2019 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION INDIA

# Uncover: Functional Coverage Made Easy

Akash S[1] (aks@nvidia.com)

Rahul Jain[1] (rahjain@nvidia.com)

Gaurav Agarwal[1] (gagarwal@nvidia.com)

[1] Nvidia Graphics Pvt Ltd, Bengaluru, India

***Abstract*—Functional coverage coding has a lot of untapped scope for automation. This paper discusses a flow developed to leverage such automation opportunities. The corollaries of the implementation: better modularity, versatility, consistency and reusability are discussed. It talks in detail about a robust and scalable shorthand notation devised to auto-generate code that makes coverage coding less tedious. The lines of code to be written reduces by 75-80% using this notation.**

***Keywords—Functional coverage, Design and verification automation, UVM, Perl scripting***

## I. INTRODUCTION

Functional coverage is the compass that guides one through the process of verification closure, the essential answer to, "Are we there yet?". It is the feedback mechanism that tells what was tested, and more importantly, what was missed. But, writing coverage code is often considered tedious; the reasons being monotony, repetitions, inconsistency and time-consumption.

Our flow is an attempt to make functional coverage coding hassle-free and organized, with the following unique selling points:

- **Auto-generation** of a complete **coverage package** that can be directly connected to the testbench.
- An **elegant shorthand notation** to generate SystemVerilog-UVM functional coverage code, which reduces the lines of code to be written by about **75-80%.**
- **Separates concerns** regarding manipulation of data from actual coverage collection.
- **Reusable** coverage collectors.
- **Versatile** flow that supports both whitebox and blackbox coverage.
- **Consistent** implementation across projects and testbenches, with support for easy review of implementation.

## II. RELATED WORK

Functional coverage code generation has largely been using a description table with various covergroups (CG), coverpoints(CP) and bins. References [1], [2] are examples, where coverage code is generated from an Excel sheet.

Though the method enhances manageability and documentation, it has the following limitations:

- **Versatility**: Whitebox coverage has full visibility to the DUT and its internal functioning, used extensively by design engineers. The table-based methodologies **do not support** such a provision.
- **Scalability and flexibility:**
  - A row-column pair has not been able to make use of a lot of automation opportunities and difficult to expand to new feature updates.
  - This method works fine with sets of key-value pairs; but does not implement the gamut of code writing scenarios, and hence **restrictive** in nature.
- **Repetition**: Coverage code tend to have a lot of repetitions and the table-based system does not help curbing this.
- **Modularity**: The existing flows do not mention anything about separating data collection and manipulation from actual coverage collection, an essential OOPs concept.

Hence, we conclude that the available methodologies have several unsolved issues and our attempt is to cater to these insufficiencies.

## III. THE PROCESS

- **Planning:** The process of coverage collection starts with figuring out various aspects of the DUT to be covered.
- **Coding:** The coverage plan is then coded in SV to be integrated with rest of the testbench. In our flow, the verification engineer jots down the coverage plan in a newly developed **shorthand notation**. This notation is **parsed** by the flow and converted to SV code. To validate if the code matches the plan, the flow generates a **consolidated XLS** with information regarding all generated coverage models.
- **Test:** Code generated by the flow consists of a complete coverage package containing **transaction-wise coverage models** in subscribers that only sample data. These subscribers are connected to components from the testbench that deal with respective transaction classes. This composite testbench is regressed to collect coverage statistics.
- **Analyse:** The coverage reports generated out of regressions are analysed as URG HTML reports or as VDB files in tools like Verdi.

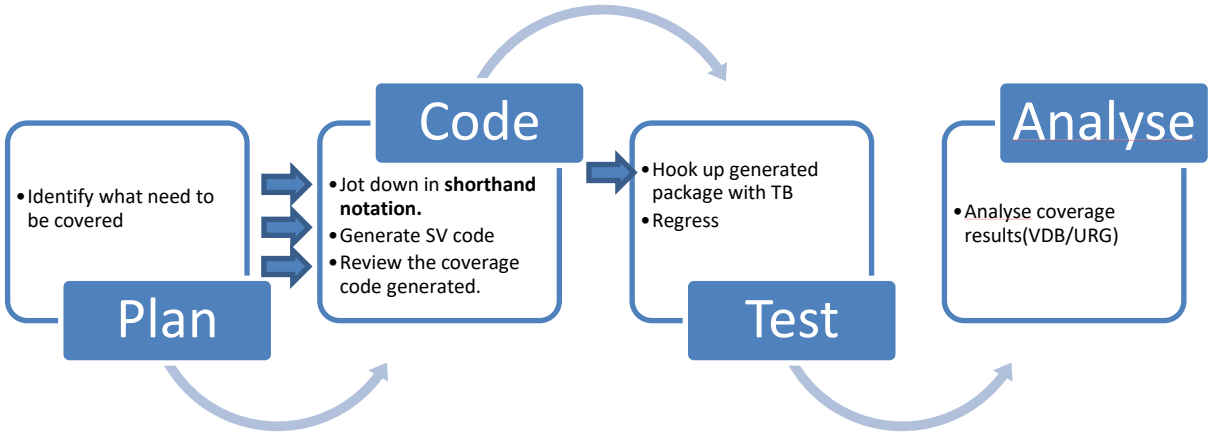



Figure 1: Coverage collection process with arrows pointing to where our flow intervenes.

The flow has the following features:

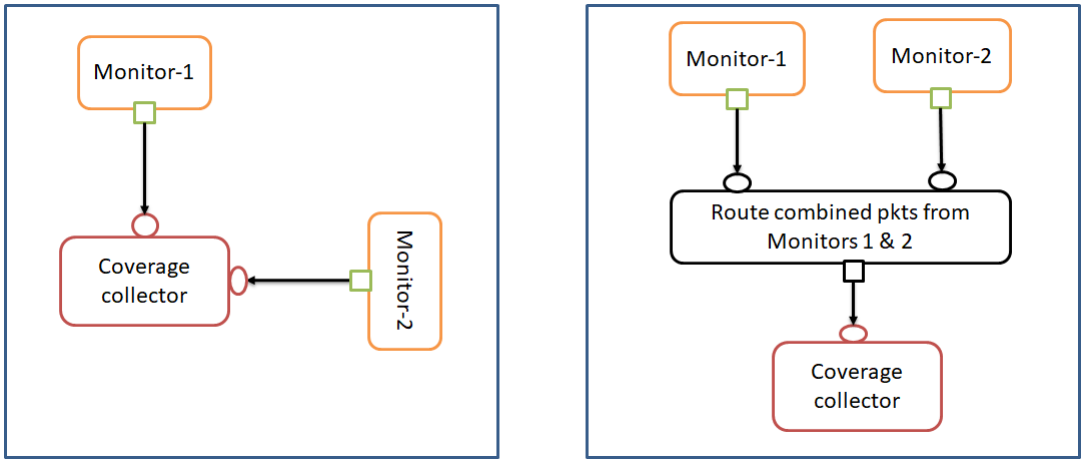
- **Separation of concerns:** Manipulation of data and it's sampling are performed by different components.
- **Whitebox coverage:** A feature widely used by design engineers, whitebox coverage has complete knowledge of the RTL and its internal functioning.
- **Consistent and reusable components:** The generated code follows uniform coding and naming conventions. Also, the coverage collectors only sample data, thus being orthogonal and easily reusable across projects and testbenches.

### *A. Separation of Concerns [3]*

Orthogonality and layering are important features of robust programs. The bottom-most component must be the "dumbest" and just perform tasks, without any decision making. Unfortunately, this is not followed while coding in many cases.

There are several scenarios where variables spread across transactions and time need to be crossed. The general practice is to do this collection, manipulation and sampling in the same component, which does not result in healthy, orthogonal structures. This further inhibits the reusability of the components.

The proposed flow mandates separating manipulation and sampling of data. By generating the bottom-most components that only samples coverage, it places the responsibility on the verification engineer to code components to process data before feeding it to the collectors.




(a) (b)

Figure 2: a. Coverage collector sorts out and combines packets across time and transaction and samples.
b. Coverage collector only samples the data it collects. Components before it takes care of the manipulation and consolidation.

### *B. Whitebox Coverage*

Whitebox coverage, as the name suggests, is cognizant of the DUT's internal structure and functioning. The flow supports generation of whitebox coverage code with just a few additional command line arguments. The structure is as shown in Figure-3. Such a feature makes our flow versatile, in the sense that it can be adapted to both whitebox and blackbox coverage.

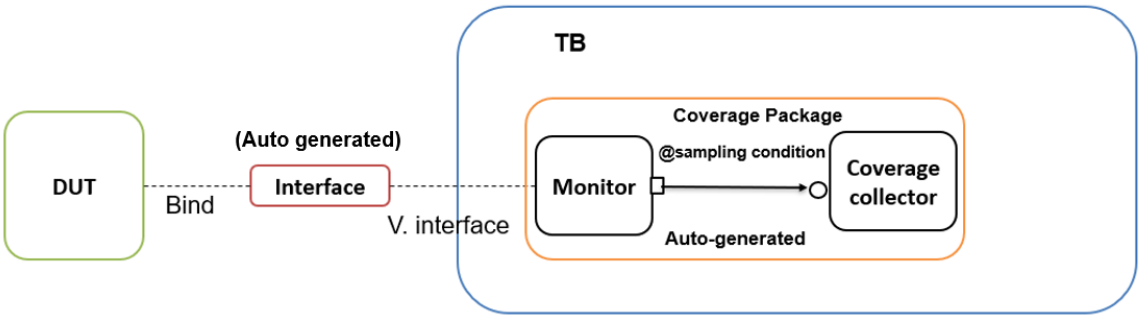



Figure 3: Structure of auto-generated code for whitebox coverage.

### *C. Consistent and Reusable components [4]*

The flow generates a coverage package with transaction class-based coverage models. These subscribers, with uniform coding style and naming structure, can be easily hooked up with rest of the testbench. Also, an XLS is generated to cross verify if the code generated matches the coverage plan.

These features ensure consistent implementation across projects and testbenches, minimizing ambiguities and disruptions due the organizational changes.

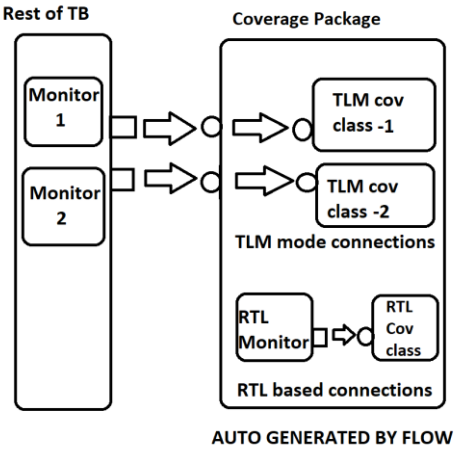



Figure 4: An overview of coverage package generated by the flow, with its constituents.

## IV. SHORTHAND NOTATION

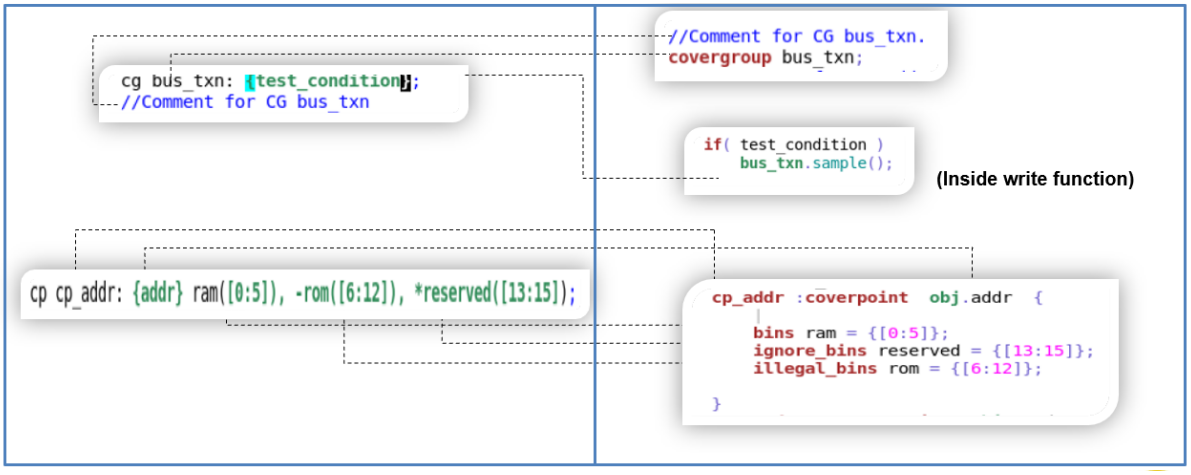
An elegant and scalable shorthand notation is at the heart of this flow. The shorthand notation is designed keeping the following points in mind:

- Brevity
- No repetitions
- Harness patterns

### *A. Brevity*

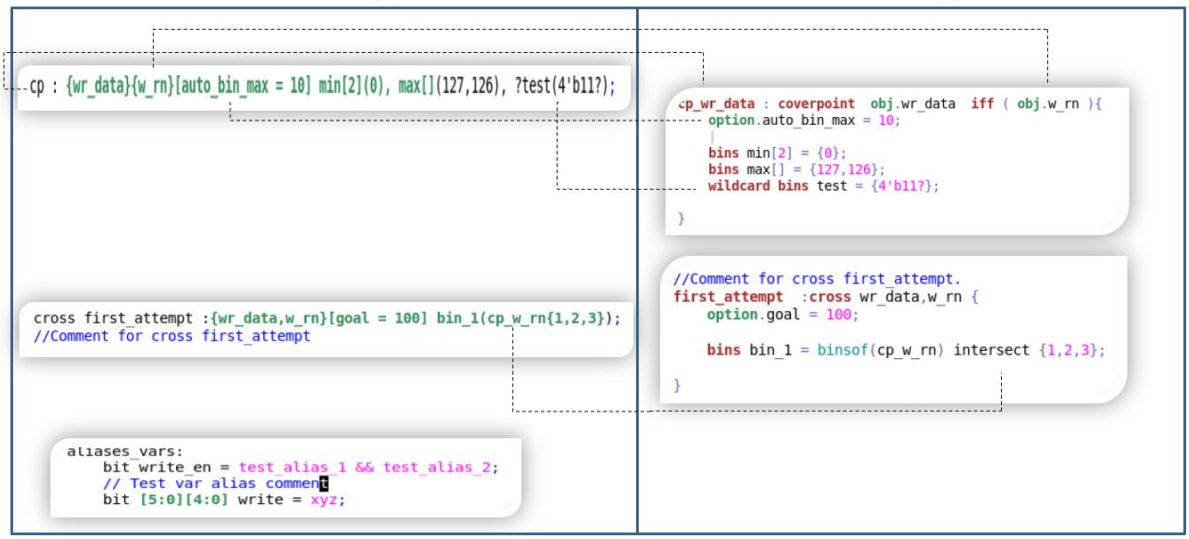
CG/CP definitions are one-liners in the shorthand notation. As shown in Figures 5 and 6, the shorthand notation needs only the crux of information and auto-generates everything else. The notation can also auto-generate meaningful names of CGs/CPs/bins when not specified (as shown in Figure 6).

The framework also provides a placeholder-signal feature to de-congest the CG/CP conditions (The section aliases_var in Figure 6.a). These placeholder signals can be used for pre/post sampling operations as well.




(a) (b)

Figure 5. Shorthand notation snippets and it's conversion for basic CG/CP definitions
a. Shorthand notation
b. Corresponding SV notation




(a) (b)

Figure 6. A few more shorthand to SV examples.
a.Shorthand notation
b. Corresponding SV generated code.

### *B. No Repetitions*

Coverpoints cannot be used across covergroups. Hence coverpoints used in more than one covergroup are defined in each of those covergroups. This leads to many repetitions and becomes tiresome to code and manage.

The shorthand notation has a separate section for repeating coverpoints, where they are defined. A command to copy the required CP is used in CGs that need the CP, as shown in Figure 6. This command, while parsing, is replaced by the CP definition. Now, the CPs are defined once and referenced wherever needed, reducing the gamut of repetitions.

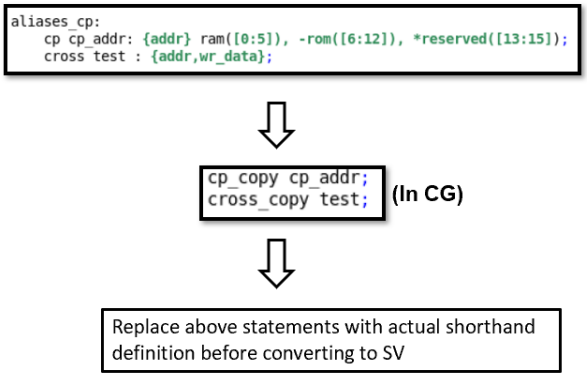
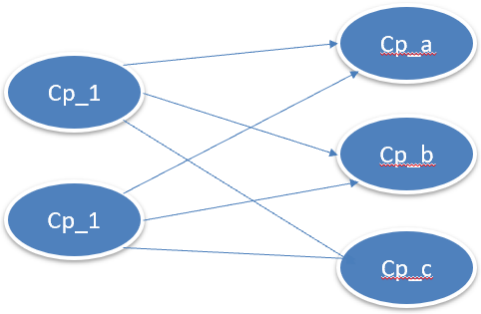



Figure 7. How CP aliases work

### *C. Harness Patterns*

Coverage coding involves a multitude of patterns. The notation features macros that can be used to generate CPs/bins that follow specific patterns. Some of them are explained below:

- **Many-many relationships:** There are cases where one set of CPs are crossed with another. Similarly, one set of states/values transit to another. Such patterns can be generated by macros as shown in Figure 8.




(a)

```
cp_speedXcp_timeout  :cross cp_speed, cp_timeout {
}
cp_speedXcp_bad_pid_after_token  :cross cp_speed, cp_bad_pid_after_token {
}
cp_speedXcp_flow_ctrl_with_nak  :cross cp_speed, cp_flow_ctrl_with_nak {
}
cp_speedXcp_stall_error  :cross cp_speed, cp_stall_error {
}
```

(b)

```
^cross_expand ("cp_speed", "cp_timeout, cp_bad_pid_after_token, cp_flow_ctrl_with_nak, cp_stall_error");
```

(c)

Figure 8. a. Many-many relationship
b. Use-case in SV syntax
c. Corresponding macro in shorthand notation

- **Irregular address ranges:** Many address ranges to be covered are non-uniform. Hence, they cannot be covered with auto bins. The macro "range_interval" generates piecewise ranges in between an address space.
- **One-hot bins/CPs:** Cases where each bit/enum member needs to be covered as a single CP/bin are generated by macros "one_cp_each" and "one_hot" respectively.
- **Hop:** Cases where numbers to be covered are in arithmetic progression (successive numbers having the same difference) are generated by macro "hop".

Tables I and II list out all the macros available for CP/bins generation.

Table I: List of macros to generate bins

| Macro | Format | Example shorthand syntax | O/P SV code |
|---|---|---|---|
| list | ^list(<vals>) | ^list{[2]{1}, *{2}, -{3}, {4}); | bins bin_list0[2] = {1};<br>ignore_bins ignore_list0 = {2};<br>illegal_bins illegal_list0 = {3};<br>bins bin_list1 = {4}; |
| hop | ^hop(min, max, step) | ^hop(1,5,2); | bins bins_hop_1 = { 1};<br>bins bins_hop_3 = {3};<br>bins bins_hop_5 = {5}; |
| interval | ^interval(h/d, min, max, break_1, break_2) | ^interval(d, 1, 10, 5,7);<br>(Similarly for hex) | bins bins_interval_1 = {[5:1]};<br>bins bins_interval_2 = {[7:6]};<br>bins bins_interval_3 = {[8:10]}; |
| range_interval | ^range_interval(min, max, step_size) | ^range_interval(1,a,4); | bins bins_range_interval_1 = {['h5:'h1]};<br>bins bins_range_interval_4 = {['ha:'h6]}; |
| expand | ^expand(<expr>) | ^expand({1=>2,3}, {4=>5}); | bins transition_1_2 = (1=>2);<br>bins_transition_1_3 = (1=>3);<br>bins transition_4_5 = (4=>5); |
| list_transition | ^list_transition(<expr>) | ^list_transition([2]{1=>2,3}, *{4=>5=>6}, -{7=>8} | bins bin_list0[2] = {1=>2,3};<br>illegal_bins illegal_list0 = {4=>5=>6};<br>ignore_bins ignore_list0 = {7=>8} |

Table II: List of macros to generate coverpoints

| Macro | Format | Example shorthand syntax | Expanded shorthand syntax |
|---|---|---|---|
| one_cp_each | ^one_cp_each(<signal>) | ^one_cp_each(test)<br>test is:<br>case 1: 2-bit vector<br>case 2: enum with pass/fail | Case-1:<br>- cp test_bit_0<br>- cp_test_bit_1<br>Case-2:<br>- cp test_enum_pass<br>- cp test_enum_fail |
| one_hot | ^one_hot(<signal>) | ^one_hot(test)<br>test is a 2-bit vector | cp test_one_hot: {test} bin_0(test[0]), bin_1(test[1]) ; |
| cross_expand | ^cross_expand("cp_1,cp_2", "cp_3, cp_4") | ^cross_expand("cp_1,cp_2", "cp_3") | cross cp_1Xcp_ 3: {cp_1,cp_3};<br>cross cp_2Xcp_3: {cp_2, cp_3}; |

## V. RESULTS

The automated infrastructure developed:

- Generates functional coverage code of healthy structure.
- Supports both whitebox and blackbox coverage.
- Is Consistent:
  - Minimal disruptions due to project/team reorganisations.
  - Minimal ambiguity in code due to extensive automation.
- Reduces efforts**:**
  - The lines of code to be written has gone down significantly, by a factor of **75-80%** in most cases.
  - Seamless connection between generated coverage package and testbench.
- Enhances reviewability: concise report generated to validate implementation.

## VI. FUTURE DEVELOPMENT

We intend to add the following features to the flow over time:

- Testbench specific heuristics: The flow could be made "TB-wise" by feeding it specific characteristics of each testbench, further automating code generation and code reuse.
- Leverage Verific: Verific SV parser can be used to get transaction level information to generate code. They are being entered by users now.

## VII. CONCLUSION

Our primary goal was to leverage the automation opportunities in functional coverage coding to channel efforts efficiently. The developed infrastructure generates code that has a scalable, consistent and reusable structure and reduces mundane work with the help of a shorthand notation. The developed shorthand notation reduces the lines to code by a significant margin.

## VIII. REFERENCES

[1] Functional Coverage Generator, Munjal Mistry- eInfochips (DVCon India 2017)

[2] SwiftCov- Automated Coverage Closure Tool, Kunal Panchal – eInfochips, Pushkar- Macom, Nisha Mallya- Macom (DVCon India 2017)

[3] Unraveling the Complexities of Functional Coverage: An advanced guide to simplify your use model, Thomas Ellis – Mentor, Rohit Jain- Mentor (DVCon US 2018)

[4] Implementing a Reusable, Auto Configurable Functional Coverage, Santosh Moharana, Synopsys (DVCon India 2016)

[5] IEEE 1800-2017 SystemVerilog

[6] Universal Verification Methodology (UVM) 1.2 Class Reference