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

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. Extended Abstract

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]. 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.

There are three problems shared by all of these implementations, and a fourth problem we observed in our original implementation of policies that this paper will address and present solutions for. The first problem 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 extend the target class, 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. The solution to this problem is to introduce class inheritance into the policy classes that mirror the target class hierarchy, supplemented by convenient macros to keep the solution simple.

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, this paper will present a solution that explicitly pairs policy definitions with the class they constrain, so users never need to hunt for the definitions or guess where they are located, and so the definitions have 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.

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, a strategy using macros is presented that 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 the complex constraints, if necessary.

A fourth problem we noticed in our usage of policy classes was occasional unexpected behavior when attempting to reuse a policy. It appeared as though the policies "remembered" being randomized and wouldn't reapply their constraints during subsequent `randomize` calls. An exact diagnosis of the issue was not discovered because we transitioned 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. We implemented practices such as static functions to create new instances of the policies and policy methods to return fresh copies of themselves.

Functional example code will be provided to demonstrate the solutions to these problems and provide other minor improvements to the original policy class implementation.

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.