Four Problems with Policy-Based Constraints and How to Fix Them

Dillan Mills
Synopsys, Inc.
Maricopa, AZ
dillan@synopsys.com

Chip Haldane
The Chip Abides, LLC
Gilbert, AZ
chip@thechipabides.com

Abstract

This paper presents solutions to problems encountered in the implementation of policy classes for SystemVerilog constraint layering. Policy classes provide portable and reusable constraints that can be mixed and matched into the object being randomized. There have been many papers and presentations on policy classes since the original presentation by John Dickol at DVCon 2015. The paper addresses three problems shared by all public policy class implementations and presents a solution to a fourth problem. The proposed solutions introduce policy class inheritance, tightly pair policy definitions with the class they constrain, reduce the expense of defining common policies using macros, and demonstrate how to treat policies as disposable and lightweight objects. The paper concludes that the proposed solution improves the usability and efficiency of policy classes for SystemVerilog constraint layering.

I. Constraints and Policy Class Review

Random objects and constraints are the foundational building blocks of constrained random verification in SystemVerilog. The simplest implementations embed fixed constraints within a class definition. Embedded constraints lack flexibility; all randomized object instances must meet these requirements exactly as they are written.

In-line constraints using the `with` construct offer marginally better flexibility. Although these external constraints allow greater variability of random objects, their definitions are still fixed within the calling context. Furthermore, all in-line constraints must be specified within a single call to `randomize()` .

Policy classes are a technique for applying SystemVerilog constraints in a portable, reusable, and incremental manner, originally described by John Dickol [1] [2]. The operating mechanism leverages an aspect of "global constraints," the simultaneous solving of constraints across a set of random objects. Randomizing an object that contains policies also randomizes the policies. Meanwhile, the policies contain a reference back to the container. Consequently, the policy container is constrained by the policies it contains. Dickol's approach is illustrated by the following code.

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

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

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

    function void add(policy_base#(ITEM) pcy);
        policy.push_back(pcy);
    endfunction

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

Fig. 1: The `policy_base` and `policy_list` classes

The first two classes provide the core definitions for policies. The **policy_base** implements the hook back to the policy container, and the **policy_list** enables related policies to be organized into groups. Both of these classes are parameterized by a container object type, so a unique specialization will be required for each policy container class that needs to be constrained.

Next, **addr_txn** is an example policy container—a generic transaction with a random address and size that implements policies.

```
class addr_txn;
    rand bit [31:0] addr;
    rand int        size;
    rand policy_base#(addr_txn) policy[$];

    constraint c_size {size inside {1, 2, 4};}

    function void pre_randomize;
        foreach(policy[i]) policy[i].set_item(this);
    endfunction
endclass
```

Fig. 2: The **addr_txn** class

The next three classes implement some policies for the **addr_txn** class. A list of valid address ranges can be stored in the **ranges** array. The **addr_permit_policy** will choose one of the ranges at random and constrain the address to be within the range, while the **addr_prohibit_policy** will exclude the address from randomizing inside any of the ranges in its list.

```
class addr_policy_base extends policy_base#(addr_txn);
    addr_range ranges[$];

    function void add(addr_t min, addr_t max);
        addr_range rng = new(min, max);
        ranges.push_back(rng);
    endfunction
endclass

class addr_permit_policy extends addr_policy_base;
    rand int selection;

    constraint c_addr_permit {
      selection inside {[0:ranges.size()-1]};

        foreach(ranges[i]) {
            if(selection == i) {
                item.addr inside {[ranges[i].min:ranges[i].max - item.size]};
            }
        }
    }
endclass

class addr_prohibit_policy extends addr_policy_base;
    constraint c_addr_prohibit {
        foreach(ranges[i]) {
            !(item.addr inside {[ranges[i].min:ranges[i].max - item.size + 1]});
        }
    }
endclass
```

Fig. 3: Policies for constraining the **addr_txn** class

The final class shows how policies might be used. The **addr_constrained_txn** class extends **addr_txn** and defines two policies, one that permits the address to be within one of two ranges, and one that prohibits the address from being within a third range. The **addr_constrained_txn** class then passes the local **pcy** list to the parent **policy** queue.

```
class addr_constrained_txn extends addr_txn;
    function new;
        addr_permit_policy     permit = new;
        addr_prohibit_policy   prohibit = new;
        policy_list#(addr_txn) pcy = new;

        permit.add('h00000000, 'h0000FFFF);
        permit.add('h10000000, 'h1FFFFFFF);
        pcy.add(permit);

        prohibit.add('h13000000, 'h130FFFFF);
        pcy.add(prohibit);

        this.policy = {pcy};
    endfunction
endclass
```

Fig. 4: The **addr_constrained_txn** subclass

At this point, an instance of **addr_constrained_txn** can be created and randomized like normal, and the address will be constrained based on the embedded policies.

```
addr_constrained_txn txn = new();
txn.randomize();
```

Fig. 5: Randomizing an instance of **addr_constrained_txn**

Further work on policy-based constraints has been presented since the original DVCon presentation in 2015. Kevin Vasconcellos and Jeff McNeal applied the concept to test configuration and added many nice utilities to the base policy class [3]. Chuck McClish extended the concept to manage real number values for User Defined Nettypes (UDN) and Unified Power Format (UPF) pins in an analog model [4]. Additionally, McClish defined a policy builder class that was used to generically build multiple types of policies while reducing repeated code that was shared between each policy class in the original implementation.

Although there has been extensive research on policies, this paper aims to address and provide solutions for three issues that have not been adequately resolved in previous implementations. Furthermore, a fourth problem that arose during testing of the upgraded policy package implementation will also be discussed and resolved.

II. Problem #1: Parameterized Policies

The first problem with the above policy implementation is that because **policy_base** is parameterized to the class it constrains, different specializations cannot be grouped and indexed. The awkward consequences of this limitation become apparent when using policies with a class hierarchy. If you extend a class and add a new random field then you need a new policy type to constrain that field. The new policy type requires its own policy list, and the new list must be traversed and mapped back to the container during **pre_randomize()** .

Imagine a complex class hierarchy with several layers of inheritance and extension, and constrainable attributes on each layer (one common example is a multi-layered sequence API library, such as the example presented by Jeff Vance [5]). Using policies becomes cumbersome in this case because each class layer requires a unique family of constraints organized into a distinct list. This stratification of polices places a burden on users to know which layer of the class hierarchy defines each attribute they want to constrain, and the name of the associated policy list for the matching policy type. For example, extending the **addr_txn** class to create a version with parity checking results in a class hierarchy that looks like this:

```
class addr_txn;
  // ... unchanged from previous example
endclass

class addr_p_txn extends addr_txn;
  rand bit parity;
  rand bit parity_err;
  rand policy_base#(addr_p_txn) addr_p_policy[$];

  constraint c_parity_err {
    soft (parity_err == 0);
    (parity_err) ^ ($countones({addr, parity}) == 1);
  }

  function void pre_randomize;
    super.pre_randomize();
    foreach(addr_p_policy[i]) addr_p_policy[i].set_item(this);
  endfunction
endclass

class addr_constrained_txn extends addr_p_txn;
  function new;
    addr_permit_policy        permit_p = new;
    addr_prohibit_policy      prohibit_p = new;

    // definition to follow in a later example - constrains parity_err bit
    addr_parity_err_policy    parity_err_p = new;

    policy_list#(addr_txn)    addr_pcy_lst = new;
    policy_list#(addr_p_txn)  addr_p_pcy_lst = new;

    permit_p.add('h00000000, 'h0000FFFF);
    permit_p.add('h10000000, 'h1FFFFFFF);
    addr_pcy_lst.add(permit_p);

    prohibit_p.add('h13000000, 'h130FFFFF);
    addr_pcy_lst.add(prohibit_p);

    parity_err_p.set(1'b1);
    addr_p_pcy_lst.add(parity_err_p);

    this.addr_policy   = {addr_pcy_lst};
    this.addr_p_policy = {addr_p_pcy_lst};
  endfunction
endclass
```

Fig. 6: Modified address transaction classes with a parity checking subclass and policies included

Scaling this implementation results in a lot of repeated boilerplate code and is not very intuitive to use. Each additional subclass in a hierarchy only increases the chaos and complexity of implementing and using policies effectively.

The solution to this problem is to replace the parameterized policy base with a non-parameterized base and a parameterized extension. We chose an interface class as our non-parameterized base for the flexibility it offers over a virtual base class—specifically, our policies are bound only to implement the interface functions and not to extend a specific class implementation.

```
interface class policy;
    pure virtual function void set_item(uvm_object item);
endclass

virtual class policy_imp#(type ITEM=uvm_object) implements policy;
    protected rand ITEM m_item;

    virtual function void set_item(uvm_object item);
        if (!$cast(m_item, item)) begin
            `uvm_warning("policy::set_item()", "Item/policy type mismatch")
            this.m_item = null;
            this.m_item.rand_mode(0);
        end
    endfunction: set_item
endclass: policy_imp

typedef policy policy_queue[$];
```

Fig. 7: The **policy** interface class and **policy_imp** class

A non-parameterized base enables all policies targeting a particular class hierarchy to be stored within a single common **policy_queue** . A parameterized template, **policy_imp** , implements the base interface and core functionality required by all policies.

One consequence of eliminating the parameter from our base type is that the policy-enabled container object, **item** , and its assignment function, **set_item()** , are no longer strongly typed. Here we make a small concession, using **uvm_object** as our default policy-enabled type. This means that all classes that implement our policies must derive from **uvm_object** , and we need to use dynamic casting to ensure that policies and their containers are type-compatible. In the example above, **item** is set to **null** and randomization is disabled when the cast fails, preventing runtime problems in the event that incompatible policies are applied.

Not much changes when it comes to defining policies; the address policies now extend **policy_imp** instead of **policy_base** , and the underlying constraints are written as implications so that they will not apply when **item** is missing.

```
class addr_policy extends policy_imp#(addr_txn);
   // ... unchanged from previous example, with updated class extension
endclass

class addr_parity_err_policy extends policy_imp#(addr_p_txn);
   protected bit parity_err;

   constraint c_fixed_value {m_item != null -> m_item.parity_err == parity_err;}

   function new(bit parity_err);
      this.parity_err = parity_err;
   endfunction
endclass

class addr_permit_policy extends addr_policy;
   // same as before
endclass
```

```
class addr_prohibit_policy extends addr_policy;
   // same as before
endclass
```

Fig. 8: Address policies updated to use the new policy implementation

However, the address transaction classes are simplified considerably. Only a single policy queue is required in the class hierarchy, and the vast majority of the boilerplate code has been eliminated, including all of the specialized lists. The **addr_txn** class now extends **uvm_object** to provide compatibility with the policy interface.

```
class addr_txn extends uvm_object;
    rand policy_queue policies;
    // policy is replaced with the above. All other members, constraints,
    // and pre_randomize are unchanged from the previous example
endclass

class addr_p_txn extends addr_txn;
    rand bit parity;
    rand bit parity_err;
    constraint c_parity_err {/*...*/}
    // The local addr_p_policy and pre_randomize are removed. Everything
    // else is unchanged from the previous example
endclass

class addr_constrained_txn extends addr_p_txn;
    function new;
        addr_permit_policy     permit_p   = new();
        addr_prohibit_policy   prohibit_p = new();
        addr_parity_err_policy parity_err_p;

        // only a single policy queue is necessary now
        permit_p.add('h00000000, 'h0000FFFF);
        permit_p.add('h10000000, 'h1FFFFFFF);
        this.policies.push_back(permit_p);

        prohibit_p.add('h13000000, 'h130FFFFF);
        this.policies.push_back(prohibit_p);

        parity_err_p = new(1'b1);
        this.policies.push_back(parity_err_p);
    endfunction
endclass
```

Fig. 9: Address transaction classes updated to use the new policy implementation

III. Problem #2: Definition Location

The second problem with policies is "where do I define my policy classes?" This is not a complicated problem to solve; most users will likely wish to place their policy classes in a file or files close to the class they are constraining. However, directly embedding policy definitions within the class they constrain offers a myriad of benefits. Not only does this convention eliminate all guesswork about where to define and discover policies, but embedded policies also gain access to all members of their container class, including protected properties and methods! This privileged access enables policies to constrain attributes of a class that are not otherwise exposed, improving encapsulation.

To further optimize the organization of potentially large families of policies, we establish a convention of defining all policy classes within an embedded wrapper class called **POLICIES** . Each layer of a class hierarchy that implements policies will have its own embedded **POLICIES** wrapper, and individual **POLICIES** wrappers extend other wrappers in a manner parallel to their container classes. This parallel inheritance pattern is shown below, with **addr_p_txn::POLICIES** extending **addr_txn::POLICIES** .

```
class addr_txn extends uvm_object;
    // class members, constraints, and pre_randomize unchanged from previous

    class POLICIES;
        class addr_policy extends policy_imp#(addr_txn);
            // ... unchanged from previous standalone class example
        endclass

        class addr_permit_policy extends addr_policy;
            // ... unchanged from previous standalone class example
        endclass

        class addr_prohibit_p_policy extends addr_policy;
            // ... unchanged from previous standalone class example
        endclass
    endclass: POLICIES
endclass

class addr_p_txn extends addr_txn;
    protected rand bit parity_err;
    // other class members and constraints unchanged from previous example

    class POLICIES extends addr_txn::POLICIES;
        class addr_parity_err_policy extends policy_imp#(addr_p_txn);
            // ... unchanged from previous example
        endclass

        static function addr_parity_err_policy PARITY_ERR(bit value);
            PARITY_ERR = new(value);
        endfunction
    endclass: POLICIES
endclass

class addr_constrained_txn extends addr_p_txn;
    function new;
        addr_constrained_txn::POLICIES::addr_permit_policy   permit_p   = new();
        addr_constrained_txn::POLICIES::addr_prohibit_policy prohibit_p = new();

        // policy constraint value setup unchanged from previous example

        this.policies.push_back(
            addr_constrained_txn::POLICIES::PARITY_ERR(1'b1)
        );
    endfunction
endclass
```

Fig. 10: Address transaction classes with embedded policies

This example also shows how we can define static constructor functions within **POLICIES** wrappers. This practice further reduces the cost of using policies since we can instantiate and initialize them with a single call, as demonstrated with the call to **addr_constrained_txn::POLICIES::PARITY_ERR()** . Note that although the **PARITY_ERR** constructor is defined in **addr_p_txn::POLICIES** , it is accessible through **addr_constrained_txn::POLICIES** because of the wrapper class inheritance. The **POLICIES::** scoping layer even helps to make code more readable and easy to understand.

What's more, the **parity_err** property has now been defined as **protected** , preventing anything but our **PARITY_ERR** policy from manipulating that "knob." In fact, a more advanced use of policies might define all members of a target class as protected, restricting the setting of fields exclusively through policies and reading through accessor functions, thus encouraging maximum encapsulation/loose coupling, which reduces the cost of maintaining and enhancing code and prevents bugs from cascading into classes that use policy-enabled classes.

IV. Problem #3: Boilerplate Overload

The third problem with using policies is that policies are relatively expensive to define since you need at a minimum: a class definition, a constructor, and a constraint. This will be relatively unavoidable for complex policies, such as those defining a relationship between multiple specific class attributes. For generic policies, such as equality constraints (property equals X), range constraints (property between Y and Z), or set membership (keyword **inside**) constraints, macros can be used to drastically reduces the expense and risk of defining common policies. The macros are responsible for creating the specialized policy class for the required constraint within the target class, as well as a static constructor function that is used to create new policy instances of the class. Additional macros are utilized for setting up the embedded **POLICIES** class within the target class. These macros can be used hand in hand with the non-macro policy classes needed for complex constraints, if necessary.

```
// Fixed-value policy class and constructor macro
`define fixed_policy(POLICY, FIELD, TYPE)                                \
`m_fixed_policy_class(POLICY, FIELD, TYPE)                               \
`m_fixed_policy_constructor(POLICY, TYPE)

`define m_fixed_policy_class(POLICY, FIELD, TYPE)                        \
    class POLICY``_policy extends base_policy;                           \
        protected TYPE l_val;                                            \
                                                                         \
        constraint c_fixed_value {                                       \
            (m_item != null) -> (m_item.FIELD == TYPE'(l_val));          \
        }                                                                \
                                                                         \
        function new(TYPE value);                                        \
            this.l_val = value;                                          \
        endfunction                                                      \
    endclass: POLICY``_policy

`define m_fixed_policy_constructor(POLICY, TYPE)                         \
    static function POLICY``_policy POLICY(TYPE value);                  \
        POLICY = new(value);                                             \
    endfunction: POLICY
```

Fig. 11: Macros for setting up the embedded **POLICIES** class and a fixed value policy

This example includes a **`fixed_policy** macro, which wraps two additional macros responsible for creating a policy class and a static constructor for the class. This **`fixed_policy** example policy class lets you constrain a property to a fixed value. A more complete macro definition can be found in Appendix B. The appendix includes **`start_policies** , **`start_extended_policies** , and **`end_policies** macros that are used to create the embedded **POLICIES** class within the constrained class instead of using hard-coded **class** and **endclass** statements. They set up class inheritance as needed and create a local typedef for the **policy_imp** parameterized type.

```
class addr_txn extends uvm_object;
```

```
    // class members, constraints, and pre_randomize unchanged from previous
↪   example

    `start_policies(addr_txn)
        `include "addr_policies.svh"
    `end_policies
endclass

class addr_p_txn extends addr_txn;
    // class members and constraints unchanged from previous example

    `start_extended_policies(addr_p_txn, addr_txn)
        `fixed_policy(PARITY_ERR, parity_err, bit)
    `end_policies
endclass

class addr_constrained_txn extends addr_p_txn;
    // ... unchanged from previous example
endclass
```

Fig. 12: Simplified address transaction classes using the policy macros

The base **addr_txn** class has complex policies with a relationship between the **addr** and **size** fields, so rather than creating a policy macro that will only be used once, they can either be left as-is within the embedded policies class, or moved to a separate file and included with **`include** as was done here to keep the transaction class simple. The child parity transaction class is able to use the **`fixed_policy** macro to constrain the **parity_err** field. The constraint block remains the same as the previous example.

V. Problem #4: Unexpected Policy Reuse Behavior and Optimizing for Lightweight Policies

A fourth problem we noticed in our initial usage of policy classes was occasional unexpected behavior when attempting to reuse a policy. It is hard to quantify and we never spent time to create a reproducible test case, but some of the policies behaved like they "remembered" being randomized when used repeatedly and wouldn't reapply their constraints during subsequent **randomize** calls. Rather than spending time to diagnose and fix this issue, we were focusing on transitioning our policy approach to a "use once and discard" approach, where we would treat a policy as disposable and apply fresh policy instances before re-randomizing a target object. This improvement happened to resolve the randomization issue as well, so we never had a reason to circle back.

Fortunately, creating new policy instances for each randomization is not prohibitively expensive. We anticipated that we would need hundreds or thousands of policies per test, so we aimed to minimize the memory requirements and streamline functionality as much as possible. To minimize the footprint of policies, we chose to implement them as simple classes rather than extensions of **uvm_object** (the policy container itself still inherits from **uvm_object**). We duplicated some useful capabilities such as a basic copy routine, while omitting more expensive functionality such as factory compatibility or object comparison. The buggy randomization behavior implied we needed to pass in fresh policy instances each time we randomized an object. Two features helped to accomplish this: static constructor functions and a **copy** method.

```
static function addr_parity_err_policy PARITY_ERR(bit value);
    PARITY_ERR = new(value);
endfunction
...
this.policies.push_back(addr_constrained_txn::POLICIES::PARITY_ERR(1'b1));
```

Fig. 13: Example static constructor function from the **PARITY_ERR** policy class

Using static functions to create new instances of the policies provides an easy way to create and use a policy with a single function call. The policy is created and immediately pushed onto the policy queue, then discarded and recreated the next time it is needed.

```
interface class policy;
    ...
    pure virtual function policy copy();
endclass: policy
```

Fig. 14: The **copy** method in the **policy** interface class

Adding a **copy** method to the policy classes provides a way for a policy to return a fresh copy of itself with the same configuration. This is useful when a policy needs to be reused multiple times. The policy can be copied and pushed onto the policy queue multiple times, or onto other policy queues, and each copy will constrain the field separately. The **copy** method is **pure virtual** , so it must be implemented by each policy class.

After migrating to this pattern of reuse, the issues we were seeing with unapplied constraints disappeared.

VI. More Improvements to the Policy Package

The examples presented so far are functional, but are lacking many features that would be useful in a real-world implementation. The following examples will present additional improvements to the policy package that will make it more practical and efficient to use.

A. Expanding the **policy** *interface class*

The following **policy** interface class adds additional methods for managing a policy.

```
    pure virtual function string type_name();
    pure virtual function string description();
    pure virtual function bit item_is_compatible(uvm_object item);
    pure virtual function void set_item(uvm_object item);
    pure virtual function policy copy();

endclass: policy
```

Fig. 15: Expanded **policy** interface class

The **name** , **description** , and **copy** methods are implemented by the policy (or policy macro) directly and provide reporting information useful when printing messages about the policy to the log for the former two, or specific behavior for making a copy for the latter. The remaining three methods are implemented by **policy_imp** and are shared by all policies.

B. Better type safety checking and reporting in **policy_imp** *methods*

Some of the benefits of above methods can be seen by examining the new **set_item** method used by **policy_imp** .

```
virtual function void set_item(uvm_object item);
    if (item == null) begin
        `uvm_error("policy::set_item()", "NULL item passed")

    end else if ((this.item_is_compatible(item)) && $cast(this.m_item, item)) begin
        `uvm_info(
            "policy::set_item()",
            $sformatf(
                "policy <%s> applied to item <%s>: %s",
                this.name(), item.get_name(), this.description()
            ),
            UVM_FULL
        )
        this.m_item.rand_mode( 1 );

    end else begin
        `uvm_warning(
            "policy::set_item()",
            $sformatf(
                "Item <%s> type <%s> is not compatible with policy <%s> type <%s>",
               item.get_name(), item.get_type_name(), this.name(), this.type_name()
            )
        )
        this.m_item = null;
        this.m_item.rand_mode( 0 );
    end
endfunction: set_item
```

Fig. 16: **set_item** method from **policy_imp**

The **set_item** method makes use of all the reporting methods to provide detailed log messages when **set_item** succeeds or fails. Additionally, the **item_is_compatible** is used before the **\$cast** method is called and the **rand_mode** state is kept consistent with the result of the cast.

C. Replacing **policy_list** *with* **policy_queue**

Eagle-eyed readers might have noticed the lack of presence of a **policy_list** class in any of the examples above after migrating to the improved policy interface. Rather, a single typedef is all that is necessary to manage policies in a class.

```
typedef policy policy_queue[$];
```

Fig. 17: The **policy_queue** typedef

The **policy_queue** type is capable of storing any policy that implements the **policy** interface. The default queue methods are sufficient for aggregating policies, and in practice we found that using policy queues as containers was more efficient than **policy_list** instances. For example, for functions expecting a **policy_queue** argument we can directly pass in array literals populated by calls to static constructor functions, allowing us to define, initialize, aggregate, and pass policies all in a single line of code!

D. Provide a common implementation for policy-enabled classes using the **policy_container** *interface and* **policy_object** *mixin*

Our solution encourages loosely coupled code by defining APIs that pass **policy_queue** arguments and hide a target object's policies queue from other classes.

```
    pure virtual function bit has_policies();

    // Assignments
    pure virtual function void set_policies(policy_queue policies);
    pure virtual function void add_policies(policy_queue policies);
    pure virtual function void clear_policies();

    // Access
    pure virtual function policy_queue get_policies();

    // Copy
    pure virtual function policy_queue copy_policies();

endclass: policy_container
```

Fig. 18: The **policy_container** interface class

This interface class should be implemented by any class that needs to use policies. The method implementations can then be used to manipulate a policy queue as needed, including adding to, replacing, or clearing the queue, and returning a handle to the queue or a copy of the queue to use elsewhere.

The **policy_container** interface class allows for a base class implementation that can be used to manage constraints across the entire class hierarchy.

```
class policy_object #(type BASE=uvm_object) extends BASE implements
↪  policy_container;

    protected policy_queue m_policies;

    // Queries
    virtual function bit has_policies();
        // returns true/false based on size of m_policies
    endfunction: has_policies

    // Assignments
    virtual function void set_policies(policy_queue policies);
        // sets m_policies to a new queue of policies
    endfunction: set_policies

    virtual function void add_policies(policy_queue policies);
        // adds new policies to m_policies
    endfunction: add_policies

    virtual function void clear_policies();
        // clears m_policies
    endfunction: clear_policies

    // Access
    virtual function policy_queue get_policies();
        // return a handle to m_policies
    endfunction: get_policies

    // Copy
    virtual function policy_queue copy_policies();
        // a copy of m_policies
    endfunction: copy_policies
endclass: policy_object
```

Fig. 19: A **policy_object** base class implementation

This base class implements all the **policy_container** functionality (a complete example is available in Appendix A). Packaging a common implementation and interface into a mixin enables sharing a common implementation and API for policy support among any classes that might benefit from the use of policies, as seen in the following example.

```
// Use policy_object for transactions
class base_txn extends policy_object #(uvm_sequence_item);

// Use policy_object for sequences
class base_seq #(type REQ=uvm_sequence_item, RSP=REQ) extends policy_object #(
↪  uvm_sequence#(REQ, RSP) );
```

```

// Use policy_object for configuration objects
class cfg_object extends policy_object #(uvm_object);
```

Fig. 20: Example classes using the **`policy_object`** base class

E. Protecting the policy queue enforces loosely coupled code

A subtle but significant additional benefit to using a base **`policy_object`** class along with the **`policy_container`** API is the ability to mark the container's policy queue as protected and prevent direct access to it.

The original implementation called **`set_item`** on each policy during the **`pre_randomize`** stage. That was necessary because the **`policy_queue`** was public, so there was nothing to prevent callers from adding policies without linking them to the target class.

Using an interface class and making the implementation private means the callers may only set or add policies using the available interface class routines, and because those routines are solely responsible for applying policies, they can also check compatibility (and filter incompatible policies) and call **`set_item`** immediately. This can be seen in the example **`policy_object`** implementation above, in the usage of the protected function **`try_add_policy`** .

By calling **`set_item`** when the policy is added to the queue, all policies will be associated with the target item automatically, so there is no need to do it during **`pre_randomize`** . This eliminates an easily-overlooked requirement for classes extending **`policy_object`** to make sure they call **`super.pre_randomize()`** in their local **`pre_randomize`** function.

VII. Conclusion

The improvements to the policy package presented in this paper provide a more robust and efficient implementation of policy-based constraints for SystemVerilog. The policy package is now capable of managing constraints across an entire class hierarchy, and the policy definitions are tightly paired with the class they constrain. The use of macros reduces the expense of defining common policies, while still allowing great flexibility in any custom policies necessary.

The complete policy package is available in Appendix A, with additional macro examples available in Appendix B. A functional package is also available for download [6] which can be included directly in a project to start using policies immediately.

References

[1] J. Dickol, "SystemVerilog Constraint Layering via Reusable Randomization Policy Classes," DVCon, 2015.
[2] J. Dickol, "Complex Constraints: Unleashing the Power of the VCS Constraint Solver," SNUG Austin, September 29, 2016.
[3] K. Vasconcellos, J. McNeal, "Configuration Conundrum: Managing Test Configuration with a Bite-Sized Solution," DVCon, 2021.
[4] C. McClish, "Bi-Directional UVM Agents and Complex Stimulus Generation for UDN and UPF Pins," DVCon, 2021.
[5] J. Vance, J. Montesano, M. Litterick, J. Sprott, "Be a Sequence Pro to Avoid Bad Con Sequences," DVCon, 2019.
[6] D. Mills, C. Haldane, "policy_pkg Source Code," https://github.com/DillanCMills/policy_pkg.

Appendix A
Policy Package

A. *policy_pkg.sv*

```
`include "uvm_macros.svh"

package policy_pkg;

    import uvm_pkg::*;

    `include "policy.svh"

    typedef policy policy_queue[$];

    `include "policy_imp.svh"

    `include "policy_container.svh"
    `include "policy_object.svh"

endpackage: policy_pkg
```

B. *policy.svh*

```
interface class policy;

    pure virtual function string name();
    pure virtual function string type_name();
    pure virtual function string description();
    pure virtual function bit item_is_compatible(uvm_object item);
    pure virtual function void set_item(uvm_object item);
    pure virtual function policy copy();

endclass: policy
```

C. *policy_imp.svh*

```
virtual class policy_imp #(type ITEM=uvm_object) implements policy;

    protected rand ITEM m_item;

    pure virtual function string name();
    pure virtual function string description();
    pure virtual function policy copy();

    virtual function string type_name();
        return( ITEM::type_name );
    endfunction: type_name

    virtual function bit item_is_compatible(uvm_object item);
        ITEM local_item;

        return( (item != null) && ($cast(local_item, item)) );
    endfunction: item_is_compatible

    virtual function void set_item(uvm_object item);
        if (item == null) begin
            `uvm_error("policy::set_item()", "NULL item passed")

        end else if ((this.item_is_compatible(item)) && $cast(this.m_item, item)) begin
            `uvm_info(
                "policy::set_item()",
                $sformatf(
                    "policy <%s> applied to item <%s>: %s",
                    this.name(), item.get_name(), this.description()
                ),
                UVM_FULL
            )
            this.m_item.rand_mode( 1 );

        end else begin
            `uvm_warning(
                "policy::set_item()",
                $sformatf(
                    "Cannot apply policy '%0s' of type '%0s' to target object '%0s' of incompatible type '%0s'",
                    this.name(), ITEM::type_name(), item.get_name(), item.get_type_name()
                )
            )
            this.m_item = null;
            this.m_item.rand_mode( 0 );
        end
    endfunction: set_item

endclass: policy_imp
```

D. *policy_container.svh*

```
interface class policy_container;

    // Queries
    pure virtual function bit has_policies();

    // Assignments
    pure virtual function void set_policies(policy_queue policies);
    pure virtual function void add_policies(policy_queue policies);
    pure virtual function void clear_policies();

    // Access
    pure virtual function policy_queue get_policies();

    // Copy
    pure virtual function policy_queue copy_policies();
```

```

endclass: policy_container
```

E. *policy_object.svh*

```
class policy_object #(type BASE=uvm_object) extends BASE implements
↪   policy_container;

    protected policy_queue m_policies;

    // Queries
    virtual function bit has_policies();
        return( this.m_policies.size() > 0 );
    endfunction: has_policies

    // Assignments
    virtual function void set_policies(policy_queue policies);
        if( this.has_policies()) begin
            `uvm_warning( "policy", "set_policies() replacing existing policies" )
        end
        this.m_policies = {};
        foreach( policies(i) ) try_add_policy( policies[i] );
    endfunction: set_policies

    virtual function void add_policies(policy_queue policies);
        foreach( policies(i) ) try_add_policy( policies[i] );
    endfunction: add_policies

    virtual function void clear_policies();
        if ( this.has_policies() ) begin
            `uvm_info( "policy", $sformatf("clearing [%0d] policies from %s",
↪   this.m_policies.size(), this.get_name()), UVM_FULL)
        end
        this.m_policies = {};
    endfunction: clear_policies

    // Access
    virtual function policy_queue get_policies();
        return( this.m_policies );
    endfunction: get_policies

    // Copy
    virtual function policy_queue copy_policies();
        copy_policies = {};
        foreach( this.m_policies[i] ) begin
            copy_policies.push_back( this.m_policies[i].copy() );
        end
    endfunction: copy_policies

    protected function void try_add_policy( policy new_policy );
        if (new_policy.item_is_compatible(this)) begin
            `uvm_info( "policy", $sformatf("adding policy %s to %s",
↪   new_policy.name, this.get_name()), UVM_FULL)
            new_policy.set_item( this );
            this.m_policies.push_back( new_policy );
        end else begin
            `uvm_warning( "policy", $sformatf("policy %s not compatible with target
↪  %s", new_policy.name, this.get_name()))
        end
    endfunction: try_add_policy
endclass: policy_object
```

Appendix B
Policy Macros

A. *policy_macros.svh*

```
`ifndef__POLICY_MACROS__
`define __POLICY_MACROS__

//=================================================
// Embedded POLICIES class macros
//=================================================

`define start_policies(cls)                                              \
class POLICIES;                                                          \
`m_base_policy(cls)

`define start_extended_policies(cls, parent)                             \
class POLICIES extends parent::POLICIES;                                 \
`m_base_policy(cls)

`define end_policies                                                     \
endclass: POLICIES

`define m_base_policy(cls)                                               \
    typedef policy_imp#cls) base_policy;


//=================================================
// Policy template macros
//=================================================

`include "constant_policy.svh"
`include "fixed_policy.svh"
`include "member_policy.svh"
`include "range_policy.svh"

`endif
```

B. *constant_policy.svh*

```
// Full policy definition
`define constant_policy(POLICY, FIELD, TYPE, CONST)                      \
`m_const_policy_class(POLICY, FIELD, TYPE, CONST)                        \
`m_const_policy_constructor(POLICY)

// Policy class definition
`define m_const_policy_class(POLICY, FIELD, TYPE, CONST)                 \
    class POLICY``_policy extends base_policy                            \
                                                                         \
        constraint c_policy_constraint {                                 \
            (m_item != null) -> (m_item.FIELD == TYPE'(CONST));          \
        }                                                                \
                                                                         \
        function new();                                                  \
        endfunction: new                                                    \
                                                                            \
        virtual function string name();                                     \
            return (`"POLICY`");                                            \
        endfunction: name                                                   \
                                                                            \
        virtual function string description();                              \
            return (`"(FIELD==CONST)`");                                    \
        endfunction: description                                            \
                                                                            \
        virtual function policy copy();                                     \
            copy = new();                                                   \
        endfunction: copy                                                   \
    endclass: POLICY``_policy

// Policy constructor definition
`define m_const_policy_constructor(POLICY)                                  \
    static function POLICY``_policy POLICY();                               \
        POLICY = new();                                                     \
    endfunction: POLICY
```

C. fixed_policy.svh

```
// Full policy definition
`define fixed_policy(POLICY, FIELD, TYPE, RADIX="%0p")                      \
`m_fixed_policy_class(POLICY, FIELD, TYPE, RADIX)                           \
`m_fixed_policy_constructor(POLICY, TYPE, RADIX)

// Policy class definition
`define m_fixed_policy_class(POLICY, FIELD, TYPE, RADIX="%0p")              \
    class POLICY``_policy extends base_policy                               \
        local TYPE          l_val;                                          \
        local string        l_radix=RADIX;                                  \
                                                                            \
        constraint c_policy_constraint {                                    \
            (m_item != null) -> (m_item.FIELD == TYPE'(l_val));             \
        }                                                                   \
                                                                            \
        function new(TYPE value, string radix=RADIX);                       \
            this.set_value(value);                                          \
            this.set_radix(radix);                                          \
        endfunction: new                                                    \
                                                                            \
        virtual function string name();                                     \
            return (`"POLICY`");                                            \
        endfunction: name                                                   \
                                                                            \
        virtual function string description();                              \
            return ({                                                       \
                `"(FIELD==",                                                \
                $sformatf(l_radix, l_val),                                  \
                `")`"                                                       \
            });                                                             \
        endfunction: description                                            \
                                                                          \
        virtual function policy copy();                                   \
            copy = new(l_val, l_radix);                                   \
        endfunction: copy                                                 \
                                                                          \
        virtual function void set_value(TYPE value);                      \
            this.l_val = value;                                           \
        endfunction: set_value                                            \
                                                                          \
        virtual function TYPE get_value();                                \
            return (this.l_val);                                          \
        endfunction: get_value                                            \
                                                                          \
        virtual function void set_radix(string radix);                    \
            this.l_radix = radix;                                         \
        endfunction: set_radix                                            \
                                                                          \
        virtual function string get_radix();                              \
            return (this.l_radix);                                        \
        endfunction: get_radix                                            \
    endclass: POLICY``_policy

// Policy constructor definition
`define m_fixed_policy_constructor(POLICY, TYPE, RADIX="%0p")             \
    static function POLICY``_policy POLICY(TYPE value, string radix=RADIX);   \
        POLICY = new(value, radix);                                       \
    endfunction: POLICY
```

D. member_policy.svh

```
// Full policy definition
`define member_policy(POLICY, FIELD, TYPE, RADIX="%0p")                   \
`m_member_policy_class(POLICY, FIELD, TYPE, RADIX)                        \
`m_member_policy_constructor(POLICY, TYPE, RADIX)

// Policy class definition
`define m_member_policy_class(POLICY, FIELD, TYPE, RADIX="%0p")           \
    class POLICY``_policy extends base_policy                             \
        typedef TYPE            l_field_array_t[];                        \
                                                                          \
        local l_field_array_t   m_values;                                 \
        local bit               l_exclude;                                \
        local string            m_radix=RADIX;                            \
                                                                          \
        constrant c_policy_constraint {                                   \
            (m_item != null) ->                                           \
                ((l_exclude) ^ (m_item.FIELD inside {m_values}));         \
        }                                                                 \
                                                                          \
        function new(                                                     \
            l_field_array_t values,                                       \
            bit             exclude=1'b0,                                 \
            string          radix=RADIX                                   \
        );                                                                \
            this.set_values(values);                                                \
            this.set_exclude(exclude);                                              \
            this.set_radix(radix);                                                  \
        endfunction: new                                                            \
                                                                                    \
        virtual function string name();                                             \
            return (`"POLICY`");                                                    \
        endfunction: name                                                           \
                                                                                    \
        virtual function string description();                                      \
            string values_str = "";                                                 \
                                                                                    \
            foreach(m_values[i])                                                    \
                values_str = {                                                      \
                    values_str,                                                     \
                    $sformatf(m_radix, m_values[i]),                                \
                    i == m_values.size()-1 ? "" : ", "};                            \
                                                                                    \
            return ({                                                               \
                `"POLICY(FIELD ",                                                   \
                l_exclude ? "outside {" : "inside {",                               \
                values_str,                                                         \
                `"})`"                                                              \
            });                                                                     \
        endfunction: description                                                    \
                                                                                    \
        virtual function policy copy();                                             \
            copy = new(m_values, l_exclude, m_radix);                               \
        endfunction: copy                                                           \
                                                                                    \
        virtual function void set_values(l_field_array_t values);                   \
            this.m_values = values;                                                 \
        endfunction: set_values                                                     \
                                                                                    \
        virtual function l_field_array_t get_values();                              \
            l_field_array_t l_array;                                                \
            foreach (this.m_values[i])                                              \
                l_array[i] = this.m_values[i];                                      \
            return (l_array);                                                       \
        endfunction: get_values                                                     \
                                                                                    \
        virtual function void set_exclude(bit exclude);                             \
            this.l_exclude = exclude;                                               \
        endfunction: set_exclude                                                    \
                                                                                    \
        virtual function bit get_exclude();                                         \
            return (this.l_exclude);                                                \
        endfunction: get_exclude                                                    \
                                                                                    \
        virtual function void set_radix(string radix);                              \
            this.m_radix = radix;                                                   \
        endfunction: set_radix                                                      \
                                                                                    \
        virtual function string get_radix();                                        \
            return (this.m_radix);                                         \
        endfunction: get_radix                                             \
    endclass: POLICY``_policy

// Policy constructor definition
`define m_member_policy_constructor(POLICY, TYPE, RADIX="%0p")             \
    typedef TYPE POLICY``_array_t[];                                       \
    static function POLICY``_policy POLICY(                                \
        POLICY``_array_t values,                                           \
        bit              exclude=1'b0,                                     \
        string           radix=RADIX                                       \
    );                                                                     \
        POLICY = new(values, exclude, radix);                              \
    endfunction: POLICY
```

E. range_policy.svh

```
// Full policy definition
`define range_policy(POLICY, FIELD, TYPE, RADIX="%0p")                     \
`m_range_policy_class(POLICY, FIELD, TYPE, RADIX)                          \
`m_range_policy_constructor(POLICY, TYPE, RADIX)

// Policy class definition
`define m_range_policy_class(POLICY, FIELD, TYPE, RADIX="%0p")             \
    class POLICY``_policy extends base_policy                              \
        local TYPE          l_low;                                         \
        local TYPE          l_high;                                        \
        local bit           l_exclude;                                     \
        local string        l_radix=RADIX;                                 \
                                                                           \
        constraint c_policy_constraint {                                   \
            (m_item != null) -> (                                          \
                (l_exclude) ^                                              \
                ((m_item.FIELD >= l_low && m_item.FIELD <= l_high))        \
            );                                                             \
        }                                                                  \
                                                                           \
        function new(                                                      \
            TYPE   low,                                                    \
            TYPE   high=low,                                               \
            bit    exclude=1'b0,                                           \
            string radix=RADIX                                             \
        );                                                                 \
            this.set_range(low, high);                                     \
            this.set_exclude(exclude);                                     \
            this.set_radix(radix);                                         \
        endfunction: new                                                   \
                                                                           \
        virtual function string name();                                    \
            return (`"POLICY`");                                           \
        endfunction: name                                                  \
                                                                           \
        virtual function string description();                             \
            return ({                                                      \
                `"(FIELD ",                                                   \
                l_exclude ? "outside [" : "inside [",                         \
                $sformatf(l_radix, l_low),                                    \
                ", ",                                                         \
                $sformatf(l_radix, l_high),                                   \
                `"])`"                                                        \
            });                                                               \
        endfunction: description                                              \
                                                                              \
        virtual function policy copy();                                       \
            copy = new(l_low, l_high, l_exclude, l_radix);                    \
        endfunction: copy                                                     \
                                                                              \
        virtual function void set_range(TYPE low, TYPE high);                 \
            if (low <= high) begin                                            \
                this.l_low = low;                                             \
                this.l_high = high;                                           \
            end else begin                                                    \
                this.l_low = high;                                            \
                this.l_high = low;                                            \
            end                                                               \
        endfunction: set_range                                                \
                                                                              \
        virtual function TYPE get_low();                                      \
            return (this.l_low);                                              \
        endfunction: get_low                                                  \
                                                                              \
        virtual function TYPE get_high();                                     \
            return (this.l_high);                                             \
        endfunction: get_high                                                 \
                                                                              \
        virtual function void set_exclude(bit exclude);                       \
            this.l_exclude = exclude;                                         \
        endfunction: set_exclude                                              \
                                                                              \
        virtual function bit get_exclude();                                   \
            return (this.l_exclude);                                          \
        endfunction: get_exclude                                              \
                                                                              \
        virtual function void set_radix(string radix);                        \
            this.l_radix = radix;                                             \
        endfunction: set_radix                                                \
                                                                              \
        virtual function string get_radix();                                  \
            return (this.l_radix);                                            \
        endfunction: get_radix                                                \
    endclass: POLICY``_policy

// Policy constructor definition
`define m_range_policy_constructor(POLICY, TYPE, RADIX="%0p")                 \
    static function POLICY``_policy POLICY(                                   \
        TYPE   low,                                                           \
        TYPE   high=low,                                                      \
        bit    exclude=1'b0,                                                  \
        string radix=RADIX                                                    \
    );                                                                        \
        POLICY = new(low, high, exclude, radix);                              \
    endfunction: POLICY
```