PyVSC: SystemVerilog-Style Constraints, and Coverage in Python

M. Ballance
Milwaukie, OR, USA
matt.ballance@gmail.com

***Abstract*—Constrained-randomization and functional coverage are key elements in the widely-used SystemVerilog-based verification flow. The use of Python in functional verification is growing in popularity, but Python has historically lacked support for the constraint and coverage features provided by SystemVerilog. This paper describes PyVSC, a library that provides these features.**

Keywords—functional verification, Python, constrained random, functional coverage

I. Introduction

Verification flows based on constrained-random stimulus generation and functional coverage collection have become dominant where SystemVerilog is used for the testbench environment. Consequently, verification engineers have significant experience with the constructs and patterns provided by SystemVerilog.

There is growing interest in making use of Python in verification environments – either as a complete verification environment or as an adjunct to a SystemVerilog verification environment. It is desirable in both of these cases to be able to make use of constrained random stimulus generation and to capture functional coverage. The PyVSC[1] library brings these features to Python, with the goal of providing as similar a user experience as possible to what SystemVerilog-knowledgeable verification engineers are familiar with. The PyVSC library also takes advantage of advances in solver technology to support constraint complexity on par with what is in use in SystemVerilog-based environments.

II. Related Work

Other projects have, of course, sought to bring constrained randomization and/or functional coverage collection to languages used for functional verification that lack built-in support.

A. cocotb-coverage

Possibly the most relevant of these packages, given the target of Python, is the cocotb-coverage library [2][3]. This library supports basic randomization and functional coverage collection. The constraints used by the cocotb-coverage library are captured as executable Python methods and are evaluated using a pure-Python SAT solver. This approach to capturing and solving constraints restricts each field to having no more than two constraints, and can result in poor performance and capacity.

Coverage is captured as a set of individual coverpoint instances, and is not organized into covergroup types and instances in the same way that SystemVerilog does.

B. SystemC

SystemC has been one of the longer-lasting languages used for modeling and verifying hardware that is built as an embedded domain-specific language within a general-purpose programming language (C++). While libraries for constraints and coverage targeting SystemC are not directly applicable in Python, it's worthwhile considering approaches taken.

The SystemC Verification Library (SCV) provides basic randomizations and constraints, using a Binary Decision Diagram (BDD) solver. BDD-based constraint solvers are known to have performance and capacity challenges, especially with sizable constraint spaces. No support for capturing functional coverage is provided.

The CRAVE [4] library sought to bring higher performance and capacity to randomization in SystemC using Satifiability Modulo Theories (SMT) constraint solvers. While there is some evidence that some support for functional coverage was developed [5], this is not part of the published CRAVE library.

III. Modeling Constrained-Random Stimulus

PyVSC is considered an embedded domain-specific language (eDSL) because it effectively extends the Python language with new semantics for constraint solving and coverage capture. There are several key Python language features that are used to layer these new semantics on top of the Python language. Library classes and methods are used to capture some constraints and coverage constructs. Python decorators are used to identify Python classes and methods as having specific significance. Python introspection is used to obtain source locations for declarations, to automatically-assign names to elements, and to dynamically modify user-declared classes. Python 'with' statements are used to identify statement blocks. PyVSC uses the Boolector SMT solver[6] to solve the user-specified constraints.

A. Data Fields and Classes

Both SystemVerilog and PyVSC use data fields with a specified bit width. This is important to ensure that both constraints and coverage constructs are interpreted correctly. PyVSC supports creating data elements individually, but data elements are typically grouped with constraints in randomizable classes.

```
@vsc.randobj
class my_s(object):

    def __init__(self):
        self.a = vsc.rand_bit_t(8)
        self.b = vsc.rand_uint8_t()

    @vsc.constraint
    def ab_c(self):
        self.a < self.b
```

Figure 1 - Randomizable Class Declaration

As shown in Figure 1, a randomizable class is identified with the @vsc.randobj decorator. This builds infrastructure into the class to support randomization, and ensures that constraint elaboration occurs after a class instance is created.
Class members that can be constrained are created using methods provided by the library. The width of data fields can either be specified directly, or via convenience methods such as vsc.rand_uint8_t that create data fields with commonly-used widths.

B. Class Constraints

Constraints are specified as Python statements within Python methods decorated with @vsc.constraint. Constraint methods are evaluated once to construct a constraint data model from the statements within the method.

```
@vsc.constraint
    def ab_c(self):
        with vsc.if_then(self.a == 1):
            self.b == 1
        with vsc.else_if(self.a == 2):
            self.b == 2
        with vsc.else_if(self.a == 3):
            self.b == 4
        with vsc.else_if(self.a == 4):
            self.b == 8
        with vsc.else_if(self.a == 5):
            self.b == 16
```

Figure 2 - If/else Constraints

The difference between Python statements and PyVSC constraints is most visible when control-flow constraints are used, as shown in Figure 2. In these cases, Python statements cannot be used directly. It is necessary to use the PyVSC-provided construct to capture the constraint intent.

Just as with SystemVerilog, constraint blocks are considered virtual, in that a same-named constraint in a sub-class overrides the constraint in the super-class.

```
@vsc.randobj
class my_base_s(object):

    def __init__(self):
        self.a = vsc.rand_bit_t(8)
        self.b = vsc.rand_bit_t(8)

    @vsc.constraint
    def ab_c(self):
       self.a < self.b

@vsc.randobj
class my_ext_s(my_base_s):

    def __init__(self):
        super().__init__()

    @vsc.constraint
    def ab_c(self):
       self.a > self.b
```

Figure 3 - Overriding Constraints

In the example in Figure 3, the relationship *a* > *b* will hold for all instances of class *my_ext_s* because the *ab_c* constraint in the *my_ext_s* type overrides the *ab_c* constraint in the *my_base_s* type.

Figure 4 summarizes the constraint statements currently supported by PyVSC and those supported by SystemVerilog.

Feature	SystemVerilog	PyVSC
Algebraic constraints	Y	Y
Integer fields	Y	Y
Enum fields	Y	Y
Fixed-size arrays	Y	Y
Variable-size arrays	Y	Y
dist constraint	Y	Y
soft constraint	Y	Y
inside constraint	Y	Y
solve ordering	Y	N
unique constraint	Y	Y
foreach constraint	Y	Y
constraint_mode	Y	Y
rand_mode	Y	Y
if/else constraint	Y	Y

Figure 4 - Supported Constraint Statements

C. Randomizing a Data Field

PyVSC randomziable classes have *randomize* and *randomize_with* methods that are used for randomizing fields within that class.

```
@vsc.randobj
class my_base_s(object):

     def __init__(self):
         self.a = vsc.rand_bit_t(8)
         self.b = vsc.rand_bit_t(8)

     @vsc.constraint
     def ab_c(self):
        self.a < self.b

item = my_base_s()
for i in range(10):
    with item.randomize_with() as it:
        it.a == i
```

Figure 5 - Randomizing a Class with Additional Constraints

PyVSC supports randomization with additional constraints, as well as randomization considering just the constraints declared within the class. Figure 5 shows an example of adding additional constraints. The *randomize_with* class method is called using a Python *with* clause. Additional constraints are specified within the *with* clause. All legal constraints supported by PyVSC may be used with an inline constraint.

```
@vsc.randobj
 class my_item(object):

     def __init__(self):
         self.a = vsc.rand_bit_t(8)
         self.b = vsc.rand_bit_t(8)

     @vsc.constraint
     def valid_ab_c(self):
        self.a < self.b

item = my_item()

# Always generate valid values
for i in range(10):
   item.randomize()

item.valid_ab_c.constraint_mode(False)

# Allow invalid values
for i in range(10):
   item.randomize()
```

Figure 6 - Using constraint_mode to disable constraints

PyVSC also supports constraint-control methods, such as *rand_mode* for variables and *constraint_mode* for constraint blocks. Figure 6 shows using constraint_mode to temporarily disable a constraint block.

IV. Modeling Functional Coverage

Modeling functional coverage constructs with PyVSC follows the same patterns used for modeling randomizable classes and constraints.

A. Declaring a Covergroup

A covergroup type is declared with PyVSC as a regular Python class decorated with the @vsc.covergroup decorator.

```
@vsc.covergroup
class my_covergroup(object):

    def __init__(self):
        self.with_sample(
             a=vsc.bit_t(4)
             )
        self.cp1 = vsc.coverpoint(
            self.a, bins={
                "a" : vsc.bin(1, 2, 4),
                "b" : vsc.bin(8, [12,15])
            })
```

Figure 7 - Declaring a covergroup type

PyVSC covergroups support several mechanisms to provide coverage data for sampling. Figure 7 shows using the *with_sample* method, which creates a method named *sample* on the covergroup class. Coverage data is provided via method parameters each time the sample function is called.

```
@covergroup
 class my_covergroup(object):

     def __init__(self, a):
         super().__init__()

         self.cp1 = coverpoint(a,
             bins=dict(
                 a = bin_array([], [1,15])
             ))
 a = 0;
 cg = my_covergroup(lambda:a)
 a=1
 cg.sample() # Hit the first bin of cp1
```

Figure 8 - Binding sampling data at instantiation

Another approach is shown in Figure 8. In this case, sampling data is provided via a lambda function that is specified when the covergroup is instanced. Calling 'sample' on the covergroup causes that lambda to be called and read the current value of the data to be sampled.

B. Specifying Coverpoints and Bins

Coverpoints and bins are specified in the covergroup class constructor. User-specified single bins and arrays of bins are supported. If no bins are specified, bins are automatically partitioned according to the auto-bin max.

```
@vsc.covergroup
class my_covergroup(object):

  def __init__(self):
    self.with_sample(
      a=vsc.bit_t(4),
      b=vsc.bit_t(4))
    self.cp1 = vsc.coverpoint(self.a, bins={
      "a" : vsc.bin_array([], [1,15])
    })
    self.cp2 = vsc.coverpoint(self.b, bins={
      "b" : vsc.bin_array([], [1,15])
    })

    self.cp1X2=vsc.cross([self.cp1, self.cp2])
```

Figure 9 - Coverpoint Cross

Coverpoints crosses are also supported, as shown in Figure 9.

A summary of coverage features supported by PyVSC and SystemVerilog is shown below.

Feature	SystemVerilog	PyVSC
covergroup type	Y	Y
coverpoint bins	Y	Y
coverpoint ignore_bins	Y	N
coverpoint illegal_bins	Y	N
coverpoint single bin	Y	Y
coverpoint array bin	Y	Y
coverpoint auto bins	Y	Y
coverpoint transition bin	Y	N
cross auto bins	Y	Y
cross bin expressions	Y	N
cross explicit bins	Y	N
cross ignore_bins	Y	N
cross illegal_bins	Y	N

V. Working with Coverage Data

Once coverage data has been collected, it can be displayed as a simple text report that shows the coverage achieved by each covergroup instance as well as the total coverage collected by all instances of a covergroup type. Coverage data can also be saved to a Unified Coverage Interchange Standard (UCIS) XML file for processing by other tools.

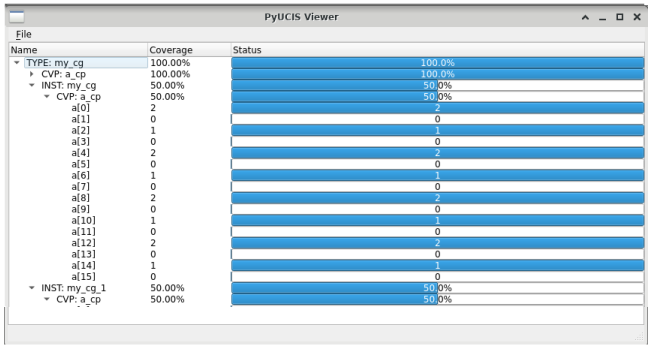

Figure 10 - Coverage Data Visualized

Figure 10 shows a graphical view of coverage data captured in UCIS XML format.

VI. Future Work

While PyVSC is currently in a usable state, there are features that are still on the near-term roadmap, as well as extensions to be considered. The major areas of effort are listed below.

A. *Complete supporting SystemVerilog Constructs*

The highest priority for future work with PyVSC is to complete implementation of missing features that SystemVerilog supports. Currently, most of the constraint constructs are supported. More work exists to complete support of all functional coverage constructs.

B. *Beyond SystemVerilog*

Thus far, PyVSC has focused on achieving feature parity with the constraint and functional coverage constructs supported by SystemVerilog. But, supporting these features using a library in the context of a general-purpose programming language leaves open the possibility to go much further.

Coverage and constraint constructs are built into the SystemVerilog language. While this makes them easily accessible as first-class language features, it also makes them difficult or impossible to use programmatically. One interesting direction to explore with PyVSC is the hybrid space where user-specified constraints are programmatically processed to build more complex constraints. Because PyVSC constraint and coverage constructs are implemented in terms of Python language features, supporting programmatic enhancement of the existing constructs is natural.

C. *Performance Improvements*

PyVSC is a pure-Python library built on top of the Boolector SMT solver. Since the Boolector solver does most of the compute-bound work of constraint solving, PyVSC's performance is quite good. That said, there are cases – specifically with arrays and iterative constraints – where Python's interpreted nature leads to slower than desirable performance.

A natural next step is to re-implement the core constraint and coverage data mode in C++ and expose it as a Python extension. Early experiments with this approach have shown significant performance improvements are possible in cases where the number of variables is significant.

VII. Conclusion

Randomization and functional coverage are key to industrial functional verification practice. PyVSC brings these features to Python in a form that will be familiar to SystemVerilog users. This boosts the capabilities of Python-based verification by making it easier for SystemVerilog practitioners to reuse their knowledge of constraints and coverage in Python.

References

[1] M. Ballance, PyVSC, GitHub, https://github.com/fvutils/pyvsc,

[2] M. Cieplucha, W. Pleskacz, "New Constrained Random and Metric-Driven Verification Methodology Using Python", DVCon Europe 2017 http://events.dvcon.org/2017/proceedings/papers/02_3.pdf

[3] M. Cieplucha, W. Pleskacz, cocotb-coverage, GitHub, https://github.com/mciepluc/cocotb-coverage

[4] F. Haedicke, H. M. Le, D. Große, and R. Drechsler, "CRAVE: An advanced constrained random verification environment for SystemC,". In International Symposium on System-on-Chip, pp. 1-7, 2012

[5] H. M. Le, R. Dreschler, "Boosting SystemC-based Testbenches with Modern C++ and Coverage-Driven Generation", Conference: DVCon Europe 2015

[6] R. Brummayer, A. Biere, "Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays", Conference: Tools and Algorithms for the Constructions and Analysis of Systems (TACAS) 2009