

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 constraint check. For example, the following `addr_txn` class contains a constraint on the `size` property to ensure the address is always word aligned.



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 `addr` property.

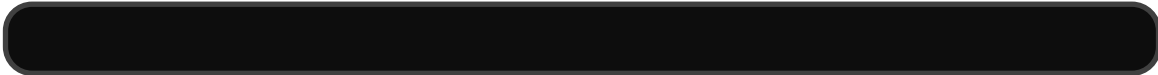


Fig. 2: En example of an inline constraint

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 [?] [?]. 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 John’s approach can be found by examining the following code.



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.



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.



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 pass 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.



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 [?]. 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 [?]. Additionally, Chuck 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 [?]). 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:



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 with a non-parameterized base and a parameterized extension. We chose to use an interface class as our non-parameterized container for the flexibility that offers over inheriting from a virtual base class—namely that our parameterized base is free to implement other interfaces, and should the need ever arise, other classes might implement the policy interface.



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

Implementing a non-parameterized base (in this case, an interface class) enables all policies targeting a particular class hierarchy to be stored within a single common **policy_queue** implemented by the base class of the target class hierarchy. 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.

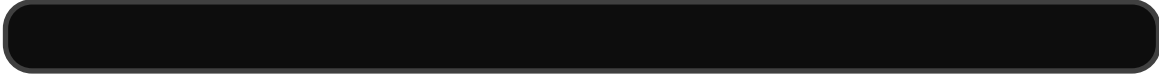


Fig. 9: Address policies updated to use the new policy implementation

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.



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.



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 error 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.

A nice side effect of embedding policy class definitions within their target classes is that policies have access to protected members of the target class. This enables meta members of the target class (e.g., “knobs” such as **parity_err**) to be defined as protected, preventing external classes from manipulating them directly. 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 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 reduce 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.




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.




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_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 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 apply fresh policy instances before re-randomizing a target object. This change happened to solve the randomization issue as well.

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**. 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.




Fig. 14: 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.




Fig. 15: 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 efficient to use.

A. Expanding the `policy` interface class

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



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`

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.



Fig. 18: 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.



Fig. 19: 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.



Fig. 20: A `policy_object` base class implementation

This base class implements all the **policy_container** functionality. 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.



Fig. 21: 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.

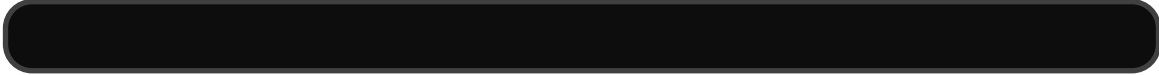
The complete policy package is available in Appendix ??, with additional macro examples available in Appendix ?. A functional package is also available for download [?] which can be included directly in a project to start using policies immediately.

REFERENCES

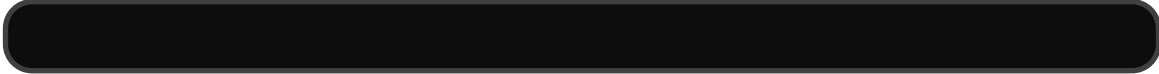
- [1] J. Dickol, "SystemVerilog Constraint Layering via Reusable Randomization Policy Classes," DVCon, 2015.
- [2] J. Dickol, "Complex Constraints: Unleashing the Power of the VCS Constraint Solver," SNUG Austin, September 29, 2016.
- [3] K. Vasconcellos, J. McNeal, "Configuration Conundrum: Managing 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. Spratt, "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*



B. *policy.svh*



C. *policy_imp.svh*



D. *policy_container.svh*



E. *policy_object.svh*



APPENDIX B
POLICY MACROS

A. *policy_macros.svh*



B. *constant_policy.svh*



C. *fixed_policy.svh*



D. *ranged_policy.svh*



E. *set_policy.svh*

