Functional Coverage Closure with Python

Seokho Lee, Youngsik Kim, Suhyung Kim, Jeong Ki Lee, Wooyoung Choe, Minho Kim
FuriosaAI, Seoul, Korea
seokholee@furiosa.ai, kaengsik@furiosa.ai, soohyk@furiosa.ai, jklee@furiosa.ai,
wooyoung.choe@furiosa.ai, minho@furiosa.ai

***Abstract*- Over the past few years, there has been a growing trend of using Python for design verification instead of traditional hardware verification languages such as SystemVerilog. However, existing Python verification frameworks focus on driving and monitoring signals or solving random constraints, but lack coverage features which make it hard to achieve functional coverage closure. This paper proposes a Python environment for enabling more efficient functional coverage closure. This environment fully utilizes rich features of SystemVerilog functional coverage as well as leverages existing tools for easier coverage analysis.**

I. Introduction

Python is easy to learn and has extended its reach into the field of hardware verification [1]. Since Python does not have native support for verification features like constrained randomization or functional coverage, open-source libraries are being developed to make these functionalities available. However, there are challenges in a real-world verification project using Python testbenches, especially for functional coverage. Coverage features supported by Python libraries are less powerful than those in SystemVerilog, functional coverage databases are not compatible with code coverage databases from logic simulators, and coverage analysis tools are either absent or only have simple functions compared to those provided by electronic design automation (EDA) vendors.

To overcome the limitations addressed above, this paper presents a new approach to closing functional coverage with Python. Using this approach, verification engineers proficient in Python but not familiar with SystemVerilog can achieve functional coverage closure with much less effort by utilizing user-friendly graphical user interface (GUI) tools provided by EDA vendors for coverage closure.

II. Background & Previous Works

Most Python-based testbenches use Cocotb as their verification framework. Cocotb is an open source coroutine based cosimulation testbench environment that allows you to drive input signals of the design under test (DUT) and monitor output signals of the DUT in a testbench described in Python [2]. However, Cocotb does not provide built-in support for class randomization and functional coverage. To tackle this, open-source libraries like cocotb-coverage and PyVSC are developed and being used [3][4].

Both cocotb-coverage and PyVSC mimic coverage concepts from SystemVerilog such as covergroup, coverpoint, and cross coverage. While these libraries claim they are equipped with coverage features, they cannot fully support all the features SystemVerilog offers. For example, cocotb-coverage does not have array bins and auto bins syntax, and PyVSC does not have transition bins and ignore bins syntax.

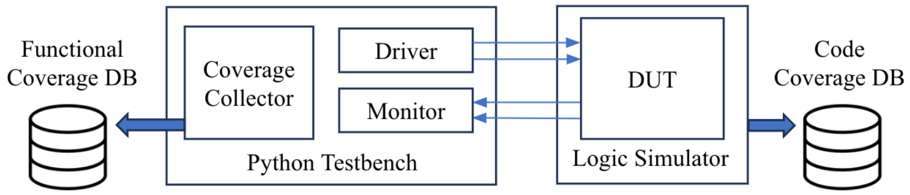


Figure 1. Python testbench with functional coverage

Figure 1 represents a typical structure of a Python testbench. Python is in charge of running testbench codes and the logic simulator is only responsible for register transfer language (RTL) simulation of the DUT. Consequently, code and functional coverage collection data are separately generated and stored. The functional coverage database has an extensible markup language (XML) format and cannot be opened directly by coverage analysis tools from EDA vendors. Verification engineers should either maintain two databases for coverage closure which is hard to manage or convert functional coverage databases into compatible formats. Even if the conversion process is successful, utilizing convenient features of the coverage analysis tool, such as source browsing, is not possible.

The lack of coverage features and compatibility of databases addressed above can be a barrier for verification engineers who wish to sign-off coverage closure with Python. Thus, we propose a new Python library that enables functional coverage closure without compromising quality and efficiency.

III. Proposed Works

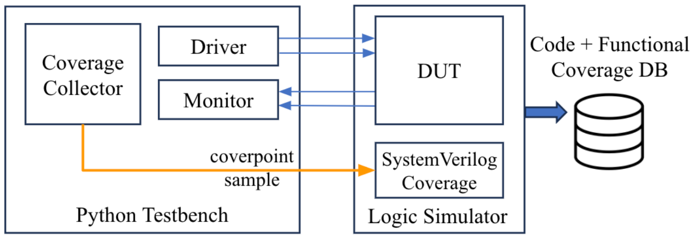


Figure 2. Python testbench with our proposed functional coverage library

Our proposed Python functional coverage library provides a more straightforward way to define functional coverage in Python, and a freedom to express it in a more detailed way than previous works. This is accomplished through a SystemVerilog layer on the logic simulator side. As you can see in Figure 2, the Python testbench still takes a role in defining and collecting coverage but offloads creating and managing functional coverage databases to the logic simulator. The SystemVerilog layer is automatically generated directly from the Python coverage implementation.

The proposed library is also designed for coverage closure which includes modeling, measuring, analyzing, and reviewing coverage. Figure 3 shows the entire process of coverage closure using the proposed library followed by a detailed description of each process.

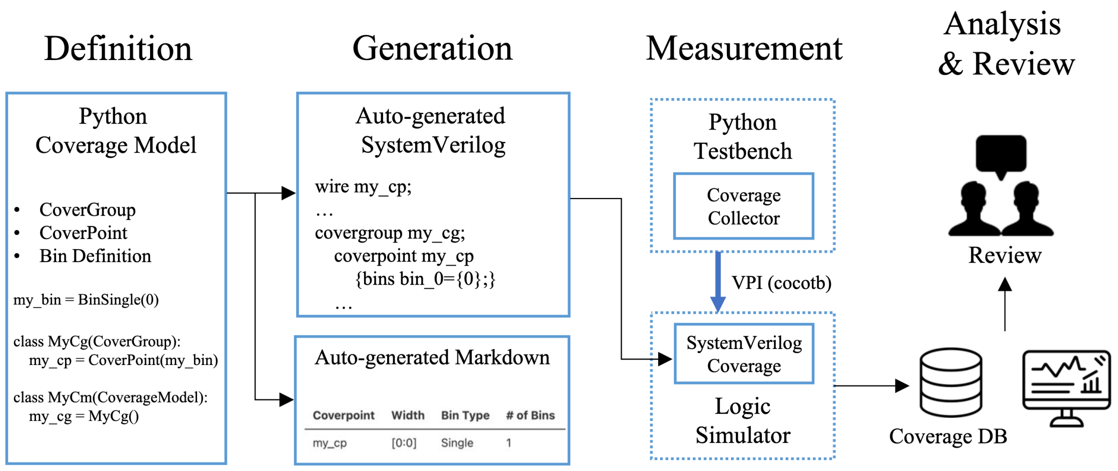


Figure 3. Overall functional coverage flow using the proposed library

A. *Define Functional Coverage in Python*

Our proposed library supports a hierarchical structure for describing coverpoints within a covergroup, similar to conventional coverage description methods. Each CoverGroup class includes one or more coverpoints and cross coverage of coverpoints. Each CoverPoint class includes bin descriptions. There are two ways to describe bin types: (1) using predefined bin types and (2) using user-defined iterators. Predefined bin types provide classes for bin constructs that are frequently used by verification engineers; some examples can be found in Table 1. Moreover, these predefined bin classes can be combined to make complicated bin expressions possible.

If it is tricky to make coverpoint bins using predefined bin types, this can be achieved by user-defined iterators. Iterator bin types can be as simple as listing through Python default data types (e.g. list, tuple), or as detailed and flexible as iterator types including generators. Figure 4 shows coverage bins for AXI strobe signals in case of narrow transfers. As in the example, Python generators offer efficient ways to replace lines of code that enumerate every single case.

Cross coverage can be defined as simple as passing a list of coverpoints as in Figure 5.

Predefined Coverage Type	Python Definition	Generated SystemVerilog Coverage	Bin Description in document	
			# of Bins	Bin Description

Single	`cp_zero =`
	`  CoverPoint(BinSingle(0))`
		`cp_zero: coverpoint signal {`
		`  bins bin_0 = {0};`
		`}`
			1	0

Bool	`cp_bool =`
	`  CoverPoint(BinBool())`
		`cp_bool: coverpoint signal {`
		`  bins TRUE = {1};`
		`  bins FALSE = {0};`
		`}`
			2	FALSE(0), TRUE(1)

Enum	`class MyEnum(IntEnum):`
	`  STATE_0 = 0b00`
	`  STATE_1 = 0b01`
	`  STATE_2 = 0b10`
	`  STATE_3 = 0b11`
	
	`cp_enum =`
	`  CoverPoint(BinEnum(MyEnum))`
		`cp_enum: coverpoint signal {`
		`  bins STATE_0 = {0};`
		`  bins STATE_1 = {1};`
		`  bins STATE_2 = {2};`
		`  bins STATE_3 = {3};`
		`}`
			4	STATE_0(0),
				STATE_1(1),
				STATE_2(2),
				STATE_3(3)

Range	`cp_range =`
	` CoverPoint(BinRange(64))`
		`cp_range: coverpoint signal {`
		`  bins bin_0_63[64] = {[0:63]};`
		`}`
			64	[0:63]/64

Uniform	`cp_uniform =`
	`  CoverPoint(BinUniform(`
	`  0, 128, num=5))`
		`cp_uniform: coverpoint signal {`
		`  bins bin_0_127[5] = {[0:127]};`
		`}`
			5	[0:127]/5

MinMax	`cp_minmax =`
	`  CoverPoint(BinMinMax(`
	`  min=0, max=127, num_bins=5))`
		`cp_minmax: coverpoint signal {`
		`  bins bin_0 = {0};`
		`  bins bin_1_126[3] = {[1:126]};`
		`  bins bin_127 = {127};`
		`}`
			5	0, [1:126]/3, 127

Exp	`cp_exp =`
	`  CoverPoint(BinExp(16))`
		`cp_exp: coverpoint signal {`
		`  bins bin_0 = {0};`
		`  bins bin_1_1 = {[1:1]};`
		`  bins bin_2_3 = {[2:3]};`
		`  bins bin_4_7 = {[4:7]};`
		`  bins bin_8_15 = {[8:15]};`
		`}`
			5	0, [1:1], [2:3], [4:7], [8:15]

Bitwise	`cp_bitwise =`
	`  CoverPoint(BinBitwise(4))`
		`cp_bitwise_0: coverpoint signal[0];`
		`cp_bitwise_1: coverpoint signal[1];`
		`cp_bitwise_2: coverpoint signal[2];`
		`cp_bitwise_3: coverpoint signal[3];`
			8	[0:1]/2 for bit 0~3

OneHot	`cp_onehot =`
	`  CoverPoint(BinOneHot(4))`
		`cp_onehot: coverpoint signal {`
		`  bins bin_0x1 = {'h1};`
		`  bins bin_0x2 = {'h2};`
		`  bins bin_0x4 = {'h4};`
		`  bins bin_0x8 = {'h8};`
		`}`
			4	0x1, 0x2, 0x4, 0x8

Transition	`cp_transition =`
	`  CoverPoint(BinTransition(`
	`  (2,5), (2,10), (3,8)`
	`  ))`
		`cp_onehot: coverpoint signal {`
		`  bins bin_2_5 = (2=>5);`
		`  bins bin_2_10 = (2=>10);`
		`  bins bin_3_8 = (3=>8);`
		`}`
			3	2=>5, 2=>10, 3=>8

Table 1. Pre-defined bin types and generated outputs

Python Definition

```
def narrow_strobe(self, bus_byte_width):
  for i in range(log2Ceil(bus_byte_width)):
    size = 1 << i
    for j in range(bus_byte_width // size):
      strobe = ((1 << size) - 1) << (j * size)
      yield (f"en{size}byte_{hex(strobe)}", strobe)

class AxiNarrowTransferCoverage(CoverGroup):
  cp_strobe = CoverPoint(narrow_strobe(8), format="x")

cg_narrow = AxiNarrowTransferCoverage()
```

Generated SystemVerilog Coverage

```
covergroup cg_narrow;
  strobe: coverpoint cg_narrow_cp_strobe {
    bins en1_0x1  = {'h1};
    bins en1_0x2  = {'h2};
    bins en1_0x4  = {'h4};
    bins en1_0x8  = {'h8};
    bins en1_0x10 = {'h10};
    bins en1_0x20 = {'h20};
    bins en1_0x40 = {'h40};
    bins en1_0x80 = {'h80};
    bins en2_0x3  = {'h3};
    bins en2_0xc  = {'hc};
    bins en2_0x30 = {'h30};
    bins en2_0xc0 = {'hc0};
    bins en4_0xf  = {'hf};
    bins en4_0xf0 = {'hf0};
}
```

Figure 4. Example of defining coverage bins using an iterable object

Python Definition

```
class MyCrossCoverage(CoverGroup):
  cp_1 = CoverPoint([0, 1])
  cp_2 = CoverPoint([2, 3])
  cp_3 = CoverPoint([4, 5])
  cr = Cross([cp_1, cp_2, cp_3])

cg_cross = MyCrossCoverage()
```

Generated SystemVerilog Coverage

```
covergroup cg_cross;
  cp_1: coverpoint cg_cross_cp_1 {
    bins bin_0 = {0};
    bins bin_1 = {1};
  }
  cp_2: coverpoint cg_cross_cp_2 {
    bins bin_2 = {2};
    bins bin_3 = {3};
  }
  cp_3: coverpoint cg_cross_cp_3 {
    bins bin_4 = {4};
    bins bin_5 = {5};
  }
  cr: cross cp_1, cp_2, cp_3;
```

Figure 5. Example of cross coverage

B. *Generate SystemVerilog Layer and Coverage Document*

Once the coverage modeling in Python is finished, our proposed library can generate a SystemVerilog layer and corresponding coverage documents in markdown format. The generation process is command-based and fully automated, so there is no human intervention. The generated coverage document consists of markdown tables and can be imported to the verification plan for a review. Each covergroup has two tables, one for coverpoints and the other for cross coverage as in Figure 11. One helpful feature in the coverage document is the column with the number of bins, to check if the defined coverage model has an appropriate number of bins.

C. *Bring-up Testbench and Collect Coverage*

The next step is writing coverage collection codes for each coverage model. To collect coverage, coverpoint signals, and sample signals in the SystemVerilog layer should be driven by the Python Testbench. The connection between the Python testbench and the SystemVerilog layer is established by calling a predefined connect method with a hierarchical path to the SystemVerilog layer as an input.

Figure 6 shows an example that measures coverage through a coverpoint handler and a sample handler. When the coverage collector wants to sample a value, it assigns the value to a coverpoint handler so that it deposits the value to the SystemVerilog layer, and then calls a sample handler to toggle a sample signal in the SystemVerilog layer.

As coverage databases are created by the logic simulator, it is possible to measure other coverage metrics such as code coverage at the same time as a unified coverage database.

```
class MyCoverageCollector:
  def __init__(self, dut):
    self.my_cov = MyCoverageModel()
    cov_inst = getattr(dut, self.my_cov.sv_instname)

    # connect python coverage objects
    self.my_cov.connect(cov_inst)

  def collect_coverage(self, coverpoint_value):
    # assign sampling values
    self.my_cov.my_cg.my_cp <= coverpoint_value

    # trigger sampling
    self.my_cov.my_cg.sample()
```

Figure 6. Code snippet of coverage collector

D. *Analyze and Review Functional Coverage*

The final step for coverage closure is to analyze measured coverage data. The EDA vendors have pretty good coverage analysis tools with GUIs which help view overall coverage status, detect coverage holes, exclude unreachable bins, etc. By leveraging those coverage analysis tools, it is possible to achieve coverage closure as efficiently as using SystemVerilog testbenches.

IV. Methodology Example

This chapter shows real use-cases with our proposed library. The first one is a simple matrix multiplier and the second one is an AXI interface. Complete source codes can be found in our GitHub repository: https://github.com/furiosa-ai/dvcon2024-functial-coverage-closure-with-python.

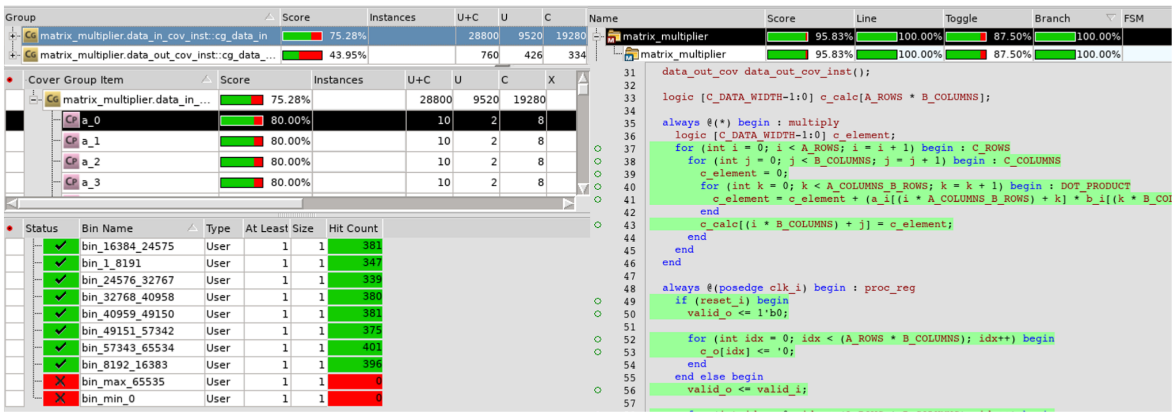
A. A Simple Use Case: Matrix Multiplier

```
class DataInCoverGroup(CoverGroup):
    a = [CoverPoint(BinMinMax(0, (1<<DATA_WIDTH) - 1, DATA_WIDTH//2)) for _ in range(A_ROWS * A_COLUMNS_B_ROWS * B_COLUMNS)]
    b = [CoverPoint(BinMinMax(0, (1<<DATA_WIDTH) - 1, DATA_WIDTH//2)) for _ in range(A_ROWS * A_COLUMNS_B_ROWS * B_COLUMNS)]
    cross_a_b = [Cross([a_i, b_i]) for a_i, b_i in zip(a, b)]

class DataOutCoverGroup(CoverGroup):
    c = [CoverPoint(BinMinMax(0, (1<<C_DATA_WIDTH) - 1, C_DATA_WIDTH//2)) for _ in range(A_ROWS * B_COLUMNS)]

class DataInCoverage(CoverageModel):
    cg_data_in = DataInCoverGroup()

class DataOutCoverage(CoverageModel):
    cg_data_out = DataOutCoverGroup()

data_in_cov = DataInCoverage("data_in_cov_inst")
data_out_cov = DataOutCoverage("data_out_cov_inst")
```

Figure 7. Coverage definition of the matrix multiplier

```
class InputDataValidMonitor(DataValidMonitor):
    def __init__(self, clk, datas, valid, dut, coverage):
        super().__init__(clk, datas, valid)
        self.coverage = coverage
        self.coverage.cg_data_in.connect(getattr(dut, coverage.sv_instname))

    def _sample(self) -> Dict[str, Any]:
        a_matrix = self._datas["A"].value
        b_matrix = self._datas["B"].value

        for i, j, n in itertools.product(range(A_ROWS), range(B_COLUMNS), range(A_COLUMNS_B_ROWS)):
            a_idx = (i * A_COLUMNS_B_ROWS) + n
            b_idx = (n * B_COLUMNS) + j
            m_idx = (j * A_COLUMNS_B_ROWS * A_ROWS) + (i * A_COLUMNS_B_ROWS) + n

            self.coverage.cg_data_in[m_idx].a <= a_matrix[a_idx].value
            self.coverage.cg_data_in[m_idx].b <= b_matrix[b_idx].value
        self.coverage.cg_data_in[m_idx].sample()

        return super()._sample()


class OutputDataValidMonitor(DataValidMonitor):
    def __init__(self, clk, datas, valid, dut, coverage):
        super().__init__(clk, datas, valid)
        self.coverage = coverage
        self.coverage.cg_data_out.connect(getattr(dut, coverage.sv_instname))

    def _sample(self) -> Dict[str, Any]:
        c_matrix = self._datas["C"].value

        for i, j in itertools.product(range(A_ROWS), range(B_COLUMNS)):
            idx = i * (B_COLUMNS) + j
            self.coverage.cg_data_out.c[idx] <= c_matrix[idx].value
        self.coverage.cg_data_out.sample()

        return super()._sample()
```

Figure 8. Coverage collection of the matrix multiplier



Group	Score	Instances	U+C	U	C
matrix_multiplier.data_in_cov_inst::cg_data_in	75.28%		28800	9520	19280
matrix_multiplier.data_out_cov_inst::cg_data_…	43.95%		760	426	334

Cover Group Item	Score	Instances	U+C	U	C	X
matrix_multiplier.data_in_…	75.28%		28800	9520	19280	
a_0	80.00%		10	2	8	
a_1	80.00%		10	2	8	
a_2	80.00%		10	2	8	
a_3	80.00%		10	2	8	

Status	Bin Name	Type	At Least	Size	Hit Count
✓	bin_16384_24575	User	1	1	381
✓	bin_1_8191	User	1	1	347
✓	bin_24576_32767	User	1	1	339
✓	bin_32768_40958	User	1	1	380
✓	bin_40959_49150	User	1	1	381
✓	bin_49151_57342	User	1	1	375
✓	bin_57343_65534	User	1	1	401
✓	bin_8192_16383	User	1	1	396
✗	bin_max_65535	User	1	1	0
✗	bin_min_0	User	1	1	0

Name	Score	Line	Toggle	Branch	FSM
matrix_multiplier	95.83%	100.00%	87.50%	100.00%	
matrix_multiplier	95.83%	100.00%	87.50%	100.00%	

```
data_out_cov data_out_cov_inst();

logic [C_DATA_WIDTH-1:0] c_calc[A_ROWS * B_COLUMNS];

always @(*) begin : multiply
  logic [C_DATA_WIDTH-1:0] c_element;
  for (int i = 0; i < A_ROWS; i = i + 1) begin : C_ROWS
    for (int j = 0; j < B_COLUMNS; j = j + 1) begin : C_COLUMNS
      c_element = 0;
      for (int k = 0; k < A_COLUMNS_B_ROWS; k = k + 1) begin : DOT_PRODUCT
        c_element = c_element + (a_i[(i * A_COLUMNS_B_ROWS) + k] * b_i[(k * B_CO
      end
      c_calc[(i * B_COLUMNS) + j] = c_element;
    end
  end
end

always @(posedge clk_i) begin : proc_reg
  if (reset_i) begin
    valid_o <= 1'b0;

    for (int idx = 0; idx < (A_ROWS * B_COLUMNS); idx++) begin
      c_o[idx] <= '0;
    end
  end else begin
    valid_o <= valid_i;
```

Figure 9. Screenshot of functional and code coverage result

Figure 7 represents coverage models for the matrix multiplier, which is one of the Cocotb examples. There are two covergroups, one for two input matrices and the other for one output matrix. The number of input coverage bins is equal to the total number of multiplications and the number of output coverage bins is equal to the number of elements of the output matrix. List comprehensions are used to efficiently generate coverpoint instances with the same type.

In Figure 8, coverage collection is done by extending the existing bus monitor. Since the _sample method is called at every valid transaction, we override this method and add coverage related codes to sample coverage.

Figure 9 shows the collected coverage metrics after running a certain number of tests. As in the figure, both code coverage and functional coverage are displayed together so that verification engineers can identify overall verification status easily, which contributes to faster coverage closure.

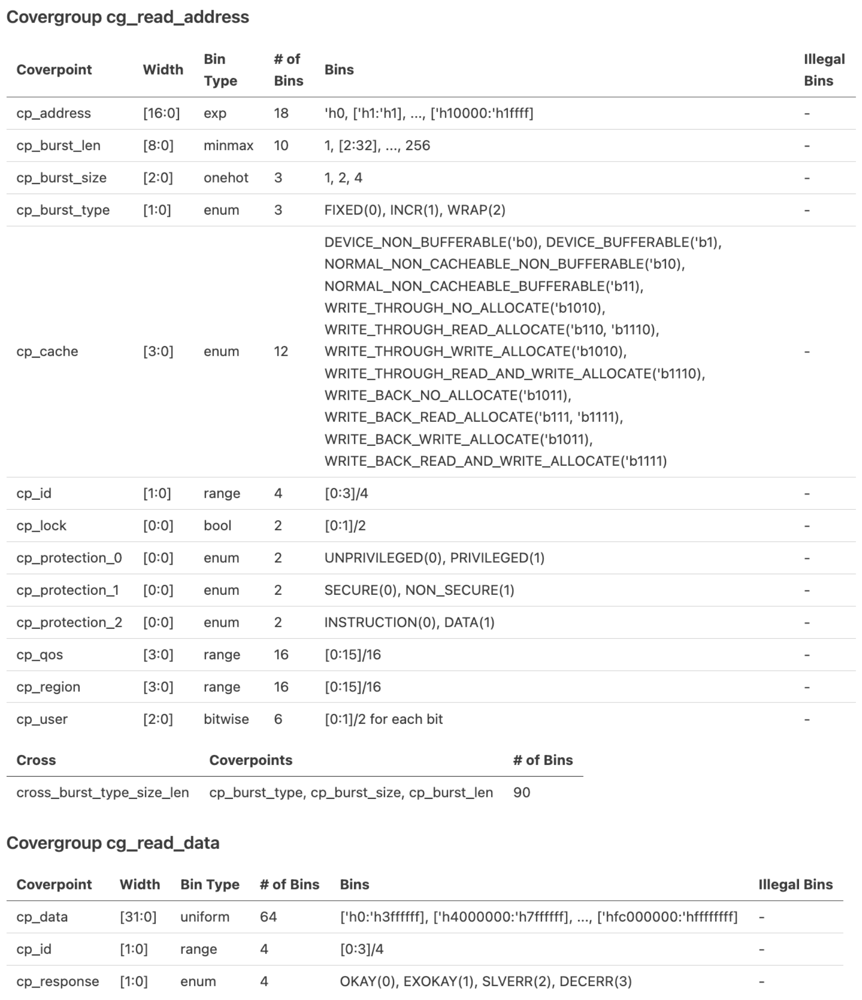
B. *A More Practical Use Case: AXI Interface*

Figure 10 is an example of a functional coverage model for the AMBA AXI using the proposed library [5]. All coverpoint classes are concisely described with predefined bin classes and all covergroup classes are simply described by inheriting base or extended covergroup classes. Figure 11 and Figure 12 are the generated markdown document on a markdown viewer and measured coverage on one of the EDA vendor's coverage analysis tools, respectively.

```
class AxiAddressChannel(CoverGroup):
    cp_id = CoverPoint(BinRange(1 << config.id_width))
    cp_address = CoverPoint(BinExp(config.addr_width), format="x")
    cp_burst_type = CoverPoint(BinEnum(AxiBurstType))
    cp_burst_size = CoverPoint(BinOneHot(int(log2(config.data_width >> 3)) + 1))
    cp_burst_len = CoverPoint(BinMinMax(min=1, max=256, num_bins=8))

    cp_protection = [CoverPoint(BinEnum(AxiProtectionPrivleged)),
                     CoverPoint(BinEnum(AxiProtectionSecure)),
                     CoverPoint(BinEnum(AxiProtectionInstruction))]

    cp_lock = CoverPoint(BinBool())
    cp_qos = CoverPoint(BinRange(16))
    cp_region = CoverPoint(BinRange(16))
    cp_user = CoverPoint(BinBitwise(config.user_width))

    cross_burst_type_size_len = Cross([cp_burst_type, cp_burst_size, cp_burst_len])

class AxiDataChannel(CoverGroup):
    cp_data = CoverPoint(BinUniform(num_bins=64, width=config.data_width), format="x")

class AxiResponseChannel(CoverGroup):
    cp_id = CoverPoint(BinRange(1 << config.id_width))
    cp_response = CoverPoint(BinEnum(AxiResponse))

class AxiWriteAddressChannel(AxiAddressChannel):
    cp_cache = CoverPoint(BinEnum(AxiWriteCache), format="b")

class AxiWriteDataChannel(AxiDataChannel):
    cp_strobe = CoverPoint(BinBitwise(config.data_width >> 3), format="x")

class AxiWriteResponseChannel(AxiResponseChannel):
    pass
```

```
class AxiReadAddressChannel(AxiAddressChannel):
    cp_cache = CoverPoint(BinEnum(AxiReadCache), format="b")

class AxiReadDataChannel(AxiDataChannel, AxiResponseChannel):
    pass

class AxiProtocolCoverage(CoverageModel):
    cg_write_address = AxiWriteAddressChannel()
    cg_write_data = AxiWriteDataChannel()
    cg_write_response = AxiWriteResponseChannel()
    cg_read_address = AxiReadAddressChannel()
    cg_read_data = AxiReadDataChannel()
```

Figure 10. Functional coverage definition for AMBA AXI



Covergroup cg_read_address

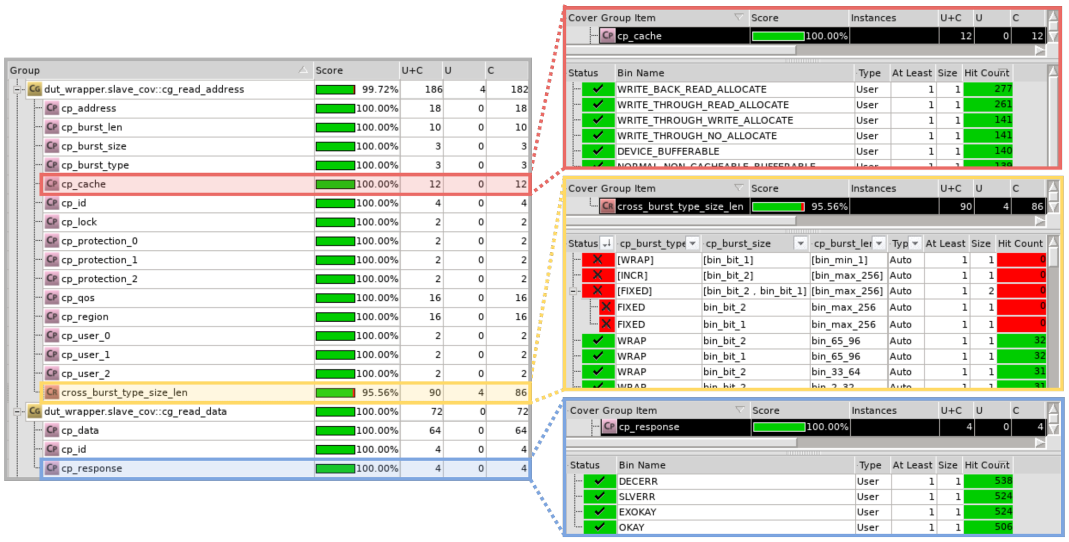
Coverpoint	Width	Bin Type	# of Bins	Bins	Illegal Bins
cp_address	[16:0]	exp	18	'h0, ['h1:'h1], ..., ['h10000:'h1ffff]	-
cp_burst_len	[8:0]	minmax	10	1, [2:32], ..., 256	-
cp_burst_size	[2:0]	onehot	3	1, 2, 4	-
cp_burst_type	[1:0]	enum	3	FIXED(0), INCR(1), WRAP(2)	-
cp_cache	[3:0]	enum	12	DEVICE_NON_BUFFERABLE('b0), DEVICE_BUFFERABLE('b1), NORMAL_NON_CACHEABLE_NON_BUFFERABLE('b10), NORMAL_NON_CACHEABLE_BUFFERABLE('b11), WRITE_THROUGH_NO_ALLOCATE('b1010), WRITE_THROUGH_READ_ALLOCATE('b110, 'b1110), WRITE_THROUGH_WRITE_ALLOCATE('b1010), WRITE_THROUGH_READ_AND_WRITE_ALLOCATE('b1110), WRITE_BACK_NO_ALLOCATE('b1011), WRITE_BACK_READ_ALLOCATE('b111, 'b1111), WRITE_BACK_WRITE_ALLOCATE('b1011), WRITE_BACK_READ_AND_WRITE_ALLOCATE('b1111)	-
cp_id	[1:0]	range	4	[0:3]/4	-
cp_lock	[0:0]	bool	2	[0:1]/2	-
cp_protection_0	[0:0]	enum	2	UNPRIVILEGED(0), PRIVILEGED(1)	-
cp_protection_1	[0:0]	enum	2	SECURE(0), NON_SECURE(1)	-
cp_protection_2	[0:0]	enum	2	INSTRUCTION(0), DATA(1)	-
cp_qos	[3:0]	range	16	[0:15]/16	-
cp_region	[3:0]	range	16	[0:15]/16	-
cp_user	[2:0]	bitwise	6	[0:1]/2 for each bit	-

Cross	Coverpoints	# of Bins
cross_burst_type_size_len	cp_burst_type, cp_burst_size, cp_burst_len	90

Covergroup cg_read_data

Coverpoint	Width	Bin Type	# of Bins	Bins	Illegal Bins
cp_data	[31:0]	uniform	64	['h0:'h3ffffff], ['h4000000:'h7ffffff], ..., ['hfc000000:'hffffffff]	-
cp_id	[1:0]	range	4	[0:3]/4	-
cp_response	[1:0]	enum	4	OKAY(0), EXOKAY(1), SLVERR(2), DECERR(3)	-

Figure 11. Generated functional coverage document for AMBA AXI



Group	Score	U+C	U	C
dut_wrapper.slave_cov::cg_read_address	99.72%	186	4	182
cp_address	100.00%	18	0	18
cp_burst_len	100.00%	10	0	10
cp_burst_size	100.00%	3	0	3
cp_burst_type	100.00%	3	0	3
cp_cache	100.00%	12	0	12
cp_id	100.00%	4	0	4
cp_lock	100.00%	2	0	2
cp_protection_0	100.00%	2	0	2
cp_protection_1	100.00%	2	0	2
cp_protection_2	100.00%	2	0	2
cp_qos	100.00%	16	0	16
cp_region	100.00%	16	0	16
cp_user_0	100.00%	2	0	2
cp_user_1	100.00%	2	0	2
cp_user_2	100.00%	2	0	2
cross_burst_type_size_len	95.56%	90	4	86
dut_wrapper.slave_cov::cg_read_data	100.00%	72	0	72
cp_data	100.00%	64	0	64
cp_id	100.00%	4	0	4
cp_response	100.00%	4	0	4

Cover Group Item	Score	Instances	U+C	U	C
cp_cache	100.00%		12	0	12

Status	Bin Name	Type	At Least	Size	Hit Count
✓	WRITE_BACK_READ_ALLOCATE	User	1	1	277
✓	WRITE_THROUGH_READ_ALLOCATE	User	1	1	261
✓	WRITE_THROUGH_WRITE_ALLOCATE	User	1	1	141
✓	WRITE_THROUGH_NO_ALLOCATE	User	1	1	141
✓	DEVICE_BUFFERABLE	User	1	1	140
✓	NORMAL_NON_CACHEABLE_BUFFERABLE	User	1	1	139

Cover Group Item	Score	Instances	U+C	U	C
cross_burst_type_size_len	95.56%		90	4	86

Status	cp_burst_type	cp_burst_size	cp_burst_ler	Typ	At Least	Size	Hit Count
✗	[WRAP]	[bin_bit_1]	[bin_min_1]	Auto	1	1	0
✗	[INCR]	[bin_bit_2]	[bin_max_256]	Auto	1	1	0
✗	[FIXED]	[bin_bit_2 , bin_bit_1]	[bin_max_256]	Auto	1	2	0
✗	FIXED	bin_bit_2	bin_max_256	Auto	1	1	0
✗	FIXED	bin_bit_1	bin_max_256	Auto	1	1	0
✓	WRAP	bin_bit_2	bin_65_96	Auto	1	1	32
✓	WRAP	bin_bit_1	bin_65_96	Auto	1	1	32
✓	WRAP	bin_bit_2	bin_33_64	Auto	1	1	31
✓	WRAP	bin_bit_2	bin_2_32	Auto	1	1	31

Cover Group Item	Score	Instances	U+C	U	C
cp_response	100.00%		4	0	4

Status	Bin Name	Type	At Least	Size	Hit Count
✓	DECERR	User	1	1	538
✓	SLVERR	User	1	1	524
✓	EXOKAY	User	1	1	524
✓	OKAY	User	1	1	506

Figure 12. Measured coverage for AMBA AXI on EDA vendor's coverage analysis tool

V. Conclusion

Existing Python coverage libraries had limited coverage features and lacked coverage analyzing functionality, making it difficult to perform coverage closure on Python-based testbenches. The proposed library seamlessly integrates SystemVerilog and Python to implement various types of coverage bins, and is fully compatible with EDA tools, allowing use of existing coverage workflows. In addition, we've leveraged the strengths of Python to concisely implement coverage and automatically generate documentation for review. These advantages empower engineers, streamlining the path to efficient and effective functional coverage closure.

Acknowledgements

This work was supported by the Institute of Information & communications Technology Planning & evaluation (IITP) grant funded by the Korean government (MSIT, Development of Artificial Intelligence Deep Learning Processor Technology for Complex Transaction Processing Server) under No. 2020-0-01309-001.

References

[1] Foster H. "The 2022 Wilson Research Group Functional Verification Study", Oct 2022. [Online]. Available: https://blogs.sw.siemens.com/verificationhorizons/2022/10/10/prologue-the-2022-wilson-research-group-functional-verification-study
[2] Rosser, Benjamin John. "Cocotb: a Python-based digital logic verification framework." In Micro-electronics Section seminar. Geneva, Switzerland: CERN, 2018
[3] Cieplucha, Marek, and W. Pleskacz. "New constrained random and metric-driven verification methodology using python." In Proceedings of the design and verification conference and exhibition US (DVCon). 2017
[4] Ballance, M. "PyVSC: SystemVerilog-Style Constraints, and Coverage in Python."
[5] ARM, "AMBA AXI Protocol Specification", no. K, Sep. 2023, [online] Available: http://www.arm.com.