2024 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE
MUNICH, GERMANY
OCTOBER 15-16, 2024

Scalable and mergeable functional coverage flow for highly configurable IP signoff and specific customer deliveries

Fryderyk Kozioł, Cadence Design Systems, Warsaw, Poland (*fkoziol@cadence.com*)
Sebastian Cieślak, Cadence Design Systems, Warsaw, Poland (*scieslak@cadence.com*)

***Abstract*— Sign-off of highly configurable design IPs via cross-configuration merging has considerable challenges with proving full functional coverage for a single chosen configuration. Scripting, templating, and refining require a lot of additional effort. We present an approach to SystemVerilog coverage coding that results in a single codebase for both multi and single-configuration functional coverage closure.**

Keywords—functional coverage; configuration; SystemVerilog; UVM; IP signoff; verification

I. Introduction

Highly configurable IP design is paramount in the current industry state. Customers require many configuration possibilities in their SoC systems, which creates a large configuration space for sign-off on the IP provider side. Proving full functional and code coverage for the Metric Driven Verification [1] approach has become an increasingly difficult challenge.

RTL code that is templated or based on defines and includes is hard to manage for designers as each configuration set is fundamentally a separate design – this does not allow code coverage merging, and each configuration requires separate regressions, linting, and synthesis trials. On the other hand, parameter-based configuration allows cross-configuration code coverage merging but still requires separate snapshots and regression runs for configuration sets. A third approach allowing for single snapshots and elaborations is wire-based configuration in which features are removed or "strapped away" during design synthesis [2][3].

These configuration approaches have different trade-offs in regard to effort needed during design IP sign-off. The traditional approach to functional coverage coding either has bins related to values that should be hit in different configuration sets (configuration "superset") or has bins that are tailored to a specific configuration [2][3]. Keeping a single codebase for functional coverage would be highly beneficial for IP providers.

II. New Approach

A. Configurable RTL design

Configurable features can be removed during synthesis based on constant configuration wire values or during elaboration via parameters. In both cases, the RTL code is identical throughout IP development, allowing for code coverage merging and easing the refinment effort of designers.

In a typical multi-configuration approach, to prove full functional coverage for a single configuration, the bins that are not hit for a selected configuration must be refined manually or with scripts. This is tedious, prone to error, and not scalable, as any implementation change or new configuration set requires a review of existing coverage. Every configuration requires different subsets of bins to be present, so the SystemVerilog codebase is different.

Writing functional coverage with a focus on gathering numbers per single configuration allows the use of the same parameters that are used to configure the device under testing. This way, each coverage model will be finely tuned to the selected configuration, eliminating the need to refine. However, this type of solution makes it

impossible to merge the functional coverage and therefore requires collecting and analyzing reports per config, which makes highly configurable IP verification practically impossible due to the large configuration space.

```
module rtl_example #(parameter DATAPATH_WD = 1024)
  (input  wire                    strap_is_addr_64b,
   input  wire [63:0]             addr_in,
   output wire [63:0]             addr_out,
   input  wire [DATAPATH_WD-1:0] data_in,
   output wire [DATAPATH_WD-1:0] data_out);

  assign addr_out = (strap_is_addr_64b) ? ~addr_in : {32'b0, ~addr_in[31:0]};
  assign data_out = ~data_in;
endmodule : rtl_example
```

Figure 1. Example of RTL code configured by both wires and parameters

A wire-based configuration is beneficial for lesser gate count features as it minimizes the number of parameters and, therefore, required snapshots per regression. However, removing constant values and logic related to them might be a computationally intensive task for synthesis tools, so greater features should be parametrized. The presented approach works for both wire-based and parameter-based, and their mix in a single codebase (Figure 1), allowing for different design trade-offs and approaches.

This paper presents an approach to SystemVerilog coverage coding that is scalable, self-refinable, and contained exclusively in the verification environment with no additional tools or scripting required. This coding style allows for a single verification plan (vPlan [1]), without any configuration-related manually added attributes, in both single and multi config sign-off flows. The coverage database to vPlan item mapping stays the same irrespective of the selected flow, drastically reducing reporting efforts.

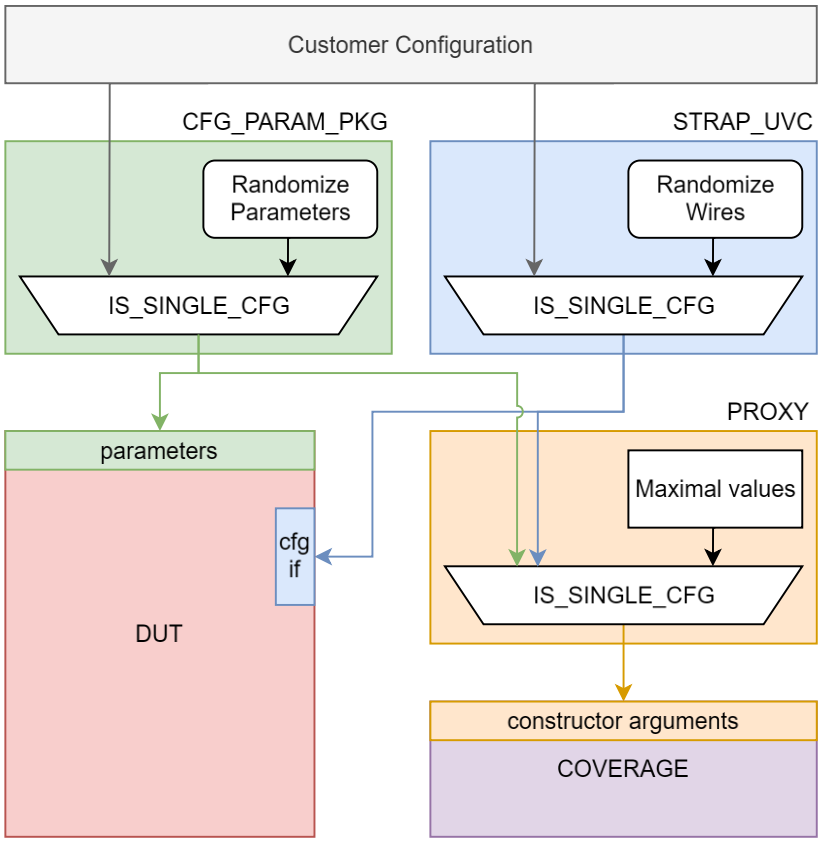

Figure 2. Connections of configuration components

B. *Proxy Class*

A regression-wide and seed-independent config object class used for coverage generation is proposed to make cross-seed functional coverage group merging possible. Depending on the regression requirements, this class contains the maximum possible configuration values from a superset specified by the IP provider or the values of a specific customer configuration. The proxy class object values are passed to cover groups as constructor arguments, which is a standard approach [3][4]. As the values are constant across all tests in the regression, the coverage model generated by each testrun is the same, which solves functional coverage merging issues [3].

Apart from the IP configuration-related values, the proxy class contains a novel is_single_config flag that distinguishes between sign-off flows. This flag is also passed to the cover groups to control bin filtering, mostly done by the SystemVerilog *with* statement. For some projects, splitting this flag into two might be useful: one for parameters and one for wire-based configuration options, which allows for sign-off of all wire-based configurations for a single set of parameters. Cover groups are always created to keep their vPlan mapping intact.

```
class cov_proxy_singleton;
  static cov_proxy_singleton proxy_obj;
  static cfg_strap_uvc       strap_uvc;

  local static bit is_single_config;

  local static int min_datapath_wd; // parameter based
  local static int max_datapath_wd; // parameter based
  local static bit is_atomic_supp;  // parameter based
  local static bit is_addr_64b;     // wire based
  local static int max_outstanding; // wire based

  function new();
    is_single_config = `ifdef SINGLE_CFG 1 `else 0 `endif ;
    if (is_single_config && strap_uvc == null) strap_uvc = new();

    min_datapath_wd = (is_single_config) ? cfg_param_pkg::DATAPATH_WD    : 128;
    max_datapath_wd = (is_single_config) ? cfg_param_pkg::DATAPATH_WD    : 1024;
    is_atomic_supp  = (is_single_config) ? cfg_param_pkg::IS_ATOMIC_SUPP : 1'b1;
    is_addr_64b     = (is_single_config) ? strap_uvc.is_addr_64b         : 1'b1;
    max_outstanding = (is_single_config) ? strap_uvc.max_outstanding     : 4;
    // [...]
  endfunction : new

  static function cov_proxy_singleton get_obj();
    if (proxy_obj == null) proxy_obj = new();
    return proxy_obj;
  endfunction : get_obj

  static function bit get_is_single_config(); return is_single_config; endfunction
  static function int get_min_datapath_wd();  return min_datapath_wd;  endfunction
  static function int get_max_datapath_wd();  return max_datapath_wd;  endfunction
  static function bit get_is_atomic_supp();   return is_atomic_supp;   endfunction
  static function bit get_is_addr_64b();      return is_addr_64b;      endfunction
  static function int get_max_outstanding();  return max_outstanding;  endfunction
endclass : cov_proxy_singleton
```

Figure 3. Example of proxy class written as singleton

The proxy class might be implemented in two ways to ensure consistency across all tests by using constant variables (*const*) or local static variables (*local static*). As the values must be the same in each testbench component,

it is safe to use a singleton design pattern that forces only one instance of the proxy class in the whole verification environment. If the coverage model is expected to be instanced at a higher level or multiple instances will be required, e.g., many instances of a single, differently configured IP on an SoC level, then const values should be used instead.

C. Coverage Coding Style

The proposed functional coverage coding style relies heavily on the *with* and *iff* SystemVerilog constructs, expanding on the work outlined in [3]. The first allows fine filtering of coverage *bins* values and adds options for more sophisticated logic in coverage creation. Run-time-based configuration bounded sampling can then be done with the *iff* statement. Additionally, these constructs help to encapsulate all coverage-related calculations directly into the cover groups, which improves reusability. There is no need to prepare supplementary constructor arguments before creating the cover group.

Controlling the cover group shape without proprietary scripts or templates is beneficial because all coverage-related code is encompassed in the SystemVerilog language, which is well-known and popular across design verification engineers. Furthermore, syntax checking and linting for the correct coverage coding are available, and the constructs are part of a widely propagated Language Reference Manual (LRM [5]). This is an improvement over scripting, which is widely used for configurable cover groups, as in [2][6].

To keep the scalability and mergeablity, several rules have to be fulfilled:

- The creation of cover point *bins* should be based on the proxy values or constant values as the coverage model needs to be the same for all simulations.
- It is a good practice to use the *pxy_* prefix for cover group constructor arguments that will be set to proxy values. This will help to distinguish them from a run-time configuration that may be sampled.
- Usage of the *with* clause for *bins* with a wide range of values may significantly increase the cover group construction time and even crash the simulator as the expression is evaluated for every value. It is better to use a cross with an enabling coverpoint for such coverage. Consider the *addr_64b_cx* item from the Figure 4 example, if the *with* clause was used directly in the *addr_64b_cp*, it would cause 2^{64} expression evaluations. Cross-coverage eliminates this problem by moving calculations to a simpler *cfg_is_addr_64b_cp* item.
- Cover groups should be created regardless of configuration to keep mapping between the coverage database and verification plan intact.
- If some cover points are not applicable for a single configuration sign-off, all their bins can be removed by using *with* expressions. It is convenient to do this by crossing the cover point with an enabling cover point. This approach will create an empty cover item that will still be correctly mapped to the verification plan but with zero bins. The *addr_64b_cx* from Figure 4 is an example of such a case.
- The *iff* expression can be used to associate a sampled value with the currently running test-case configuration. The *max_outstanding_per_cfg_hit_cp* from Figure 4 utilizes this structure to cover that the maximal number of packets was sent for each configuration.
- *Cross* coverage tweaking can be done by *ignore_bins* with *binsof* and *intersect* expressions. Do not use the *illegal_bins* feature as cover groups should be used only for collecting functional coverage and not error reporting.

```
covergroup example_cg (bit pxy_is_single_config, int pxy_min_datapath_wd, int pxy_max_datapath_wd,
                       bit pxy_is_atomic_supp, bit pxy_is_addr_64b, int pxy_max_outstanding)
  with function sample (int cfg_datapath_wd, bit cfg_is_atomic_supp, bit cfg_is_addr_64b,
                        int cfg_max_outstandig, bit [1:0] atomic_type, bit [63:0] addr,
                        int num_of_pkts);
  option.per_instance = 1;
  option.name = "example_cg";

  cfg_datapath_wd_cp : coverpoint cfg_datapath_wd {
    bins datapath_wd [] = {[pxy_min_datapath_wd:pxy_max_datapath_wd]} with ($onehot(item));
  }

  atomic_type_cp : coverpoint atomic_type {
    bins non_atomic = {2'b00};
    bins store      = {2'b01}
      with ((pxy_is_single_config && !pxy_is_atomic_supp) ? (item == 2'd0) : (item > 2'd0));
    bins load       = {2'b10}
      with ((pxy_is_single_config && !pxy_is_atomic_supp) ? (item == 2'd0) : (item > 2'd0));
  }

  cfg_is_addr_64b_cp : coverpoint cfg_is_addr_64b {
    bins is_addr_64b [] = {0,1} with ((pxy_is_single_config) ? (item == pxy_is_addr_64b) :
                                                               (item <= pxy_is_addr_64b));
  }
  addr_32b_cp : coverpoint addr {
    bins min_32b_addr    = { 32'h0000_0000};
    bins med_32b_addr[2] = {[32'h0000_0001 : 32'hFFFF_FFFE]};
    bins max_32b_addr    = {                 32'hFFFF_FFFF };
  }
  addr_64b_cp : coverpoint addr {
    bins min_64b_addr    = { 64'h0000_0000_0000_0000                          };
    bins med_64b_addr[2] = {[64'h0000_0000_0000_0001 : 64'hFFFF_FFFF_FFFF_FFFE]};
    bins max_64b_addr    = {                           64'hFFFF_FFFF_FFFF_FFFF };
  }
  addr_32b_cx : cross cfg_is_addr_64b_cp, addr_32b_cp {
    ignore_bins addr_64b = binsof(cfg_is_addr_64b_cp) intersect {1};
  }
  addr_64b_cx : cross cfg_is_addr_64b_cp, addr_64b_cp {
    ignore_bins addr_32b = binsof(cfg_is_addr_64b_cp) intersect {0};
  }

  max_outstanding_per_cfg_hit_cp : coverpoint num_of_pkts
                                   iff (num_of_pkts == cfg_max_outstanding) {
    bins num_of_pkts [] = {[1:pxy_max_outstanding]}
                          with ((pxy_is_single_config) ? (item == pxy_max_outstanding) :
                                                         ($onehot(item)))              ;
  }

endgroup : example_cg
```

Figure 4. Example of a cover group that is configured by proxy values

D. Verification Plan and Reporting

One of the first steps of a metric-driven verification [1] approach is preparing a verification plan that will contain information about all necessary checks, coverage metrics, and tests that must be added to a newly started verification environment. Such documents can be typically created as CSV files, XML files, or in vendor-specific format (e.g. .vPlan or .vPlanx). To allow effective progress tracking, verification plans need to be connected to the coverage database generated by test regressions. When coverage items are mapped to the respective vPlan items, verification reports can be generated to review the current status of verification efforts.

The verification plan format is not standardized across the industry, so due to this, most of the vPlan parametrization and generation methods are done by in-house-built tools and scripts, making the transfer of knowledge and flows difficult [2][6]. Our work minimizes the impact on a once-created vPlan and transfers reconfigurability to a well-known and standardized SystemVerilog [5].

A vPlan should contain all DUT features and "maximal configuration" for development purposes. This superset is mandatory for IP-level sign-off and product development, where all possible design configurations must be tested. However, for a specific customer delivery, the verification report should be refined and concentrated on the chosen configuration. Refining can be done manually or automated with additional tools, but both approaches require working with non-standard formats and flows. The same problem applies when a generic module needs to be used on a higher level of RTL, for example, on the SoC level.

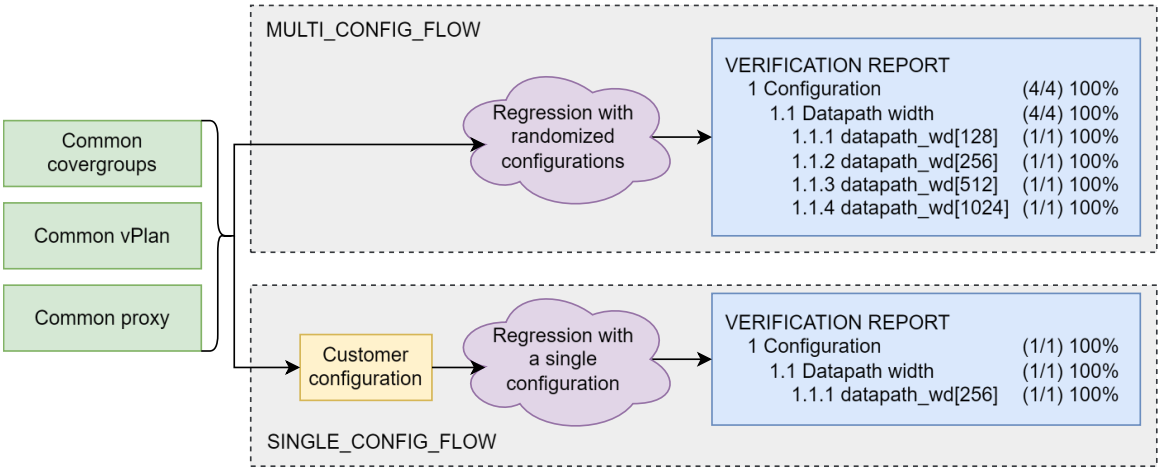

Figure 5. Report generation flows

Using the aforementioned coverage coding style and a coverage-config-proxy class, it is possible to have a completely configuration-agnostic vPlan, in which items are mapped to the same cover points for both the single and multi config flows. The distinction is made at the level of the verification environment by selecting the appropriate flow. As shown in Figure 5, the same common files are used for both flows, and the only difference will be in a verification environment configuration – once randomized and once taken from a separate customer configuration file.

In the case of bins not existing for a specific configuration, the mapping will be consistent, and number of bins will be reduced. For feature-based cover groups, the mappings will stay the same, but the coverpoints will be empty and will not appear unmapped. Such empty items can be left or, if needed, easily hidden by tool filters or by scripting.

Figure 6 shows an example of a verification report for both single and multi config flows. Bold lines correspond to vPlan elements to which cover items are mapped. These lines are the same regardless of the selected flow. The only change is in the number of bins shown under them. In the case of an empty section "3.2 Address 64b values", the vPlan item still exists, but the section is empty.

```
MULTI_CONFIG                                  |  SINGLE_CONFIG:dp256,addr32b,noatomics,2otst
VERIFICATION REPORT              (20/20)      |  VERIFICATION REPORT              (8/8)
  1 Configuration                  (6/6)      |    1 Configuration                (2/2)
    1.1 Datapath width             (4/4)      |      1.1 Datapath width           (1/1)
      1.1.1 datapath_wd[128]       (1/1)      |
      1.1.2 datapath_wd[256]       (1/1)      |        1.1.1 datapath_wd[256]     (1/1)
      1.1.3 datapath_wd[512]       (1/1)      |
      1.1.4 datapath_wd[1024]      (1/1)      |
    1.2 Address width              (2/2)      |      1.2 Address width            (1/1)
      1.2.1 is_addr_64b[0]         (1/1)      |        1.2.1 is_addr_64b[0]       (1/1)
      1.2.2 is_addr_64b[1]         (1/1)      |
  2 Atomic type                    (3/3)      |    2 Atomic type                  (1/1)
    2.1 non_atomic                 (1/1)      |      2.1 non_atomic               (1/1)
    2.2 store                      (1/1)      |
    2.3 load                       (1/1)      |
  3 Address                        (8/8)      |    3 Address                      (4/4)
    3.1 Address 32b values         (4/4)      |      3.1 Address 32b values       (4/4)
      3.1.1 min_32b_addr           (1/1)      |        3.1.1 min_32b_addr         (1/1)
      3.1.2 med_32b_addr[0]        (1/1)      |        3.1.2 med_32b_addr[0]      (1/1)
      3.1.3 med_32b_addr[1]        (1/1)      |        3.1.3 med_32b_addr[1]      (1/1)
      3.1.4 max_32b_addr           (1/1)      |        3.1.4 max_32b_addr         (1/1)
    3.2 Address 64b values         (4/4)      |      3.2 Address 64b values       (0/0)
      3.2.1 min_64b_addr           (1/1)      |
      3.2.2 med_64b_addr[0]        (1/1)      |
      3.2.3 med_64b_addr[1]        (1/1)      |
      3.2.4 max_64b_addr           (1/1)      |
  4 Max outstanding per config     (3/3)      |    4 Max outstanding per config   (1/1)
    4.1 num_of_pkts[1]             (1/1)      |
    4.2 num_of_pkts[2]             (1/1)      |      4.1 num_of_pkts[2]           (1/1)
    4.3 num_of_pkts[y]             (1/1)      |
```

Figure 6. Verification report in both single and multi configuration flow

III. RESULTS

The multi-config flow allows for precise roadmap planning and feature metric analysis during the IP development process. A single coverage report generated from a single regression contains both code and functional coverage results for all possible configurations and features.

For chosen configurations, the single-config flow transparently and automatically generates coverage bins and cover groups tuned to the configuration. The vPlan mappings, reports, and regression flows are identical to the ones created during multi-config IP development and require no extra effort.

The presented approach has been successfully used in a PCIe Controller IP project, which is a highly configurable and complex design. This flow has been used for a few years so that it can be treated as stable and well-tested. Additionally, the presented flow works seamlessly with continuous integration requirements and improves on other works that require custom scripts and tools [2][6].

IV. CONCLUSIONS

The newly proposed coverage coding style and coverage-config-proxy class allow for effective IP development regardless of the configuration space size. Scripts and templating approaches are not transferrable from project to project and put more burden on design and verification engineers while additionally requiring interworking between tools in daily regression flows. Using only SystemVerilog, the coverage is entirely encompassed within the environment.

The proposed code style may increase the complexity of cover groups compared with typically written coverage and thus slow down its writing. However, code written in this way is significantly more scalable. It removes the need for manual report refining, allows for a single report tracking during development, and eliminates the need for additional scripting, ultimately saving engineering time. This scalability for customer configurations is crucial when the project is no longer in the development phase, as it can shorten the time needed to prepare coverage reports for future deliveries.

The presented flow can be used in a wire-based configurable design, a traditionally parametrizable RTL, and a combination of both. As this solution is based on pure SystemVerilog, it can be used in all SV testbenches regardless of the methodology used (pure SV, UVM, OVM, etc.). Verification environments written in other languages, for example, Python (cocotb) or VHDL, can utilize mixed language simulations to include coverage-related SystemVerilog code prepared as described in this paper's guidelines. If this type of simulation is not desired, the concept of a proxy class with maximal values can be easily transferred to other languages.

REFERENCES

[1] N. Heaton, “Maximizing Verification Effectiveness Using MDV,” Cadence Design Systems, 2014.
[2] J. Ridgeway, “Managing Highly Configurable Design and Verification,” Design and Verification Conference (DVCON), San Jose, 2018.
[3] C. Lovett and M. Horn, “Relieving the Parameterized Coverage Headache,” Design and Verification Conference (DVCON), San Jose, 2012.
[4] B. Ramirez and M. Horn, “OVM & Parameters: Why Can’t They Just Get Along?,” Design and Verification Conference (DVCON), San Jose, 2011.
[5] “IEEE Standard for SystemVerilog--Unified Hardware Design, Specification, and Verification Language,” IEEE Std 1800-2023 (Revision of IEEE Std 1800-2017), vol., no., pp.1-1354, 2024.
[6] J. Ridgeway, “Molding Functional Coverage for Highly Configurable IP,” Design and Verification Conference (DVCON), San Jose, 2016.