New Constrained Random and Metric-Driven Verification Methodology using Python

Functional Verification is Software Engineering

Marek Ciepłucha and Witold Pleskacz

Warsaw University of Technology
Institute of Micro- and Optoelectronics
Agenda

• Functional verification scope – a different view
• Functional Coverage and Constrained Randomization
• Cocotb – Python-based verification environment
• New Functional Coverage architecture proposal
• New Constrained Randomization architecture proposal
• Working examples
• Summary

This project was partially supported by Polish National Centre for Research and Development under project No. DOBR/0053/R/ID1/2013/03
What is Functional Verification?

- Applying stimuli to the DUT and checking response
  - DUT as an RTL model – simulation or emulation
  - DUT as a (pseudo-)HLS model – pure software concern

- Testbench is a software program at the transaction level
  - Only Monitors and Drivers are time-aware

Do we need *simulators* supporting verification process?
Functional Verification – a different approach

• Assume we only simulate DUT
  – Simulator not required to support HVLs
  – Free simulators can be used (e.g. Icarus Verilog)

• Testbench is a standalone application
  – API implementation needed to access simulator flow (e.g. VPI)

• Problem – VPI performance
Modern verification methodology

- Constrained Randomization
  - Mechanisms that simplify applying random stimuli

- Functional Coverage
  - Need to observe whether all expected scenarios executed

- Regression Testing – Coverage Closure
  - A process that examines verification against a plan – meeting defined metrics
Modern verification methodology

• High-level software verification - similar approach
• CRV an FC available as a part of HVLs syntax
• Regression Management – automated by EDA tools
• UVM – just a framework
  – Well understood only by UVM users
  – Resolves some typical use-case issues (like design patterns in software engineering)
  – UVM ≠ QUALITY
Constrained Random Verification

- Randomization of the data with given constraints:
  - Requires a constraint solver to be implemented
  - Part of the simulator – why here?

- System Verilog CRV syntax
  - Random variables
  - Constraints (soft constraints in SV2012)
  - Weighted distributions
  - Constraint = function?

```verbatim
constraint frame_sizes {
  size == NORMAL -> {
    length dist {
      [64 : 127] := 10,
      [128 : 511] := 10,
      [512 : 1500] := 10
    };
    ...
  }
}
```
source: http://www.asic-world.com/
• Metrics need to be defined for verification process:
  – Test scenarios
  – Code Coverage
  – Functional Coverage

• SystemVerilog Functional Coverage syntax
  – `covergroup, coverpoint, cross`
    • `bins`: signals, variables, sequences
  – Only countable features can be easily covered!
Functional Coverage limitations

• Flat coverage structure
• Only naive bins matching
• Coverage data separated from the testbench data
• Example implementation issue: coverage driven test generation

source: C. Kuznik and W. Mueller: Aspect enhanced functional coverage driven verification in the SystemC HDVL
Motivation for work

• Functional Verification as a software engineering
• Slow evolution of HVLs (and then – simulators)
• Lack of the „agility” of the verification process
• Expensive digital simulation environments
Cocotb

• Cocotb is a COroutine based COsimulation TestBench environment for verifying VHDL/Verilog RTL using Python
• Cocotb is completely free, open source (under the BSD License) and hosted on GitHub
• Cocotb requires a simulator to simulate the RTL

source: http://cocotb.readthedocs.io
Cocotb

• Base testbench classes: Driver, Monitor, Scoreboard
• Easy interfacing to other languages
• Missing features: functional coverage, randomization mechanisms, regression management
Python functions

• A function – first class citizen in Python (and many other modern languages)
  – Assigned or passed as an argument
  – Any object that is callable
  – Defined anywhere (inside other functions)
  – Lambda expressions

• Decorator design pattern

```python
@decorator(x,y)
def my_func(f1, w, z):
    def inside_func(a,b):
        return a + b

    f2 = lambda a,b: a - b

    if (w < z):
        return f1(w, z)
    elif (w > z):
        return inside_func(w,z)
    else:
        return f2(w, z)

my_func = decorator(my_func, x, y)
```
New Constrained Randomization Mechanism

• A constraint – an arbitrary function
  – Returns true/false – hard constraint
  – Returns numeric value – distribution
    • Can be used for soft constraints

• API:
  – addRand(var), addConstraint(function)
  – post/pre_randomize(), randomize[with]()
  – SolveOrder (solve … before)
Example: SV vs. Cocotb – randomization

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)
Example: Cocotb – advanced randomization

class TripleInt(Randomized):
    def __init__(self, x):
        Randomized.__init__(self)
        # this is a non-random value, determined at class instance creation
        self.x = x
        self.y = 0
        self.z = 0
        addRand(y, list(range(1000))) # 0 to 999
        addRand(z, list(range(1000))) # 0 to 999
        # HARD CONSTRAINT
        addConstraint(lambda x, y, z : x + y + z == 1000)
        # TRIANGULAR DISTRIBUTION
        addConstraint(lambda z: 500 - abs(500 - z))
        # MULTI-DIMENSIONAL DISTRIBUTION
        addConstraint(lambda y, z : 100 + abs(y - z))
        # SOFT CONSTRAINT
        addConstraint(lambda x, y : 0.01 if (y > x) else 1)
New Functional Coverage Mechanism

• A tree (trie) structure
• Coverage primitive – a function decorator
  – Called each time at the function call
  – User can define own coverage types
  – SystemVerilog originated:
    • CoverPoint
    • CoverCross
Example: SV vs. Cocotb – functional coverage

```plaintext
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):
  ...
```
```python
simple_bins = [] #bins generation for coverage.tuple:
for i in range (1, 21): #for i = 1 to 20
    simple_bins.extend([(i, 'y'), (i, 'n')])

#transition function for coverage.transition
prev_value = 0; #previous value defined outside the function
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

@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)
def decorated_function(inta, intb, string):
    ...
```
Working test example

• DUT: calculates mean value of bus_width inputs
• Verification requirement: check all possible data combinations on first and last input
• Random data order, random data on other inputs
class StreamTransaction(Randomized):
    """
    randomized transaction
    """
    def __init__(self, bus_width, data_width):
        Randomized.__init__(self)
        self.bus_width = bus_width
        self.data_width = data_width
        self.data = ()

        list_data = range(0, 2**data_width)

        combs = list(itertools.product(list_data, repeat=bus_width))

        self.addRand("data", combs)

    def mean_value(self):
        return sum(self.data) // self.bus_width
#functional coverage - check if all possible data values were sampled at first and last input

```python
@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):
    """
    We need this sampling function inside the class function, as transaction object needs to exist (required for bins creation). If not needed, just "send" could be decorated.
    """
    pass

sample_coverage(transaction)
```
@cocotb.test()
def mean_mdv_test(dut):
    """ Test using functional coverage measurements and
    Constrained-Random mechanisms. Generates random transactions
    until coverage defined in Driver reaches 100% """

dut_out = StreamBusMonitor(dut, "o", dut.clk)
dut_in = StreamBusDriver(dut, "i", dut.clk)

exp_out = []

scoreboard = Scoreboard(dut)
scoreboard.add_interface(dut_out, exp_out)

data_width = int(dut.DATA_WIDTH.value)
bus_width = int(dut.BUS_WIDTH.value)

cocotb.fork(clock_gen(dut.clk, period=clock_period))
dut.rst <= 1
for i in range(bus_width):
    dut.i_data[i] = 0
dut.i_valid <= 0

yield RisingEdge(dut.clk)
yield RisingEdge(dut.clk)
dut.rst <= 0

coverage1_hits = []
coverageN_hits = []

#define a constraint function, which prevents
#from picking already covered data
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)
while coverage < 100:
    #randomize with constraint
    if not "top.data1" in coverage_db:
        xaction.randomize()
    else:
        coverage1_new_bins = coverage_db["top.data1"].new_hits
        coverageN_new_bins = coverage_db["top.dataN"].new_hits
        coverage1_hits.extend(coverage1_new_bins)
        coverageN_hits.extend(coverageN_new_bins)
xaction.randomize_with(data_constraint)

    yield dut_in.send(xaction)
    exp_out.append(xaction.mean_value())

coverage = coverage_db["top"].coverage*100/
            coverage_db["top"].size

dut._log.info("Current Coverage = %d %", coverage)
Summary

- Verification is software engineering!
- SystemVerilog/UVM-based implementation is not efficient for complex programming tasks
- Cocotb may be an alternative for expensive simulators
  - Less code
  - Fast ramp-up

Cocotb with presented extensions is available online: https://github.com/mciepluc/cocotb
Thank you!
New Constrained Random and Metric-Driven Verification Methodology using Python

Functional Verification is Software Engineering

Marek Ciepłucha and Witold Pleskacz

Warsaw University of Technology
Institute of Micro- and Optoelectronics