2020 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

Does It Pay Off To Add Portable Stimulus Layer On Top Of UVM IP Block Test Bench?

Xia Wu, Team Lead, Syosil ApS, Taastrup, Denmark (xia@syosil.com)

Jacob Sander Andersen, CTO, Syosil ApS, Taastrup, Denmark (jacob@syosil.com)

Ole Kristoffersen, Project Manager, Ericsson Lund, Sweden (ole.kristoffersen@ericsson.com)

Abstract*—The work presented is the outcome of deploying portable stimulus to an existing UVM block level test bench. The target is to verify a highly configurable filter chain system which has extensively used generics and flexible run-time configuration. Accellera Portable Stimulus Standard Domain Specific Language (PSS-DSL) is used to create the abstract PSS model. The paper investigates the effort to create a PSS based solution on a traditional UVM test environment, and if it contributes to better verification quality. We present five concrete challenges that we have experienced when creating a PSS model: Compile-time parameters, Run-time configuration, Inheritance, Partial Configuration and Semantics equivalence, and solutions are proposed to overcome or mitigate these challenges. In the end we conclude a few guidelines and consideration for future projects use.

***Keywords*— Functional verification; Portable Stimulus; SystemVerilog; UVM**

I. INTRODUCTION

Portable stimulus is in recent years becoming the emerging trend in functional verification world. Portable stimulus has shown its benefit of creating reusable tests which are constraint-randomized with the aim at fast coverage closure. By reusing the stimuli and test cases on different platforms, we also make sure that the people across the teams agree on the same requirements and test in a more uniform way. This creates a bridge across any target platform and is a key motivation of using PSS. Without doubt, many large ASIC companies are working towards integrating it into the standard ASIC methodology flow.

Accellera System Initiatives announced the release of the Portable Test and Stimulus Standard (PSS)[1] on June 2018. The standard provides a language to describe the test requirement and content in a highly abstract way. Due to the abstract and implementation-agnostic nature of the PSS language, early adoptions are mostly done and evaluated on the system level test bench. The connectivity test which verify the data flow between different subsystems are the main target. If we are aiming at reusing the test and stimuli on multiple platforms and test bench implementations, it makes also sense to start from block level tests. Because the subsystem and system test will most likely be built on a subset from the existing block level PSS implementations. It allows maximum verification reuse and improves verification readiness across platforms.

When we look through the recent publication and articles, we rarely find evaluation of portable stimulus on IP block level test bench. Is portable stimulus only suitable for high level test bench? This paper explores the effort to create a PSS based solution on a traditional UVM test environment [2]. This paper is related to the work presented in [3] but instead of starting to use PSS from scratch, it is based on an existing UVM test bench to examine if it is possible to create an abstract stimuli model for vertical reuse.

II. MOTIVATION

While PSS was only recently released, portable stimulus has been an existing industry solution for years. Various EDA vendors have announced several proprietary commercial solutions which address the portability challenge. We choose Cadence tool to conduct our experiment on. Our motivation is to find out what the effort and challenge is to migrate the UVM test bench into PSS-based solution.

PSS provides the control of activities with loops, branches, parallelism etc. Also, it introduces concepts like resources, data flow objects, constraints and more to allow the correct and automated creation of a scenario from a user's partial scenario definition. But does the PSS-DSL provide sufficient language support for the low level block test, which are mainly focus on the bus transactions? How efficient is the partial configuration in the block level

tests which have more details and constraints than the system-level test? And more importantly, how to ensure the semantics equivalence with UVM tests when migrating to PSS, as there are already existing tests?

This paper is aiming to answer these questions based on an evaluation of using PSS to model and test a configurable filter engine chain system. We have categorized the five major challenges we have met during our experiment, and discuss the solution or pitfalls for each of them in the following chapters.

A. Compile-time parameters
B. Run-time configuration
C. Inheritance
D. Partial description
E. Semantics equivalence

III. SYSTEM OVERVIEW

IP block details

Our target DUT is a highly configurable filter chain system, which can be viewed as a series of filter engine blocks, processing data independently or in chain as shown in Figure 1.

The number of the filter engine blocks are generics set in compile time. The way these blocks are cascaded are configurable in run time. They can be running independently (by taking the red paths), or in chain (by taking the blue paths), or in a mixed way. The existing UVM testbench verifies around 10 different generic combinations for the DUT, and for each combination there are different run-time configurations. Hence, a lot of test cases are needed in the regression to close coverage. Collecting functional and code coverage is usually tedious, repetitive work.

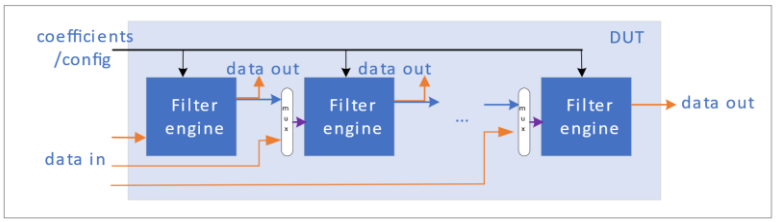

Figure 1 DUT is a highly configurable filter engine system

This DUT is carefully chosen to be the evaluation target, because

- The combination of the extensively used generics and highly flexible run-time configuration is in itself a challenge to model correctly in PSS.
- The benefit in closing coverage by using PSS is high.
- The complexity level of the UVM sequence is high. It requires the correct timing and stimuli. Therefore it shows how fine-grain the PSS control is.

Cadence's implementation of the PSS frame work

The evaluation project is based on the PSS tool from Cadence called Perspec. It is a modelling tool which can be used in batch or GUI mode. Perspec enables PSS model development, simplified scenario creation, scenario randomization, target code generation and abstract debugging. Perspec supports PSS C++ and DSL standard input formats from the official PSS LRM, as well as their own System Level Notation (SLN) language as a supplement.

While PSS and UVM are in two different language domains, we need a bridge to make these two parts interact with each other. This is also what we achieved by using Perspec, which through its automation provides the framework to link the PSS implementation to the traditional UVM test bench in SystemVerilog. As shown in Figure 2, we start with implementing a model in PSS-DSL language. Then we use Perspec to translate the PSS model into scenario code in C language. The scenario code controls the synchronization and timing of the

underlying UVM sequences by DPI calls. Parallelization are implemented as threads in C. A Perspec test which is uvm_test object is manually created in SystemVerilog to initiate all the threads defined in the C code.

Perspec test itself has no logic complexity implemented. It is only served as an initiator to the whole process and therefore can be shared by all PSS test cases that are created in this way. The control of activities all resides in the generated scenario code (C code). And the low level bus transactions are still taken care by UVM sequences. Since the Perspec test is a uvm_test object, it can be located directly in the UVM test directory as another UVM

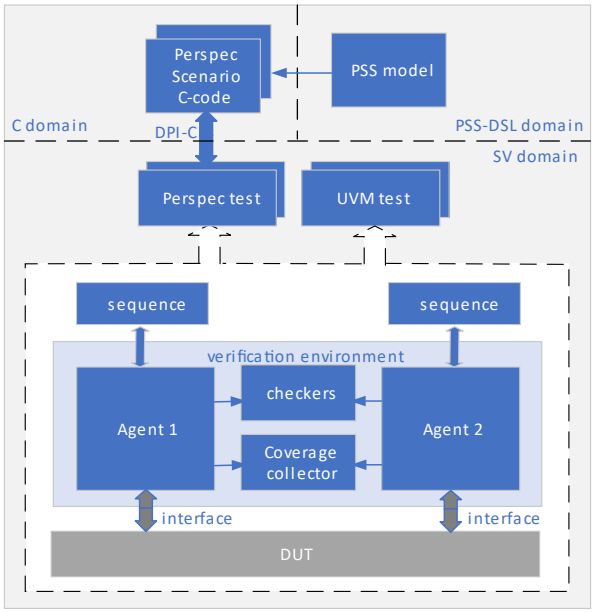

Figure 2 PSS as a layer on top of UVM PSS as a layer on top of UVM

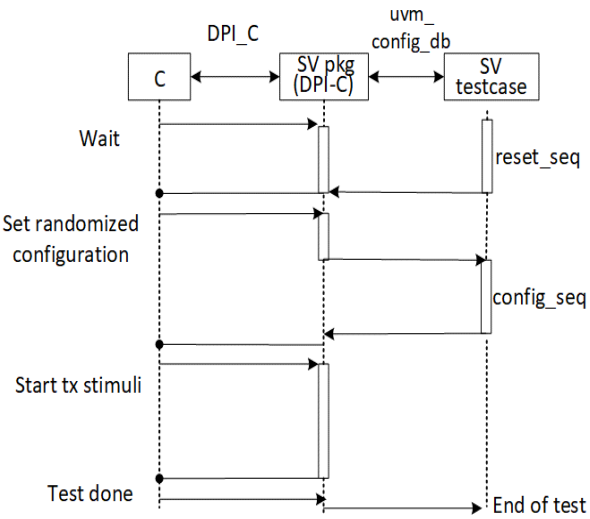

Figure 3 Interaction of different parts in PSS test bench

test case. All the existing UVM tests are still unchanged and are completely independent from the new portable stimulus layer. Therefore it does not introduce any back-ward compatible problem. The interaction of different parts is shown in Figure 3.

IV. Compile-time parameters

Modern generic IP blocks are in many cases reused in different platforms to reduce time-to-market and manufacturing cost. By setting the generics in different value, the same IP block can accommodate for different product portfolio. Thus, verifying the IP block with all valid combination of generics is crucial. Our target DUT is also highly generic. The UVM test bench has a SystemVerilog package with ifdef around each set of parameters. The regression runs all the tests on all the setups.

To migrate this into PSS, it is not a trivial task. The PSS model also need to be “generic” so that it can generate test cases based on each parameter set. However, the PSS-DSL language in PSS 1.0a does not provide support for this. Users have to find workarounds to implement the generics support. Based on our previous knowledge with Cadence SLN language, we know that it supports a table construct, which is a macro-like syntactic mechanism to capture code repetition. Table enables importing model parameters from external data files, and factoring out common model definition into shared code.

The parameters are defined in a comma-separate-values file(.csv). In SLN file, csv_to_table command choose a parameter set with a specific name. The parameters can be passed on as individual constants to PSS model. This is shown in Figure 4.

In this example, one can define all the needed parameter sets in the same table. By defining one parameter SETUP, the parameter set is selected and applied in the PSS model. Without this, it is still possible to manually code the generics in the PSS model. But when we hit the large amount of generics combinations, in our case 10,

the table construct simplifies the parameter set selection and avoid repetition in coding the model. We consider it as a very useful add-on to the existing PSS DSL specification.

CSV

```
#name, #nfilt, #nsec, #ncpsec, #wc,...
SETUP_DEFAULT,  6, 2, 4, 18,...
SETUP_2_3_8__0, 2, 3, 8, 17,...
```

SLN

```
#ifndef SETUP {
#define SETUP "SETUP_DEFAULT";
};
table from csv_to_table("../csv/setups.csv","Setup Info",(csv_column("name")
== SETUP)) with {
   const NUM_OF_CHANNELS: int = <#nfilt>;
   const NCPSEC_P: int = <#ncpsec>;
   const NUM_OF_SECTIONS: int = <#nsec>;
   const COEFFICIENT_WORD_WIDTH: int = <#wc>;
   ...
```

Figure 4 Table construct in Cadence SLN language

V. RUN-TIME CONFIGURATION

Another challenge of migrating the IP block test bench to PSS is to cover all the dynamic configurations of DUT. In our case, one must cover all the possible scenario of the chaining of the filter engines, ie. all the possible starting point and end point of a chain, as well as all the possible chain size. When the number of filter engine blocks increases, the permutation of the different filter chains also increases significantly.

In the existing UVM test bench, we define constraints of valid chain and rely on the randomization by running a high number of seeds in the regression to generate all the possible chain combinations. By ranking the seeds, we found a set of optimal scenarios which contributes to achieving the coverage goal.

One of the advantages to use PSS is that we can define the coverage goal beforehand and generate the scenario which directly cover that goal. With that in mind, our focus when creating the PSS model should be

1. How to abstract the test bench behavior into smallest possible actions
2. How to best possible layer the model so that we can distribute the complexity into each layer.

We have identified the following steps to solve the problem

1. Randomization should be one action per filter engine to keep the test bench scalable.
2. Utilizing the input/output data stream in action to model a virtual chain.
3. Defining variables in the action which directly links to the cover point, e.g. starting position, role, etc.

To model a single element in a chain, we define the randomization action called channel. The action need to know the starting point of the chain, the size of the chain, the current index, the previous element. Besides, it also need to pass down some design specific variable(`npsec_in_use`), which should be the same throughout the chain. These variables are defined as the data element which is passed between actions. Furthermore, a lock (`channel_r`) is created to ensure a unique id on each channel action.

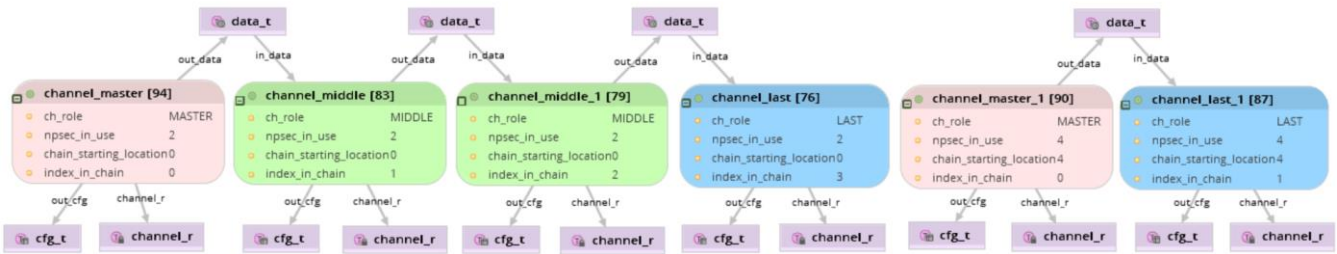

Figure 5 Example of modelling a cascaded filter chain

To reduce the complexity in coding, we extended the channel action according to its role in the chain and assign it to a variable called `ch_role`. The role can be master (first), middle, or last. Figure 5 shows an example of a cascaded filter chain, with the first 4 filter engines in one chain, and the last 2 filter engines in another chain. Figure 6 is an example of how to implement the middle channel.

```
stream data_t {
    rand int in [0..NUM_OF_CHANNELS-1] source;
    rand int in [1..NUM_OF_CHANNELS-1] step;
    rand int in [0..NUM_OF_CHANNELS-1] chain_starting_location;
    rand int in [1..NUM_OF_CHANNELS] chain_size;
    rand int in [1..NCPSEC_P] npsec_in_use;
};
action channel {
    rand role_in_ch_e ch_role;
    rand int in [1..NCPSEC_P] npsec_in_use;
    rand int in [0..NUM_OF_CHANNELS-1] chain_starting_location;
    rand int in [1..NUM_OF_CHANNELS] chain_size;
    rand int in [0..NUM_OF_CHANNELS-1] index_in_chain;
    lock channel_t channel_r;
};
action channel_middle : channel {
  constraint ch_role == MIDDLE;
  input data_t in_data;
  output data_t out_data;
  // Use this to find the previous channel
  constraint in_data.source == channel_r.instance_id - 1;
  // Assign the value from the input data
  constraint chain_starting_location == in_data.chain_starting_location;
  constraint index_in_chain == in_data.step;
  constraint chain_size ==in_data.chain_size;
  constraint npsec_in_use == in_data.npsec_in_use;
  // Pass the value to the output data
  constraint out_data.source == channel_r.instance_id;
  constraint out_data.chain_starting_location == chain_starting_location;
  constraint out_data.step == index_in_chain + 1;
  constraint out_data.chain_size == chain_size;
  constraint out_data.npsec_in_use == npsec_in_use;
};
```

Figure 6 Modelling a channel in PSS

The middle channel has both input and output data, and it either increment the index or pass the variable from input to output by using constraints. And it will always hook up its input to the element of `instance id -1`.

The first channel action has a subset of the constraints of the middle channel. It only has output data. It has the `index_in_chain` as 0, `chain_starting location` as its `instance_id`, and it has the `chain_size` from the output data's `chain_size`. The last channel action, similarly, only has the input data. It gets the `index_in_chain` from the `input_data.step`. And it constrains the `chain_size` as `index_in_chain` + 1. Another derived channel is the "single" channel which only has itself in the chain. The single channel can be simply modelled by constraining the `index_in_chain` to be 0, `chain_size` to be 1 and `chain_starting_point` as its own `instance_id`.

When the model is created, we use the solver to create the chain. This is done neatly by only three constraints in the test case (shown in Figure 7). By filling each channel with all the possible roles, we achieved the coverage in all the possible cascading options.

Using PSS to model the configuration can achieve faster coverage closure by reducing tests in the regression. In the brutal-force UVM-only regression without seed ranking, we need approximately ~1000 tests and 24 hours of machine time (depends on how many generics sets) to get full functional coverage. For PSS-based solution, the number of tests is reduced to about 50%, since it is doing the seed ranking upfront. We got 100% functional coverage in an early verification stage. This reduces the license usage and machine power for running regressions.

VI. INHERITANCE

Inheritance and polymorphism are the most used principles in object-oriented programming to facilitate reuse. In PSS it is also possible to exploit the inheritance to simplify the coding. It is therefore important to plan a good structure before starting to implement the PSS model.

Test case is the highest layer in PSS model and it is defined in action. Most of the UVM test case is extended from some base class test, which configures DUT and sends in stimuli. It is also possible to follow the same approach in PSS modelling, thanks to the inheritance support in the PSS-DSL. Example in Figure 7 shows that by changing the constraints, we can easily extend test cases from the base test.

```
action base_test {
  rand role_in_ch_e channel_ch_roles[NUM_OF_CHANNELS];
  rand int in [1..NCPSEC_P] npsec_in_use;
  rand int in [16..2000] m_itr_min;
  rand int in [16..2000] m_itr_max;
  constraint m_itr_min < m_itr_max;

  activity {
    sequence {
      do config_channel with {
        channel_ch_roles == this.channel_ch_roles;
        npsec_in_use == this.npsec_in_use;
      }
      do activate_phase_config;
      do run_tx_all_channels with {
        m_itr_min == this.m_itr_min;
        m_itr_max == this.m_itr_max;
      }
      do end_phase_tx;
    };
  };
};

action cascade_test : base_test {
  constraint channel_ch_roles[0] in [SINGLE, MASTER];
  constraint channel_ch_roles[NUM_OF_CHANNELS-1] in [SINGLE, LAST];
  constraint foreach (channel_ch_roles[i]) {(channel_ch_roles[i] in [MIDDLE, LAST]) ==
    (channel_ch_roles[i-1] in [MASTER, MIDDLE]);};
  constraint {
    m_itr_min == 500;
    m_itr_max == 1000;
  };
};
```

Figure 7 Test case inheritance

For tests with more details in the control part, we need to extend the PSS model to add complexity into it. The basic approach is to create an action for a UVM sequence. The complexity in low level transactions should still be kept in UVM sequences. The controlling of the sequence should be leveraged into PSS model. Adding this abstract layer on top of UVM gives us better graph-based control over sequences, and the portability is much improved.

One limitation to the reusability here is that PSS model usually have many control knobs. The control knobs are passed by DPI function variables, and since there is no support in polymorphism in SystemVerilog, we had repetitive DPI functions which starts the same UVM sequence but with different number of randomized variables. A possible solution is to pass the control parameters in a struct, and define wrapper function in the SV/UVM test bench to extract and use them.

VII. PARTIAL CONFIGURATION

PSS supports abstract partial configuration. This means the model does not need to specify all the steps but can instead exploit the specification of the data flow and rely on the solver to get a complete test scenario. This approach works, until we have multiple parallel process running, with each of them passing a different configuration from action to action. During this experiment we faced constraint solver problems. We think the success of abstract partial configuration is highly dependent on the coding style.

One difficult test case we encountered is a power saving test case. In this test, we configured a number of filter engines, then send out stimuli several times depending on a randomized variable from the configuration randomization. Between each time stimuli is sent, we turn on and off the power saving mechanism and check, if the output is same as we expected.

The main problem is that we have 6 channel actions, with each representing a filter engine and randomizing a variable `npsec_in_use`. This variable is needed in action `run_1`, `run_2` and `run_3`. The variable should only be passed to the run actions with the same channel number as shown in Figure 8. In PSS actions, we define input and output data, but it is completely depending on the solver to connect the data to correct subsequent action. We found that in a highly parallelized system, if without many detailed constraints, the solver has difficulties to connect two actions correctly, so that it generates invalid or unwanted scenarios. If we don't want to sacrifice the flexibility that partial configuration provides, we must find a balance between partial and fixed configuration.

Our solution is to group the actions into a compound action, and use the lock to ensure each compound action related to one and only one channel. Figure 9 shows the compound action. First we create input data with type `cfg_t`. It contains the `npsec_in_use` and `channel_num` from the `channel_single` action. Then we

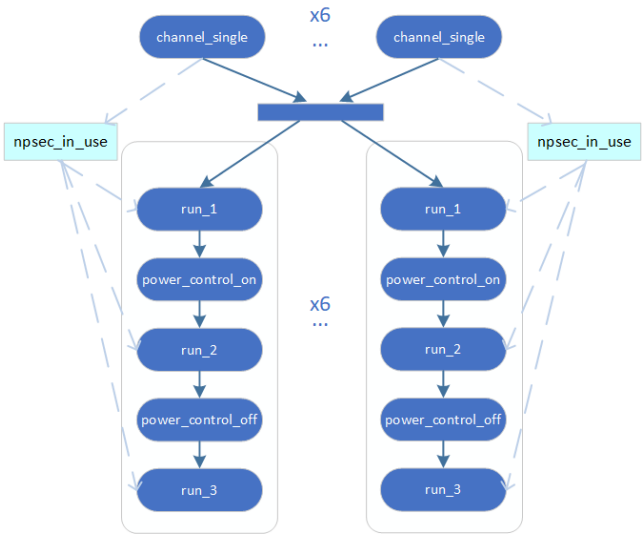

Figure 8 Example of a partial configuration challenge

```
action run_on_one_chan  {
  input cfg_t in_cfg;
  lock  cfg_rsrc_t cfg_rsrc;
  // force different id by lock
  constraint in_cfg.chan_no == cfg_rsrc.slot_num;
  rand int in [0..NUM_OF_CHANNELS-1] channel_num;
  rand int in [1..NCPSEC_P] npsec_in_use;
  constraint npsec_in_use == in_cfg.npsec_in_use;
  activity {
    sequence {
      do run_1 with {
        npsec_in_use == this.npsec_in_use;
        channel_num == this.channel_num;
      };
      do power_control_on with {
        channel_num == this.channel_num;
      };
      do run_2 with {
        npsec_in_use == this.npsec_in_use;
        channel_num == this.channel_num;
      };
      do power_control_off with {
        channel_num == this.channel_num;
      };
      do run_3 with {
        npsec_in_use == this.npsec_in_use;
        channel_num == this.channel_num;
      };
    };
  };
};
```

Figure 9 Compound action to solve partial configuration challenge

use a lock to constrain the `channel_num` from the input data. This constraint forces different channel number by utilizing the unique id of the lock.

The corresponding output data should be created on the channel action. And for the test case, we simply replicate this compound action as many times as `NUM_OF_CHANNELS`. In this way, the solver actually resolved the scenario as we expected.

VIII. SEMANTICS EQUIVALENCE

It is essential that when migrating test case to PSS, the same semantics of the existing UVM tests should be kept. But due to the difference in the PSS-DSL and SystemVerilog language, we often encounter implementation details which do not exist in both languages.

An example is the constraint distribution in SystemVerilog. It is very useful to randomize a variable according to the distribution of different values. But it has unfortunately not been implemented in PSS-DSL.

Another example is the reconfiguration test case which we have migrated to PSS. The initial PSS model produced semantics which did not match the existing semantics. Two problems are identified

- Synchronization between C and SystemVerilog as shown in Figure 3 needs to be extended. Since our synchronization is event-trigger based, we cannot reuse the same mechanism for the re-configuration. Therefore simply putting a loop around the actions won't work. We had to create new events to synchronize the C and SystemVerilog.
- The test generated from the PSS model are direct tests with constraint-randomized parameters to cover a pre-specified coverage goal. Randomization for the reconfiguration needs to be done upfront, i.e.. before test is running. Therefore, the reconfiguration test is indeed pre-generating configuration twice, and apply them in two different steps of the test. This is different from the SystemVerilog randomization, which is seed related and done during run-time.

It is difficult to do the equivalence check between UVM test and PSS model generated test. One may look at the test results as a reference. The regression should not have new failures after migrating to PSS. Another important measure is the coverage. Here we are talking about the SystemVerilog coverage model which is not used for PSS scenario generation. It is still valuable to keep the original coverage model to make sure the PSS generated test does not deviate from the UVM tests, so that we have unexpected missing coverage.

IX. CONCLUSION

Our experiment shows it is realistic to add portable stimulus layer to an UVM test bench with reasonable effort, especially when the tests are configuration and sequence based. It is also possible for both UVM and PSS based test cases to co-exist. We summarize our five major challenges:

A. Compile-time parameters: We have shown that using Cadence SLN can simplify the coding, and is therefore a good complement to the current standard.

B. Run-time configuration: It is essential to create the model in layers and distribute the complexity into each layer. PSS model accelerate coverage closure by reducing test number in the regression.

C. Inheritance: It is a powerful tool to create test cases, but it can be further improved for code reuse.

D. Partial configuration: Successful solving of the partial configuration is heavily dependent on good constraint sets and coding style.

E. Semantics: One should be aware of the potential pitfalls about the semantics inequivalence between the PSS model and UVM tests, so that one does not get misled by false positive results.

In conclusion, the effort in deploying an extra portable stimulus layer is paid off by improved verification quality, faster functional coverage closure and decreased number of tests in the regression. Also the visual representation of the test scenario offers a uniform understanding in test requirements and strategy across different teams. It promotes reusability and will in the long term reduce redundant test development time in other target platforms such as SoC environment. It is valuable to have PSS-based solution as an add-on to the existing dynamic verification techniques.

ACKNOWLEDGMENT

The authors would like to acknowledge the contributions made by Cadence.

REFERENCES

[1] Portable Test and Stimulus Standard Version 1.0a Accellera, Februar 2019

[2] IEEE Standard 1800.2-2017, "IEEE Standard for Universal Verification Methodology Language Reference Manual", 2017.

[3] A.Vintila, I. Tolea, AMIQ Consulting, "Portable Stimulus Driven SystemVerilog/UVM verification environment for the verification of a high-capacity Ethernet communication bridge, ", Design and Verification Conference US, 2019

[4] Cadence, "Perspec System Verifier User Guide"