What Just Happened? Behavioral Coverage Tracking in PSS

Tom Fitzpatrick, Siemens EDA, Groton, MA, USA (tom.fitzpatrick@siemens.com)

Wael Abdelaziz Mahmoud, Siemens EDA, Cairo, Egypt (wael.mahmoud@siemens.com)

Mohamed Nafea, Siemens EDA, Cairo, Egypt (mohamed.nafea@siemens.com)

***Abstract*—PSS3.0 introduces the concept of Behavioral Coverage of a PSS specification, allowing for the specification of specific action sequences and data combinations that must be observed to exercise critical functionality. Due to the randomization employed when solving a PSS scenario, it is not always possible to determine *a priori* what the behavioral coverage of a particular scenario or set of scenarios might be, so some amount of run-time or post-run processing must be employed to determine the behavioral coverage. This paper describes a methodology that may be employed to determine the behavioral coverage of a test or regression suite of tests generated from a PSS specification.**

Keywords— Portable Stimulus; Coverage; Behavioral Coverage; UVM; Constrained-Random

I. Introduction

The Portable Test and Stimulus Standard (PSS) from Accellera [1] defines an abstract modeling language to specify critical verification intent and allow the generation of target-specific implementations of a set of correct-by-construction *scenarios*, each of which includes the critical intent supplemented by additional behaviors to meet the requirements of the critical intent. The declarative nature of a PSS description allows a concise model to be the basis for a potentially large number of scenarios. Since each scenario, by definition, includes the critical intent, it is not necessary to track coverage of these behaviors. However, it is often necessary to understand whether additional interesting behavior sequences occur as part of the scenario set generated from the model.

Conceptually like using covergroups in SystemVerilog (SV) [5] to understand which interesting values were generated by the random solver, PSS added Behavioral Coverage in PSS3.0, to allow the description of ancillary sub-scenarios and track which of those are exercised as part of the generated scenarios. Similarly, just as SystemVerilog covergroups are sampled throughout a simulation and the results typically aggregated at the end of simulation – essentially allowing randomization throughout the simulation and then asking, "what happened?" – it is often not possible to determine whether specific sub-scenarios are traversed until the end of simulation. This paper will describe a methodology that can be used to determine this coverage.

II. Background

When SystemVerilog was originally developed, the performance of a simulator's constraint solver quickly became a competitive differentiator between tools. Initially, several benchmarks were used to measure how long it took a solver to reach a solution for a problem like Einstein's Puzzle [2]. It soon became clear that the real benchmark is how long it took a given simulator to generate a set of random values within the constraint space to reach the desired functional coverage goals, as specified by the covergroups.

In SystemVerilog, it is possible to analyze the set of constraints and determine statically the set of possible solutions. Thus, when some set of random values is collected during a simulation, it is possible to calculate a "coverage percentage" that indicates how many of the coverage bins have been hit.

PSS includes a key feature that adds a dimension of solver complexity beyond a typical SystemVerilog solver. If a PSS action defines an input data flow object, then the solver must *infer* a compatible action to provide a compatible object. If there are multiple compatible actions available, the solver will randomly pick one. If that inferred action has its own input data object, then an additional action must be inferred to supply that object. Thus, different solutions based on different random seeds could lead to substantially different overall scenarios, each of which would include the originally specified sub-scenario that represents the critical intent.

III. Coverage and Quality

One useful metric in evaluating both the quality of the solver as well as the quality of the overall PSS model, is to record how many unique scenarios are actually generated from a given PSS specification. For example, if the critical

intent starts with an action that has a particular input object type, the solver must infer another action to provide it. If there are two possible actions to infer, invoking the solver with multiple seeds should eventually cause each action type to be inferred. If every seed causes the same action to be inferred, while the solution is technically correct, being able to see that only one solution is generated is nonetheless useful information. If the user expects both actions to be inferred eventually, it would be helpful to determine if that may be due to incorrect constraints. Additionally, a PSS description may include conditional (if/else, match) choices between possible actions to be traversed as well as a *schedule* construct that specifies a set of actions that must be traversed in a random order. Being able to determine which choices were actually made by the solver is also helpful.

IV. PSS Behavioral Coverage

The Behavioral Coverage feature in PSS allows the user to define a coverage monitor that may specify a sub-scenario of interest. This sub-scenario may relate to the set of inferred or selected/scheduled actions that get traversed by the solver. Similar to a simulator reporting on which SV covergroups have been hit, the Behavioral Coverage feature allows a PSS tool to report whether and how many times a specified behavioral coverage monitor was hit.

While there may be some coverage scenarios that can be determined at solve-time to be unreachable or will always be reached, many coverage scenarios may not be determined successfully until the simulation is actually completed. The proposed methodology shows how a set of Behavioral Coverage scenarios may be applied to set of simulation runs and report whether and how many times each coverage scenario was traversed within the simulation. Once coverage "holes" are thus identified, the PSS model can be adjusted, either by modifying the original source or by layering in additional constraints, to allow the user may to direct the solver to fill in the holes and reach the behavioral coverage goals.

V. Methodology

As shown in previous sections, PSS introduces new capabilities to assess the critical verification intent, by identifying the key action order and data combinations that need to be observed to exercise the key verification intent. Also, action inferencing is the key aspect of PSS. It's one of the most powerful features of PSS, it provides the ability to focus purely on the user's verification intent, while delegating the means to achieve that intent to the PSS tool itself. Also, PSS introduces activity *schedule* construct, which randomly traverses the enclosed actions as per the semantics of actions' data flow objects, to be traversed either sequentially or parallelly. Finally, there are other activity branching constructs, like: *select*, *match* and *if/else* constructs. Accordingly, PSS-complaint tools can create many different unique scenarios randomly at the solve time, which in most of the cases is hard to calculate by hand.

The paper introduces a methodology using Questa Portable Stimulus [3] - Siemens EDA tool as well as some utilities to post-process the solved PSS model during simulation to extract the behavioral coverage metrics. The methodology provides the ability to analyze the simulation results, determine which scenarios were traversed, and generate trends to assess the distribution of the uniquely traversed scenarios. These metrics are useful to verification engineers to help assess the executed scenarios and can be used to adjust the original model to improve results and hence improving the overall behavioral coverage of the model.

Basically, the methodology focuses on measuring two important metrics:

1. The breadth of possible solutions from a relatively simple PSS model is shown by the total number of randomly generated scenarios in a single simulation run or across a regression of runs.
2. The coverage of scenarios of interest within the set of solved scenarios, i.e. the new Behavioral Coverage in PSS3.0. Methodology of measuring this metric will be covered in "Behavioral coverage of key actions" next section.

a. Measuring the distribution of generated scenarios

This section covers the proposed methodology to assess and measure the quality and coverage of generated PSS scenarios after solving the user's PSS abstract model.

The flow chart in Figure 1 describes the proposed methodology, which can be summarized as following:

1. Start by developing PSS model, and define the critical verification intent
2. Optionally, Questa Portable Stimulus - Siemens EDA (SEDA) - User Interface (UI) debug tool can be used to debug the developed model
3. On-the-Fly solving PSS model during simulation, generate tests and run them using Questa Portable Stimulus SEDA tool
4. Using internally developed utilities, the executed/traversed PSS scenarios can be extracted post-simulation

5. Reporting trends and coverage of randomly solved and traversed scenarios during simulation, along with number of occurrences of each unique scenario as well as trend graphs to measure the distribution quality of the generated random scenarios

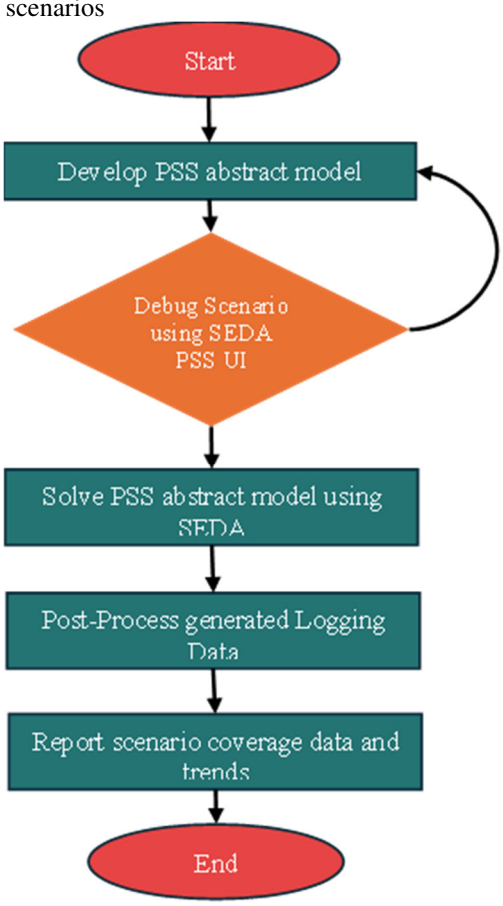

Figure 1: Proposed methodology for measuring generated scenarios

b. Behavioral coverage of key actions

Previous section gives an overview about PSS 3.0 behavioral coverage. This section provides a methodology to measure the behavioral coverage of complete scenarios or partial scenarios (i.e. focusing on critical intent).

The methodology is pretty much the same as the proposal shown in Figure 1, with the difference that the generated reports are focusing on the number of occurrences of the specified behavioral coverage in the input PSS model.

More examples of the proposed methodologies will be shown in the next section.

VI. Examples

This section covers some different examples with varying complexities to illustrate the above proposed methodology. Also, for each example the results of generated scenario coverage will be shown in different formats to illustrate the following metrics:

1. The breadth of possible solutions, in terms of
    a. Uniquely generated and traversed scenarios
    b. Distribution quality of generated scenarios during multiple simulation regressions with varying seeds, or single simulation with repeat loops, shown in Figure 2
2. Behavioral coverage of critical intent

A. Simple Example

Example 1 shows a simple PSS model, where there are 4 simple atomic actions (A, B, C, and D), traversed inside the “top_act” compound action, with each pair of actions traversed inside a *select* activity statement. The traversal of

this action should be repeated for 150 iterations, as specified in the *repeat* loop statement. There are four possible traversals of the activity, as shown below:

1. {sequence {A}, {C}}
2. {sequence {A}, {D}}
3. {sequence {B}, {C}}
4. {sequence {B}, {D}}

```
component pss_top {
    action A { }
    action B { }
    action C { }
    action D { }
    action top_act {
        activity {
            repeat (150) {
                    select {do A; do B;}
                    select {do C; do D;}
            }
        }
    }
}
```

Example 1: Simple PSS actions traversed inside repeat loop and select statements

Example 2 adds one extra simple activity statement, called *schedule*, to the PSS model described in Example 1. Running this modified version of the PSS model, we see that the number of possible and valid unique scenarios jumped from 4 to 12 scenarios, i.e. 3X increase in the number of unique scenarios.

```
component pss_top {
    action A { }
    action B { }
    action C { }
    action D { }
    action top_act {
        activity {
            repeat (150) {
                schedule {
                    select {do A; do B;}
                    select {do C; do D;}
                }
            }
        }
    }
}
```

Example 2: Simple PSS actions traversed inside repeat loop, schedule and select statements

Below is a list of all possible unique scenarios that can be generated from Example 2:

1. sequence {parallel {A, C}}
2. sequence {parallel {A, D}}
3. sequence {parallel {B, C}}
4. sequence {parallel {B, D}}
5. sequence {sequence {A, C}}

6. sequence {sequence {A, D}}
7. sequence {sequence {B, C}}
8. sequence {sequence {B, D}}
9. sequence {sequence {C, A}}
10. sequence {sequence {C, B}}
11. sequence {sequence {D, A}}
12. sequence {sequence {D, B}}

As the reader can imagine, the number of unique scenarios that can be created from even a simple PSS model can easily be very large, which makes it harder to calculate by hand. Today's complex SoCs can easily require PSS models that can generate large number of unique scenarios to cover the different verification intents. Accordingly, the proposed methodology helps assure verification engineers that they have verified all the required scenarios and the distribution quality of such scenarios.

We ran Questa Portable Stimulus Siemens EDA tool on the code in Example 2 and randomly generated the different scenarios. We then applied the proposed methodology to post-process the results to identify the 12 unique solutions shown above. Figure 2 shows the distribution quality of the randomly generated solutions, after 150 iterations, measured by the proposed methodology.

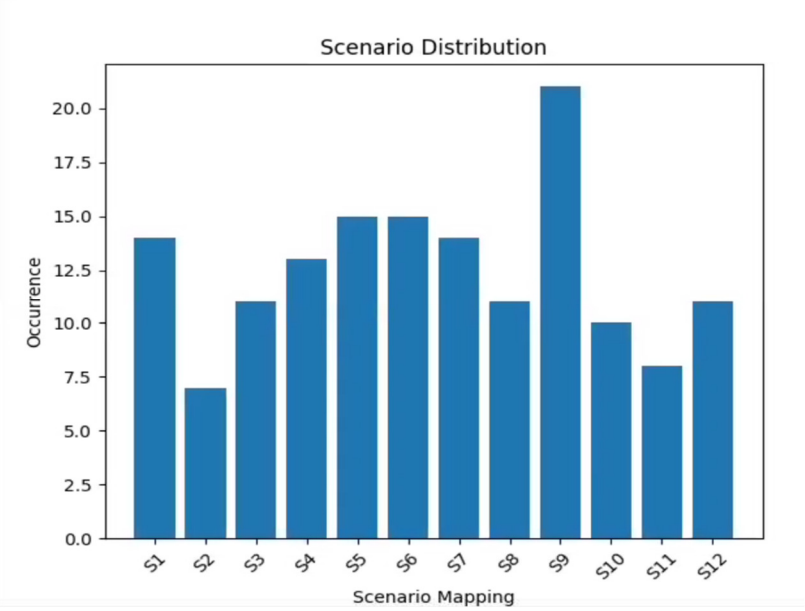

Figure 2: Scenario distribution quality of generated random scenarios for PSS example 2

In this simple example, each pair-wise combination of [A,B] and [C,D] is possible. It may be that the verification engineer wants to pay particular attention to the sequence of B followed by C. The behavioral coverage features in PSS3.0 allows us to add a coverage monitor as shown in Example 3 to identify this scenario explicitly.

```
component pss_top {
    action A { }
    action B { }
    action C { }
    action D { }
    c1: cover {
        activity {
            do B;
            do C;
        }
    }
```

```
    action top_act {
        activity {
            repeat (150) {
                schedule {
                    select {do A; do B;}
                    select {do C; do D;}
                }
            }
        }
    }
}
```

Example 3: PSS model with behavioral coverage

The proposed methodology is applied to measure the coverage of this critical behavior, and Figure 3 shows a sample of the verbose output of behavioral coverage results.

Extracting Behavioral Coverage Defined in PSS Model ...

ex3.pss[6] top_act::c1: sequence{sequence{B, C}}

Occurrences of defined Behavioral Coverage: (sequence, sequence, B, C) - Occurrences: 11

Behavioral Coverage matches (sequence, sequence, B, C):

loop[10]{sequence{sequence{B, C}}}
loop[11]{sequence{sequence{B, C}}}
loop[14]{sequence{sequence{B, C}}}
loop[20]{sequence{sequence{B, C}}}
loop[37]{sequence{sequence{B, C}}}
loop[46]{sequence{sequence{B, C}}}
loop[55]{sequence{sequence{B, C}}}
loop[57]{sequence{sequence{B, C}}}
loop[71]{sequence{sequence{B, C}}}
loop[99]{sequence{sequence{B, C}}}
loop[139]{sequence{sequence{B, C}}}

Figure 3: Sample of generated Behavioral Coverage Results

B. DMA Example

The previous examples demonstrated the randomization that can result from using PSS '*schedule*' construct. In this example, we explore another one of the powerful capabilities of PSS, where randomization can occur through action inferencing.

Consider the simple subsystem design shown in Figure 4. The memory element includes data storage that can be directly written to or read from, and a set of channel-specific registers to support DMA transfers of data chunks from one address to another, as well as transferring data to or from the peripheral port. The peripheral element contains a FIFO data buffer and a set of control registers to support transferring data to or from the memory port. Both RTL elements support the same simple read/write protocol for communication on their control ports, and the data bus between them follows a simple handshake protocol. The details of these interfaces are not important for the purposes of this paper.

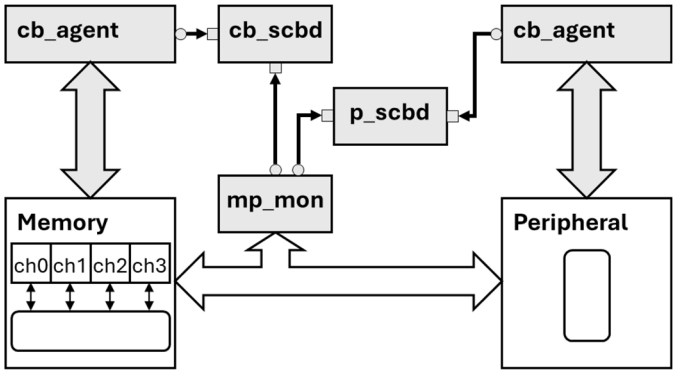

Figure 4: Memory/Peripheral Subsystem with UVM Environment

The UVM Environment

In the UVM [4] environment, each `cb_agent` supports transaction-level sequences to transfer data to/from the connected design element. The `mp_mon` monitor component tracks the data transferred between the two devices and the two scoreboard components serve as reference models to ensure that the data in the memory and peripheral devices is correct. A typical UVM test would consist of a UVM virtual sequence that coordinates the execution of agent-level transaction sequences to implement the desired scenario.

Suppose we want to test that we can read data out of the memory. In order to test that, we must first write data into the memory. We can either do a "`memory fill`" operation to write the data into the memory from the `cb_agent`, or we can load the data into the peripheral device and transfer it from the peripheral to the memory, or we could fill a memory buffer, do a DMA transfer to a different location in the memory, and dump it from there, or any combination of these. A basic set of such sequences is shown in Figure 5.

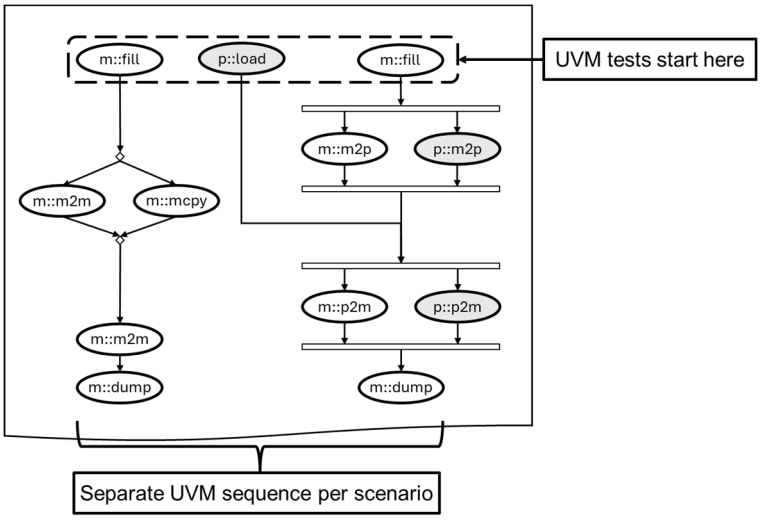

Figure 5: UVM Sequences to Model Scenarios

The UVM test writer would need to start at the top of this graph to consider the different ways to get data into the memory in order to dump the data and check that it is correct. This can start by filling the memory or by loading the peripheral. If we wish to keep the data local to the memory (such as in a block-level test that involves only the memory component), we could follow the initial fill operation with either a memory-to-memory DMA transfer (`m2m`) or we could do a copy operation (`mcpy`) where each memory word is read from the source address and written to the destination address. We could then do a separate m2m operation to get the data to the address from which we want to do the dump operation. Alternately, after the initial memory fill operation, we could choose to transfer the data over to the peripheral (`m2p`), but then we need to transfer the data back to the memory (`p2m`) before we can dump the data from the memory. Notice that the `m2p` and p2m operations require the parallel execution of complementary operations on both design elements. The third scenario shown is to load the data originally into the peripheral, followed by the `p2m` operation prior to the dump.

In addition to the test writer having to conceive of these scenarios, each one requires the test writer to keep track of the individual transactions to make sure that the output of one transaction is the input to the next. Even for this relatively simple example, there could be a lot of details to keep track of to ensure that each scenario is correct.

The other thing the astute reader will notice is that each scenario described above is essentially a "directed-random" test. The elements of each transfer, such as source, destination and size, could be randomized, but the overall scenario is the same so that, even if the same test were run with different random seeds, the set of scenarios would be the same albeit with different data characteristics of each transfer. In this case, there may also be some scenarios that use the `m2m` operation while others use the `mcpy` operation, but each of those operations has the same type of inputs and outputs so it's not difficult to add an `if` statement in the scenario. It is much more difficult to randomize the entire scenario in UVM.

The test writer could start with an `if` statement to begin the scenario with a memory fill or a peripheral load. It rapidly becomes untenable to specify the follow-on transactions based on that choice, especially if there are more than just two choices at any particular node. Trying to write a single UVM virtual sequence with the required nesting of if or case statements is extraordinarily difficult.

Modeling the Tests in PSS

PSS on the other hand makes this task much easier. A PSS model begins by defining the set of actions that must be supported, and the data flow requirements for them. For the memory operations, the data communication will be modeled using a PSS buffer data flow object, which is similar to a struct type in SystemVerilog. We then define the memory fill and dump actions as output or input of a buffer object of that type, respectively, as shown in Example 4. Similarly, we can also define the `m2m` and `mcpy` actions that each input and output the corresponding buffer type.

```
struct cb_struct {
    rand bit [ADDR_WIDTH] addr;
    rand bit [ADDR_WIDTH] size; // number of bytes
    rand bit[8] step;
}
buffer cb_mbuf : cb_struct {
    rand MemKindE kind; // RAM Type (DRAM or SRAM)
    rand bit[32]  alignment; // Data alignment in the associated memory location
    constraint alignment >= 64;
}
action cb_mem_fill {
    output cb_mbuf obuf;
    constraint c1 {obuf.size in [16..64];}
    constraint c5 {obuf.step == 0;}
}
action cb_mem_dump {
    input cb_mbuf ibuf;
}
action cb_mem_xfer {
    input cb_mbuf ibuf;
    output cb_mbuf obuf;
    rand int in [0..DMA_CHANNELS-1] chan;
    constraint c1 {obuf.size == ibuf.size;}
    constraint c2 {obuf.step == ibuf.step+1;}
}
action cb_mem_m2m : cb_mem_xfer { }
action cb_mem_mcpy : cb_mem_xfer { }
```

Example 4: Simple Memory Actions and Buffer Type

In Example 4, we can see that we start with the `struct` element `cb_struct`, which defines the `addr` and `size` fields, which will be common to other types that will be derived from `cb_struct`, such as `cb_mbuf`, which adds the `kind` and `alignment` fields. The `step` field will be used to track how many transfers actually occur. The constraints on this field in `cb_mem_fill` and `cb_mem_xfer` allow this tracking.

The `cb_mem_m2m` and `cb_mem_mcpy` actions are both derived from the `cb_mem_xfer` base action type, and each action inputs and outputs a `cb_mbuf` object, with the added constraint that the output and input buffers are constrained to have the same value for `size`. At the abstract model level, these two actions are identical, but we create two separate action types so that we can later add different realization layers for each.

Although not shown, a similar set of actions is defined for the peripheral, including the `m2p` and `p2m` actions, which share a data flow object type with the (not shown) `m2p` and `p2m` in the memory. If the solver chooses to infer a `p2m` action to supply the buffer input to `cb_mem_dump`, then the complementary `p2m` action must also be inferred to be traversed concurrently by the peripheral, and then that action will require the inferencing of either `load` or `m2p` to supply its input as shown in Figure 5. The same model can obviously produce many more scenarios than just those shown in the Figure 5.

The PSS language was designed specifically to define stimulus scenarios, so the semantics include several features that make it easier to create multiple scenarios from a single model. Perhaps the most useful of these features is the concept of *action inferencing*. The semantic rules define that if an action inputs a buffer type, there **must** be corresponding action that outputs a compatible buffer type. Further, if the model includes both a producer and a consumer of the same type, then the output of the producer is automatically bound to the input of the consumer. If we consider the scenarios illustrated in Figure 5, the implications for this are staggering.

Even if we limit ourselves to just the memory operations, without yet considering the peripheral operations, we can "begin with the end in mind" by simply defining our PSS model to simply traverse the `mem_dump` action, as in Example 5.

```
action pss_top {
    activity {
        repeat (4) {
           do cb_mem_dump;
        }
    }
}
```

Example 5: Simple PSS Model to Create Multiple Scenarios

In this model, for each traversal of the `cb_mem_dump` action, the PSS tool will need to infer an action that can provide the input `cb_mbuf` object. If the the `cb_mem_fill` action is inferred, then we're done because that action doesn't require any additional input. However, if the `cb_mem_m2m` or `cb_mem_mcpy` actions are inferred, then we'll have to infer another action to supply its input buffer type, for which we are presented with the choice of inferring one of the same three action types. Eventually, each scenario must start with `cb_mem_fill` since that is the only action that produces the correct buffer type without requiring an input.

Thus, with a very simple test specification, a PSS tool can automatically create a large number of scenarios and manage the data connections and scheduling constraints between them. PSS also provides a mechanism to specify a target-specific implementation for each action, but that is beyond the scope of this paper since Behavioral Coverage focuses only on the generated scenarios, not how the scenarios are implemented.

The user can limit the number of inferences in the chain and can choose to add additional fields and constraints to the model to ensure that a minimum number of operations is inferred. We can also add a few more details to the model to, for example, avoid the simple fill/dump scenario, as shown in Example 6.

```
action pss_top {
    constraint cstep {mdump.ibuf.step == 2;};
    activity {
        repeat (4) {
             select {
                do cb_mem_p2m with {obuf == mdump.ibuf;};
                do cb_mem_m2m with {obuf == mdump.ibuf;};
```

```
            }
            do cb_mem_dump;
        }
    }
}
```

Example 6: PSS Model to Add Complexity

The constraint `cstep` requires that there be two transfers before the `cb_mem_dump` action may be traversed. For example, since the constraint `c5` in `cb_mem_fill` (Figure 4) forces the value of `obuf.step` to be 0, and each `cb_mem_xfer` variant increments the `step` field, there must be two `cb_mem_xfer` actions traversed before the `cb_mem_dump`. It's not shown, but each `p2m/m2p` combination also increments the `step` field, so, before `cb_mem_dump` is traversed, the solver will choose either `cb_mem_m2m` or `cb_mem_p2m` to traverse, and this choice will cause another action traversal to be inferred to increment the `step` field. Ultimately, the originating action, either `cb_mem_fill` or `cb_periph_load` will be inferred, which sets the value of `step` to 0.

In this example, we are running a regression of simulations, each of them runs with a different seed value, this will ensure different starting point for each simulation, and then creation of different scenarios randomly. After applying the proposed methodology, we found that, given the constraints discussed above, there are 16 unique scenarios that can be generated from the PSS model in Example 6. After running 200 simulations, each with a different seed. Figure 6 shows the distribution of these scenarios using the proposed methodology.

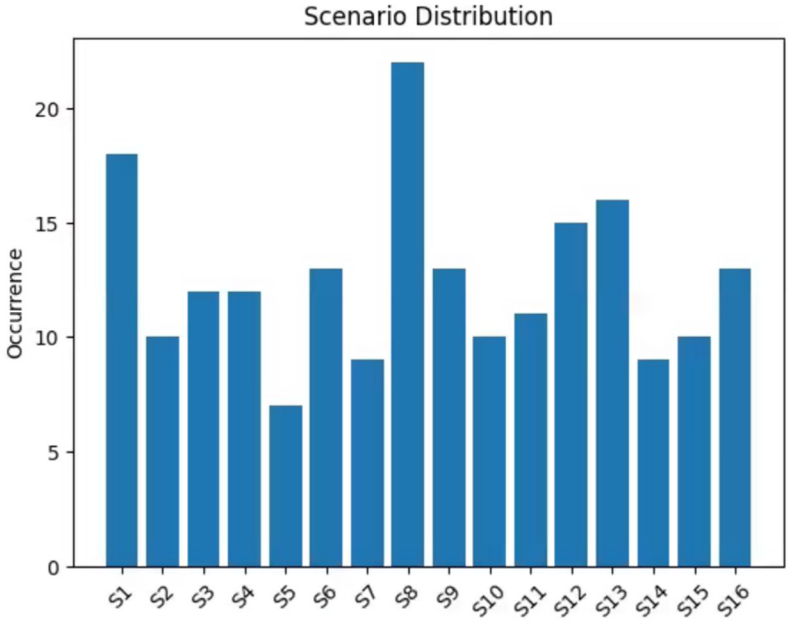

Figure 6: Scenario distribution quality of generated random scenarios for DMA example

Now, let's measure the behavioral coverage, Example 7 shows a PSS cover statement which is added to measure the behavioral coverage of a critical actions.

```
    c1: cover {
        activity {
            do cb_mem_c::cb_mem_mcpy;
            do cb_mem_c::cb_mem_m2p;
        }
    }
```

Example 7: Behavioral coverage for DMA example

Figure 7 shows the occurrences of behavioral coverage after applying the proposed methodology to measure behavioral coverage of the specified 'c1' cover statement after running 200 simulations.

Extracting Behavioral Coverage Defined in PSS Model ...

cb_exec_ex.pss[78] cb_vip_nobind_ex_a::c1: cb_mem_mcpy, cb_mem_m2p

Occurrences of defined Behavioral Coverage: (cb_mem_mcpy, cb_mem_m2p) - Occurrences: 27

{inferred_cb_mem_fill,inferred_cb_mem_mcpy,parallel{inferred_cb_mem_m2p,inferred_cb_periph_m2p},parallel{inferred_cb_periph_p2m, p2m}, mdump}

{inferred_cb_mem_fill,inferred_cb_mem_mcpy,parallel{inferred_cb_mem_m2p,inferred_cb_periph_m2p},parallel{inferred_cb_periph_p2m, p2m}, mdump}

{inferred_cb_mem_fill,inferred_cb_mem_mcpy,parallel{inferred_cb_mem_m2p,inferred_cb_periph_m2p},parallel{inferred_cb_periph_p2m, p2m}, mdump}

{inferred_cb_periph_ldmem,parallel{inferred_cb_mem_p2m,inferred_cb_periph_p2m},inferred_cb_mem_mcpy,parallel{inferred_cb_mem_m2p, inferred_cb_periph_m2p}, parallel{inferred_cb_periph_p2m, p2m}, mdump}

…

Figure 7: Snippet from the behavioral coverage report generated by proposed methodology

Please note that the behavioral coverage report shown in Figure 7 is not showing the complete list of occurrences. As shown the occurrences include all the inferred actions that are required to complete the scenarios, check actions with "*inferred_*" prefix.

VII. Conclusion

With the recent increase in SoCs complexities, PSS models are getting bigger and requiring covering all the required verification scenarios of the different verification levels, block, sub-system, and full system. Hence, there is a requirement to develop methodologies and frameworks that can help verification engineers to assess and measure the quality and coverage of the generated PSS scenarios used to verify such SoCs.

This paper introduced a methodology to measure the generated PSS scenarios with different metrics. The paper showed as assessment of the overall generated scenarios, the total number of unique scenarios and the distribution quality of the overall generated random scenarios. Also, the methodology explains a post-processing technique to measure the PSS3.0 behavioral coverage of specified critical verification intent.

VIII. References

[1] Accellera, "Portable Test and Stimulus Standard Version 3.0", https://www.accellera.org/images/downloads/standards/pss/Portable_Test_Stimulus_Standard_v3.0.pdf

[2] "Einstein's Puzzle", https://www.brainzilla.com/logic/zebra/einsteins-riddle/

[3] Questa – Siemens EDA product family, https://support.sw.siemens.com/en-US/product/852852103/download/202411017

[4] Accellera, "Standard Universal Verification Methodology", https://www.accellera.org/downloads/standards/uvm

[5] IEEE Standard for SystemVerilog--Unified Hardware Design, Specification, and Verification Language, https://standards.ieee.org/ieee/1800/7743/