

Improvement of UVM Scenario Generation, Control and Reproducibility using Portable Stimulus (PSS) for IP Validation

Robert R. Martin
Intel Corporation
1900 Prairie City Road,
Folsom, CA, 95630

Alan M. Curtis
Intel Corporation
1900 Prairie City Road,
Folsom, CA, 95630

Gopinath L. Narasimhan
Synopsys
690 E Middlefield Rd,
Mountain View, CA, 94043

Qingwei Zhou
Intel Corporation
1900 Prairie City Road,
Folsom, CA, 95630

Abstract - In recent years, validators have tried to augment their UVM test-suites by employing the Portable Test & Stimulus Standard (PSS) methodology where the main advertised benefit was translation of stimulus across different levels of validation (LOVs). 'Portability' has been the primary focus in creating random stimulus to hit platform agnostic DUT coverage. While these features are popular, they aren't the only advantages. Conventional validation test-suites which are often hand-written and rely heavily on UVM sequence libraries, suffer from two problems (1) Test reproducibility, due to test and testbench changes over time & (2) Lack of indicators of test paths and total actual validation space covered. This paper highlights a PSS Proof-of-Concept (POC) performed on the Intel Graphic's Memory fabric verification framework. We investigated how the following never-done-before features can be realized in a UVM environment: (1) Use PSS as a test configuration manager or to generate interesting test scenarios (2) Iterative stimulus coverage improvement (via a coverage report analysis without running DUT model elaboration / simulation) (3) The best PSS generated tests (high coverage / other criteria) selected for reproducibility. This is a huge deal as the same seeds run over the course of RTL development may produce varied results if the testbench, test or constraints change.

PSS constraints are defined in a Domain Specific Language (DSL) with C++ / SV-like syntax. Stimulus intent can be modeled with scenario entities such as 'actions' & 'flow objects'. Actions may be atomic or instantiate other actions & traverse them via activity blocks. Typically, atomic actions are mapped to UVM sequences & tasks respectively to drive hardware interfaces. The PSS scenario model, its constraints and mapping information are passed to a novel Synopsys PSS tool - VCPS. The tool picks valid paths "P" for each scenario and can be made to generate random cycles "R" per path. Users specify the total number of tests "N". The tool generates transactions depending on the DSL constraints.

$$\text{Total number of transactions (T)} = N \text{ tests} * (P \text{ paths / test}) * (R \text{ transactions / path}).$$

All these new PSS tests can be included along with legacy UVM tests and elaborated together along with the DUT. They add interesting scenarios to the existing UVM validation test plan and can be selected to run later, on demand. These deterministic, scenario-specific tests ensure consistent behavior across the project timeline. The PSS model generates stimulus pre-simulation aka PSS solve time. This way the stimulus is not influenced at run-time by seeding the simulator's constraint solver. This solves problem #1 - stimulus reproducibility. PSS augments the IP test coverage by generating stimulus using two modes of abstraction: (1) Unique PSS Tests: The DSL Scenario Generator directly schedules unique solutions of transactions that the existing UVM framework converts into hardware bus cycles. (2) Hybrid Randomization: PSS control knobs pass info to their UVM counterparts, which control sequence variables to drive target interfaces. Both modes create a coverage report of PSS test stimulus via user written DSL covergroups. If unsatisfactory, the DSL constraints were iteratively tweaked for better results. This solves problem #2 - test-suite validation space coverage. This is all done pre-simulation in a fraction of the time compared to post-simulation coverage analysis. For example, a test with 1 path & 1 random cycle took 1.79 seconds for PSS compilation / elaboration / linking & solve time while in comparison a test with 100 paths & 100 random cycles per path (10K transactions) took only 27 seconds! So, after just 27 seconds a 10K test can be analyzed for coverage holes even before a simulation is run saving hours depending on the LOV. This reduces overall compute cost resulting in huge cost savings! Thus, a coverage-based golden test can be run throughout the project lifecycle to ascertain RTL health. This is crucial for continuous integration code storage systems like Gatekeeper. It provides deterministic behavior, eliminates randomness in simulations, facilitating fixed runtimes, saving a lot of Intel \$\$\$ (spent in extra compute, high-mem machines & failures). Using these two modes not only were we able to achieve 100% stimulus coverage (aka PSS scenario coverage) but we also achieved 100% functional coverage (typically SV, hardware or traditional cover groups and crosses). In addition, DSL action knobs were also mapped to UVM framework knobs that controlled traditional validation features such as idle delays, uvm config object modifications, cycle out of order-ness, clock & mid-test-reset testing, memory map loading/dumping and uvm callbacks. DSL also allows powerful aspect-oriented programming concepts (code extension) beyond traditional OOP, great for sharing stimulus across environments / teams.

PSS tool can enable user queries for pattern or path coverage on the tests generated from user's PSS scenario model. In the near future, any design engineer, validator or architect can obtain confirmation if a required scenario was generated in the PSS tests by sending assertion style queries to the tool. The tool will also report which generated test satisfied the query, which can be goldenized for feature specific testing! To add icing on the cake, machine learning algos could also create PSS tool queries from past regressions truly revolutionizing how content is generated, analyzed and deployed on future products. This successful POC proves that the industry has only begun to scratch the surface with what can actually be accomplished with PSS unleashing the true potential of this methodology.

I. INTRODUCTION

Over the past couple of years, the widespread use-model and advertised features of the *Portable Test and Stimulus Standard* (or PSS) has been the translation of stimulus across different levels of validation. Common stimulus is used to validate RTL across multiple units, clusters, pipes, fullchip simulation and finally emulation. In other words, its namesake, the ‘portability’ aspect has been given a lot of attention. Rightly so, there are clear benefits to re-use the stimulus across simulation/emulation [1] and to use the random stimulus generation to hit platform agnostic coverage of the underlying RTL [2]. However, while these capabilities seem to be the most popular, they are definitely not the only ones. The industry has only begun to scratch the surface with what can *actually* be accomplished with PSS.

This paper highlights the use of PSS during Intel’s graphics IP development. The graphics *Memory Fabric Interface (MFI)* verification environment was used as a proof of concept (POC) to investigate how PSS can be used as not just a random stimulus generator, but also as a *Scenario Generator* or a *test configuration manager* and a showcases reproducible stimulus using coverage. We used a brand-new Synopsys Tool **VCPS** (*Verification Continuum ‘Portable Stimulus’*) [3] [4] to implement and test PSS features and execute MFI IP testplan scenarios.

The flow implements control knobs in PSS that can be applied to either UVM variables or knobs which in turn control the actual flow of a uvm test. In other words, SV uvm constraints, which provide random distributions per *test seed*, are replaced by PSS constraints to determine possibilities of UVM *unique test scenarios* that can be inserted directly into the verification test-suite. With this concept rather than use PSS’ random stimulus nature and hope that some of the seeds might hit an interesting scenario, we program rules into PSS and the tests that VCPS creates generate the ideal conditions required by the spec.

Finally using coverage-analysis, a set of ideal tests can be picked for reproducibility which can replace legacy UVM tests. As project development warrants TB and uvm test modifications, a certain seed will not retain the same functionality across the program. So, a passing-test “A” with seed 1 at the end of a project does not guarantee that the scenarios of this test “A” with seed 1 at project start was the same. Since PSS scenarios are not seed based, (and are static) using coverage, the best golden test can be selected to be used across validation milestones to determine RTL health and ensuring backwards test compatibility. This is crucial when applied to continuous integration systems such as gatekeeper-code-controlled repositories. It provides deterministic behavior and eliminates randomness in daily-simulations providing a consistent turn-in timeline.

II. THE PSS ENVIRONMENT SETUP AND FLOW

The *Portable Test and Stimulus Standard v2.0* specification [5] contains the *Domain Specific Language* [6] (DSL) LRM used for test modeling and scenario description. Fig. 1 shows the PSS ‘test generation flow’ steps that demonstrate test control and reproducibility.

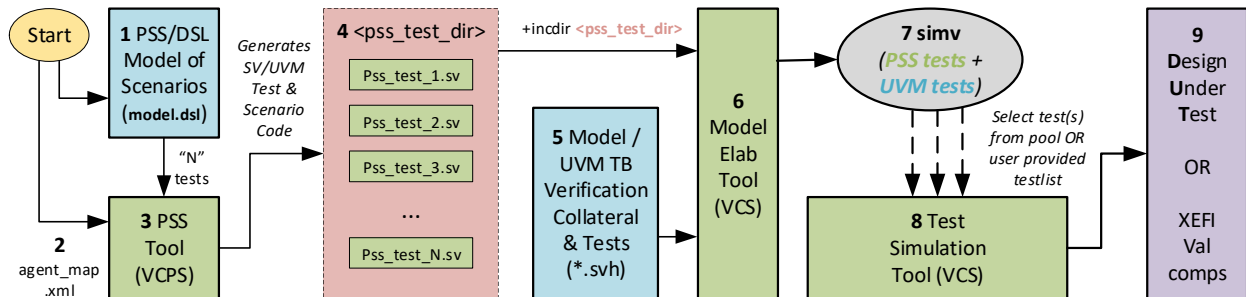


Fig. 1: PSS Test Generation Flow

1. The flow starts with test scenarios described in DSL (**model.dsl**. Refer Fig. 2). They are a list of actions that can be scheduled either in parallel or sequential. Actions in PSS/DSL are analogous to UVM sequences. Compound actions are like virtual sequences and atomic actions are like sequences or sub-sequences.

Note: In UVM, the `uvm_test_top` component specifies the sequencer on which the test sequence must run. Likewise, in DSL, the highest hierarchy component - "`pss_top`" needs to be able to map its actions to run on real objects. These `pss` objects are called "resource pools" and must be mapped to `uvm` sequencers. In Fig. 2 the `pss_top` component declares multiple resources which will be mapped to `uvm` sequencers in Step 2.

```

1 // -----
2 // Filename: model.dsl
3 // -----
4
5 // Top level PSS Component
6 component pss_top {
7
8   // PSS Package Import (contains global struct & type declarations)
9   import mfi_pss_pkg::*;
10
11   // Declare DSL/PSS Resources (types below are defined in mfi_pss_pkg)
12   // These Resources will be mapped to real UVM Agent Sequencers
13   pool [1] mfi_virt_seqr_res mfi_vseqr_res_pool; // One Resource 'A'
14   pool [3] mfi_seqr_res      mfi_seqr_res_pool; // Three Resources 'B'
15   pool [1] mfi_clk_res       mfi_clk_res_pool; // One Resource 'C'
16
17   // Bind resource pools to all actions within component
18   bind mfi_vseqr_res_pool *;
19   bind mfi_seqr_res_pool *;
20   bind mfi_clk_res_pool *;
21
22   // Top level Root Action definition (invoked by tool to generate PSS test)
23   action pss2uvm_action {
24
25     activity { // Activity definition
26
27       parallel { // Schedule Parallel Actions
28
29         do clock_action; // Controls TB Clock Sequence
30
31         sequence { // Runs actions below 'sequentially'
32
33           do pss_test_config_action; // PSS/UVM Test config/init (Optional)
34
35           do mfi_compound_action; // <- Testplan Scenario (MFI VIRT SEQ)
36
37           do mfi_atomic_action; // <- Testplan Scenario (MFI SUB SEQ)
38
39         } // sequence
40
41       } // parallel
42
43     } // activity
44
45   } // action
46
47 } // component
48
49 // -----

```

Fig. 2: PSS/DSL Model of Scenarios (Filename: model.dsl)

2. Next, an **agent_map.xml** file is defined to map a resource pool to a `uvm` sequencer. Fig. 3 depicts a mapping file where 3 sets of resource pools - `mfi_virt_seqr_res`, `mfi_seqr_res` and `mfi_clk_res` need to be mapped to their corresponding UVM sequencers. (Each PSS resource is assigned an `instance_id` that can be looked up in the `config_db` to complete the mapping to a UVM sequencer)

```

1 // -----
2 // Filename: agent_map.xml
3 // -----
4
5 <mapping>
6   <sv>
7     <executors>
8
9       <resource id="0" type="mfi_pss_pkg:mfi_virt_seqr_res" // 1 VSEQR 'A'
10
11       <executor uvm_path="uvm_test_top.env.mfi_virt_seqr" instance_id="0" //
12
13       </resource>
14
15       <resource id="1" type="mfi_pss_pkg:mfi_seqr_res" // 3 Agent SEQRe 'B'
16
17       <executor uvm_path="uvm_test_top.env.mfi_agent0.seqr" instance_id="0" //
18       <executor uvm_path="uvm_test_top.env.mfi_agent1.seqr" instance_id="1" //
19       <executor uvm_path="uvm_test_top.env.mfi_agent2.seqr" instance_id="2" //
20
21       </resource>
22
23       <resource id="2" type="mfi_pss_pkg:mfi_clk_res" // 1 Clock SEQR 'C'
24
25       <executor uvm_path="uvm_test_top.env.clk_agent.seqr" instance_id="0" //
26
27       </resource>
28
29     </executors>
30
31   </sv>
32 </mapping>
33 // -----

```

Fig. 3: PSS resource to UVM sequencer mapping (Filename: agent_map.xml)

3. The **model.dsl** scenario file, the **agent_map.xml** along with the **number of tests "N"** and runline switches are passed to VCPS. The tool-invoking runline command indicates what top level 'root_action' is to be executed to

generate testcases. From Fig. 2, the model.dsl contains the “pss2uvvm_action” used as the root_action. The tool interprets DSL actions defined and creates N uvvm_test extensions to allow sequences to be controlled via PSS.

4. All the **unique tests created by PSS** can be suitably stored in a common directory `<pss_test_dir>`.
5. Any existing **UVM collateral / tests** that PSS is intended to interface with/control is created and represented in this step.
6. The UVM framework (**Step 5**) and the contents of `<pss_test_dir>` (via `+incdir`) (**Step 4**) are now elaborated together using a standard compiler (VCS).
7. The output of elaboration is a simv object that contains any traditional **UVM tests** or any **PSS tests** passed in from Step 4.
8. Now that the user’s test suite contains all these flavors (pss & uvm), all that remains is to set the uvm runtime switch `+UVM_TESTNAME` to the intended test to run.
9. Existing traditional UVM tests if picked will be run as normal. If a PSS test is picked, then the scenarios depicted in **model.dsl** file are what get executed during this step.

With this basic framework in place, the agent sequencers that the DSL actions were mapped to (Fig. 3) will get the required items. If multiple actions are mapped to multiple sequencers, then the ordering picked by the DSL test is honored. This is how the MFI testplan (normally executed in UVM), can now have choreographed scenarios (DSL actions) that are equivalent to sequence layering scheduling.

III. SOLUTION MODE 1: THE PSS SCENARIO-GENERATION AND COVERAGE

All DSL actions are expected to have constraints on random variables. These constraints allow for coverpoint and cross coding constructs. VCPS, after code compilation, interprets the DSL scenarios and “solves” any constraints, rules, conditional logic and directed actions to create test scenarios. After the mapping file of action->sequencer is interpreted, a coverage urg report is generated. This crucial feature indicates the “solve/generation-action” coverage of the DSL action, ***without actually running a simulation***. The tool only takes a few seconds (Refer Table 1) to generate the scenario coverage which can be analyzed. If the analysis proves unsatisfactory, the DSL constraints, generation logic, number of random tests or cycles can be tweaked. This is done iteratively (Refer Fig. 4 & Table 2) until expected scenarios are hit.

Table 1: VCPS Tool Computation Time (seconds)

Num Tests	Random Cycles / Test	Total Random Cycles	Compile Time (s)	Elab Time (s)	Link Time (s)	Solve Time (s)	Total Generation Time (s)
1	1	1	.616 s	.172 s	.612 s	0.390 s	1.79
10	1	10	.620 s	.174 s	.603 s	0.800 s	2.20
10	10	100	.614 s	.180 s	.602 s	1.360 s	2.76
100	10	1,000	.619 s	.177 s	.609 s	6.690 s	8.10
100	100	10,000	.618 s	.179 s	.605 s	25.610 s	27.00

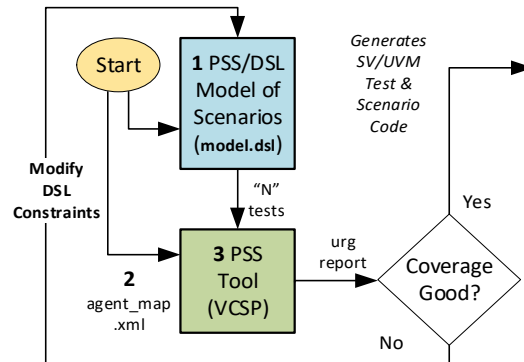


Fig. 4: PSS solve-time generated coverage analysis

Table 2: Iterative Coverage Analysis & Improvement

Iteration #1: PSS generation coverage for 10 tests

Summary for Group pss_top::base_fmt_a::fi_pkt_cg

CATEGORY	EXPECTED	UNCOVERED	COVERED	PERCENT
Variables	42	2	40	94.44
Crosses	480	396	84	17.50

Variables for Group pss_top::base_fmt_a::fi_pkt_cg

VARIABLE	EXPECTED	UNCOVERED	COVERED	PERCENT	GOAL	WEIGHT	AT LEAST	AUTO BIN MAX	COMMENT
CP_fmtid	10	0	10	100.00	100	1	1	1	0
CP_atype	12	0	12	100.00	100	1	1	1	0
CP_ctype	10	0	10	100.00	100	1	1	1	0
CP_chnl	2	0	2	100.00	100	1	1	1	0
CP_rdwrr	2	0	2	100.00	100	1	1	1	0
CP_dlen	6	2	4	66.67	100	1	1	1	0

Crosses for Group pss_top::base_fmt_a::fi_pkt_cg

CROSS	EXPECTED	UNCOVERED	COVERED	PERCENT	GOAL	WEIGHT	AT LEAST	PRINT	MISSING	COMMENT
CR_cmds1	480	396	84	17.50	100	1	1	1	0	

Iteration #2: PSS generation coverage for 30 tests (Improved)

Summary for Group pss_top::base_fmt_a::fi_pkt_cg

CATEGORY	EXPECTED	UNCOVERED	COVERED	PERCENT
Variables	42	1	41	97.22
Crosses	480	293	187	38.96

Variables for Group pss_top::base_fmt_a::fi_pkt_cg

VARIABLE	EXPECTED	UNCOVERED	COVERED	PERCENT	GOAL	WEIGHT	AT LEAST	AUTO BIN MAX	COMMENT
CP_fmtid	10	0	10	100.00	100	1	1	1	0
CP_atype	12	0	12	100.00	100	1	1	1	0
CP_ctype	10	0	10	100.00	100	1	1	1	0
CP_chnl	2	0	2	100.00	100	1	1	1	0
CP_rdwrr	2	0	2	100.00	100	1	1	1	0
CP_dlen	6	1	5	83.33	100	1	1	1	0

Crosses for Group pss_top::base_fmt_a::fi_pkt_cg

CROSS	EXPECTED	UNCOVERED	COVERED	PERCENT	GOAL	WEIGHT	AT LEAST	PRINT	MISSING	COMMENT
CR_cmds1	480	293	187	38.96	100	1	1	1	0	

For illustration, during the dev process, the cross-coverage (CR_*) urg report snapshots in Table 2 show an improvement in covered cases using the iterative cov-analysis process. Coverage improved when the number of unique tests generated by VCPS was increased from 10 to 30. Cross coverage improved from 17.5% to **38.96 %** with very little user effort. (Overall variable coverage also improved from 94.44% to **97.22%**)

Thus, any testplan scenario can be evaluated at generation time (in seconds) for a fraction of the cost.

(This is compared model elaboration and regression test run times!).

Note: VCPS determines the max number of legal paths (**num_legal**) that the DSL code implies. If a user requests to generate N tests where $N < \text{num_legal}$ then the tool generates N tests. If $N > \text{num_legal}$ then the number of tests will get capped at the **num_legal**. At this point coverage will not improve by increasing the number of tests/seeds. Fig 5 shows the VCPS solution graph outputs from the tool that represents the possible PSS scenario paths/trajectory that each generated PSS test can take. Again, since these paths are possible based off validation space, it is not seed based.

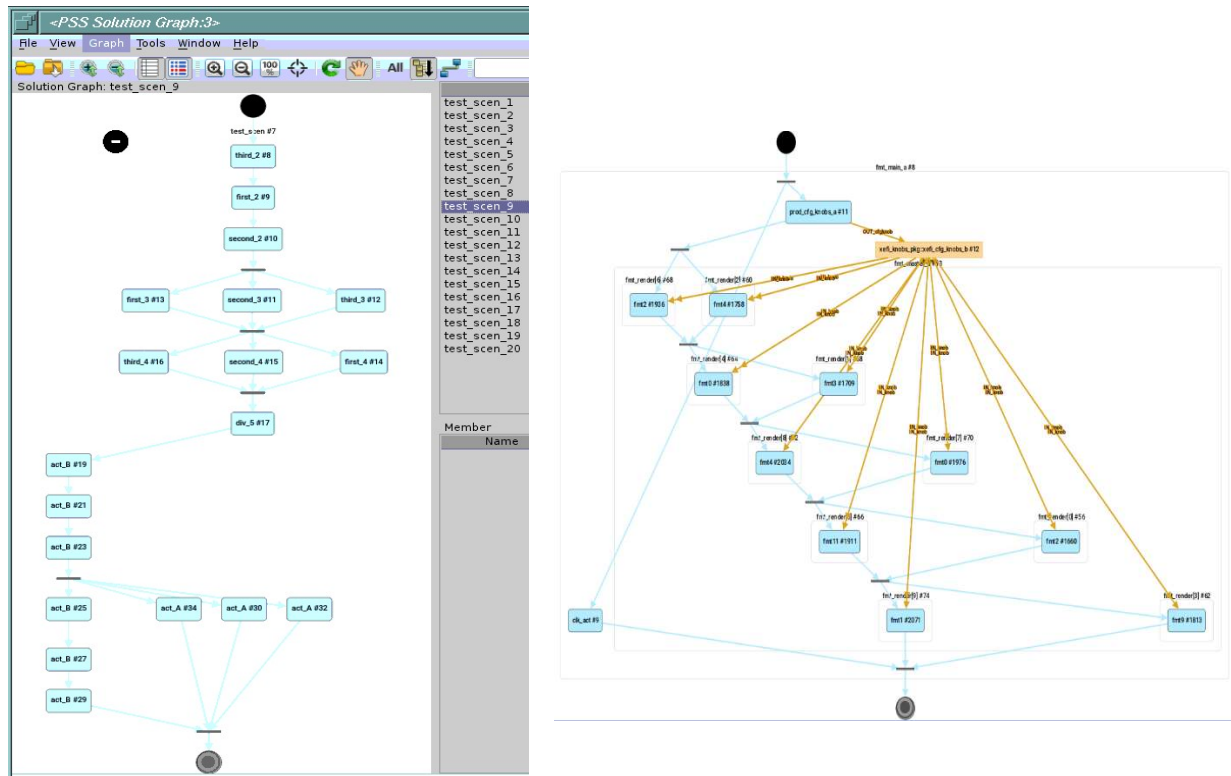


Fig. 5: VCPS Solution Graph: representing test paths and possibilities (with & without DSL flow objects)

IV. SOLUTION MODE 2: PSS- UVM HYBRID RANDOMIZATION

The **MFI** Validation testplan consisted of testing two libraries – a front-end UVM scheduler of traffic (MFI_uvm_pkg) and a back-end hardware implementation of the spec (MFI_val_common). Testing required injecting cycles of varying attributes from a UVM sequence to a UVM agent which would drive out a hardware library component. This emulation-friendly hardware library had components which housed a lot of internal fifos, arbiters and combinatorial logic. Cycles of varying DLEN's (data lengths) would arrive on multiple command and data channels creating different fifo (full/almost-full) conditions received from UVM. Later, arbitration of these cycles across channels had to be done which would determine a winner to be sent to the RTL. These cycles were also back-pressured by RTL based off credits.

A robust validation platform was created which could send in all possible cycles. This required 100% coverage of back-to-back cases of all types in the hardware backend library to ascertain accurate functionality. While the UVM sequence generation was successful in generating controlled traffic, the results were not fully deterministic (based off seed) and lead to a lot of wasted cycles without much coverage improvement. Visual analysis of logs and sequence code indicated that an expected golden series of cycles should be realized, but varying hardware conditions, arbitration and delays across parallel channels didn't accurately reflect the intent. This coupled by random seeds and behavior almost never generated the optimal distribution expected from such an environment. This is where PSS testcases, that were uniquely generated to perform a certain sequence of events, came to the rescue!

Fig. 6 depicts how DSL knobs/variables (randomized by actions) and the starting of sequences can get randomized or scheduled / choreographed into the PSS test code. This in-turn would control their UVM knob / variable counterparts. The DSL code employed control knobs (KA_p , KB_p , KC_p , KD_p) and variables ($V1_p$, $V2_p$, $V3_p$) which were either randomized completely or constrained based on the MFI spec. These knobs were then configured to either control sending in sequences of different types or would in-turn pass on their settings their UVM knob counterparts (KA_u , KB_u , KC_u , KD_u) and variables ($V1_u$, $V2_u$, $V3_u$). The **big difference** here is that the UVM knobs (normally random in a purely uvm setup) now received a pre-determined biasing per test mapped and randomized from DSL actions. This was not per seed but rather per unique PSS test. So, if the same test was run with different seeds the order of cycles on the UVM side wouldn't change as the stimulus scheduling came from PSS.

V. PSS TEST REPRODUCIBILITY

The “PSS Generated Test” from Fig. 6 is an example of any of the “pss_test_*.sv” tests from Fig. 1 that are generated by VCPS. This test, now-pregenerated can be run as part of the normal regression suite on the UVM side. Each week, VCPS could be run to generate new tests and they could be used to produce unique (and deterministic) flavors previously statistically (but not deterministically) possible with purely randomized UVM tests.

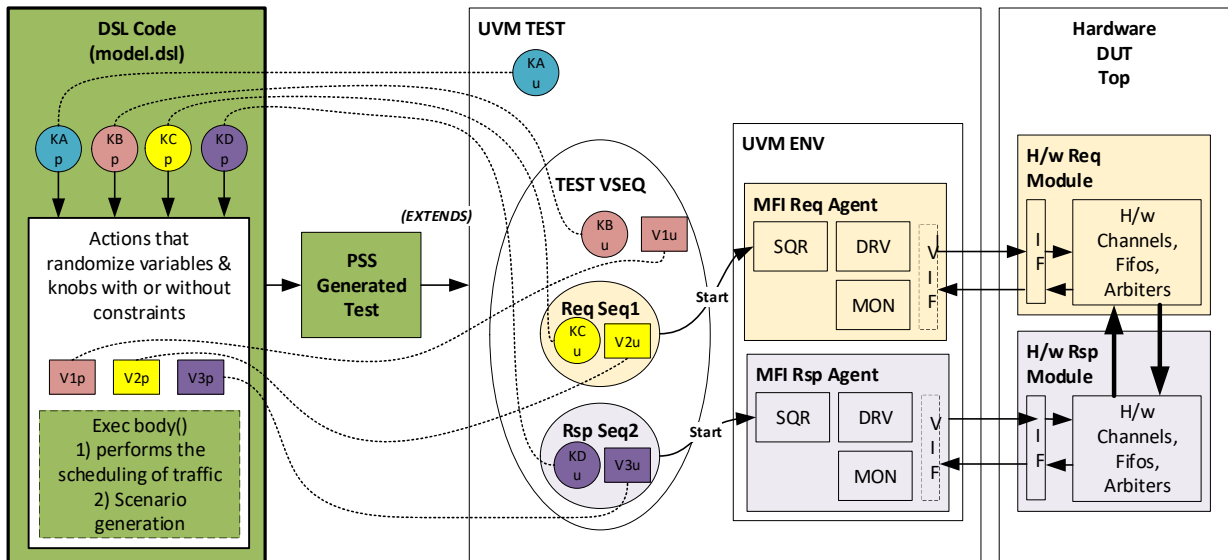


Fig. 6: PSS knob & variable mapping to their UVM counterparts

Hardware coverage can be collected using VCS and coverage urg reports can be generated. When certain important features need dedicated tests, the PSS regression coverage can be analyzed to find a golden-test that fits the scheduling of traffic for the feature. This golden test (let's say *pss_test_3.sv* run from a weekly regression) can now be separated from the pss generated tests and moved into the traditional UVM testsuite. This test can now be run at any point in the future and it will always generate the same stimulus giving the uvn framework an automatically generated focused test that is feature specific. As mentioned in the introduction, a test like this will be deterministic across all milestones as it is scenario-based and not seed-based.

This is how PSS tests can be implemented as reproducible UVM tests. If such a set of scheduled PSS tests could be picked after coverage analysis, then they be used in a Continuous-Integration systems like gatekeeper. This way the Turn-in (TI) process will execute not a random UVM test with a fixed seed but a feature-based, deterministically scheduled/sequenced test with only random attributes such as delay or flows that do not alter the feature expectation.

This reduces randomness during the TI process allowing for a deterministic gatekeeper-turn-around compute-time saving a lot of Intel \$\$\$ that are currently spent in extra compute, high memory machines, tool-licenses and regression failures.

VI. RESULTS: MFI TESTPLAN IMPLEMENTATION, STIMULUS CONTROL & COVERAGE

The MFI testplan required a robust constrained random UVM testsuite. This was made possible by leveraging powerful PSS features. The UVM test scenarios defined in the validation test plan were implemented using PSS/DSL control knobs (*efficient control*) producing deterministic tests (*scenario generation*) that can be re-used for feature specific testing (*reproducibility*). The generation time stimulus urg coverage reports were a very good indication of scheduled traffic to the MFI validation framework. This was tweaked very quickly, and coverage improvements observed without the need for model-elaboration and large regression simulation times (Table 1). This is a huge benefit. Hard to hit scenarios can now be easily devised without sacrificing uvm randomness.

The MFI libraries also required to provide other validation capabilities to the GFX and Media validation teams. These features, part of the validation test plan were also verified using PSS. Some of novel ideas summarized here were never done in the industry using portable stimulus. Actions were written to control UVM knobs for – *Idle delays*, mid-simulation *config-object modification*, out-of-order enable/disable, clock and reset modification, *mid-test-reset* issue and recovery, MFI *memory and map loading/dumping* and mid-simulation MFI/UVM *callback* inclusion were all tested. These could be choreographed on top of any random sequences that the UVM test was already calling for MFI cycle attribute verification. The PSS test could add any choreographed functionality before, during or after the existing UVM test it was extending, proving to be an invaluable additional to a traditional UVM verification framework.

















































Technical Comparison of Coverage: Our prior traditional UVM testplan approach (non-PSS) that was manually coded, analyzed, maintained and regressed over time eventually resulted in 100% functional coverage. This was a result of a many weeks of UVM development, test writing, coverage analysis, coding and regressions and 1000s of seeds. We then repeated our functional coverage experiments using the Portable Stimulus tests from Solution Modes 1 and 2 and achieved the same results in a fraction of the time with a reduced number of tests. Refer to Table 3.

The main-takeaway is that none of the PSS tests were hand-coded but rather were auto-generated from the tool. In the future if the constraints are defined and functional coverage indicators are available, test writers do not need to spend hours creating constrained random tests to manually bring up coverage. This is a huge benefit. To normalize UVM vs PSS results we reran 300 UVM test instances vs 300 PSS tests.

The 300 UVM tests: were 3 hand-written unique test templates developed, run with 100 associated configurations by way of seeds (ie 3x100 tests). The 300 PSS tests: were automatically generated in seconds by the VCPS tool from 1 DSL model with 300 unique random testcases (ie 1x300 tests). The 300 PSS tests showed better results as indicated in Table 3.

Thus the functional coverage obtained from the PSS tool (Results II) provided near comparable and slightly higher coverage than manually written UVM tests (Results I) in a fraction of the time. This time savings is many magnitudes of orders of time saving weeks of development time.

UVM vs PSS Functional Coverage Analysis & Comparison

Case I: UVM Testplan / Testsuite Coverage Results – I (Average = 98.34%) (Total tests = 300)	Case 2: PSS Auto-Generated Test Coverage Results – II (Average = 99.58%) (Total tests = 300)																				
Functional Black Box Coverage I(a)	Functional Black Box Coverage II(a)																				
<table> <tr> <td>ifi_bb_cov1::cg_cmd_pkt</td><td> 96.34%</td></tr> <tr> <td>ifi_bb_cov1::cg_comp_pkt</td><td> 100.00%</td></tr> <tr> <td>ifi_bb_cov1::cg_data_rsp_pkt</td><td> 100.00%</td></tr> </table>	ifi_bb_cov1::cg_cmd_pkt	 96.34%	ifi_bb_cov1::cg_comp_pkt	 100.00%	ifi_bb_cov1::cg_data_rsp_pkt	 100.00%	<table> <tr> <td>ifi_bb_cov1::cg_cmd_pkt</td><td> 96.76%</td></tr> <tr> <td>ifi_bb_cov1::cg_comp_pkt</td><td> 100.00%</td></tr> <tr> <td>ifi_bb_cov1::cg_data_rsp_pkt</td><td> 100.00%</td></tr> </table>	ifi_bb_cov1::cg_cmd_pkt	 96.76%	ifi_bb_cov1::cg_comp_pkt	 100.00%	ifi_bb_cov1::cg_data_rsp_pkt	 100.00%								
ifi_bb_cov1::cg_cmd_pkt	 96.34%																				
ifi_bb_cov1::cg_comp_pkt	 100.00%																				
ifi_bb_cov1::cg_data_rsp_pkt	 100.00%																				
ifi_bb_cov1::cg_cmd_pkt	 96.76%																				
ifi_bb_cov1::cg_comp_pkt	 100.00%																				
ifi_bb_cov1::cg_data_rsp_pkt	 100.00%																				
Functional White Box Coverage I(b)	Functional White Box Coverage II(b)																				
<table> <tr> <td>fi_req_to_arb_wb_cov1::req_to_arb_2ch</td><td> 99.80%</td></tr> <tr> <td>fi_req_to_arb_wb_cov1::write_arb_2ch</td><td> 90.61%</td></tr> <tr> <td>fi_rsp_protocol_layer_wb_cov1::protocol_layer_cmd</td><td> 100.00%</td></tr> <tr> <td>fi_rsp_protocol_layer_wb_cov1::protocol_layer_pop</td><td> 100.00%</td></tr> <tr> <td>fi_rsp_protocol_layer_wb_cov1::protocol_layer_state</td><td> 100.00%</td></tr> </table>	fi_req_to_arb_wb_cov1::req_to_arb_2ch	 99.80%	fi_req_to_arb_wb_cov1::write_arb_2ch	 90.61%	fi_rsp_protocol_layer_wb_cov1::protocol_layer_cmd	 100.00%	fi_rsp_protocol_layer_wb_cov1::protocol_layer_pop	 100.00%	fi_rsp_protocol_layer_wb_cov1::protocol_layer_state	 100.00%	<table> <tr> <td>fi_req_to_arb_wb_cov1::req_to_arb_2ch</td><td> 99.95%</td></tr> <tr> <td>fi_req_to_arb_wb_cov1::write_arb_2ch</td><td> 100.00%</td></tr> <tr> <td>fi_rsp_protocol_layer_wb_cov1::protocol_layer_cmd</td><td> 100.00%</td></tr> <tr> <td>fi_rsp_protocol_layer_wb_cov1::protocol_layer_pop</td><td> 100.00%</td></tr> <tr> <td>fi_rsp_protocol_layer_wb_cov1::protocol_layer_state</td><td> 100.00%</td></tr> </table>	fi_req_to_arb_wb_cov1::req_to_arb_2ch	 99.95%	fi_req_to_arb_wb_cov1::write_arb_2ch	 100.00%	fi_rsp_protocol_layer_wb_cov1::protocol_layer_cmd	 100.00%	fi_rsp_protocol_layer_wb_cov1::protocol_layer_pop	 100.00%	fi_rsp_protocol_layer_wb_cov1::protocol_layer_state	 100.00%
fi_req_to_arb_wb_cov1::req_to_arb_2ch	 99.80%																				
fi_req_to_arb_wb_cov1::write_arb_2ch	 90.61%																				
fi_rsp_protocol_layer_wb_cov1::protocol_layer_cmd	 100.00%																				
fi_rsp_protocol_layer_wb_cov1::protocol_layer_pop	 100.00%																				
fi_rsp_protocol_layer_wb_cov1::protocol_layer_state	 100.00%																				
fi_req_to_arb_wb_cov1::req_to_arb_2ch	 99.95%																				
fi_req_to_arb_wb_cov1::write_arb_2ch	 100.00%																				
fi_rsp_protocol_layer_wb_cov1::protocol_layer_cmd	 100.00%																				
fi_rsp_protocol_layer_wb_cov1::protocol_layer_pop	 100.00%																				
fi_rsp_protocol_layer_wb_cov1::protocol_layer_state	 100.00%																				

VII. CONCLUSIONS AND FUTURE PLANS/USE

Portable stimulus means a lot more than transposing content across different levels of validation or environments. There are powerful DSL constructs, aspect-oriented programming concepts that allow for code extension well beyond traditional object-oriented programming which is great for sharing stimulus across envs and teams. Stimulus generation times are now greatly reduced, and content can be ratified and perfected even without simulation, using coverage. Teams will benefit greatly by using highly deterministically scheduled UVM tests eliminating gross variation in test-run times saving compute \$\$\$.

Currently, VCPS provides an excellent supplement to any UVM environment but there is more coming! In the near future, the stimulus generation tool will also keep track of scenarios in a coverage database (covdb) automatically. Users will not need to code DSL coverage to interpret the scenario. Instead, an SVA (system verilog assertion) query-based system will allow any validator, designer or architect to query the stimulus database generated to obtain confirmation if a required scenario was generated. In addition, VCPS will also determine and report which generated test satisfied the input query so this can be goldenized. This is perfect for feature specific testing! To add icing on the cake this PSS tool capability integrated with queries that are the output of Machine Learning algorithms from past regressions will truly revolutionize how content is generated, analyzed and deployed on complicated ASIC products.

ACKNOWLEDGMENT

Special thanks to our technical colleagues from Synopsys: Bernie Delay and Hillel Miller who we collaborated on this PSS proof of concept, tool development and testing. Working with a brand-new methodology was exciting and cutting edge but it had its challenges along the way which we went past together. It was a rewarding experience to learn a new language DSL along with different validation BKM's to achieve UVM controllability. Their feedback and aid in realizing a UVM testplan via PSS was greatly appreciated and we look forward to seeing more mainstream adoption and standardization of the tool. We would like to also thank the Intel Graphics tools teams for providing the necessary back-end help (VCPS Installation, Synopsys licensing etc) so we could perform all our experiments.

REFERENCES

- [1] S. Vasu, N. Venkatesh, J. Maitra, Intel Corporation, "Media Performance Validation in Emulation and Post Silicon Using Portable Stimulus Standard", Paper ID 7067, DVCON 2021.
- [2] J. Maitra, V. Singh, "Low Power Validation of Heterogeneous SoCs Using Portable Stimulus Standard", ID 124.13, DAC 2019.
- [3] Synopsys VCPS datasheet - <https://www.synopsys.com/content/dam/synopsys/verification/datasheets/costart-portable-stimulus-ds.pdf>
- [4] Synopsys VCPS User Guide - https://spdocs.synopsys.com/dow_retrieve/latest/vg/vc_ps/PDFs/vcps_user_guide.pdf
- [5] PSS 2.0 Specification - https://www.accellera.org/images/downloads/standards/Portable_Test_Stimulus_Standard_v20.pdf
- [6] Domain Specific Language - https://en.wikipedia.org/wiki/Domain-specific_language