New Architecture of the Object-Oriented Functional Coverage Mechanism for Digital Verification

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

***Abstract*—Functional Coverage is a mechanism used in digital integrated circuits functional verification to measure whether the executed test set covered the declared functionality. It helps to examine test scenarios, by providing metrics that give information about the testcase or design-under-test (DUT) reached states (coverage points). SystemVerilog language provides dedicated syntax to make it possible, such as covergroup and coverpoint objects. However, these features have very limited functionality. It is difficult to manipulate functional coverage data on the testbench level, e.g. to make the test scenario dependent on the real-time coverage metrics. It is also not clear how to define coverage objects for complex design features.**

The architecture of a new, object-oriented functional coverage mechanism for digital verification, implemented in Python, is proposed in this paper. The testbench is based on the Cocotb open verification framework. The implemented solution gives more flexibility than standard SystemVerilog syntax and enables more agile creation of verification environments.

I. Introduction

Functional (or dynamic) verification is one of the major tasks during the digital integrated circuits design. 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 behaviour. Today, a constrained random verification technique is preferred over directed testing. Its principle is to generate a test with random stimuli, which makes the test scenario different at each execution. This allows for significant reduction of the number of required independent test scenarios to cover a specific functionality. However, some metric is required to check whether the test indeed exercised the expected function.

As an example, let us consider a test which checks a packet transmission correctness with different packet sizes. If directed tests were used, we would need several scenarios with predefined packet size. Our random test would be a simple data transmission with the packet size defined as a random variable, different at each test execution. If we run our test several times, we would get some of the possible packet sizes covered. We usually focus on corner cases, e.g. maximum, minimum sizes or implementation-specific difficulties. Therefore a "feedback" is required to check whether the executed amount of tests really verified what was expected. It is possible with the functional coverage mechanism. Tracking functional coverage may be done using verification plans. Modern EDA tools make this process easier and allow to automate verification plan implementation and binding it to the functional coverage results. This is described in more detail in Sec. II.

SystemVerilog language provides special constructs for collecting functional coverage. The principal idea is to sample some signal or variables at a specific time and check whether they belong to the pre-defined "coverage bins". Verification engineer defines these bins and sampling conditions and at the end of the simulation gets the detailed coverage result. It is also possible to cover signal sequences instead of values using the syntax adopted from SystemVerilog Assertions (SVA). Functional coverage data can be helpful for improving the efficiency of test stimuli generation. This aspect and related work in the functional coverage alternative implementations are discussed in Sec. III. SystemVerilog coverage features are described in Sec. IV.

In this paper, a new, object-oriented functional coverage mechanism is proposed. The main goal of this implementation was to provide a solution, which allows to easily use functional coverage collecting functionality known from SystemVerilog. Moreover, some new ideas (such as function-based bins matching or coverage callbacks) are introduced, which are difficult or not possible to put into operation using SystemVerilog. The designed mechanism was implemented using Python language and is based on Cocotb verification framework.

Cocotb [1] is an open-source based testbench which enables an RTL design verification with arbitrary simulator engine. The role of the external simulator is limited only to the logic-level simulation and all transactions are translated to Python using the VPI (Verilog Procedural Interface). It means that all stimuli generation and signals monitoring is done in Python, so no additional Verilog or SystemVerilog testbench is needed at all. The Python language has many advantages over SystemVerilog and is considered to more effective for complex programs development. 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. Additionally, Python provides a readable decorator syntax. Decorators can be used in order to extend (decorate) the function that is already implemented without function modifications [3]. In the proposed architecture, the testbench functions (such as send/receive transactions) are decorated with functional coverage objects. Features and examples of the implemented architecture are discussed in Sec. V.

978-1-5090-1141-4/16/$31.00

II. Verification Planning and Completeness

The idea of regression testing [4] can be described as a process to fully verify the DUT against a verification (or test) plan. It means that any functional change in the design should be followed by running the full regression test set to make sure no new issues were unintentionally introduced. The number of tests executed per regression run depends on coverage metrics definition. When DUT is big and coverage definition contains many corner cases, the regression testing may be long, as satisfying coverage metrics requires many repetitive test runs. Thanks to random tests, a single test scenario can be run several times with different *seeds*, which leads to overall coverage improvement. The regression process itself does not depend on coverage result. Collected coverage is only a process output, monitoring the verification progress.

Verification planning [4], [5] is a process of defining a list of features that need to be tested during the functional verification. This list corresponds to the implemented elements of the verification environment e.g. test scenarios, functional coverage points or scoreboard checks. After execution of a full regression test set, the coverage results can be mapped to the verification plan to check the verification completeness. This process is automated by commercial EDA tools [6], [7]. The presented approach is called a metric-driven verification (MDV) and is shown in Fig. 1.

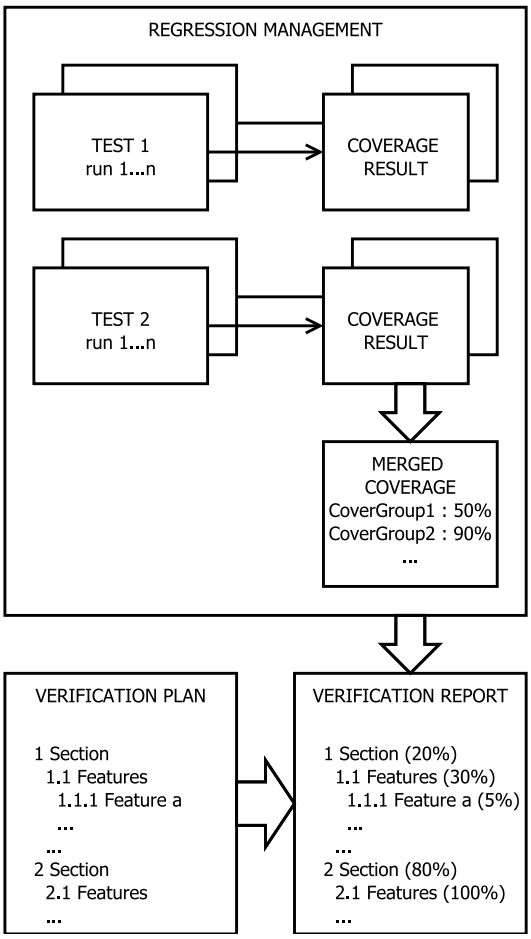

Fig. 1: Metric-driven verification flow

The verification plan structure is organised in various sections. Some of them correspond to the specification-based features (e.g. modes of operation, supported data structures) or implementation-specific features (e.g. performance, logic interfaces behaviour). Mapping plan items to implemented verification environment elements has to be done manually. This process may be confusing as it is not clear how to define a feature (or extract it from the specification) in the plan in a readable way. Moreover, it may be ambiguous how the feature will be validated in the verification environment. SystemVerilog functional coverage syntax (discussed in Sec. IV) is suitable for features which are quantifiable, for example "packet size may be 1 to 100 bytes" or "mode X may be enabled or not". If the verification plan item describes a complex dependency or functionality, binding several environment elements to a single feature may be required, which increases the risk of making a mistake.

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 (seeds)? By default, MDV automation tools would append only coverage results from the successful test run. Should the appended metrics be considered valid when it was proved that the test scenario may sometimes lead to the DUT malfunction? It may depend on the project assumptions. Similar problems appear when the regression management is done independently from the verification environment itself. 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.

III. Related work and motivation for the project

The related work on functional coverage improvements mainly focuses on the effective stimuli generation, which allows the testbench to satisfy the coverage metrics faster than in the case of straightforward executing random tests. There is a "coverage-directed test generation" idea introduced in [8]. This solution has been later expanded with the use of evolutionary algorithms (i.e. [9], [10]). The main principle of a coverage-directed test generation is to observe how the test stimuli result in covering specific points. Using a feedback from the coverage result, new test constraints are generated. No information about the DUT itself is processed during new tests generation. A different idea is proposed in [11]. The covergroups are being "synthesized" to the logic level of abstraction. Afterwards, a boolean satisfiability (SAT) solver, analysing data from DUT with synthesized covergroups and target coverage, helps to find test patterns to hit missing

coverage points. All these solutions assume that an engine improving the verification closure process is placed outside the testbench (e.g. C++ library in [9]).

Another group of the related publications are implementations of the coverage mechanism in SystemC, as this language itself does not contain any coverage collecting features. A basic example is proposed in [12]. The coverage monitor was implemented to observe the variables and their dependencies, however its functionality is very limited when compared to SystemVerilog costructs. A comprehensive implementation of SystemVerilog coverage features is presented in [13] and later extended in [14] and [15]. In [15] an Aspect-oriented programming enhancements of the C++ are used to implement coverage features. This is an elegant way to extend SystemC-based verification testbenches and make the coverage implementation similar to SystemVerilog. This paper also presents an idea of a verification flow based on a feedback from coverage metrics. Another recent proposal is described in [16] – an extension of the UVM-SystemC library.

A coverage-driven distribution (CDD) idea is presented in [17]. Its principle is to adjust the test randomization constraints automatically according to the coverage bins distribution. However, a direct relationship between constraints and coverage is assumed, which is a requirement unlikely to be satisfied in complex testbenches. Nevertheless, this idea may be useful for e.g. covering bus drivers functionality. One of the most advanced recent works is described in [18]. In this paper several enhancements of SystemC verification libraries are presented, considering the new C++11 standard. An extension of functional coverage features considering definition of a logical expression as a coverage bin has also been proposed. In SystemVerilog only a set of values or transitions can be defined as a bin.

The main observation from the cited papers is that the functional coverage features are widely used, but the SystemVerilog mechanism is not sufficient. Many projects consider a dependence between real-time coverage results and stimuli generation. This is technically possible with SystemVerilog syntax but with very limited scope. SystemC-based functional coverage implementations in general follow the SystemVerilog concepts and provide some extra features. However, their main limitation is operating at the low level of abstraction, such as signals, variables or sequences.

The work presented in this paper introduces a new functional coverage mechanism which can be used in a SystemVerilog style, but also allows designers to create complex structures. This mechanism itself is a class-based implementation, therefore it can be extended by users to create new coverage primitives. As the coverage objects become fully-featured testbench elements, it is more simple to manipulate them in real time. It is also easier to bind them to the verification plan items, as the coverage objects may be defined at different levels of abstraction. The main motivation for this work was to provide a scalable solution which is not only convenient to use for simple tasks, but also helps to build comprehensive coverage-driven verification environments.

IV. Functional Coverage in SystemVerilog

In SystemVerilog [19] a fundamental coverage unit is a coverpoint. Coverpoint 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 checked whether it matches any defined bin. If yes, then the number of hits of the particular bin is incremented. By default, at the end of test (or tests) the bin is considered covered if its number of hits is higher than zero. The overall coverage level of the coverpoint is therefore the number of hit bins divided by the number of defined bins.

Coverpoints are organised in covergroups, which are specific class-like structures. After defining a covergroup we need to create its instance. 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.

An example covergroup *packet* with single coverpoint *packet_size* is presented in the listing below.

```
covergroup packet @ (posedge clk);
  packet_size : coverpoint pkt_size {
    bins small    = {[1:19]};
    bins medium   = {[20:49]};
    bins max      = {50};
  }
endgroup
```

A bin is a unit that contains a set of values that need to be matched to a variable being covered. A bin may contain:

- an implicit value(s) (e.g. numeric, bit vector, enum type),
- a wildcard value(s) (e.g. “1?” meaning “10” and “11” satisfy the matching),
- a numeric value range (e.g. [1:10]),
- a sequence (e.g. “1=>2” meaning “1” and “2” were sampled consecutively).

SVA 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.

In the example presented below, a cross is defined as all possible combinations of the bins from coverpoints *transfer_direction*, *transfer_length* and *transfer_type*. This gives 12 possible bin combinations overall. All cross-bins with coverpoint *transfer_length* matched to bin “1” are ignored. Finally, the defined *transfer_cross* item contains 8 cross-bins.

```
covergroup transfer @(posedge clk);
  transfer_direction : coverpoint dir {
    bins read       = {0};
    bins write      = {1};
  }
  transfer_length : coverpoint length {
    bins single     = {1};
    bins short      = {[2:10]};
    bins long       = {[10:100]};
  }
  transfer_type : coverpoint type {
    bins type_a     = {A};
    bins type_b     = {B};
  }
  transfer_cross : cross
    transfer_direction, transfer_length,
    transfer_type {
    ignore_bins ign =
      binsof(transfer_length) intersect {1};
  }
endgroup
```

In SystemVerilog 2012 a feature allowing the user to define a function returning a set of ignore bins for cross item has been added to be used instead of "binsof ... intersect" construct. The function should return a "CrossQueueType" list of bins to be ignored. This controls the cross-bins definition in a more legible way.

Coverage objects in SystemVerilog may have additional attributes. The most important are:

- *weight* – specifies a particular covergroup or coverpoint coverage impact on the overall coverage level by setting a value different than 1 (default),
- *at_least* – specifies how many times a bin in a coverpoint needs to be hit to be covered (by default 1),
- *per_instance* – when true, each instance of the covergroup coverage is tracked independently (by default 0).

It is possible to get the coverage level of a particular covergroup in real time using the *get_coverage()* function. Also a system function *$get_coverage()* is defined that gets the overall coverage level of all created covergroups.

The functional coverage features are also implemented in e language [20]. 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 – covergroups cannot contain other covergroups, which would better correspond to a verification plan scheme,
- not possible to get the detailed coverage information in real time (e.g. when a specific bin was hit).

V. Implemented Functional Coverage Mechanism

This section describes in detail the syntax and features of the implemented functional coverage mechanism. Based on the discussion from the previous sections, the general assumptions for the proposed architecture 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, they can be created inside other functions) this is a convenient and compact solution.

A. Coverage structure

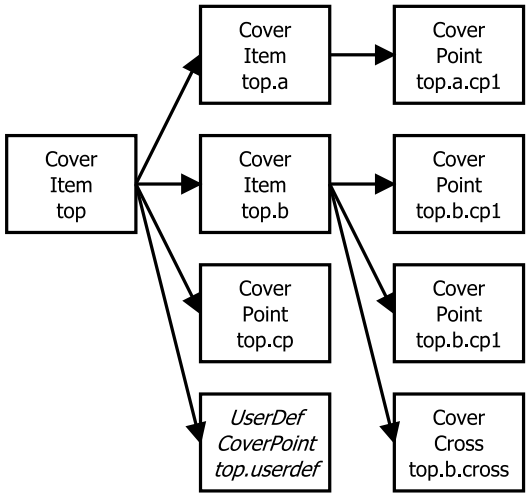

Fig. 2: An example of the coverage trie structure

The implemented coverage structure is based on a prefix tree (a trie). The main coverage primitive is a CoverItem, which roughly corresponds to a SystemVerilog covergroup. CoverItem may contain more 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.

B. CoverPoint

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, bins matching condition can be described by a formula:

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

Additional attributes of a CoverPoint are:

- *weight* – corresponds to SystemVerilog *weight* function (by default equal to 1),
- *at_least* – corresponds to SystemVerilog *at_least* function (by default equal to 1),
- *inj* – defines whether a bin matching relation is injective (more than one bin can satisfy the matching).

In the example below, the *decorated_function()* has several CoverPoints which are associated with the function arguments *inta*, *intb* and *string*.

```
simple_bins = [] #bins generation for coverage.simple
for i in range (1, 5):
  for j in range (1, 5):
    simple_bins.extend([(i,j,'y'), (i,j,'n')])

#transformation function for coverage.transformation
def transformation(inta, intb, string):
  ...
  return HashableObject(inta, intb, string)

#transition function for coverage.transition
prev_value = 0;
def transition_inta(inta, intb, string):
  transition = (prev_value, inta)
  prev_value = inta
  return transition

@CoverPoint( "coverage.simple",
  bins = simple_bins
  )
@CoverPoint( "coverage.transformation",
  f = transformation, bins = [H1, H2, H3]
  )
@CoverPoint( "coverage.relation",
  f = lambda inta, intb, string : inta,
  rel = lambda x, y : x < y, bins = [1, 5, 10, 50]
  )
@CoverPoint( "coverage.transition",
  f = transition_inta,  bins = [(1,2), (2,3), (3,4)]
  )
@CoverPoint( "coverage.primefactors",
  f = lambda inta, intb, string : inta,
  rel = has_prime_factor,  inj = True,
  bins = [2, 3, 5, 7, 11, 13, 17]
  )
def decorated_function(inta, intb, string):
  ...
```

The *coverage.simple* CoverPoint checks whether arguments match predefined bins listed in the *simple_bins* container. 50 bins are generated to fill the list up. As the *decorated_function()* takes 3 arguments, each bin has a form of a tuple (*int*, *int*, *string*).

The *coverage.transformation* CoverPoint performs argument transformation defined by the *transformation()* function and returns some new object. Bins are some constants of the same type as the *transformation()* function result.

In the *coverage.relation* CoverPoint transformation is used only to take a single *inta* argument for further matching analysis. The relation is defined as a "<" operator, which means a bin will be matched when the sampled *inta* value is less than the bin value. Bin definition order is important as matching will be analysed from the lowest index element of the bin list. In this CoverPoint we can see a lambda expression, which is a simple way of defining one-line functions in Python.

The *coverage.transition* CoverPoint 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* CoverPoint defines the 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".

C. CoverCross

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. It is also possible to set a *weight* and *at_least* attributes the same way as in CoverPoint. 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.

In the example below, two CoverPoints are defined and each contains 10 bins. CoverCross *coverage.cross* is implemented as a Cartesian product of these CoverPoints. Ignore bins list is defined as a list containing all items with *coverage.coverpoint1* matched to bin "1" and all items with *coverage.coverpoint2* matched to bin "10". As a result, *coverage.cross* contains 81 bins.

```
@CoverPoint( "coverage.coverpoint1",
  f = lambda inta, intb : inta, bins = range(1,10)
  )
@CoverPoint( "coverage.coverpoint2",
  f = lambda inta, intb : intb, bins = range(1,10)
  )
@CoverCross( "coverage.cross", items =
  ["coverage.coverpoint1", "coverage.coverpoint2"]
  ign_bins = [(1, None), (None, 10)]
  )
def decorated_function(inta, intb):
  ...
```

D. Using coverage objects during test execution

It is possible to read a coverage level in real time from any existing CoverItem. For example, *coverage_db["a.b.c"].coverage* returns a coverage level of the *a.b.c* CoverItem, which applies to all the children of node *a.b.c* in the coverage trie.

It is also possible to add a callback function to any CoverItem. Two types of callbacks are defined:

- threshold callback – called when coverage level of the CoverItem exceeds a given value,
- bins callback – called when a specified bin was hit in the CoverPoint or CoverCross.

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, the callback function will be called automatically.

VI. Experimental Results and Conclusion

The Cocotb-based verification environment with presented functional coverage additions has been implemented in order to verify the Bluetooth Low Energy controller (RTL code). The main purpose was to use external, high level sequences (driver-based) and to check the coverage monitored on a physical Bluetooth interface (packet exchange). Previously, a comprehensive UVM-based verification flow has been conducted for the same design [21].

The testbench written in Python showed several advantages over existing verification environment. First of all, the productivity of the implementation process is greater, as the Python syntax is more intuitive and consistent. It is also easier to manage testbench features at a higher level of abstraction. Complex transactions monitoring, such as Bluetooth-specific packet sequences, is more readable and concise. The proposed decorator-based approach can be adopted for any functions (e.g. scoreboard checks), therefore coverage may be easily added anywhere in the testbench. The environment was tested using Cadence Incisive, Mentor Graphics Questa and Synopsys VCS. No performance issues related to communication between Python and simulator engine (based on VPI) have been observed, because simulating the RTL design is much more complex task than high-level verification environment routines.

Implementing coverage items in the testbench allows for easy extraction of the coverage tree structure. Therefore, a verification plan can be automatically created based on implemented coverage. When new coverage objects are added, there is no need to manually modify the plan.

The verification strategy based on Cocotb framework with presented functional coverage features showed many advantages over SystemVerilog-based methodologies. The described mechanism allows for easy setup of MDV environments using Python. With the presented approach, the digital verification tasks are shifted more in the direction of software development issues, which eventually leads to the improvement of overall verification process agility. The effectiveness of the verification environment creation remains one of the major concerns of the digital integrated circuit design flow.

The described functional coverage framework is available as an open source.

Acknowledgment

This work was supported in part by the Polish National Centre for Research and Development under project No. DOBR/0053/R/ID1/2013/03.

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] *The Python Language Reference*, Python Software Foundation, Oct. 2013, release 2.6. [Online]. Available: https://docs.python.org/2.6/reference/

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

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

[6] "Cadence Redefines Verification Planning and Management with Incisive vManager Solution," Cadence Design Systems, Inc., 2014. [Online]. Available: http://www.cadence.com/cadence/newsroom/press_releases/Pages/pr.aspx?xml=022414_vmanager

[7] "Functional Verification Choice of Leading SoC Design Teams," Synopsys, Inc., 2015. [Online]. Available: http://www.synopsys.com/Tools/Verification/FunctionalVerification/Pages/VCS.aspx

[8] 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.

[9] 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.

[10] A. M. Cruz, R. B. Fernandez, and H. M. Lozano, "Automated functional coverage for a digital system based on a binary differential evolution algorithm," in *BRICS Congress on Computational Intelligence and 11th Brazilian Congress on Computational Intelligence (BRICS-CCI and CBIC)*. IEEE, Sep. 2013, pp. 92–97.

[11] A.-C. Cheng, C.-C. Yen, and J.-Y. Jou, "A formal method to improve SystemVerilog functional coverage," in *IEEE International High Level Design Validation and Test Workshop (HLDVT)*. IEEE, Nov. 2012, pp. 56–63.

[12] S. Park and S.-I. Chae, "A C/C++-based functional verification framework using the SystemC verification library," in *The 16th IEEE International Workshop on Rapid System Prototyping (RSP)*. IEEE, Jun. 2005, pp. 237–239.

[13] G. Defo, C. Kuznik, and W. Mueller, "Verification of a CAN bus model in SystemC with functional coverage," in *International Symposium on Industrial Embedded Systems (SIES)*. IEEE, Jul. 2010, pp. 28–35.

[14] 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.

[15] C. Kuznik and W. Mueller, "Aspect enhanced functional coverage driven verification in the SystemC HDVL," in *International SoC Design Conference (ISOCC)*. IEEE, Nov. 2011, pp. 154–157.

[16] 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.

[17] 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.

[18] 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.

[19] *1800-2012 - IEEE Standard for SystemVerilog – Unified Hardware Design, Specification, and Verification Language*, IEEE Std. 10.1109/IEEESTD.2013.6 469 140, Feb. 2013.

[20] *1647-2011 - IEEE Standard for the Functional Verification Language e*, IEEE Std. 10.1109/IEEESTD.2011.6 006 495, Aug. 2011.

[21] M. Moskala, P. Kloczko, M. Cieplucha, and W. Pleskacz, "UVM-based verification of Bluetooth Low Energy controller," in *Design and Diagnostics of Electronic Circuits Systems (DDECS), 2015 IEEE 18th International Symposium on*. IEEE, Apr. 2015, pp. 123–124.