from test writers
4. Allow some configuration settings to perform necessary overrides of other settings

Using the policy or strategy design pattern to control the randomization of our configuration classes, the authors were able to greatly simplify the job of the test writer, while giving them a straightforward way to control the VIP and testbench configuration from a test.

By doing all this several benefits became clear. We almost completely ended the copy-paste problem the project was having. We were able to understand the configuration of a test at-a-glance, as most tests ended up with three to five lines of configuration control. By simplifying everybody's understanding of the configuration of a test, we were able to audit the configuration controls in tests. This led us to catch several test problems where a configuration had been constrained for debug purposes and checked in unintentionally. The authors plan to use this strategy for future controls of configuration moving forward when things end up being inter-dependent or complex.

Through the example presented in this paper and the added example available in the GitHub repository [7], we hope we have planted the seed that the use of policies to further constrain objects can be applied to any object that can be randomized. You may find your own unique use as we did and go beyond sequence items and sequences.

References

[1] J. Dickol, "SystemVerilog Constraint Layering via Reusable Randomization Policy Classes," DVCon 2015.
http://events.dvcon.org/2015/proceedings/papers/04P_11.pdf
[2] J. Dickol, “Complex Constraints –Unleashing the Power of the VCS Constraint Solver,” SNUG 2016.
[3] J. Dickol, “I Didn’t Know Constraints Could Do That!,” Samsung DV Club, DV Club Europe 2018.
[4] E. Nelson, "Design Patterns by Example for SystemVerilog Verification ...." DVCon 2016.
https://static1.squarespace.com/static/521153a8e4b01a5565d75104/t/571d3b08746fb91673c11962/1461533459432/08_2.pdf
[5] J. Vance, et.al., “My Testbench Used to Break! Now it Bends: Adapting to Changing Design Configurations,” DVCon 2018.
[6] *IEEE Standard for SystemVerilog - Unified Hardware Design, Specification, and Verification Language*, IEEE Std 1800-2012.
[7] http://www.github.com/jmcneal/config_policy_pattern