DVCon India 2019

Benefits of PSS coverage at SOC and its limitations

Sundararajan Haran, Principal Engineer, Qualcomm India Private Limited,
Bangalore, India (**sharan@qti.qualcomm.com**)
Saleem Khan, Senior Staff Engineer, Qualcomm India Private Limited,
Bangalore, India (**salekhan@qti.qualcomm.com**)

***Abstract*— With growing adoption of Portable test stimulus standard (PSS) in complex SOC Verification testbenches, complex soc tests scenarios like multi-core, cache-coherent, low power scenarios get generated automatically using PSS model-based test generator solution tools. Now It's important to review and qualify this auto generated test stimulus to match our desired verification test scenario goals as per plan. To analyze the generated stimulus user needs to have CDV like metric approach. In Accellera PSS 1.0a release coverage support feature got added, now this allows user to have coverage groups developed at PSS model itself. This helps to grade and qualify the valid test stimulus even before running test stimulus.**

In this paper we like to provide an overview of Accellera' s PSS generation coverage feature support and benefits. We also share our experience in adoption this coverage solution and the limitations we observed. This paper also describes on the need for run time coverage which is currently undefined in PSS.

Keywords— Portable test and stimulus standard (PSS); Application programming Interface (API), Domain-specific language (DSL), System under test (SUT), Hardware/Software Interface (HIS), Coverage driven Verification (CDV); procedural Interface (PI)

I. Introduction

Accellera PSS offers a standard way to specify test stimulus intent and behaviour which is reusable across target platforms (e.g., Simulation, emulation, silicon, etc.). While Portable Stimulus is about test portability across various verification platforms, the real value is in the improvements to engineering productivity while creating a higher quality test stimulus. PSS abstracts intent capture, coverage, and provide project-to-project reuse. PSS Coverage is really a new type, at the system level. It provides an opportunity to better understand your design and the status of your SoC verification efforts. The PSS coverage application chapter describes our experience in adoption of PSS gen coverage for our Multimedia sub-system blocks, sharing the benefits we observed with this PSS coverage addition to our existing PSS model.

PSS Gen time coverage

PSS standard offers two kinds of gen coverage constructs i.e. Implicit, Explicit covergroup in both DSL and C++ syntax flavors.

The **covergroup** construct is a user-defined type. There are two forms of the **covergroup** construct. The first form allows an explicit type definition to be written once and instantiated multiple times in different contexts. The second form allows an in-line specification of an anonymous **covergroup** type and a single instance.

a) An *explicit* **covergroup** type can be defined in a **package, component** (see Appendix ex-1), **action** (see Appendix ex -2), or **struct** (Appendix example - 3). In order to be reusable, an explicit **covergroup** type shall specify a list of formal parameters and shall not reference fields in the scope in which it is declared. An instance of an explicit **covergroup** type can be created in an **action** or **struct**.

b) An *in-line* **covergroup** can be defined in an action or struct scope. An in-line covergroup can reference fields in the scope in which it is defined. (see Appendix example -4).

17.1.1 DSL syntax

The syntax for **covergroup**s is shown in Syntax 82.

```
covergroup_declaration ::=
    covergroup covergroup_identifier ( covergroup_port {, covergroup_port } )
    { {covergroup_body_item} } [;]
covergroup_port ::= data_type identifier
covergroup_body_item ::=
    covergroup_option
  | covergroup_coverpoint
  | covergroup_cross
covergroup_option ::= option . identifier = constant_expression ;
```

Syntax 82—DSL: covergroup declaration

Fig 1: Reference from Portable_Test_Stimulus_Standard_v10a.pdf – Accellera,

The following also apply to covergroup definition

a) The identifier associated with the **covergroup** declaration defines the name of the coverage model type.

b) A **covergroup** can contain one or more coverage points. A *coverage point* can cover a variable or an expression.

c) Each coverage point includes a set of bins associated with its sampled value. The bins can be user defined or automatically created by a tool.

d) A **covergroup** can specify cross coverage between two or more coverage points or variables. Any combination of more than two variables or previously declared coverage points is allowed. See also Example (Appendix, example 1)

e) A **covergroup** can also specify one or more options to control and regulate how coverage data are structured and collected. Coverage options can be specified for the **covergroup** as a whole or for specific items within the **covergroup**, i.e., any of its coverage points or crosses. In general, a coverage option specified at the **covergroup** level applies to all its items unless overridden by them.

- **Covergroup Instantiation**

A **covergroup** type can be instantiated in struct and action contexts. If the **covergroup** declared formal parameters, these shall be bound to variables visible in the instantiation context. Instance-specific coverage options may be specified as part of instantiation.
In many cases, a **covergroup** is specific to the containing type and will not be instantiated independently multiple times. In these cases, it is possible to declare a covergroup instance in-line. In this case, the **covergroup** type is *anonymous (Ref: Appendix examples)*

II PSS coverage Application

In our SOC TB, we adopted PSS generation coverage flow as pilot in out Multimedia sub system, this sub-system comprises of multiple IP blocks like Camera, Display, Graphics, DSP, APU ...etc. With PSS adoption we get major benefit or re-use of PSS models across different projects. Since Each project is unique with its Architecture changes, we used to have changes to model specific to project which will impact the project configuration and the respective test config for all the components based on the feature additions or enhancements plan.

Now , there is a possibility of missing updating the project config settings or test user can miss out to specify the test configuration of a given component feature which will result in test Before the introduction of PSS coverage there is no mechanism or kind of metric existed as part of PSS to identify the missing holes , reporting any illegal test stimulus generation . user happen to notice test hang when an illegal test ran, and it takes lots of effort to root cause the failure at SOC level and get the test fixed.

We started to adopt the coverage metric to address these challenges to all our Multimedia blocks , The start point to implement this generation coverage is first to identify all the common actions that are shared across these different PSS components and we looked at the attributes which gets configured randomly e.g. : different frequency modes , Image processing algorithms , different image formats , image sizes and so on . We tried to model basic cover groups with the intention to make sure to cover all the random configuration whether we exercise different frequency modes, all different data format kinds like JPEG. later we added cross cover to all the interesting valid bins.

Once we had all the basic component cover group coded, we ran the full regression and started analyzing the coverage results. The cool thing about this PSS gen coverage is that coverage collection doesn't need to wait till simulation run complete instead user can obtain the results post the test generation itself.

- **Gen Coverage Analysis:**

- Coverage analysis helped to root cause the missing test "SMMU Invalidation" which got added in one of our previous project , but missed to include in current project in the test configuration file , which acts as constraint to generate the specific test .Action coverage block attribute which checks cache enable field hole helped to catch this issue.

- In initial migration from older project we used to notice few camera tests failing caused by unsupported frequency mode configuration generated , This gets generated as the frequency "enum" field of previous project supported more frequency modes of operation and we didn't happen to have any constraint on this project to limit the valid modes. Now we added a cover group bins for only valid modes and keep the unsupported as illegal now the tool will flag generation error when unsupported frequency gets generated.

```
Covergroup :  mdss_freq_switch_cg
     -  SVS2, TURBO L0, L1,L2,L3, PLL_TEST_SE cover bins
        should not be hit
     -  NOMINAL is the default value
 Covergroup : mdss_power_seq_cg
     -  In mdss_soc_state coverpoint, except
        MDSS_HM_COLLAPSE rest should be excluded
Covergroup : camss_power_seq_cov:
freq_switch_cov_struct
     -  SVS3 is not applicable for camss, but it gets
        covered (3 hits) – illegal stimulus

Covergroup : GFX: gfx_test_select_cov:
vdd_collapse,vdd_collapse_imem should be excluded
```

Figure 3: Reference from Qualcomm Multimedia code snippet

We started defining explicit cover group in package for the commonly used action attributes in a separate cov file. This cover group is instantiated along the component instance with define switch to enable/disable coverage from inline run argument. Few of component specific covergroup are specified inline extending the action in this newly added coverage file. Once all the cover groups are integrated with the model the central team runs full suite test generation keeping coverage ON and post the results to component owners.

Each component lead analyzes the coverage report against his plan to ensure all the intended tests are generated by the model and identifies any stimulus holes from the coverage report , As this reported coverage is in the ucd (unified coverage database) format , user can even use this coverage and map along with coverage obtained from other target platforms ,(e.g.: formal , system Verilog based – simulation runs).

This coverage analysis helps us to enhance the test much earlier even before running the actual simulation, thereby saving lots of simulation run time efforts and disk usage. The overall coverage flow is shown in below Fig : 1. Though this gen time coverage metric helps to improve our test stimulus , it lacks defining scenario coverage and the LRM currently doesn't provide any details on how a use case specific test scenario coverage to be implemented , e.g. user is interested to cover sequence of chain of actions , concurrent execution of actions , interaction with external events and so on …etc.

SOC tests is mostly intended to verify dataflow across more than one or multiple sub-systems, in such cases we need to cover parallel or sequence of action executions. It will be good if we have some way to express coverage metric at run time associating my action sequence along with external event occurrences so that I can ensure at run time the test executed actions in parallel or in a sequence as intended along with external events as well . Today we do analyze the traces from the run logs and associate with waves to determine the sequence of actions which is a difficult task to do, unlike examine the cover group to identify this missing scenario.

- **Benefits of PSS coverage :**
  - With PSS Gen time coverage data obtained during from test generator engines allows to create optimized set of tests based on the specific coverage goals this helps to improve quality of stimulus even before simulation runs.
  - Able to analyse and avoid if invalid scenario gets generated
  - Eliminate redundancy in tests
  - Check the test intent is followed from the action sequence coverage details

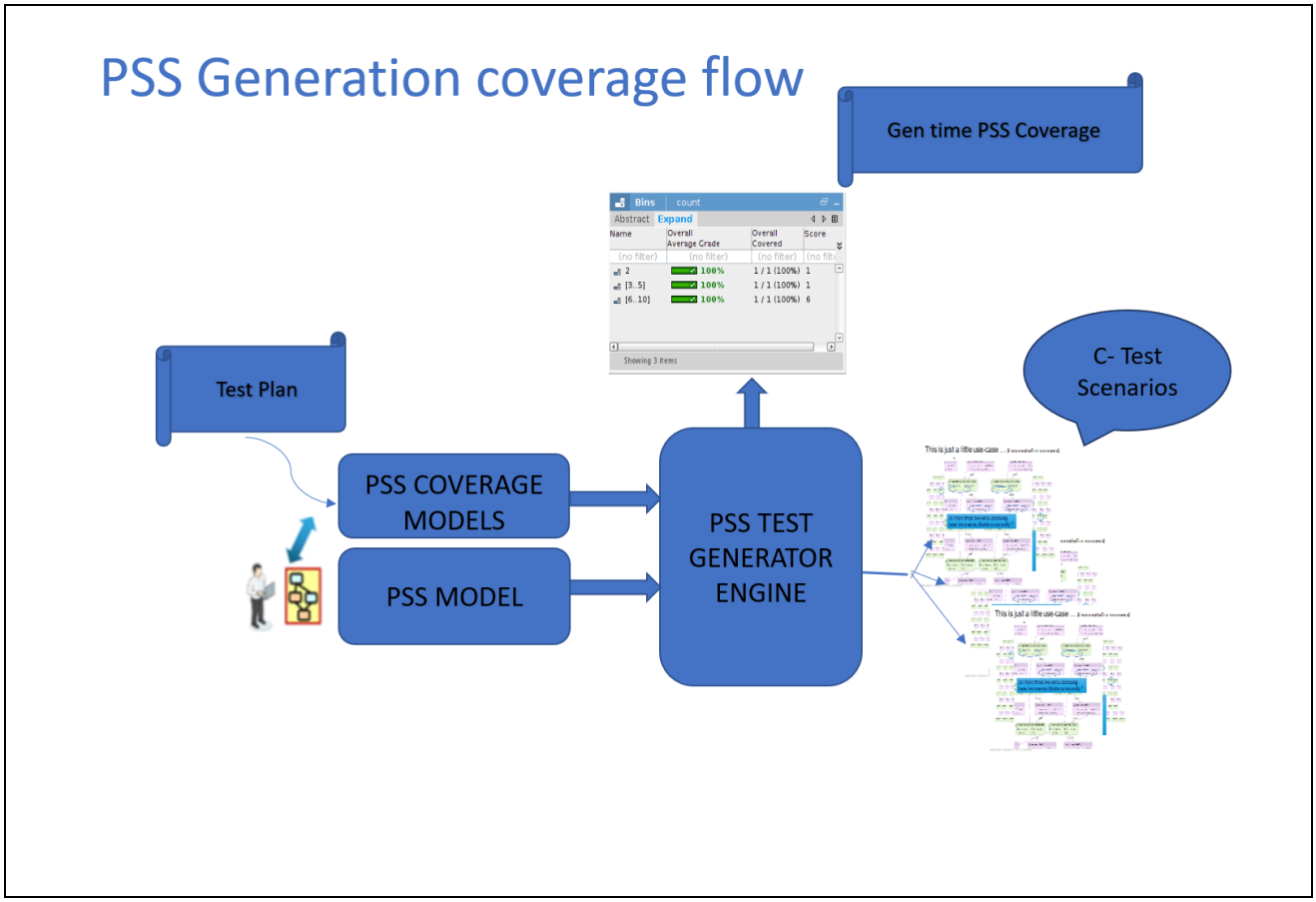

Figure 4: PSS Generation coverage flow
[Reference from Perspec user guide]

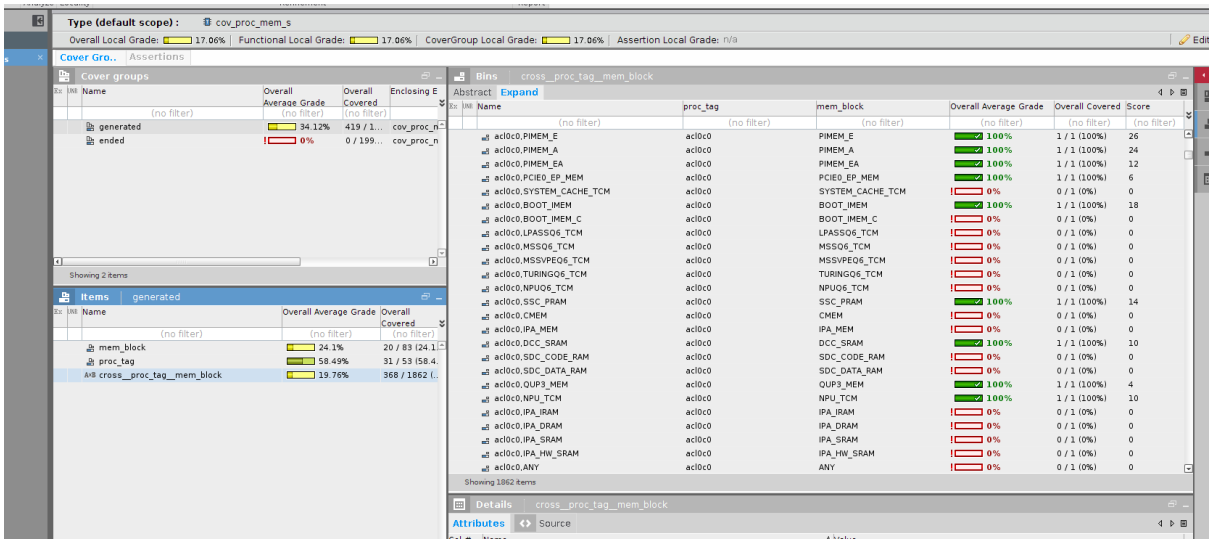

Figure 5: Sample PSS Generation coverage snapshot report from Perspec tool

III RUN TIME COVERAGE

Most of the SOC complex test have concurrent actions execution ,passive models in HVL's to respond or generate stimulus traffic though the test scenario is generated via PSS we never know on the reality whether those actions got really executed in parallel or not since most of the SOC Test benches are hybrid in nature involving HVL's passive models we need feature support in PSS to allow to pass action attributes or its occurrence time message from PSS to HVL enabling to write cover group which can capture run time scenario behavior . Perspec tool from Cadence offers run-time coverage which is a "Mirrored" gen-time Coverage which is collected by post process the simulation logs this flow adoption is progress in our TB, we tried to provide an overview of the flow in this paper in the below chapter.

- **Need for run time coverage & limitations of gen coverage :**
  - Ability to define coverage goals that span across multiple actions within a scenario is not supported in PSS.
  - Ability to define coverage goals in terms of temporal relationships between events at runtime is not supported in PSS

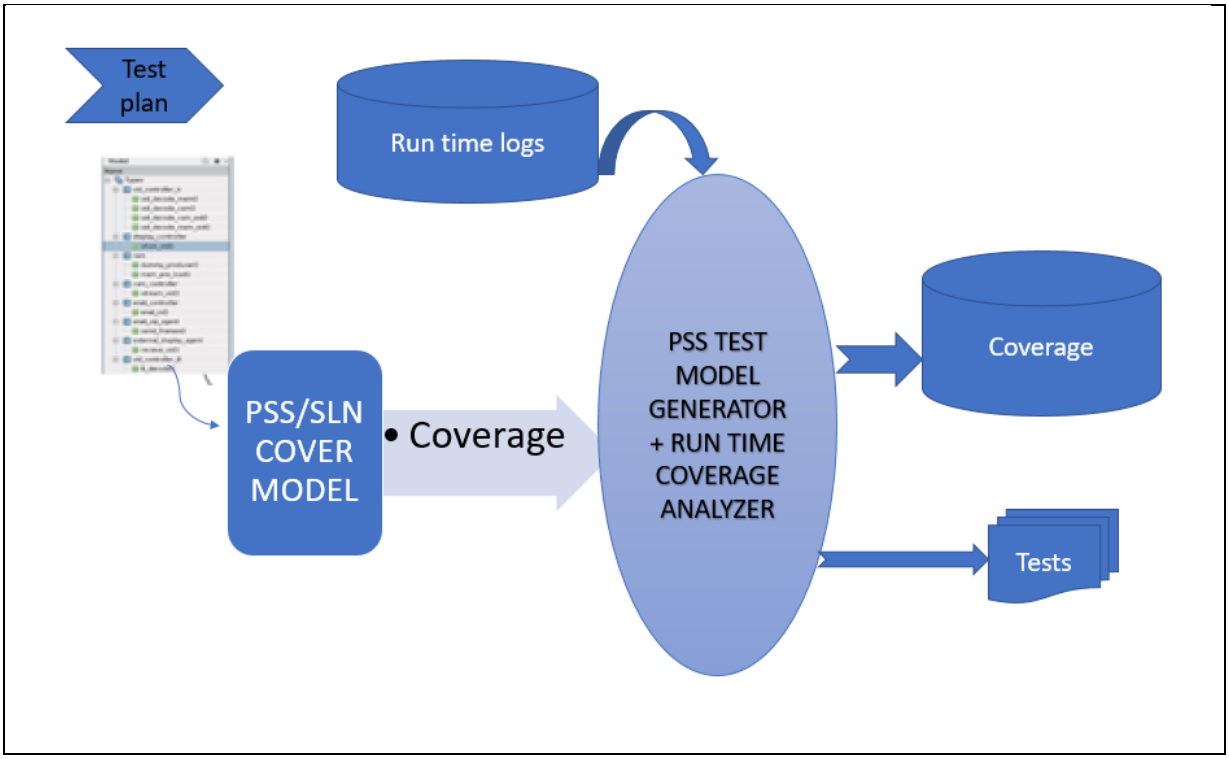

Figure 4: RUN time coverage flow [Reference: Perspec coverage user guide]

IV SUMMARY

In summary PSS coverage at SOC provides the below major benefits:

- Productivity - less time running tests by reducing number of tests needed.
- Predictability - concise goals and measurement towards them
- Quality – non subjective completeness metrics

Gen time coverage limitation can be overcome with Run time coverage flow but to get this flow working user need to have handshake events to communicate across PSS to SV , this requires additional effort in developing cover groups , we adopted string parser to notify the handshake event occurrences by parsing the log file , this mechanism allows to collect info on a given action sequence which has dependency with SV tasks/function we used to map the action calls and its respective function calls with TAG ID to notify the occurrence . Post simulation run this TAG ID Sequence pattern is matched for expected cover intent, thereby we could able to say the test scenario is simulated as per the coverage goal.

V REFERENCES

[1] https://www.accellera.org/community/portable-stimulus

[2] https://www.cadence.com/content/cadence-global/en_US/home/tools/system-design-and-verification/software-driven-verification/perspec-system-verifier.html

[3] http://www.deepchip.com/items/0578-03.html

APPENDIX – CODE SAMPLES

Sample A: Implicit gen coverage example

```
//Explicit cover group
 covergroup cg_explicit (color_e color,
                  shape_e shape,
                  bit[3] v_a,
                  config_modes_e config_mode) {
       //option.per_instance = true;
       option.at_least = 2;

       c_l1 : coverpoint config_mode;
       c_l2 : coverpoint color;
       c_l3 : coverpoint shape;
       c_l5 : coverpoint v_a { bins a = [0..1,7];
                        bins others[] = default; }

       all : cross color,shape,config_mode;

 };
```

```
//Explicit cover group Instance

extend component graphics_c {
      cg_explicit cg_explicit_inst
(color,shape,v_a,config_mode) with {
      option.at_least = 2;
      };
};
```

Sample B: Explicit gen coverage example

```
//Implicit cover group sample code
extend component graphics_c {
  action sample_action_a {
     covergroup {
       option.at_least = 2;

       c_l1 : coverpoint color;
       c_l2 : coverpoint shape;
       c_l3 : coverpoint s0 iff (is_s0_enabled) {
                  illegal_bins not_legal = [3,5]; }

       all : cross color,shape

     } cg_implicit;
   };
};
```