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

When working with randomized objects in SystemVerilog, it is typically desired to control the randomization of the item being randomized. The common method of doing so is by using constrained random value generation. Constraints for the randomization of class properties can be provided within the class definition, which will require randomized properties to always pass the specified constaint check. For example, the following **addr_txn** class contains a constraint on the **addr** property to ensure it is always word aligned.

```
class addr_txn;
    rand bit [31:0] addr;
    rand int        size;

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

Fig. 1: A basic address transaction class

Constraints can also be provided outside the class definition at the point of randomization by using the **with** construct. This allows variation to the constraints based on the context of the randomization. For example, the same **addr_txn** class can have an instance of it randomized with the following constraint on its **data** property.

```
addr_txn txn = new();
txn.randomize() with {addr == 32'hdeadbeef;};
```

Fig. 2: En example of an inline constraint

Policy classes are a technique for SystemVerilog constraint layering with comparable performance to traditional constraint methods, while additionally providing portable and reusable constraints that can be mixed and matched into the object being randomized, originally presented by John Dickol [1] [2]. This is possible through SystemVerilog “Global Constraints”, which randomizes all lower-level objects whenever a top-level object is randomized, and solves all rand variables and constraints across the entire object hierarchy simultaneously. A summary of his approach can be found by examining 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. 3: The **policy_base** and **policy_list** classes

These first two classes provide the base class types for the actual policies. This allows different, related policies to be grouped together in **policy_list** and applied to the randomization call concurrently. Both of these classes are parameterized to a specific object type, so separate instances will be required for each class type that needs to be constrained.

```
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. 4: Policies for constraining the **addr_txn** class

The next three classes contain implementations of the **policy_base** class 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_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

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. 5: The **addr_txn** class and a constrained subclass

The final two classes show how these policies are used. The **addr_txn** class creates a **policy** queue, and during the **pre_randomize** function sets the handle to the **item** object within each policy to itself. The **addr_constrained_txn** subclass adds two policies to a local **pcy** list, 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 can then passes the local **pcy** list to the parent **policy** queue.

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 policies provided.

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

Fig. 6: 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, he 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 policies are specialized using parameterization to the class they constrain, different specializations of parameterized classes cannot be grouped and indexed. This is most apparent when creating a class hierarchy. If you wanted to extend a class and add a new field to constrain then you need a new policy type. The new policy type requires an extra policy list to modify any new attributes in the child class and requires the user to know which layer of the class hierarchy defines each attribute they want to constrain. Imagine a complex class hierarchy with many layers of inheritance and extension, and new 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 you need to manage a separate policy list for each level of the hierarchy and know what each list can and cannot contain. 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;
  rand bit [31:0] addr;
  rand int        size;
  rand policy_base#(addr_txn) addr_policy[$];

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

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

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

  constraint c_parity {parity == $countones(addr) % 2;}

  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 = new;
    addr_prohibit_policy      prohibit = new;
    addr_parity_policy        parity = new;
    policy_list#(addr_txn)    addr_pcy = new;
    policy_list#(addr_p_txn)  addr_p_pcy = new;

    permit.add('h00000000, 'h0000FFFF);
```

```
    permit.add('h10000000, 'h1FFFFFFF);
    addr_pcy.add(permit);

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

    parity.set(1'b1);
    addr_p_pcy.add(parity);

    this.addr_policy   = {addr_pcy};
    this.addr_p_policy = {addr_p_pcy};
  endfunction
endclass
```

Fig. 7: Modified address transaction classes with a parity checking subclass and policies included

This class ends up with a lot of boilerplate code and is not very intuitive to use. The user needs to keep track of what the difference is between each policy list and apply constraints properly. Each additional subclass in a hierarchy will cause the constrained class to grow further. The solution to this problem is to replace the parameterized policy base classes with a combination of a policy interface class and a parameterized policy implementation class.

```
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. 8: The **policy** interface class and **policy_imp** class

By abstracting the policy functionality into an interface class, all policy implementations under a common base class are given compatibility with each other and can be stored in a shared policy queue. The **uvm_object** class is a good base class when using policies within UVM, as it will likely cover every class you will be randomizing. If working with another framework, a reasonable base class will need to be chosen. Because the **set_item** function no longer receives a parameterized type input, the input needs to be cast to the implementation type. If this cast fails, the item is set to null and randomization is disabled.

```
class addr_policy extends policy_imp#(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_parity_policy extends policy_imp#(addr_p_txn);
   protected bit parity;

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

   function new(int parity);
      this.parity = parity;
   endfunction
endclass

class addr_permit_policy extends addr_policy;
   rand int selection;

   constraint c_addr_permit {
      m_item != null -> (
         selection inside {[0:ranges.size()-1]};

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

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

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

The address policies are updated to inherit from **`policy_imp`** parameterized to the relevant transaction class. Additionally, the constraints are updated to verify the internal item handle is not null.

```
class addr_txn extends uvm_object;
    rand bit [31:0]   addr;
    rand int          size;
    rand policy_queue policy;

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

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

```
    endfunction
endclass

class addr_p_txn extends addr_txn;
  rand bit parity;

  constraint c_parity {parity == $countones(addr) % 2;}
endclass

class addr_constrained_txn extends addr_p_txn;
    function new;
        policy_queue pcy;

        addr_permit_policy   permit   = new();
        addr_prohibit_policy prohibit = new();
        addr_parity_policy   fixed_p;

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

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

        fixed_p = new(1'b1);
        pcy.push_back(fixed_p);

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

Fig. 10: Address transaction classes updated to use the new policy implementation

The new address transaction classes are simplified considerably. Only a single policy queue is required in the class hierarchy, and the constraint block does not need to keep track of any specialized lists. The **addr_txn** class now subclasses **uvm_object** to provide compatibility with the policy interface.

III. Problem #2: Definition Location

The second problem is "where do I define my policy classes?" This is not a complicated problem to solve, and most users likely place their policy classes in a file close to the class they are constraining. However, by explicitly pairing policy definitions with the class they constrain, users never need to hunt for the definitions or guess where they are located, and the definitions are given an appropriate and consistent home in the environment. This is done by embedding a **POLICIES** class inside the class being constrained. This class acts as a container for all relevant policies and allows policies to be created by simply referencing the embedded class through the `::` operator.

```
class addr_txn extends uvm_object;
    rand bit [31:0]   addr;
    rand int          size;
    rand policy_queue policy;

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

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

```
    endfunction

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

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

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

class addr_p_txn extends addr_txn;
    rand bit parity;

    constraint c_parity {parity == $countones(addr) % 2;}

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

        static function addr_parity_policy FIXED_PARITY(bit value);
            FIXED_PARITY = new(value);
        endfunction
    endclass: POLICIES
endclass

class addr_constrained_txn extends addr_p_txn;
    function new;
        policy_queue pcy;

        addr_constrained_txn::POLICIES::addr_permit_policy   permit   = new();
        addr_constrained_txn::POLICIES::addr_prohibit_policy prohibit = new();

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

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

        pcy.push_back(addr_constrained_txn::POLICIES::FIXED_PARITY(1'b1));

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

Fig. 11: Address transaction classes with embedded policies

In the above code, there were few changes from the previous iteration. All the policy classes were migrated into an embedded

POLICIES class, and a static constructor was created for the parity policy to simplify its creation. The parity **POLICIES** class was set as a child of the initial **POLICIES** class, which further simplified the usage of the classes in the constraint block. Finally, policies in the constraint block are accessed through the **addr_constrained_txn::POLICIES::** accessor.

IV. Problem #3: Boilerplate Overload

The third problem 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, range constraints, 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.

```
// Create a base embedded POLICIES class
`define start_policies(cls)                                                 \
class POLICIES;                                                             \
`LOCAL_POLICY_IMP(cls)

// Create a child embedded POLICIES class
`define start_extended_policies(cls, parent)                                \
class POLICIES extends parent::POLICIES;                                    \
`LOCAL_POLICY_IMP(cls)

// End the embedded POLICIES class
`define end_policies                                                        \
endclass: POLICIES

// Create the base policy type
`define LOCAL_POLICY_IMP(cls)                                               \
    typedef policy_imp#(cls) base_policy;


// Fixed-value policy class and constructor macro
`define fixed_policy(POLICY, TYPE, field)                                   \
`FIXED_POLICY_CLASS(POLICY, TYPE, field)                                    \
`FIXED_POLICY_CONSTRUCTOR(POLICY, TYPE, field)

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

`define FIXED_POLICY_CONSTRUCTOR(POLICY, TYPE, field)                       \
    static function POLICY``_policy POLICY(TYPE value);                     \
        POLICY = new(value);                                                \
```

```
    endfunction: POLICY                                                    \
```

Fig. 12: Macros for setting up the embedded **POLICIES** class and a fixed value policy

The **start_policies**, **start_extended_policies**, and **end_policies** macros replace the embedded **class POLICIES** and **endclass** statements within the constrained class. They set up class inheritance as needed and create a local typedef for the **policy_imp** parameterized type. The **fixed_policy** macro wraps two additional macros responsible for creating the policy class and a static constructor for the class. In this example, a policy class that lets you constrain a property to a fixed value is defined.

```
class addr_txn extends uvm_object;
    rand bit [31:0]   addr;
    rand int          size;
    rand policy_queue policy;

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

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

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

class addr_p_txn extends addr_txn;
    rand bit parity;

    constraint c_parity {parity == $countones(addr) % 2;}

    `start_extended_policies(addr_p_txn, addr_txn)
        `fixed_policy(FIXED_PARITY, bit, parity)
    `end_policies
endclass

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

Fig. 13: 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** 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 find an exact diagnosis of the issue, we were focused on transitioning our policy approach to a "use once and discard" approach, where we would treat a policy as disposable and model them as extremely lightweight classes. This change happened to solve the randomization issue as well.

We expected 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** . 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_policy FIXED_PARITY(bit value);
    FIXED_PARITY = new(value);
endfunction
...
pcy.push_back(addr_constrained_txn::POLICIES::FIXED_PARITY(1'b1));
```

Fig. 14: Example static constructor function from the **FIXED_PARITY** 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. 15: Example static constructor function from the **FIXED_PARITY** policy 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 efficient to use.

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

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

```
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
```

Fig. 16: 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. 17: **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** *and* **policy_container**

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. 18

The **policy_queue** type is capable of storing any policy that implements the **policy** interface. The default queue methods are sufficient for basic usage, but by introducing another interface class, the direct management of the queue can be abstracted away from the user.

```
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
```

Fig. 19: The **policy_container** interface class

This interface class should be implemented by any class that needs to manage a policy queue. 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.

D. Creating a **policy_container** *base class implementation*

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();
        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
```

Fig. 20: A **policy_object** base class implementation

This base class implements all the **policy_container** functionality. Treating it as a mixin allows you to apply it to different categories of transactions.

```
class base_txn extends policy_object #(uvm_sequence_item);

class base_seq #(type REQ=uvm_sequence_item, RSP=REQ) extends policy_object #(
    uvm_sequence#(REQ, RSP) );

class cfg_object extends policy_object #(uvm_object);
```

Fig. 21: Example classes using the **policy_object** base class

Each of these classes will contain a separate policy queue, but each will share an API and can be used in the same way. The **policy_object** base class can be used as a mixin to any class that needs to manage a policy queue.

E. Protecting the policy queue

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 rountines, 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*

```
`ifndef __POLICY_PKG__
`define __POLICY_PKG__

`include "uvm_macros.svh"

package policy_pkg;

    import uvm_pkg::*;

    `include "policy.svh"

    typedef policy policy_queue[$];

    `include "policy_container.svh"

    `include "policy_imp.svh"
    `include "policy_object.svh"

endpackage: policy_pkg

`endif
```

B. *policy.svh*

```
`ifndef __POLICY__
`define __POLICY__

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

`endif
```

C. *policy_imp.svh*

```
`ifndef __POLICY_IMP__
`define __POLICY_IMP__

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;

        if (item == null) return( 0 );
        else return( $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(
                    "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

endclass: policy_imp
```

```
`endif
```

D. policy_container.svh

```
`ifndef __POLICY_CONTAINER__
`define __POLICY_CONTAINER__

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

`endif
```

E. policy_object.svh

```
`ifndef __POLICE_OBJECT__
`define __POLICE_OBJECT__

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
```

```
21
22      virtual function void add_policies(policy_queue policies);
23          foreach( policies(i) ) try_add_policy( policies[i] );
24      endfunction: add_policies
25
26      virtual function void clear_policies();
27          if ( this.has_policies() ) begin
28              `uvm_info( "policy", $sformatf("clearing [%0d] policies from %s",
    ↪   this.m_policies.size(), this.get_name()), UVM_FULL)
29          end
30          this.m_policies = {};
31      endfunction: clear_policies
32
33      // Access
34      virtual function policy_queue get_policies();
35          return( this.m_policies );
36      endfunction: get_policies
37
38      // Copy
39      virtual function policy_queue copy_policies();
40          copy_policies = {};
41          foreach( this.m_policies[i] ) begin
42              copy_policies.push_back( this.m_policies[i].copy );
43          end
44      endfunction: copy_policies
45
46      protected function void try_add_policy( policy new_policy );
47          if (new_policy.item_is_compatible(this)) begin
48              `uvm_info( "policy", $sformatf("adding policy %s to %s",
    ↪   new_policy.name, this.get_name()), UVM_FULL)
49              new_policy.set_item( this );
50              this.m_policies.push_back( new_policy );
51          end else begin
52              `uvm_warning( "policy", $sformatf("policy %s not compatible with target
    ↪   %s", new_policy.name, this.get_name()))
53          end
54      endfunction: try_add_policy
55  endclass: policy_object
56
57  `endif
```

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 "ranged_policy.svh"
`include "set_policy.svh"

`endif
```

B. *constant_policy.svh*

```
`ifndef __CONSTANT_POLICY__
`define __CONSTANT_POLICY__

// Full policy definition
`define constant_policy(POLICY, TYPE, field, constant)                          \
`CONST_POLICY_CLASS(POLICY, TYPE, field, constant)                              \
`CONST_POLICY_CONSTRUCTOR(POLICY)

// Policy class definition
`define CONST_POLICY_CLASS(POLICY, TYPE, field, constant)                       \
    class POLICY``_policy extends base_policy                                   \
        typedef TYPE l_field_t;                                                 \
                                                                                \
        virtual function string name();                                         \
```

```
            return (`"POLICY`");                                                \
        endfunction: name                                                       \
                                                                                \
        virtual function string description();                                  \
            return (`"(field==constant)`");                                     \
        endfunction: description                                                \
                                                                                \
        virtual function policy copy();                                        \
            copy = new();                                                       \
        endfunction: copy                                                       \
                                                                                \
        constraint c_const_value {                                              \
            (item != null) -> (item.field == l_field_t'(constant));             \
        }                                                                       \
                                                                                \
        function new();                                                         \
        endfunction: new                                                        \
    endclass: POLICY``_policy

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

`endif
```

C. *fixed_policy.svh*

```
`ifndef __FIXED_POLICY__
`define __FIXED_POLICY__

// Full policy definition
`define fixed_policy(POLICY, TYPE, field, RADIX="%0p")                          \
`FIXED_POLICY_CLASS(POLICY, TYPE, field, RADIX)                                 \
`FIXED_POLICY_CONSTRUCTOR(POLICY, TYPE, RADIX)

// Policy class definition
`define FIXED_POLICY_CLASS(POLICY, TYPE, field, RADIX="%0p")                    \
    class POLICY``_policy extends base_policy                                   \
        typedef TYPE        l_field_t;                                          \
                                                                                \
        protected TYPE      m_fixed_value;                                      \
        protected string    m_radix=RADIX;                                      \
                                                                                \
        constraint c_fixed_value {                                              \
            (item != null) -> (item.field == l_field_t'(m_fixed_value));        \
        }                                                                       \
                                                                                \
        function new(TYPE value, string radix=RADIX);                           \
            this.set_value(value);                                              \
            this.set_radix(radix);                                              \
        endfunction: new                                                        \
                                                                                \
```

```
        virtual function string name();                                    \
            return ({                                                      \
                `"POLICY(field==",                                         \
                $sformatf(m_radix, m_fixed_value),                         \
                `")`"                                                      \
            });                                                            \
        endfunction: name                                                  \
                                                                           \
        virtual function void set_value(TYPE value);                      \
            this.m_fixed_value = value;                                    \
        endfunction: set_value                                             \
                                                                           \
        virtual function TYPE get_value();                                 \
            return (this.m_fixed_value);                                   \
        endfunction: get_value                                             \

        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 FIXED_POLICY_CONSTRUCTOR(POLICY, TYPE, RADIX="%0p")                \
    static function POLICY``_policy POLICY(TYPE value, string radix=RADIX);\
        POLICY = new(value, radix);                                        \
    endfunction: POLICY

`endif
```

D. ranged_policy.svh

```
`ifndef __RANGED_POLICY__
`define __RANGED_POLICY__

// Full policy definition
`define ranged_policy(POLICY, TYPE, field, RADIX="%0p")                    \
`RANGED_POLICY_CLASS(POLICY, TYPE, field, RADIX)                           \
`RANGED_POLICY_CONSTRUCTOR(POLICY, TYPE, RADIX)

// Policy class definition
`define RANGED_POLICY_CLASS(POLICY, TYPE, field, RADIX="%0p")              \
    class POLICY``_policy extends base_policy                              \
        typedef TYPE        l_field_t;                                     \
                                                                           \
        protected TYPE      m_low;                                         \
        protected TYPE      m_high;                                        \
        protected bit       m_inside;                                      \
        protected string    m_radix=RADIX;                                 \
                                                                           \
        constrant c_ranged_value {                                         \
```

```
            (item != null) ->                                                   \
                ((!m_inside) ^ (item >= m_low && item <= m_high));              \
        }                                                                       \
                                                                                \
        function new(                                                           \
            TYPE   low,                                                         \
            TYPE   high=low,                                                    \
            bit    inside=1'b1,                                                 \
            string radix=RADIX                                                  \
        );                                                                      \
            this.set_range(low, high);                                          \
            this.set_inside(inside);                                            \
            this.set_radix(radix);                                              \
        endfunction: new                                                        \
                                                                                \
        virtual function string name();                                         \
            return ({                                                           \
                `"POLICY(field ",                                               \
                m_inside ? "inside [" : "outside [",                            \
                $sformatf(m_radix, m_low),                                      \
                ", ",                                                           \
                $sformatf(m_radix, m_high),                                     \
                `"])`"                                                          \
            });                                                                 \
        endfunction: name                                                       \
                                                                                \
        virtual function void set_range(TYPE low, TYPE high);                   \
            if (low <= high) begin                                              \
                this.m_low = low;                                               \
                this.m_high = high;                                             \
            end else begin                                                      \
                this.m_low = high;                                              \
                this.m_high = low;                                              \
            end                                                                 \
        endfunction: set_range                                                  \
                                                                                \
        virtual function TYPE get_low();                                        \
            return (this.m_low);                                                \
        endfunction: get_low                                                    \
                                                                                \
        virtual function TYPE get_high();                                       \
            return (this.m_high);                                               \
        endfunction: get_high                                                   \
                                                                                \
        virtual function void set_inside(bit inside);                           \
            this.m_inside = inside;                                             \
        endfunction: set_inside                                                 \
                                                                                \
        virtual function bit get_inside();                                      \
            return (this.m_inside);                                             \
        endfunction: get_inside                                                 \
                                                                                \
        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 RANGED_POLICY_CONSTRUCTOR(POLICY, TYPE, RADIX="%0p")                    \
    static function POLICY``_policy POLICY(                                     \
        TYPE   low,                                                             \
        TYPE   high=low,                                                        \
        bit    inside=1'b1,                                                     \
        string radix=RADIX                                                      \
    );                                                                          \
        POLICY = new(low, high, inside, radix);                                 \
    endfunction: POLICY

`endif
```

E. set_policy.svh

```
`ifndef __SET_POLICY__
`define __SET_POLICY__

// Full policy definition
`define set_policy(POLICY, TYPE, field, RADIX="%0p")                            \
`SET_POLICY_CLASS(POLICY, TYPE, field, RADIX)                                   \
`SET_POLICY_CONSTRUCTOR(POLICY, TYPE, RADIX)

// Policy class definition
`define SET_POLICY_CLASS(POLICY, TYPE, field, RADIX="%0p")                      \
    class POLICY``_policy extends base_policy                                   \
        typedef TYPE        l_field_array_t[];                                  \
                                                                                \
        protected l_field_array_t   m_values;                                   \
        protected bit               m_inside;                                   \
        protected string            m_radix=RADIX;                              \
                                                                                \
        constrant c_set_value {                                                 \
            (item != null) ->                                                   \
                ((m_inside) ^ (item.field inside {m_values}));                  \
        }                                                                       \
                                                                                \
        function new(                                                           \
            l_field_array_t values,                                             \
            bit             inside=1'b1,                                        \
            string          radix=RADIX                                         \
        );                                                                      \
            this.set_values(values);                                            \
            this.set_inside(inside);                                            \
            this.set_radix(radix);                                              \
        endfunction: new                                                        \
                                                                                \
```

```
        virtual function string name();                                          \
            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 ",                                                \
                m_inside ? "inside {" : "outside {",                             \
                values_str,                                                      \
                `"})`"                                                           \
            });                                                                  \
        endfunction: name                                                        \
                                                                                 \
        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_inside(bit inside);                            \
            this.m_inside = inside;                                              \
        endfunction: set_inside                                                  \
                                                                                 \
        virtual function bit get_inside();                                       \
            return (this.m_inside);                                              \
        endfunction: get_inside                                                  \
                                                                                 \
        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 SET_POLICY_CONSTRUCTOR(POLICY, TYPE, RADIX="%0p")                        \
    typedef TYPE POLICY``_array_t[];                                             \
    static function POLICY``_policy POLICY(                                      \
        POLICY``_array_t values,                                                 \
        bit              inside=1'b1,                                            \
        string           radix=RADIX                                             \
    );                                                                           \
        POLICY = new(values, inside, radix);                                     \
```

```
    endfunction: POLICY

`endif
```