PyVSC Documentation

Release 0.0.1

Matthew Ballance and Contributors

Sep 12, 2023

CONTENTS

Contents:

CHAPTER

ONE

INTRODUCTION

1.1 What is PyVSC?

PyVSC is a Python library that implements random verification-stimulus generation and coverage collection. PyVSC provides this capability in two forms: an object-oriented Model API, and a Python-embedded domain-specific language built on top of the Model API. This allows coverage and randomization features to be programmatically built, defined with user-friendly constructs, or defined using a mix of the two.

1.1.1 Blog Posts

One great way to get an overview of PyVSC is to read a series of blog posts about PyVSC. Links are below:

- The fundamentals of modeling stimulus and functional coverage in Python.
- Modeling verification data types in Python.
- Modeling and capturing functional coverage in Python.
- Making use of captured coverage data.
- Python Verification Stimulus and Coverage: Constraints.

1.1.2 Papers

- PyVSC: SystemVerilog-Style Constraints and Coverage in Python.

Currently, the Python-embedded domain-specific language supports similar features to those supported by SystemVerilog. Not all SystemVerilog features are supported, but in some cases features not supported by SystemVerilog are also supported. Please see the following section *PyVSC Features* for a comparison of the user-level coverage and randomization features supported by PyVSC compared to SystemVerilog.

Here is a quick example showing capturing random data fields, constraints, coverage, and inline randomization.

```
@vsc.randobj
class my_item_c(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 != 0
```

(continues on next page)

(continued from previous page)

```
        self.a <= self.b
        self.b in vsc.rangelist(1,2,4,8)

@vsc.covergroup
class my_cg(object):

    def __init__(self):
        # Define the parameters accepted by the sample function
        self.with_sample(dict(
            it=my_item_c()
         ))

         self.a_cp = vsc.coverpoint( self.it.a, bins=dict(
            # Create 4 bins across the space 0..255
            a_bins = bin_array([4], [0,255])
         )
         self.b_cp = vsc.coverpoint(self.it.b, bins=dict(
            # Create one bin for each value (1,2,4,8)
            b_bins = bin_array([], 1, 2, 4, 8)
         )
         self.ab_cross = vsc.cross([self.a_cp, self.b_cp])

# Create an instance of the covergroup
my_cg_i = my_cg()

# Create an instance of the item class
my_item_i = my_item_c()

# Randomize and sample coverage
for i in range(16):
    my_item_i.randomize()
    my_cg_i.sample(my_item_i)

# Now, randomize keeping b in the range [1,2]
for i in range(16):
    with my_item_i.randomize_with() as it:
        it.b in vsc.rangelist(1,2)
    my_cg_i.sample(my_item_i)

print("Coverage: %f \%" % (my_cg_i.get_coverage()))
```

1.2 Contributors

CHAPTER

TWO

QUICKSTART GUIDE

2.1 Installing PyVSC

2.1.1 Installation via PyPi

```
pip install pyvsc
```

2.1.2 Installation from Source

```
cd pyvsc
pip install -e .
```

2.2 Running a Simple Example

CHAPTER

THREE

PYVSC DATA TYPES

Generating good random data requires characterizing the data to be randomized. PyVSC provides specific data types to characterize the size and signedness of fields to be used in constraints.

First, a quick example

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

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

    @vsc.constraint
    def ab_c(self):

        self.a in vsc.rangelist(1, 2, 4, 8)

        self.c != 0
        self.d != 0

        self.c < self.d
        self.b in vsc.rangelist(vsc.rng(self.c,self.d))
```

The example above shows using the *rand_bit_t* type to specify class attributes that are random, unsigned (bit), and 8-bits wide.

In much the same way that C/C++ and SystemVerilog provide more than one way to capture types that are equivalent, PyVSC provides several ways of capturing the same type information.

3.1 Scalar Standard-Width Attributes

PyVSC provides a set of standard-width data types, modeled after the types defined in `stdint.h`. Both random and non-random variants of these attribute classes are provided.

Width	Signed	Random	Non-Random
8	Y	*rand_int8_t*	*int8_t*
8	N	*rand_uint8_t*	*uint8_t*
16	Y	*rand_int16_t*	*int16_t*
16	N	*rand_uint16_t*	*uint16_t*
32	Y	*rand_int32_t*	*int32_t*
32	N	*rand_uint32_t*	*uint32_t*
64	Y	*rand_int64_t*	*int64_t*
64	N	*rand_uint64_t*	*uint64_t*

The constructor for the classes above accepts the initial value for the class attribute. By default, the initial value will be 0.

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

    def __init__(self):
        self.a = vsc.rand_uint8_t()
        self.b = vsc.uint16_t(2)
        self.c = vsc.rand_int64_t()
```

In the example above, a random unsigned 8-bit field, a non-random unsigned 16-bit field, and a random signed 64-bit field is created.

3.2 Scalar Arbitrary-Width Attributes

PyVSC provides four classes for constructing arbitrary-width scalar class attributes. The first parameter of the class constructor is the width. The second parameter specifies the initial value for the attribute.

Signed	Random	Non-Random
Y	*rand_int_t*	*int_t*
N	*rand_bit_t*	*bit_t*

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

    def __init__(self):
        self.a = vsc.rand_int_t(27)
        self.b = vsc.rand_bit_t(12)
```

The example above creates a random signed 27-bit attribute and a random unsigned 12-bit attribute.

3.3 Enum-type Attributes

PyVSC supports Python `Enum` and `IntEnum` enumerated types. Attributes are declared using the *enum_t* and *rand_enum_t* classes.

```
class my_e(Enum):
  A = auto()
  B = auto()

@vsc.randobj
class my_s(object):

    def __init__(self):
        self.a = vsc.rand_enum_t(my_e)
        self.b = vsc.enum_t(my_e)
```

3.4 Class-type Attributes

Random and non-random class attributes can be created using classes decorated with *randobj*. Non-random class attributes can optionally be decorated with *attr*.

```
@vsc.randobj
class my_sub_s(object):
    def __init__(self):
        self.a = vsc.rand_uint8_t()
        self.b = vsc.rand_uint8_t()

@vsc.randobj
class my_s(object):

    def __init__(self):
        self.i1 = vsc.rand_attr(my_sub_s())
        self.i2 = vsc.attr(my_sub_s())
```

3.5 Accessing Attribute Values

The value of scalar attributes can be accessed in two ways. All PyVSC scalar attribute types provide a *get_val()* and *set_val()* method. These methods can be called to get or set the current value.

PyVSC also provides operator overloading for *randobj*-decorated classes that allows the value of class attributes to be accessed directly.

3.6 List-type Attributes

Random and non-random class attributes of list type can be created using the *list_t* class. The size of the list can be later changed by appending or removing elements, or clearing the list. Randomizing the array will not change its size.

```
@vsc.randobj
class my_item_c(object):
    def __init__(self):
      self.my_l = vsc.rand_list_t(vsc.uint8_t(), 4)
```

The *randsz_list_t* class creates a list whose size will be randomized when the list is randomized. A list with a randomized size must have a top-level constraint bounding the list size.

```
@vsc.randobj
class my_item_c(object):
    def __init__(self):
      self.my_l = vsc.randsz_list_t(vsc.uint8_t())

    @vsc.constraint
    def my_l_c(self):
        self.my_l.size in vsc.rangelist((1,10))
```

The *list_t* class must be used for any list that will be used in constraints.

```
@vsc.randobj
class my_item_c(object):
    def __init__(self):
      self.a = vsc.rand_uint8_t()
      self.my_l = vsc.list_t(vsc.uint8_t(), 4)

      for i in range(10):
          self.my_l.append(i)

    @vsc.constraint
    def a_c(self):
      self.a in self.my_l

it = my_item_c()
it.my_l.append(20)

with it.randomize_with():
  it.a == 20
```

CHAPTER

FOUR

PYVSC CONSTRAINTS

4.1 Constraint Blocks

Constraint blocks are class methods decorated with the *constraint* decorator. Dynamic constraint blocks are decorated with the *dynamic_constraint* decorator.

Constraint blocks are 'virtual', in that constraints can be overridden by inheritance.

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

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

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

Instances of `my_base_s` will ensure that `a` is less than `b`. Instances of `my_ext_s` will ensure that `a` is greater than `b`.

4.2 Expressions

4.2.1 Dynamic-constraint Reference

Constraint blocks decorated with *constraint* always apply. Dynamic-constraint blocks, decorated with *dynamic_constraint* only apply when referenced. A dynamic constraint is referenced using syntax similar to a method call.

Dynamic constraints provide an abstraction mechanism for applying a condition without knowing the details of what that condition is.

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

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

    @vsc.constraint
    def a_c(self):
        self.a <= 100

    @vsc.dynamic_constraint
    def a_small(self):
        self.a in vsc.rangelist(vsc.rng(1,10))

    @vsc.dynamic_constraint
    def a_large(self):
        self.a in vsc.rangelist(vsc.rng(90,100))

my_i = my_cls()

with my_i.randomize()

with my_i.randomize_with() as it:
    it.a_small()

with my_i.randomize_with() as it:
    it.a_large()

with my_i.randomize_with() as it:
    it.a_small() | it.a_large()
```

The example above defines two dynamic constraints. One ensures that the range of `a` is inside 1..10, while the other ensures that the range of `a` is inside 90..100.

The first randomization call results in a value of a across the full value of `a` (0..100).

The second randomization call results in the value of `a` being 1..10.

The third randomization call results in the value of `a` being 90..100.

The final randomization call results in the value of `a` being either 1..10 or 90..100.

4.2.2 in

PyVSC provides two ways of expressing set-membership constraints. Python's `in` operator may be used directly to express simple cases. More complex cases, including negation of set-membership, may be captured using the *inside* and *not_inside* methods on PyVSC scalar data types.

The `in` constraint ensures that the value of the specified variable stays inside the specified ranges. Both individual values and ranges may be specified. In the example below, the value of `a` will be 1, 2, or 4..8. The value of `b` will be between `c` and `d` (inclusive).

The right-hand side of an 'in' constraint must be a `rangelist` expression. Elements in a `rangelist` may be: - individual expressions - ranges of expressions, using `rng` or a tuple of two expressions - a list of expressions or ranges

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

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

    @vsc.constraint
    def ab_c(self):

       self.a in vsc.rangelist(1, 2, vsc.rng(4,8))
       self.c != 0
       self.d != 0

       self.c < self.d
       self.b in vsc.rangelist(vsc.rng(self.c,self.d))
```

PyVSC scalar data types provide *inside* and *not_inside* methods that to express set membership.

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

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

    @vsc.constraint
    def ab_c(self):

       self.a in vsc.rangelist(1, 2, vsc.rng(4,8))
       self.c != 0
       self.d != 0

       self.c < self.d
       self.b.inside(vsc.rangelist(1, 2, 4, 8))
       self.c.not_inside(vsc.rangelist(1, 2, 4, 8))
```

In the example above, the `b` variable will be inside the range (1,2,4,8). The `c` variable will be outside (ie not equal to)

(1,2,4,8)

4.2.3 Mutable Rangelists

It is sometimes useful to change the value/range list used in an `Membership test operations` constraint between randomizations. The *`rangelist`* class can be constructed as a class member, referenced in constraints, and modified between calls to `randomize`.

The *`rangelist`* class provides three methods to modify the values in a rangelist after it has been created:

- append() – Add a new value or range tuple
- clear() – Remove all previously-added ranges
- extend() – Add a list of values and/or range tuples to the rangelist

```
@vsc.randobj
class Selector():
    def __init__(self):
        self.availableList = vsc.rangelist((0,900))
        self.selectedList = vsc.rand_list_t(vsc.uint32_t(), 15)

    @vsc.constraint
    def available_c(self):
        with vsc.foreach(self.selectedList) as sel:
            sel.inside(self.availableList)

    def getSelected(self):
        '''Returns a sorted list of selected integers.'''
        selected = []
        for resource in self.selectedList:
            selected.append(int(resource))
        selected.sort()
        return selected

selector = Selector()

selector.randomize()

selector.availableList.clear()
selector.availableList.extend([(1000, 2000)])

selector.randomize()
```

In the example above, the rangelist is initially created to contain a value range of 0..900. All values in the `selectedList` produced by the first randomization will fall in this range.

The rangelist is subsequently cleared, and a new range 1000..2000 added. The second randomization will produce values in the 1000..2000 range.

4.2.4 part select

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

    def __init__(self):
        self.a = vsc.rand_bit_t(32)
        self.b = vsc.rand_bit_t(32)
        self.c = vsc.rand_bit_t(32)
        self.d = vsc.rand_bit_t(32)

    @vsc.constraint
    def ab_c(self):

        self.a[7:3] != 0
        self.a[4] != 0
        self.b != 0
        self.c != 0
        self.d != 0
```

4.3 Statements

4.3.1 dist

Distribution constraints associate weights with values or value ranges of the specified variable.

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

    def __init__(self):
        self.a = vsc.rand_uint8_t()

    @vsc.constraint
    def dist_a(self):
        vsc.dist(self.a, [
            vsc.weight(1, 10),
            vsc.weight(2, 20),
            vsc.weight(4, 40),
            vsc.weight(8, 80)])
```

Any otherwise-legal values for the variable that does not have a non-zero weight associated will be excluded from the legal value set. The example above associates non-zero weights with 1, 2, 4, 8. So, a value such as '3' will not be produced.

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

    def __init__(self):
        self.a = vsc.rand_uint8_t()

    @vsc.constraint
```

(continues on next page)

(continued from previous page)

```
    def dist_a(self):
        vsc.dist(self.a, [
            vsc.weight((10,15),  80),
            vsc.weight((20,30),  40),
            vsc.weight((40,70),  20),
            vsc.weight((80,100), 10)])
```

Ranges for weights are specified as a tuple, as shown above.

4.3.2 foreach

foreach constraints are modeled with the *foreach* class. By default, the foreach iterator is a reference to the current element of the array.

```
@vsc.randobj
class my_s(object):
    def __init__(self);
        self.my_l = vsc.rand_list_t(vsc.uint8_t(), 4)

    @vsc.constraint
    def my_l_c(self):
        with vsc.foreach(self.my_l) as it:
            it < 10
```

The *foreach* class supports control over whether the item, index, or both is provided for use in constraints.

Here is an example of requesting the index instead of the iterator.

```
@vsc.randobj
class my_s(object):
    def __init__(self);
        self.my_l = vsc.rand_list_t(vsc.uint8_t(), 4)

    @vsc.constraint
    def my_l_c(self):
        with vsc.foreach(self.my_l, idx=True) as i:
            self.my_l[i] < 10
```

Here is an example of explicitly requesting the iterator.

```
@vsc.randobj
class my_s(object):
    def __init__(self);
        self.my_l = vsc.rand_list_t(vsc.uint8_t(), 4)

    @vsc.constraint
    def my_l_c(self):
        with vsc.foreach(self.my_l, it=True) as it:
            it < 10
```

Now, finally, here is an example of having both an iterator and index.

```
@vsc.randobj
class my_s(object):
    def __init__(self);
        self.my_l = vsc.rand_list_t(vsc.uint8_t(), 4)

    @vsc.constraint
    def my_l_c(self):
        with vsc.foreach(self.my_l, it=True, idx=True) as (i,it):
            it == (i+1)
```

4.3.3 if/else

if/else constraints are modeled using three statements:

- *if_then* – simple if block
- *else_if* – else if clause
- else_then – terminating else clause

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

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

    @vsc.constraint
    def ab_c(self):

        self.a == 5

        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
```

4.3.4 implies

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

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

    @vsc.constraint
    def ab_c(self):

        self.a == 5

        with vsc.implies(self.a == 1):
            self.b == 1

        with vsc.implies(self.a == 2):
            self.b == 2

        with vsc.implies(self.a == 3):
            self.b == 4

        with vsc.implies(self.a == 4):
            self.b == 8

        with vsc.implies(self.a == 5):
            self.b == 16
```

4.3.5 soft

Soft constraints are enforced, except in cases where they violate a hard constraint. Soft constraints are often used to set default values and relationships, which are then overridden by another 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 ab_c(self):
        self.a < self.b
        vsc.soft(self.a == 5)

item = my_item()
item.randomize() # a==5
```

(continues on next page)

(continued from previous page)

```
with item.randomize_with() as it:
  it.a == 6
```

The *soft* constraint applies to a single expression, as shown above. Soft constraints are disabled if they conflict with another hard constraint declared in the class or introduced as an inline constraint.

4.3.6 solve_order

Solve-order constraints are used to provide the user control over value distributions by ordering solve operations. The PyVSC *solve_order* statement corresponds to the SystemVerilog `solve a before b` statement.

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

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

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

        with vsc.if_then(self.a == 0):
            self.b == 4
        with vsc.else_then:
            self.b != 4
```

In the example above, te *solve_order* statement causes b to have values evenly distributed between the value sets [4] and [0..3,5..255].

4.3.7 unique

The *unique* constraint ensures that all variables in the specified list have a unique value.

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

    def __init__(self):
        self.a = vsc.rand_bit_t(32)
        self.b = vsc.rand_bit_t(32)
        self.c = vsc.rand_bit_t(32)
        self.d = vsc.rand_bit_t(32)

    @vsc.constraint
    def ab_c(self):
        self.a != 0
        self.b != 0
        self.c != 0
        self.d != 0

        vsc.unique(self.a, self.b, self.c, self.d)
```

4.4 Customizing Constraint Behavior

In general, the bulk of constraints should be declared inside a class and should always be enabled. However, there are often cases where these base constraints need to be customized slightly when the class is used in a test. PyVSC provides several mechanisms for customizing constraints.

4.4.1 Randomize-With

Classes decorated with the *randobj* decorator are randomized by calling the `randomize` method, as shown in the example below.

```
@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()
item.randomize()
```

PyVSC also provides a `randomize_with` method that allows additional constraints to be added in-line. The example below shows using this to constraint `a` to explicit values.

```
@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
```

4.4.2 Constraint Mode

All constraints decorated with the *constraint* decorator can be enabled and disabled using the *constraint_mode* method. This allows constraints to be temporarily turned off. For example, a constraint that enforces valid ranges for certain variables might be disabled to allow testing design response to illegal values.

```
@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):
   with item.randomize():

item.valid_ab_c.constraint_mode(False)

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

4.4.3 Rand Mode

The random mode of rand-qualified fields can be changed using the *rand_mode* method. This allows randomization of rand-qualified fields to be programmatically disabled.

Due to the operator overloading that PyVSC uses to enable direct access to the value of class attributes, a special mode must be entered in order to access or modify rand_mode.

```
@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()
# Randomize both 'a' and 'b'
for i in range(10):
   with item.randomize():

# Disable randomization of 'a'
```

(continues on next page)

(continued from previous page)

```
with vsc.raw_mode():
    item.a.rand_mode = False

# Randomize only 'b'
for i in range(10):
   with item.randomize():
```

CHAPTER

FIVE

PYVSC COVERAGE

5.1 Covergroups

With PyVSC, a covergroup is declared as a Python class that is decorated with the *covergroup* decorator.

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

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

my_cg_1 = my_covergroup()
my_cg_2 = my_covergroup()
```

Data to be sampled for coverage can be passed to the covergroup as parameters of the *sample* method when the covergroup is sampled, or may be specified as a reference parameter when the covergroup is instanced.

Coverage goals, coverage options, and sampling details are specified within the `__init__` method.

Covergroup instances are created by creating an instance of a *covergroup*-decorated class.

5.1.1 Specifying Covergroup Options

PyVSC covergroups contain an `options` attribute and a `type_options` attribute with which to configure covergroup options. Options may only be changed within the `__init__` method.

Note: `options` and `type_options` attributes are provided, but the values are currently ignored

Option name	Default	Description
name=*string*	""	Unique name	Specifies a name fo the covergroup instance. If unspecified, a unique name will be generated based on the type name.
weight=number	1	Specifies the weight of this covergroup instances relative to other instances.
goal=number	100	Specifies the target goal for this covergroup instance
comment=*string*	""	Specifes a comment for this covergroup
at_least=number	1	Minimum number of hits for each coverage bin
auto_bin_max=number	64	Maximum number of automatically-created bins when bins are not explicitly specified
per_instance=*bool*	False	When true, instance-specific coverage information must be saved for each covergroup instance
get_inst_coverage	False	Only applies when the *merge_instances* type option is set. Enables tracking of per-instance coverage with the *get_inst_coverage* method. When False, *get_coverage* and *get_inst_coverage* return the same value.

Options can be configured in two ways. Options maybe configured within the `__init__` method. They can also be configured after construction, and before the covergroup is sampled for the first time, by referencing the options fields directly.

```
@covergroup
 class my_covergroup(object):

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

 cg1 = my_covergroup(10)
 cg2 = my_covergroup(20)
```

The example above sets the weight of the covergroup to the specified weight passed to `__init__`

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

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

 cg1 = my_covergroup()
 cg1.options.weight=10
 cg2 = my_covergroup()
 cg2.options.weight=20
```

The example above configures the *weight* option by setting it post-cconstruction.

5.2 Coverpoints

A coverpoint is declared using the *coverpoint* method. The name of the coverpoint will be the same as the class attribute to which it is assigned.

The first argument to a coverpoint is its target expression. This can be an expression involving PyVSC-typed variables, or it can be a simple reference to a callable field that returns a value.

5.2.1 Specifying Bins

Bins are specified as a Python `dict`, and passed via the `bins` keyword argument to the coverpoint method. Both individual bins and arrays of bins can be specified.

Individual Bins

Individual bins are specified with the *bin* method. The *bin* method accepts a list of individual values and value ranges that the bin contains.

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

    def __init__(self, a : callable):

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

In the example above, the a bin contains the values 1, 2, 4. The b bin contains the value 8 and the value range 12..15.

Bin Arrays

Bin arrays partition a list of values and ranges into a specified number of bins. Bin arrays are specified using the *bin_array* method. The first parameter to this method specifies how values are to be partitioned. This parameter can be specified either as a number, or a single value in a list. The list format is similar to SystemVerilog syntax.

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

    def __init__(self, a : callable):

        self.cp1 = vsc.coverpoint(a, bins={
            "a" : vsc.bin_array([], 1, 2, 4),
            "b" : vsc.bin_array([4], [8,16])
            })
```

In the example above, bin a will consist of three individual value bins, with a bin for value 1, 2, and 4 respectively. Bin b will consist of four bins, each covering two values of the range 8..16.

Auto-Bins

Auto-binning can be used in many cases to cause bins to be created for all values of an enumerated type, or to cause the legal value range to be partitioned evenly based on the `auto_bin_max` option. When auto-binning is used and the type of the coverpoint isn't apparent, the `cp_t` parameter must be used to specify the type of the value being sampled.

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

     def __init__(self, a : callable):

         self.cp1 = vsc.coverpoint(a, cp_t=vsc.uint8_t())
```

In the example above, the type of the coverpoint is not apparent because a callable is providing the target value. Consequently, the cp_t parameter is used to specify that the value being sampled is an 8-bit unsigned integer.

Wildcard Bins (Single)

A wildcard specification may be used to specify the values within single bins. The checked value may either be specified as a string that contains wildcard characters ('x', '?') or may be specified as a tuple of (value, mask).

When using the string form of specifying a wildcard bin, the specification string must start with "0x" (hexadecimal), "0o" (octal), or "0b" (binary).

Here is an example showing specification of a wildcard bin that matches any value 0x80..0x8F:

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

    def __init__(self):
        self.with_sample(
            dict(a=vsc.bit_t(8)))

        self.cp_a = vsc.coverpoint(self.a, bins=dict(
            a=vsc.wildcard_bin("0x8x")))
```

Here is the same coverpoint specification using the value/mask form:

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

    def __init__(self):
        self.with_sample(
            dict(a=vsc.bit_t(8)))

        self.cp_a = vsc.coverpoint(self.a, bins=dict(
            a=vsc.wildcard_bin((0x80,0xF0))
            ))
```

Wildcard Bins (Array)

A wildcard specification may also be used to specify arrays of bins. In this case, the wildcard characters specify a location where all possibilities must be expanded.

The example below creates 16 bins for the values 0x80..0x8F:

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

    def __init__(self):
        self.with_sample(
            dict(a=vsc.bit_t(8)))

        self.cp_a = vsc.coverpoint(self.a, bins=dict(
            a=vsc.wildcard_bin_array([], "0x8x")
            ))
```

Ignore and Illegal Bins

`Ignore` and `illegal` bins may be specified on coverpoints in addition to the other bins described above. An ignore or illegal bin trims values from other bins if it intersects values within those bins. Please note that, as in SystemVerilog, bins are partitioned *after* ignore and illegal bin values are removed from regular bins.

```
@vsc.covergroup
   class val_cg(object):
       def __init__(self):
           self.with_sample(dict(
               a=vsc.uint8_t()
               ))
           self.cp_val = vsc.coverpoint(self.a, bins=dict(
                                rng_1=vsc.bin_array([4], [1,3], [4,6], [7,9], [10,12])
                           ),
                           ignore_bins=dict(
                               invalid_value=vsc.bin(4)
                           ))
```

In the example above, the user specifies an array of four auto-partitioned bins and an ignored value of 4. In the absence of ignore bins, the 12 values to be paritionted would be divided into bins of three (1..3, 4..6, 7..9, 10..12). Because bins are partitioned after excluded bins have been applied, the bins in the example above are: - 1..2 - 3,5 - 6,7 - 8..12

5.2.2 Coverpoint Crosses

Coverpoint crosses are specified using the *cross* method. The first parameter to the *cross* method is a list of the coverpoints that compose the coverpoint cross.

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

    def __init__(self):
        self.with_sample(
            a=bit_t(4),
```

(continues on next page)

(continued from previous page)

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

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

5.2.3 Specifying Coverpoint Sampling Conditions

A sampling condition can be specified on both coverpoints and coverpoint crosses using the `iff` keyword parameter to the *coverpoint* and *cross* methods.

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

    def __init__(self, a : callable, b : callable):

        self.cp1 = vsc.coverpoint(a, iff=b, bins={
            "a" : vsc.bin_array([], 1, 2, 4),
            "b" : vsc.bin_array([4], [8,16])
            })
```

5.2.4 Coverpoint Options

Both type options and instance options can specified on both coverpoints and coverpoint crosses. Only the following options are currently respected:

Option name	Default	Description
weight=number	1	Specifies the weight of this covergroup instances relative to other instances.
goal=number	100	Specifies the target goal for this covergroup instance
at_least=number	1	Minimum number of hits for each coverage bin
auto_bin_max=number	64	Maximum number of automatically-created bins when bins are not explicitly specified

Options are specified via a `dict` attached to the coverpoint during construction. The example below shows overriding the covergroup-level `at_least` option for one coverpoint.

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

    def __init__(self):
        self.with_sample(dict(
            a=vsc.uint8_t(),
            b=vsc.uint8_t()))
```

(continues on next page)

(continued from previous page)

```
        self.options.at_least = 2

        self.cp1 = vsc.coverpoint(self.a, bins={
            "a" : vsc.bin_array([], 1, 2, 4, 8),
            }, options=dict(at_least=1))
        self.cp2 = vsc.coverpoint(self.b, bins={
            "b" : vsc.bin_array([], 1, 2, 4, 8)
            })
```

5.3 Providing Coverage Data to Sample

PyVSC supports several methods for providing data for a covergroup instance to sample. - Data in a *randobj*-decorated class object can be provided by reference to the covergroup `__init__` method. - Scalar data can be specified to the `__init__` method using lambda expressions to obtain the data from the instantiating context - Data can be provided via the *sample* methods, using a user-specified sample-method signature.

5.3.1 Declaring a Custom Sample Method

Use of a custom *sample* method that accepts parameters is specified by calling the `with_sample` method and passing either a `dict` of parameter-name/parameter-type pairs or a list of keyword arguments. The `with_sample` method declares class members with the same name and type as the key/value pairs in the dict passed to the `with_sample` method. The `with_sample` method should be called early in the `__init__` method body to ensure that the sample parameters are declared early and present when referenced in coverpoints.

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

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

The example above shows specifying the *sample* method parameter list using a `dict`.

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

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

The example above shows specifying the *sample* method parameter list using individual keyword arguments.

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

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

cg = my_covergroup()
cg.sample(1)
cg.sample(12)
```

In both cases, data is passed as parameters to the *sample* method, as shown in the example above.

5.3.2 Specifying Sampling Data at Instantiation

PyVSC supports specifying coverage-sampling data when the covergroup is instanced, as well as specifying it each time the sample method is called. In this case, no parameters are passed to the *sample* method.

This mode of specifying coverage-sampling data requires that a `lambda` is used to connect the calling context to the data used for coverage sampling.

```
@covergroup
 class my_covergroup(object):

     def __init__(self, a, b): # Need to use lambda for non-reference values
         super().__init__()

         self.cp1 = coverpoint(a,
             bins=dict(
                 a = bin_array([], [1,15])
             ))

         self.cp2 = coverpoint(b, bins=dict(
             b = bin_array([], [1,15])
             ))


 a = 0;
 b = 0;

 cg = my_covergroup(lambda:a, lambda:b)

 a=1
 b=1
 cg.sample() # Hit the first bin of cp1 and cp2
```

In the example above, calling the *sample* method will sample the current value of a and b in the context and sample the coverpoints with those values.

5.4 Coverage API

PyVSC covergroup classes implement methods for querying achieved coverage.

- *get_coverage* - Reports coverage achieved by all covergroup instances (0..100)
- *get_inst_coverage* - Reports coverage by this instance (0..100)

```
@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_array([], [1, 2, 4, 8])
            })

cg1 = my_covergroup()
cg2 = my_covergroup()

cg1.sample(1)
print("Type=%f cg1=%f cg2=%f" % (
  cg1.get_coverage(),
  cg1.get_inst_coverage(),
  cg2.get_inst_coverage()))

cg2.sample(2)
print("Type=%f cg1=%f cg2=%f" % (
  cg1.get_coverage(),
  cg1.get_inst_coverage(),
  cg2.get_inst_coverage()))
```

Running this example produces:

```
Type=25.000000 cg1=25.000000 cg2=0.000000
Type=50.000000 cg1=25.000000 cg2=25.000000
```

Sampling the first covergroup instance results in its instance coverage being increased to 25% (1/4 bins have been hit) and the combined type coverage incrasing to 25%. Sampling the second covergroup instance raises its instance coverage to 25% as well, while increasing the total type coverage achieved to 50%.

5.5 Coverage Reports

PyVSC provides three methods for obtaining a coverage report.

vsc.**get_coverage_report**(*details=False*) → str

Returns a textual coverage report of all covergroups

Parameters
details (*bool*) – Write details, such as the hits in each bin (False)

Returns
String containin coverage report text

vsc.**get_coverage_report_model**() → CoverageReport

Returns a coverage report model of all covergroups

Returns
Object describing collected coverage

vsc.**report_coverage**(*fp=None*, *details=False*)

Writes a coverage report to a stream (stdout by default)

Parameters

- **fp** – Stream to which to write the report
- **details** (*bool*) – Write details, such as the hits in each bin (False)

Let's using a derivative of the example show above to see the differences between a coverage report with and without details.

```
@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_array([], 1, 2, 4, 8)
            })

cg1 = my_covergroup()
cg2 = my_covergroup()

cg1.sample(1)
cg2.sample(2)

print("==== Without Details ===")
vsc.report_coverage()
print()
print("==== With Details ===")
vsc.report_coverage(details=True)
```

The output from this code is shown below:

```
TYPE my_covergroup : 50.000000%
  CVP cp1 : 50.000000%
  INST my_covergroup : 25.000000%
      CVP cp1 : 25.000000%
  INST my_covergroup_1 : 25.000000%
      CVP cp1 : 25.000000%

==== With Details ===
TYPE my_covergroup : 50.000000%
  CVP cp1 : 50.000000%
  Bins:
      a[0] : 1
      a[1] : 1
      a[2] : 0
      a[3] : 0
  INST my_covergroup : 25.000000%
      CVP cp1 : 25.000000%
      Bins:
          a[0] : 1
          a[1] : 0
          a[2] : 0
          a[3] : 0
  INST my_covergroup_1 : 25.000000%
      CVP cp1 : 25.000000%
      Bins:
          a[0] : 0
          a[1] : 1
          a[2] : 0
          a[3] : 0
```

The coverage report without details shows the coverage achieved for the covergroup and coverpoints without showing which bins were hit or how many times. The coverage report with details shows hit counts for each bin in addition to the coverage percentage achieved for the covergroups and coverpoints.

5.6 Saving Coverage Data

PyVSC uses the PyUCIS library to export coverage data using the API or XML interchange format defined by the Accellera UCIS standard.

Using the PyUCIS library, PyVSC can write coverage data to an XML-format coverage interchange file. Or, can write coverage data directly to a coverage database using a shared library that implements the UCIS C API.

PyVSC provides the *write_coverage_db* method for saving coverage data.

vsc.**write_coverage_db**(*filename*, *fmt='xml'*, *libucis=None*)

Writes coverage data to persistent storage using the PyUCIS library.

Parameters

- **filename** (*str*) – Destination for coverage data
- **fmt** (*str*) – Format of the coverage data. 'xml' and 'libucis' supported
- **libucis** (*str*) – Path to a library implementing the UCIS C API (default=None)

5.6.1 Saving to XML

By default, the *write_coverage_db* method saves coverage data to an XML file formatted according to the UCIS interchange-format schema.

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

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

my_cg_1 = my_covergroup()
my_cg_1.sample(1)
my_cg_1.sample(2)
my_cg_1.sample(8)

vsc.write_coverage_db('cov.xml')
```

5.6.2 Saving via a UCIS API Implementation

When an implementation of the UCIS C API is available, PyVSC can write coverage data using that API implementation. In this case, the `fmt` parameter of the *write_coverage_db* method must be specified as `libucis`. The `libucis` parameter of the method must specify the name of the shared library that implements the UCIS API.

In the example below, the tool-provided shared library that implements the UCIS API is named `libucis.so`.

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

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

my_cg_1 = my_covergroup()
my_cg_1.sample(1)
my_cg_1.sample(2)
my_cg_1.sample(8)

vsc.write_coverage_db('cov.db', fmt='libucis', libucis='libucis.so')
```

Calling *write_coverage_db* in this way causes the PyUCIS library to load the specified shared library and call UCIS C API functions to record the coverage data collected by the PyVSC library.

5.7 Using Coverage Data

Coverage data saved from PyVSC can be used in several open-source and closed-source commercial tool flows. The sections below describe flows that PyVSC data is known to have been used in.

Note: The information below with respect to closed-source/commercial tool flows represents data collected from users of those flows and tools. You are well-advised to confirm the accuracy of the information with the relevant vendor's documentation and/or Application Engineers.

Please report other tool flows that accept coverage data from PyVSC via the project's Issues or Discussion areas.

5.7.1 Viewing Coverage with PyUCIS-Viewer

PyUCIS-Viewer is a very simple graphical viewer for functional coverage data. It currently supports reading coverage data from UCIS XML-interchange-formatted files.

5.7.2 Siemens Questa: Writing Coverage Data

Siemens Questa[1] is reported to provide a library that implements the UCIS C API. Using this library, coverage data can be written directly to a Questa coverage database. See the information above about writing coverage data to a UCIS API implementation for more information on how to utilize this flow.

5.7.3 Synopsys VCS: Importing Coverage Data

Bringing coverage in UCIS XML-interchange format into the Synopsys VCS[2] metric analysis flow has been described using an import command. To follow this flow, write coverage data out from PyVSC in UCIS XML-interchange format.

Use the following VCS import command to read the data from the XML coverage file into a VCS coverage database:

```
% covimport -readucis <cov.xml> -dbname <cov.vdb>
```

[1] Questa is a trademark of Siemens Industry Software Inc.
[2] VCS is a trademark of Synopsys Inc.

CHAPTER

SIX

PYVSC METHODS

6.1 Randomization Methods

In addition to the `randomize` and `randomize_with` methods provided by the randobj class, PyVSC provides global methods for randomizing variables.

```
a = vsc.rand_uint8_t()
b = vsc.rand_uint8_t()

vsc.randomize(a, b)
```

The global randomize method randomizes the list of PyVSC variables, both scalar and composite.

```
a = vsc.rand_uint8_t()
b = vsc.rand_uint8_t()

for i in range(10):
    with vsc.randomize_with(a, b):
        a < b
```

The global randomize_with method randomizes the list of PyVSC variables, both scalar and composite, subject to the inline constraints.

6.2 Managing Random Stability

Each PyVSC *rand_obj* instance maintains its own random state. The random state for each *rand_obj* can be established explicitly by the user. If the random state has not been initialized at the time of the first call to `randomize` on a object, the random state will be automatically derived using the Python `random` package.

PyVSC uses the `RandState` class to store and manipulate random state. The *rand_obj* class provides functions for obtaining a copy of the current random state, and for setting the current random state to that of a previously-obtained random state.

The example below shows using a RandState object to produce the same sequence of random number twice.

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

    def __init__(self):
        self.a = vsc.rand_uint8_t()
```

(continues on next page)

(continued from previous page)

```
        self.b = vsc.rand_uint8_t()

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

ci = item_c()

v1 = []
v2 = []

print("Iteration 1")
rs1 = RandState.mkFromSeed(0)
ci.set_randstate(rs1)
for _ in range(10):
    ci.randomize()
    v1.append((ci.a,ci.b))

print("Iteration 2")
ci.set_randstate(rs1)
for _ in range(10):
    ci.randomize()
    v2.append((ci.a,ci.b))
```

The RandSeed.mkFromSeed method is the preferred way to create a random state object from user-specified values. The mkFromSeed method accepts an integer seed and an optional string. The example below shows producing the same sequence of random values based on two independently-created random state objects.

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

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

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

ci = item_c()

v1 = []
v2 = []

print("Iteration 1")
rs1 = RandState.mkFromSeed(10, "abc")
ci.set_randstate(rs1)
for _ in range(10):
    ci.randomize()
    v1.append((ci.a,ci.b))

print("Iteration 2")
```

(continues on next page)

(continued from previous page)

```
rs2 = RandState.mkFromSeed(10, "abc")
ci.set_randstate(rs2)
for _ in range(10):
    ci.randomize()
    v2.append((ci.a,ci.b))
```

6.3 Weighted-Random Selection Methods

It is often useful to perform a weighted-random selection in procedural code. SystemVerilog provides the `randcase` construct for this purpose. PyVSC provides two methods for performing a weighted-random selection from a set of candidates.

6.3.1 distselect

The `distselect` method accepts a list of weights and returns the selected index.

```
hist = [0]*4

for i in range(100):
    idx = vsc.distselect([1, 1, 10, 10])
    hist[idx] += 1

print("hist: " + str(hist))
```

In the example above, the index returned will vary 0..3.

6.3.2 randselect

The `randselect` method accepts a list of weight/lambda tuples. It performs a weighted selection and calls the selected lambda. In most cases, the lambda will need to call a function to perform useful work.

```
hist = [0]*4

def task(idx):
    hist[idx] += 1

for i in range(100):
    vsc.randselect([
            (1, lambda: task(0)),
            (1, lambda: task(1)),
            (10, lambda: task(2)),
            (10, lambda: task(3))])
print("hist: " + str(hist))
```

In the example above, the lambda functions invoke the same Python method with different arguments. Different methods could be called instead.

CHAPTER

SEVEN

PYVSC FEATURES

Constraint Features

Feature	PyVSC	SystemVerilog	PSS	Description
<	Y	Y	Y	
>	Y	Y	Y	
<=	Y	Y	Y	
>=	Y	Y	Y	
==	Y	Y	Y	
!=	Y	Y	Y	
+	Y	Y	Y	
-	Y	Y	Y	
/	Y	Y	Y	
*	Y	Y	Y	
%	Y	Y	Y	
&	Y	Y	Y	
|	Y	Y	Y	
^	Y	Y	Y	
&&	Y (&)	Y	Y	
||	Y (|)	Y	Y	
<<	Y	Y	Y	
>>	Y	Y	Y	
~ (unary)	Y	Y	Y	
unary |	N	Y	N	
unary &	N	Y	N	
unary ^	N	Y	N	
scalar signed field	Y	Y	Y	
scalar unsigned field	Y	Y	Y	
scalar enum field	Y	Y	Y	
scalar fixed-size array	Y	Y	Y	
scalar dynamic array	Y	Y	N	
class fixed-size array	Y	Y	Y	
class dynamic array	Y	Y	N	
class (in)equality	N	N	Y	
array sum	N	Y	Y	
array size	N	Y	Y	
array reduction OR	N	Y	N	
array reduction AND	N	Y	N	
array reduction XOR	N	Y	N	

continues on next page

Table 1 – continued from previous page

part select `[bit]`	Y	Y	Y	
part select `[msb:lsb]`	Y	Y	Y	
default	N	N	Y	
dist	Y	Y	N	
dynamic	Y	N	Y	
inside (in)	Y	Y	Y	
soft	Y	Y	N	
solve before	N	Y	N	
unique	Y	Y	Y	
foreach	Y	Y	Y	
forall	N	Y	Y	
pre_randomize	Y	Y	Y	
post_randomize	Y	Y	Y	
constraint override	Y	Y	Y	
constraint_mode	Y	Y	N	

Coverage Features

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

CHAPTER

EIGHT

DEBUG

There are several situations in which you may need to enable or configure debug with PyVSC. The most common is when a set of constraints fails to solve, and diagnostics must be enabled to help understand the reason for the failure. PyVSC targets execution speed over verbosity, so default behavior is to create no diagnostics when a solve failure occurs.

8.1 Enabling Solve-Fail Debug

PyVSC provides an optional argument to the `randomize` and `randomize_with` method to enable solve-fail debug on a per-call basis.

```
class my_e(IntEnum):
    A = auto()
    B = auto()
    C = auto()

@vsc.randobj
class my_c(object):

    def __init__(self):
        self.e = vsc.rand_enum_t(my_e)
        self.a = vsc.rand_uint8_t()

    @vsc.constraint
    def a_c(self):
        self.a == 1
        with vsc.if_then(self.a == 2):
            self.e == my_e.A

it = my_c()

with it.randomize_with(solve_fail_debug=1):
    it.a == 2
```

In the example above, the class-level constraint set forces a==1, while the user's inline constraints forces a==2. The `randomize_with` call sets solve_fail_debug=1, which triggers creation of diagnostic information when a solve failure occurs.

In this case, the output is of the following form:

```
Problem Set: 2 constraints
  <unknown>:
    (a == 1);
  <unknown>:
    (a == 2);
```

Enabling solve-fail debug can also be enabled globally by calling `vsc.vsc_solvefail_debug(1)` from Python code. The environment variable VSC_SOLVEFAIL_DEBUG can also be set to 1.

8.2 Capturing Source Information

Note that no source information is available for the constraints. This is because querying source information in Python is quite time-consuming.

Enabling the capture of source information for constraints can be done in two ways: via the *randobj* decorator, and via an environment variable.

```
class my_e(IntEnum):
    A = auto()
    B = auto()
    C = auto()

@vsc.randobj(srcinfo=True)
class my_c(object):

    def __init__(self):
        self.e = vsc.rand_enum_t(my_e)
        self.a = vsc.rand_uint8_t()

    @vsc.constraint
    def a_c(self):
        self.a == 1
        with vsc.if_then(self.a == 2):
            self.e == my_e.A

it = my_c()

with it.randomize_with(solve_fail_debug=1):
    it.a == 2
```

In the code-block above, the `srcinfo` parameter to the *randobj* decorator causes source information to be collected for constraints in the class. The solve-fail diagnostics will now be of the following form:

```
Problem Set: 2 constraints
  /project/fun/pyvsc/pyvsc-partsel-rand/ve/unit/test_solve_failure.py:30:
    (a == 1);
  /project/fun/pyvsc/pyvsc-partsel-rand/ve/unit/test_solve_failure.py:38:
    (a == 2);
```

Source-information capture may also be enabled globally via an environment variable. Set VSC_CAPTURE_SRCINFO=1 to cause all source information for all random classes to be captured.

CHAPTER

NINE

API REFERENCE

9.1 Domain-Specific Language API

9.1.1 Data and Constraints

vsc.attrs.**attr**(*t*)

Wraps a recognized datatype as a non-rand field

vsc.attrs.**rand_attr**(*t*)

Wraps a VSC datatype, or recognized datatype, as a rand field

class vsc.constraints.**constraint_t**(*c*)

constraint_mode(*en*)

set_model(*m*)

elab()

class vsc.constraints.**dynamic_constraint_t**(*c*)

set_model(*m*)

class call_closure(*c*, **args*, ***kwargs*)

vsc.constraints.**dynamic_constraint**(*c*)

vsc.constraints.**constraint**(*c*)

class vsc.constraints.**weight**(*val*, *w*)

vsc.constraints.**dist**(*lhs*, *weights*)

Applies distribution weights to the specified field

class vsc.constraints.**if_then**(*e*)

class vsc.constraints.**else_if**(*e*)

class vsc.constraints.**else_then_c**

class vsc.constraints.**implies**(*e*)

vsc.constraints.**soft**(*e*)

vsc.constraints.**unique**(**args*)

class vsc.constraints.**forall**(*target_type*)

class vsc.constraints.**foreach**(*l*, *it=None*, *idx=None*)

class idx_term_c(*index*)

to_expr()

class it_term_c(*em*)

to_expr()

vsc.constraints.**solve_order**(*before*, *after*)

vsc.rand_obj.**randobj**(**args*, ***kwargs*)

vsc.rand_obj.**generator**(*T*)

Mark a class as a generator

vsc.types.**unsigned**(*v*, *w=-1*)

vsc.types.**signed**(*v*, *w=-1*)

class vsc.types.**expr**(*em*)

bin_expr(*op*, *rhs*)

inside(*rhs*)

outside(*rhs*)

not_inside(*rhs*)

class vsc.types.**dynamic_constraint_proxy**(*em*)

class vsc.types.**expr_subscript**(*em*)

class vsc.types.**rng**(*low*, *high*)

class vsc.types.**rangelist**(**args*)

clear()

extend(*ranges*)

append(*a*)

vsc.types.**to_expr**(*t*)

class vsc.types.**field_info**(*is_composite=False*)

Model-specific information about the field

set_is_rand(*is_rand*)

class vsc.types.**type_base**(*width*, *is_signed*, *i=0*)

Base type for all primitive-type fields that participate in constraints

get_model()

build_field_model(*name*)

to_expr()

property rand_mode

property val

get_val()

set_val(*val*)

bin_expr(*op*, *rhs*)

inside(*rhs*)

outside(*rhs*)

not_inside(*rhs*)

clone()

class vsc.types.**type_bool**(*i=False*)

Base class for boolean fields

build_field_model(*name*)

get_val()

Gets the field value

set_val(*val*)

Sets the field value

class vsc.types.**type_enum**(*t*, *i=None*)

Base class for enumerated-type fields

build_field_model(*name*)

get_val()

Returns the enum id

set_val(*val*)

Sets the enum id

class vsc.types.**enum_t**(*t*, *i=None*)

Creates a non-random enumerated-type attribute

class vsc.types.**rand_enum_t**(*t*, *i=None*)

Creates a random enumerated-type attribute

class vsc.types.**bit_t**(*w=1*, *i=0*)

Creates an unsigned arbitrary-width attribute

class vsc.types.**bool_t**(*i=False*)

Creates a boolean field

class vsc.types.**uint8_t**(*i=0*)

Creates an unsigned 8-bit attribute

class vsc.types.**uint16_t**(*i=0*)

Creates an unsigned 16-bit attribute

class vsc.types.**uint32_t**(*i=0*)

Creates an unsigned 32-bit attribute

class vsc.types.**uint64_t**(*i=0*)

Creates an unsigned 64-bit attribute

class vsc.types.**rand_bit_t**(*w=1*, *i=0*)

Creates a random unsigned arbitrary-width attribute

class vsc.types.**rand_uint8_t**(*i=0*)

Creates a random unsigned 8-bit attribute

class vsc.types.**rand_uint16_t**(*i=0*)

Creates a random unsigned 16-bit attribute

class vsc.types.**rand_uint32_t**(*i=0*)

Creates a random unsigned 32-bit attribute

class vsc.types.**rand_uint64_t**(*i=0*)

Creates a random unsigned 64-bit attribute

class vsc.types.**int_t**(*w=32*, *i=0*)

Creates a signed arbitrary-width attribute

class vsc.types.**int8_t**(*i=0*)

Creates a signed 8-bit attribute

class vsc.types.**int16_t**(*i=0*)

Creates a signed 16-bit attribute

class vsc.types.**int32_t**(*i=0*)

Creates a signed 32-bit attribute

class vsc.types.**int64_t**(*i=0*)

Creates a signed 64-bit attribute

class vsc.types.**rand_int_t**(*w=32*, *i=0*)

Creates a random signed arbitrary-width attribute

class vsc.types.**rand_int8_t**(*i=0*)

Creates a random signed 8-bit attribute

class vsc.types.**rand_int16_t**(*i=0*)

Creates a random signed 16-bit attribute

class vsc.types.**rand_int32_t**(*i=0*)

Creates a random signed 32-bit attribute

class vsc.types.**rand_int64_t**(*i=0*)

Creates a random signed 64-bit attribute

class vsc.types.**list_t**(*t*, *sz=0*, *is_rand=False*, *is_randsz=False*, *init=None*)

get_model()

build_field_model(*name*)

property size

property sum

property product

append(*v*)

extend(*v*)

clear()

to_expr()

class vsc.types.**rand_list_t**(*t*, *sz=0*)

List of random elements with a non-random size

class vsc.types.**randsz_list_t**(*t*)

List of random elements with a random size

9.1.2 Data and Constraints

vsc.coverage.**covergroup**(*T*)

Covergroup decorator marks as class as being a covergroup

class vsc.coverage.**bin**(**args*)

Specifies a single coverage bin

build_cov_model(*parent*, *name*, *exclude_bins:* RangelistModel)

class vsc.coverage.**bin_array**(*nbins*, **args*)

Specifies an array of bins

args may be one of two formats - Single list of tuples or lists of values or ranges (eg [[1], [2,4], [8]]) - Arguments comprising values and ranges (eg 1, [2,4], 8)

build_cov_model(*parent*, *name*, *exclude_bins:* RangelistModel)

class vsc.coverage.**wildcard_bin**(**args*)

Specifies a single wildcard coverage bin

build_cov_model(*parent*, *name*, *excluded_bins*)

class vsc.coverage.**wildcard_bin_array**(*nbins*, **args*)

Specifies an array of bins using wildcard specifications

args may be one of two formats - Single list of wildcard strings (eg "0x82x") - Single list of value/mask tuples (eg [(0x820,0xFF0)]) - Single string - Single tuple

build_cov_model(*parent*, *name*, *excluded_bins*)

class vsc.coverage.**binsof**(*cp*)

intersect(*rng*)

class vsc.coverage.**coverpoint**(*target*, *cp_t=None*, *iff=None*, *bins=None*, *ignore_bins=None*, *illegal_bins=None*, *options=None*, *type_options=None*, *name=None*)

get_coverage()

get_inst_coverage()

build_cov_model(*parent*, *name*)

get_model()

get_val(*cp_m*)

set_val(*val*)

class vsc.coverage.**cross**(*target_l*, *bins=None*, *options=None*, *name=None*, *iff=None*)

build_cov_model(*parent*, *name*)

get_coverage()

9.2 Model API

9.2.1 Data and constraints

class vsc.model.field_composite_model.**FieldCompositeModel**(*name*, *is_rand=False*, *rand_if=None*)

finalize()

property is_declared_rand

set_used_rand(*is_rand*, *level=0*)

build(*builder*)

add_field(*f*) → FieldModel

get_field(*idx*)

set_field(*idx*, *f*)

find_field(*name*)

add_constraint(*c*)

get_constraint(*name*)

add_dynamic_constraint(*c*)

get_constraints(*constraint_l*)

get_fields(*field_l*)

pre_randomize()

Called during the randomization process to propagate *pre_randomize* event

post_randomize()

Called during the randomization process to propagate *post_randomize* event

accept(*v*)

dispose()

class vsc.model.constraint_block_model.**ConstraintBlockModel**(*name*, *constraints=None*)

Information about a top-level constraint block described by the user

set_constraint_enabled(*en*)

accept(*v*)

clone(*deep=False*) → *ConstraintModel*

class vsc.model.constraint_expr_model.**ConstraintExprModel**(*e*)

build(*btor*, *soft=False*)

accept(*visitor*)

clone(*deep=False*) → *ConstraintModel*

class vsc.model.constraint_if_else_model.**ConstraintIfElseModel**(*cond: ExprModel*, *true_c: Optional[ConstraintScopeModel] = None*, *false_c: Optional[ConstraintScopeModel] = None*)

build(*btor*, *soft=False*)

accept(*visitor*)

clone(*deep=False*) → *ConstraintModel*

class vsc.model.constraint_implies_model.**ConstraintImpliesModel**(*cond*, *constraints=None*)

build(*btor*, *soft=False*)

get_nodes(*node_l*)

accept(*visitor*)

clone(*deep=False*) → *ConstraintModel*

class vsc.model.constraint_model.**ConstraintModel**

Base class for all constraint models

dispose()

build(*btor*, *soft=False*) → BoolectorNode

get_nodes(*node_l*)

static and_nodelist(*node_l*, *btor*)

Creates a boolean AND across a list of expression nodes

accept(*visitor*)

clone(*deep=False*) → *ConstraintModel*

class vsc.model.constraint_scope_model.**ConstraintScopeModel**(*constraints=None*)

addConstraint(*c*) → *ConstraintModel*

build(*btor*, *soft=False*)

get_nodes(*node_l*)

accept(*visitor*)

clone(*deep=False*) → *ConstraintModel*

class vsc.model.constraint_unique_model.**ConstraintUniqueModel**(*unique_l*)

build(*btor*, *soft=False*)

get_nodes(*node_l*)

accept(*visitor*)

clone(*deep=False*) → *ConstraintModel*

9.2.2 Coverage

class vsc.model.covergroup_model.**CovergroupModel**(*name: str*, *options=None*)

finalize()

sample()

add_coverpoint(*cp*)

get_coverage()

coverage_ev(*cp*, *bin_idx*)

get_inst_coverage()

accept(*v*)

dump(*ind=''*)

equals(*oth:* CovergroupModel) → bool

clone() → *CovergroupModel*

class vsc.model.covergroup_registry.**CovergroupRegistry**

accept(*v*)

static inst()

class vsc.model.coverpoint_bin_array_model.**CoverpointBinArrayModel**(*name*, *low*, *high*)

finalize(*bin_idx_base: int*) → int

Accepts the bin index where this bin starts ; returns number of bins

get_bin_expr(*idx*)
Builds expressions to represent a single bin

get_bin_name(*bin_idx*)

sample()

dump(*ind=''*)

get_bin_range(*bin_idx*)

get_n_bins()

hit_idx()

accept(*v*)

equals(*oth*)

clone() → *CoverpointBinArrayModel*

class vsc.model.coverpoint_bin_collection_model.**CoverpointBinCollectionModel**(*name*)

DEBUG_EN = False

finalize(*bin_idx_base: int*) → int
Accepts the bin index where this bin starts ; returns number of bins

get_bin_expr(*idx*)
Builds expressions to represent the values in this bin

get_bin_name(*idx*)

add_bin(*bin_m*) → *CoverpointBinModelBase*

get_coverage()

sample()

get_bin_range(*idx*)

dump(*ind=''*)

get_hits(*idx*)

get_n_bins()

hit_idx()

set_bin_type(*bin_type*)

accept(*v*)

equals(*oth*) → bool

clone() → *CoverpointBinCollectionModel*

static mk_collection(*name: str*, *rangelist:* RangelistModel, *n_bins*) → *CoverpointBinCollectionModel*
Creates a bin collection by partitioning a rangelist

class vsc.model.coverpoint_bin_enum_model.**CoverpointBinEnumModel**(*name*, *val*)

finalize(*bin_idx_base: int*) → int

Accepts the bin index where this bin starts ; returns number of bins

get_bin_name(*bin_idx*)

sample()

accept(*v*)

equals(*oth*) → bool

clone() → *CoverpointBinEnumModel*

class vsc.model.coverpoint_bin_model_base.**CoverpointBinModelBase**(*name*)

finalize(*bin_idx_base: int*) → int

Accepts the bin index where this bin starts ; returns number of bins

get_bin_expr(*idx*)

get_bin_name(*bin_idx*)

sample()

get_n_bins()

hit_idx()

set_bin_type(*bin_type*)

equals(*oth*)

class vsc.model.coverpoint_bin_single_bag_model.**CoverpointBinSingleBagModel**(*name*, *binspec: Optional[*RangelistModel*] = None*)

Coverpoint single bin that is triggered on a set of values or value ranges

finalize(*bin_idx_base: int*) → int

Accepts the bin index where this bin starts ; returns number of bins

get_bin_expr(*bin_idx*)

Builds expressions to represent the values in this bin

get_bin_name(*bin_idx*)

sample()

dump(*ind=''*)

accept(*v*)

equals(*oth*) → bool

clone() → *CoverpointBinSingleBagModel*

class vsc.model.coverpoint_bin_single_range_model.**CoverpointBinSingleRangeModel**(*name*, *target_val_low: int*, *target_val_high: int*)

finalize(*bin_idx_base: int*) → int

Accepts the bin index where this bin starts ; returns number of bins

get_bin_expr(*bin_idx*)

Builds expressions to represent the values in this bin

get_bin_name(*bin_idx*)

sample()

accept(*v*)

equals(*oth*) → bool

clone() → *CoverpointBinSingleRangeModel*

class vsc.model.coverpoint_bin_single_val_model.**CoverpointBinSingleValModel**(*name*, *target_val: int*)

finalize(*bin_idx_base: int*) → int

Accepts the bin index where this bin starts ; returns number of bins

get_bin_expr(*bin_idx*)

Builds expressions to represent the values in this bin

get_bin_name(*bin_idx*)

sample()

get_bin_range(*idx*)

accept(*v*)

equals(*oth*) → bool

clone() → *CoverpointBinSingleValModel*

class vsc.model.coverpoint_cross_model.**CoverpointCrossModel**(*name*, *options*, *iff=None*)

set_target_value_cache(*iff*)

reset()

coverpoints()

get_coverage()

get_n_bins()

get_bin_expr(*bin_idx: int*) → ExprModel

select_unhit_bin(*r: RandIF*) → int

get_bin_hits(*bin_idx*)

get_bin_name(*bin_idx*) → str

add_coverpoint(*cp_m*)

finalize()

accept(*v*)

sample()

dump(*ind=''*)

equals(*oth:* CoverpointCrossModel) → bool

clone(*coverpoint_m*) → *CoverpointCrossModel*

class vsc.model.coverpoint_model.**CoverpointModel**(*target: ExprModel*, *name: str*, *options*, *iff: Optional[ExprModel] = None*)

set_target_value_cache(*val*, *iff=True*)

reset()

add_bin_model(*bin_m*)

add_ignore_bin_model(*bin_m*)

add_illegal_bin_model(*bin_m*)

finalize()

get_bin_expr(*bin_idx*)

get_coverage() → float

get_inst_coverage()

sample()

select_unhit_bin(*r: RandIF*) → int

get_bin_range(*bin_idx*) → *RangelistModel*

coverage_ev(*bin_idx*, *bin_type*)

Called by a bin to signal that an uncovered bin has been covered

get_val()

accept(*v*)

dump(*ind=''*)

get_n_bins()

get_n_ignore_bins()

get_n_illegal_bins()

get_n_hit_bins()

get_bin_hits(*bin_idx*)

get_ignore_bin_hits(*bin_idx*)

get_illegal_bin_hits(*bin_idx*)

get_bin_name(*bin_idx*) → str

get_ignore_bin_name(*bin_idx*) → str

get_illegal_bin_name(*bin_idx*) → str

get_hit_bins(*bin_l*)

equals(*oth:* CoverpointModel) → bool

clone() → *CoverpointModel*

9.2.3 Expressions

class vsc.model.rangelist_model.**RangelistModel**(*rl: Optional[List[List[int]]] = None*)

add_value(*v*)

add_range(*low*, *high*)

compact()

intersect(*other*)

Intersects another list or ranges with this one, trimming values that overlap

equals(*oth*) → bool

toString()

clone()

accept(*v*)

CHAPTER

TEN

INDICES AND TABLES

PYTHON MODULE INDEX

V

INDEX

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W