New Constrained Random and Metric-Driven Verification Methodology using Python

Marek Cieplucha and Witold A. Pleskacz
Institute of Microelectronics and Optoelectronics
Warsaw University of Technology
e-mail: {m.cieplucha, w.pleskacz}@imio.pw.edu.pl

***Abstract*–The work described in this paper presents a solution which allows Python to be used for complex functional verification tasks. This is an extension of the Cocotb framework by functional coverage and constrained-random verification mechanisms. The implemented functionality is a set of base classes which can be used directly or extended by verification engineers according to project-specific requirements. The presented solution allows for much easier set-up of complex verification environments and, due to Python language features, it facilitates use of advanced object-oriented constructs.**

I. Introduction

Functional (or dynamic) verification is one of the major tasks of the digital integrated circuits design process. Its main goal is to ensure the equivalence between the hardware model and its specification. It is achieved by simulating the design under test (DUT) using test stimuli and observing the DUT behavior. Today, random verification is preferred over directed testing. Its principle is to generate a test with random stimuli, which makes a test scenario different at each execution. There are two fundamental techniques used in random verification: constrained randomization and functional coverage. Hardware verification languages (HVL), such as SystemVerilog or *e* provide a special syntax enabling use of these techniques. Also, modern methodologies, such as Universal Verification Methodology (UVM), are built upon this functionality.

As the HVLs syntax for constrained randomization and functional coverage is fixed, it is difficult to extend its features, or adapt it for some specific tasks. On the other hand, the growth of various programming languages allows for building complex applications faster and easier. HVLs however are unable to benefit from this advantage, due to their slow adoption and focus on a limited scope.

Cocotb [1] is an open-source based framework, which enables the RTL verification with arbitrary simulator engine. The role of an external simulator is limited only to a logic-level simulation and all transactions are translated to Python using the Verilog Procedural Interface (VPI). It means that all stimuli generation and signals monitoring is executed in Python, so no additional Verilog or SystemVerilog testbench is needed at all. The Python language syntax has many advantages over SystemVerilog and is considered to be more effective for the development of complex programs. One of the most important features of Python, regarding the assumptions of this work, is that functions in Python are first-class citizens [2]. It means, they can be manipulated the same way as variables or objects. For example, they can be passed as arguments of other functions or returned. This is an additional level of abstraction, not available in HVLs.

The main goal of this work was to implement a Cocotb framework extension that provides the constrained randomization and functional coverage mechanisms. The framework is based on Python implementation, which means that the presented features are part of the testbench, instead of being part of the fixed language syntax (like in HVLs). This approach enables implemented mechanisms to be easily adopted or extended by verification engineers, considering project-specific requirements. For example, new coverage primitives or new, non-standard constraint types can be implemented.

From the methodical point of view, the presented solutions correspond to Constrained Random Verification (CRV) and Metric Driven Verification (MDV) subjects. The framework allows for building verification environments according to these methodologies. Together with Cocotb core features, it is possible to set up a verification environment similar to UVM-based one, for example implemented in SystemVerilog. Moreover, due to Python syntax benefits and object-oriented approach, constructing testbenches is faster and more intuitive.

One of the goals was also to provide a solution which will be easy to understand for SystemVerilog users. As the implemented coverage and constraint objects become fully-featured testbench elements, it is simpler to manipulate them in real time. The main motivation for this work was to provide a scalable solution that is not only convenient to use for simple tasks, but also facilitates building comprehensive environments.

II. Related work

As mentioned already, constrained randomization an functional coverage features are present in SystemVerilog [3] and *e* [4] as a part of the language syntax. These constructs will be discussed later in comparison to similar features implemented in Python. Also SystemC language may be used for digital verification. Basic support for randomization mechanisms is available in SystemC [5], but there are no default functional coverage constructs. There are several enhancements proposed for elementary SystemC features. A framework called CRAVE [6] has been presented for advanced constrained randomization tasks. The goal of the work [7] was to implement functional coverage extensions, similar to existing features in SystemVerilog. The most recent works enable the use of C++11 standard within SystemC and present enhancements for constrained randomization, functional coverage and other add-ons [8], [9]. It can be clearly seen from these papers that high-level features of C++ syntax allow for more agile implementation of complex programs, including verification environments.

Another important group of projects is related to the topic of improving stimuli generation for random tests. The goal is to satisfy the functional coverage metrics faster than in the case of straightforward executing random tests. The idea of coverage-directed test generation is introduced in [10] and later expanded with evolutionary algorithms [11]. The main principle of this idea is to observe how test stimuli lead to covering specific coverage points. Using the feedback from the coverage result, new test constraints can be dynamically generated. No information about the DUT itself is processed during new tests generation. Nevertheless, all these solutions assume that the engine improving the verification closure process is placed outside the testbench core code (e.g. external C++ library in [11]).

There is a lot of interest in CRV and MDV topics, but for advanced ideas ([10], [11]) HVLs limitations are an important issue. On the other hand, in [12] it is proposed that a simulator tool can handle the presented problem. This is however a non-scalable approach which can be used only for very specific tasks. It seems that due to rather slow growth and adoption of HVLs, there is a room for different, more software-concerned approach to verification.

This idea is incorporated in the Cocotb framework. As stated in the introduction, the complete testbench, based on Cocotb, can be written in Python. Some primitives, such as base classes for drivers or monitors are already provided. However, the lack of constrained randomization and functional coverage features makes advanced MDV and CRV tasks impossible.

III. Verification closure and its limitations

The methodical approach to the modern verification process [13] is presented in Fig. 1. First of all, the verification plan must be created, which contains coverage items organized in various sections. The plan elements are extracted from the specification (functional requirements) or may be implementation-specific (e.g. performance or system-level assumptions). There are a number of test scenarios implemented which intend to examine the verification plan elements. According to the CRV approach, they are complex tests with highly randomized flow, which are executed several times using different *seeds*. Therefore the same scenario verifies different features at each test run.

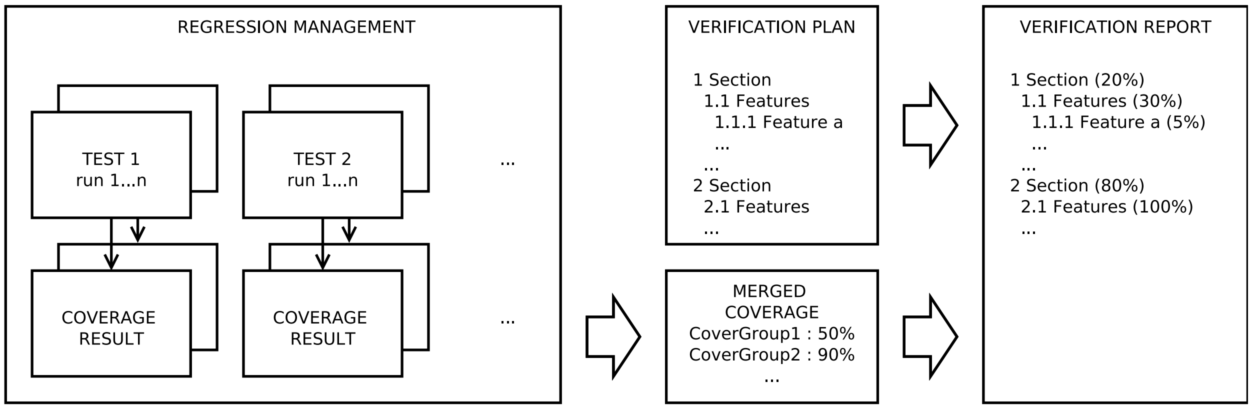

Fig. 1: Metric-driven verification flow

The coverage metrics are extracted from the executed tests. There are several types of metrics that can be monitored. It can be code coverage, functional coverage or just information that specific scenario finished successfully.

The last one is however a very "directed" approach and usually requires the application of dedicated constraints for a test scenario, or even a unique test design. The limitations of code coverage metrics are that the results may not be relevant [14]. It is possible to achieve full code coverage while DUT lacks important functional requirements. Therefore the functional coverage metrics are the most important and most widely used.

There is an additional step required which is usually not mentioned explicitly in the verification flow. This is the mapping of the functional coverage implementation to the verification plan. This task may be unfortunately performed bottom-up – the plan may be composed of items which are expressed in the language of HVL functional coverage constructs. This may lead to missing important higher-level features. On the other hand, abstract design features are difficult to be monitored by functional coverage primitives known from HVLs. For example, problems appear when the functionality is not easy to be described by numbers, states or sequences. This is often resolved by mapping several coverage items to a single plan feature, which also may lead to a mistake (e.g. not sufficient coverage implementation).

Due to these problems, there is a need to use functional coverage constructs for higher-level and abstract features. This is technically possible in HVLs, but requires additional effort, e.g. implementation of helping functions, additional variables or custom sampling conditions. It introduces an additional stage in the verification environment where a mistake can be made. On the other hand, functional coverage should be kept simple due to the fact that it examines the test execution. When coverage implementation requires additional implemented code, it obviously may contain bugs and thus incorrect coverage metrics can be provided.

The regression testing is a process of a full verification of the DUT against the verification plan. The verification may be considered complete when all elements in the verification plan are successfully tested. However, a requirement for a successful feature validation may be ambiguous. What happens, when for example, a test is finished with and without errors for different executions? Another problem is the test session definition. How many times the individual test needs to be executed to cover all corner cases? Is there a number of repetitions that guarantees a full coverage? The answer is no, as each test execution is independent. It means that running tests takes a lot of time to make sure all corner cases are hit. But most of the time, the already covered features are examined repeatedly.

The first step towards a dynamic and controlled regression management is to make more use of the coverage metrics in real time. This is not possible using the current commercial approach, the main goal of which is only to collect and merge coverage results from independently executed tests. The solution of this problem could be a coverage-directed test generation idea presented in Sec. II. However, there is no methodical support for the described solutions, as presented verification closure flow (also valid in UVM) is not adaptive.

IV. Functional Coverage features

In SystemVerilog [3] a fundamental coverage unit is a *coverpoint*. It contains several bins and each bin may contain several values. Every *coverpoint* is associated with a variable or signal. At sampling event, the *coverpoint* variable value is compared with each defined bin. If there is a match, then the number of hits of the particular bin is incremented. *Coverpoints* are organized in *covergroups*, which are specific class-like structures. A single *covergroup* may have several instances and each instance may collect coverage independently. A *covergroup* requires sampling, which may be defined as a logic event (e.g. a positive clock edge). Sampling may also be called implicitly in the testbench procedural code by invoking a *sample()* method of the *covergroup* instance. SVA (SystemVerilog Assertions) syntax may be used to build advanced sequences containing repetitions, multi-stage successions or wildcard transitions. A bin may be also defined as an *ignore_bins*, which means its match does not increase a coverage count, or an *illegal_bins*, which results in error when hit during the test execution.

Another coverage construct in SystemVerilog is a *cross*. It automatically generates a Cartesian product of bins from several *coverpoints*. It is a useful feature simplifying the functional coverage generation. As it may be difficult or unnecessary to cover all the cross-bins, some of them may be excluded from the analysis. This is possible using the *binsof ... intersect* syntax.

The functional coverage features are also implemented in *e* language [4]. They are similar to SystemVerilog, but some constructs are simplified and their functionality is reduced.

The most important limitations of the SystemVerilog functional coverage features are:

- straightforward bins matching criteria – only satisfied by equality or inclusion relation;
- bins may be only constants or transitions (possibly wildcard);
- flat coverage structure – cover groups cannot contain other cover groups, which would correspond better to a verification plan scheme;
- not possible to get the detailed coverage information in real time (e.g. when a specific bin was hit).

A. Implemented mechanisms

The general assumptions for the proposed architecture of the functional coverage features are as follows:

- functional coverage structure should better match a real verification plan;
- its syntax should be more flexible, but a separation between coverage and executable code should be maintained;
- features for analysing the coverage during test execution should be added or extended;
- coverage primitives should be able to monitor testbench objects at a higher level of abstraction.

The implemented mechanism is based on the idea of decorator design pattern. In Python, a decorator syntax is readable and easy to use. Instead of sampling coverage items by an additional method, decorators are by default invoked at each decorated function call. As it is easy to create functions in Python (for example anonymous functions can be created as *lambda expressions* - single-line function definitions), this is a convenient solution.

The implemented coverage structure is based on a prefix tree (a trie). The main coverage primitive is a *CoverItem*, which corresponds to a SystemVerilog *covergroup*. *CoverItem* may contain other *CoverItems* or objects extending *CoverItems* base class, which are *CoverPoints*, *CoverCrosses* or arbitrary new, user-defined types. *CoverItems* are created automatically, user defines only *CoverPoint* or *CoverCross* primitives (lowest level nodes in the trie). Each created primitive has a unique ID – a dot-divided string. This string denotes the position of an object in the coverage trie. For example, a *CoverPoint a.b.c* is a member of the *a.b CoverItem*, which is then a member of the *a CoverItem*. The structure of the coverage trie is presented in Fig. 2.

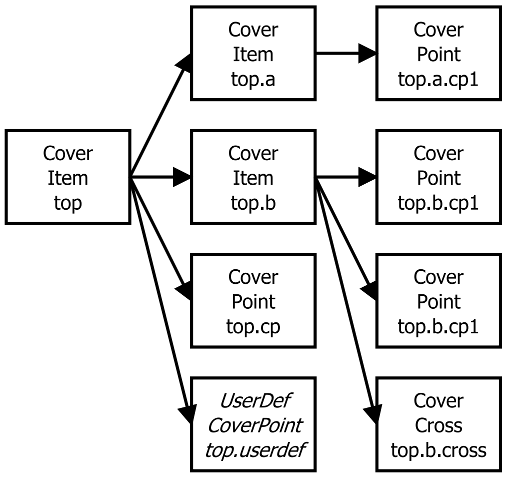

Fig. 2: An example of the coverage trie structure

A *CoverPoint* decorator functionality corresponds to a *coverpoint* in SystemVerilog. It checks whether the arguments of a decorated function match the predefined bins. In a simple case, variables equality to the bins is checked.

Additionally, it is possible to define:

- a **transformation function**, which transforms the arguments of a decorated function and the transformation result is compared to the bins,
- a **relation function**, which defines the binary relation of bin comparison (which is by default an equality operator).

Bins in the *CoverPoint* may be a list of arbitrary objects which are hashable. In Python they are constants, tuples or even functions. In general, the bins matching condition can be described by a formula:

relation(transformation(arguments), bin) is *True*

A *CoverCross* decorator functionality corresponds to a *cross* in SystemVerilog. Its main attributes are a list of items (*CoverPoints*), a list of bins to be ignored and an optional ignore relation function. *CoverCross* bins are tuples generated as a Cartesian product of bins from *CoverPoint* items. An item in (*ign_bins*) may contain a *None* object which corresponds to *binsof ... intersect* syntax, meaning a specific *CoverPoint* bin value may be wildcard.

An example below presents the same coverage implementation in SystemVerilog and in Python. As the *CoverPoint length* bins contain value range, a relation must be defined in Python implementation, which uses a tuple (in this case a pair) and finds out whether the variable is within a given range.

```
covergroup transfer;
  direction : coverpoint dir {
    bins read       = {0};
    bins write      = {1};
  }
  length : coverpoint length {
    bins short      = {[1:10]};
    bins long       = {[10:100]};
  }
  type : coverpoint type {
    bins type_a     = {A};
    bins type_b     = {B};
  }
  tr_cross : cross
    direction , length , type {
    ignore_bins ign =
      binsof(type) intersect {A};
  }
```

```
@CoverPoint( "transfer.direction",
  xf = lambda xfer : xfer.dir , bins = [0, 1]
  )
@CoverPoint( "transfer.length",
  xf = lambda xfer.length ,
  bins = [(1,10), (10,100)],
  rel = lambda val , b: b(0) <= val <= b(1)
  )
@CoverPoint( "transfer.type",
  xf = lambda xfer.type , bins = [A, B]
  )
@CoverCross( "transfer.tr_cross", items =
  ["transfer.direction", "transfer.length",
   "transfer.type"],
  ign_bins = [(None, None, A)]
  )
def decorated_function(xfer):
  ...
```

More complex examples of coverage mechanisms are presented below. The *coverage.transition* defines a transformation by a *transition_inta()* function. This function returns a tuple containing the previous and the current value of *inta*. It is a simple example of the transition bins. The *coverage.primefactors* defines a relation by a function *has_prime_factor()* checking if a bin value is a prime factor of *inta*. The *inj* attribute is set *True*, which means that more than one bin can be matched at a single sampling. For example, *inta* value of 30 matches bins "2", "3", and "5". The *coverage.tuple* presents how arbitrary hashable type may be used as a bins. The bins are predefined in a *simple_bins* list containing 40 elements of *(int, string)* pairs. The *coverage.check* is an example of a higher-level assertion. This is a new defined coverage primitive which checks whether the *string* variable is not empty. If at least one empty string is sampled, coverage level is forced zero.

```
simple_bins = [] #bins generation for coverage.tuple: create a 40-elements list
for i in range (1, 21): #for i = 1 to 20
  simple_bins.extend([(i, 'y'), (i, 'n')]) #extend list by two elements - tuples (int, str)

#transition function for coverage.transition
prev_value = 0; #previous value defined outside the function (global variable)
def transition_inta(inta, intb, string): #function definition
  transition = (prev_value, inta) #transition as a tuple of (int, int)
  prev_value = inta #update previous value
  return transition

#sampling function and its coverage decorators
@CoverPoint("coverage.transition", xf = transition_inta,  bins = [(1,2), (2,3), (3,4)])
@CoverPoint("coverage.primefactors", xf = lambda inta, intb, string : inta,
  rel = has_prime_factor, inj = True, bins = [2, 3, 5, 7, 11, 13, 17])
@CoverPoint("coverage.tuple", xf = lambda inta, intb, string : (inta + intb, string),
  bins = simple_bins)
@CoverCheck("coverage.check", f_fail = lambda inta, intb, string : string == "")
def decorated_function(inta, intb, string):
  ...
```

There are some higher-level functions available for *CoverItems*. They can be used in real time in the testbench, which allows for processing coverage data dynamically. It is possible to easily get the coverage data from each primitive or define a callback, called when coverage level has been exceeded or a specific bin was hit. Callbacks may be used in order to adjust a test scenario when specific coverage goal has been achieved. Instead of monitoring the coverage during the test execution, a callback function will be called automatically. A callback function may be simply appended to any *CoverItem* primitive by the testbench designer. More information about functional coverage background and this implementation can be found in [15].

V. CONSTRAINED RANDOM VERIFICATION FEATURES

SystemVerilog user may define random variables using *rand[c]* modifier. Calling *randomize()* function on a class instance (object) results in picking random values of the defined random variables, satisfying given constraints. Also a *with* modifier can be used together with *randomize()* which allow for appending additional constraints dynamically. Constraints are defined in a special section in the class named *constraint*. They describe a range values that a single variable may have or a relation between variables. It is also possible to define solution ranges with weights (using *dist* modifier). The *solve ... before* is an additional construction which organizes variable randomization order.

Constraints are unique constructs of SystemVerilog. They are class members, but they are not functions or objects. Basic operations can be performed on constraints, such as enable/disable or inheritance. Soft constraints have been introduced in SystemVerilog 2012. They are resolved only when it is possible to satisfy them together with all other hard constrains. Every SystemVerilog simulator must implement a constraint solver. Although many open-source constraint solvers are available, testbench designers cannot use them, as they have no control over the simulator engine.

The most important limitations of the existing constrained randomization features are related to their fixed syntax. In the proposed solution implemented in Python, it is assumed that a constraint may be any callable object – an arbitrary function or a class with *__call__* method. It allows for creating various functionalities quite easily and manipulating them in a flexible way.

A. Implemented mechanisms

The main assumption for the proposed constrained randomization features was to provide only a flexible API, and let the testbench designer to adjust it depending on project needs. There is a default open-source based hard constraint solver included in the described framework code [16], but it can be replaced by end user if required. The general idea of the implemented mechanisms is that all classes that intended to use randomized variables should extend the base class *Randomized*. Afterwards, random variables and their ranges should be defined. Constraints are just arbitrary functions with only one requirement: their argument names must match class member names. It is possible to define two types of constraints:

- functions that return a true/false value, corresponding to SystemVerilog hard constraints;
- functions that return a numeric value, corresponding to a variables distribution (or cross-distribution) which also may be used as soft constraints.

The full *Randomized* class API consists of the following functions:

- ***addRand(var, domain)*** - specifies *var* as a randomized variable taking values from the *domain* list;
- ***addConstraint(cstr)*** - adds a constraint function to the solver;
- ***delConstraint(cstr)*** - removes a constraint function from the solver;
- ***solveOrder(vars0, vars1 ...)*** - optionally specifies the order of randomizing variables (can be used for problem decomposition or in case some random variables must be fixed before randomizing the others);
- ***pre_randomize()*** - function called before *randomize[_with]()*, corresponding to similar function in SV;
- ***post_randomize()*** - function called after *randomize[_with]()*, corresponding to similar function in SV;
- ***randomize()*** - main function that picks random values of the variables satisfying added constraints;
- ***randomize_with(cstr0, cstr1 ...)*** - similar to *randomize()*, but satisfies additional given constraints.

The example below presents the corresponding implementation of the randomized class with use of hard constraints.

```
class rand_frame;
  typedef enum {SMALL,MED,BIG} size_t;
  rand logic [15:0] length;
  rand logic [15:0] pld;
  rand size_t size;

  constraint frame_sizes {
    if (size == MED) {
      length >= 64;
      length < 2000;
    } else if (size == SMALL)  {
      length > 0;
      length < 64;
    } else if (size == BIG) {
      length >= 2000;
      length < 5000;
    }
    pld < length;
    pld % 2 == 0;
  }
endclass
```

```
class rand_frame(crv.Randomized):
  def __init__(self):
    crv.Randomized.__init__(self)
    self.length = 0
    self.pld = 0
    self.size = ”SMALL”

    self.addRand(”size”,[”SMALL”, ”MED”, ”BIG”])
    self.addRand(”length”, list(range(1, 5000)))
    self.addRand(”pld”, list(range(0, 4999)))

    def frame_sizes(length, size):
      if (size == ”SMALL”) length < 64
      elif (size == ”MED”) 64 <= length < 2000
      else length >= 2000

    self.addConstraint(frame_sizes)
    self.addConstraint(
      lambda length, pld : pld < length
    )
    self.addConstraint(lambda pld : pld %2 == 0)
```

A more complex example is presented below. The class *TripleInt* contains three unsigned integer members, *y* and *z* are randomized. The first defined constraint combines all variables (random and non-random). The second constraint defines a triangular distribution for variable *z*. It is achieved by defining a function that has its maximum in the middle of the variable range (for solution $z = 500$). The third one is a cross-distribution of variables *y* and

z. The weight function defines higher probability for solutions with higher difference between both variables. The last one is a kind of a soft constraint – very low probability is set for condition $x > y$, which means that solutions satisfying $x \leq y$ will be strongly preferred.

```
class TripleInt(Randomized)
  def __init__(self, x):
    Randomized.__init__(self)
    self.x = x #this is a non-random value, determined at class instance creation
    self.y = 0
    self.z = 0
    addRand(y, list(range(1000))) #0 to 999
    addRand(z, list(range(1000))) #0 to 999
    addConstraint(lambda x, y, z : x + y + z == 1000)     #hard constraint
    addConstraint(lambda z: 500 - abs(500 - z))           #triangular distribution of z variable
    addConstraint(lambda y, z : 100 + abs(y - z))         #multi-dimensional distribution
    addConstraint(lambda x, y : 0.01 if (y > x) else 1)   #soft constraint
```

It is assumed that only one hard constraint and one distribution may be associated with each set of random variables. So, for the example presented above, it is possible to define no more than six constraint functions: separately for variables *y* and *z* and both (*y* and *z*). It means that constraints may be overwritten, for example by *randomize_with()* function arguments.

VI. Methodology example - basic test

This section presents a basic test demonstrating how the provided mechanisms can be used. The DUT is a simple module calculating the mean value of *bus_width* inputs, each of size *data_width* bits. The verification goal of the test is to achieve a 100% functional coverage. The coverage monitors data values on the first and last input buses and we expect to cover the whole data range of both of them.

The *StreamTransaction* class defines a random variable *data*, which is a list of tuples of all possible data combinations. In the *StreamBusDriver* we define a functional coverage inside driver's *send* function. Data on the first (0) and last (*bus_width*-1) input bus is monitored using two *CoverPoints*. The main test function *mean_mdv_test()* checks the data using Scoreboard and performs the main while loop. The loop checks the coverage level and sends the randomized transaction to the DUT satisfying a given constraint. To speed up the verification closure the constraint prevents the random data, that has already been covered, from being generated again. This corresponds to the coverage-directed test generation approach discussed in Sec. II.

```
class StreamTransaction(Randomized):
    ...
    def __init__(self, bus_width, data_width):
        ...
        list_data = range(0, 2**data_width) #a list of all possible data (0, 1 ... 2^data_width-1)
        #generate the Cartesian product of tuples of all possible input combinations
        #e.g. (0,0), (0,1) ... (2^data_width-1, 2^data_width-1) for bus_width = 2,
        # (0,0,0) ... (2^data_width-1, 2^data_width-1, 2^data_width-1) for bus_width = 3 etc.
        combinations = list(itertools.product(list_data, repeat=bus_width))
        self.addRand("data", combinations)

class StreamBusDriver(BusDriver):
    ...
    @cocotb.coroutine
    def send(self, transaction):
        ...
        @cocotb.coverage.CoverPoint("top.data1", xf = lambda transaction : transaction.data[0],
            bins = range(0, 2**transaction.data_width))
        @cocotb.coverage.CoverPoint("top.dataN",
            xf = lambda transaction : transaction.data[transaction.bus_width-1],
            bins = range(0, 2**transaction.data_width))
        def sample_coverage(transaction):
            pass
        sample_coverage(transaction)

@cocotb.test()
def mean_mdv_test(dut):
    dut_out = StreamBusMonitor(dut, "o", dut.clk) #DUT outputs Monitor
    dut_in = StreamBusDriver(dut, "i", dut.clk)   #DUT inputs Driver
    exp_out = []
    scoreboard = Scoreboard(dut) #Scoreboard compares DUT output with exp_out list content
    scoreboard.add_interface(dut_out, exp_out)
```

```
...
#a constraint function: do not pick values that have been already covered
def data_constraint(data):
    return (not data[0] in coverage1_hits) & (not data[bus_width-1] in coverageN_hits)

coverage = 0
xaction = StreamTransaction(bus_width, data_width) #create transaction instance
while coverage < 100:
    #observe newly hit bins after each transaction
    coverage1_new_bins = coverage.coverage_db["top.data1"].new_hits
    coverageN_new_bins = coverage.coverage_db["top.dataN"].new_hits
    coverage1_hits.extend(coverage1_new_bins)
    coverageN_hits.extend(coverageN_new_bins)
    xaction.randomize_with(data_constraint) #randomize transaction
    yield dut_in.send(xaction) #execute transaction on Driver
    exp_out.append(xaction.mean_value()) #calc mean value and send result to the Scoreboard
    #calculate the current coverage level
    coverage = coverage.coverage_db["top"].coverage*100/coverage.coverage_db["top"].size
```

For *data_width* = 6 the simulation time was about 5 times shorter than without this constraint. For *data_width* = 8 the difference is about 10 times. This basic demonstration shows the flexibility and ease of use of the presented MDV and CRV features. The implementation of a similar test scenario in SystemVerilog would be much more complicated.

VII. Conclusion

The presented approach for MDV and CRV techniques using Python shows many benefits over similar verification strategy using HVLs. First of all, the simulator is performing only RTL simulations, it is not required to support HVLs. It means that, for example, open source simulators can be used with Cocotb for complex verification tasks. The presented functional coverage features allow for a higher level of abstraction objects to be monitored and tracked. The coverage structure also better matches a real verification plan. That shortens the gap between a verification plan structure and functional coverage implementation. The described CRV features demonstrate higher flexibility than fixed HVLs syntax. Finally, due to Python syntax clarity, composing programs that use the presented mechanisms together is straightforward.

The only issue not discussed in this paper is performance. Python, as an interpreted language has no excellent performance results. This is however a relatively minor issue, as most of the complexity of the functional verification task is the DUT RTL simulation. Nevertheless, a VPI implementation differs between simulators and they may demonstrate different performance capabilities.

The Cocotb framework and presented extensions are available as an open source.

References

[1] *COCOTB 1.0 documentation*, Potential Ventures, 2014. [Online]. Available: http://cocotb.readthedocs.org/en/latest/introduction.html

[2] H. Abelson and G. J. Sussman, *Structure and Interpretation of Computer Programs*, 2nd ed. Cambridge, MA, USA: MIT Press, 1996.

[3] *1800-2012 - IEEE Standard for SystemVerilog – Unified Hardware Design, Specification, and Verification Language*, IEEE Std. 10.1109/IEEESTD.2013.6469140, Feb. 2013.

[4] *1647-2011 - IEEE Standard for the Functional Verification Language e*, IEEE Std. 10.1109/IEEESTD.2011.6006495, Aug. 2011.

[5] *SystemC Verification Standard Specification 1.0e*, SystemC Verification Working Group, May 2003.

[6] F. Haedicke, H. M. Le, D. Groe, and R. Drechsler, "CRAVE: An advanced constrained random verification environment for SystemC," in *System on Chip (SoC), 2012 International Symposium on*, Oct 2012, pp. 1–7.

[7] C. Kuznik and W. Mueller, "Functional coverage-driven verification with SystemC on multiple level of abstraction," *Proceedings of the Design and Verification Conference and Exhibition US (DVCon)*, Mar. 2011.

[8] T. Vortler, T. Klotz, K. Einwich, Y. Li, Z. Wang, M.-M. Louerat, J.-P. Chaput, F. Pecheux, R. Iskander, and M. Barnasconi, "Enriching UVM in SystemC with AMS extensions for randomization and functional coverage," *Proceedings of the Design and Verification Conference and Exhibition Europe (DVCon)*, Oct. 2014.

[9] H. M. Le and R. Drechsler, "Boosting SystemC-based testbenches with modern C++ and coverage-driven generation," *Proceedings of the Design and Verification Conference and Exhibition Europe (DVCon)*, Nov. 2015.

[10] S. Fine and A. Ziv, "Coverage directed test generation for functional verification using Bayesian networks," in *Proc. of the Design Automation Conference (DAC)*. IEEE, Jun. 2003, pp. 286–291.

[11] C. Ioannides, G. Barrett, and K. Eder, "Introducing XCS to coverage directed test generation," in *IEEE International High Level Design Validation and Test Workshop (HLDVT)*. IEEE, Nov. 2011, pp. 57–64.

[12] M. Teplitsky, A. Metodi, and R. Azaria, "Coverage driven distribution of constrained random stimuli," *Proceedings of the Design and Verification Conference and Exhibition US (DVCon)*, Mar. 2015.

[13] S. Vasudevan, *Effective Functional Verification - Principles and Processes*, 1st ed. Springer US, 2006.

[14] A. Meyer, *Principles of Functional Verification*, 1st ed. Newnes, 2003.

[15] M. Cieplucha and W. Pleskacz, "New architecture of the object-oriented functional coverage mechanism for digital verification," in *2016 1st IEEE International Verification and Security Workshop (IVSW)*, July 2016, pp. 1–6.

[16] *python-constraint – Constraint Solving Problem resolver for Python*, Gustavo Niemeyer, 2011. [Online]. Available: https://labix.org/python-constraint