DVCon India 2023

TITLE OF PAPER	**Tackling the verification complexities of a processor subsystem through Portable stimulus**
AUTHOR 1	**Name: Vivek Gopalkrishna** **Organization: Analog Devices** **Job Title: Design Verification Engineer** **Email ID: Vivek.Gopalkrisna@analog.com** **Mobile no: 8792324950**
AUTHOR 2	**Name: Ponnambalam Lakshmanan** **Organization: Analog Devices** **Job Title: Senior Design Verification Manager** **Email ID: Ponnambalam.Lakshmanan@analog.com** **Mobile no: 9945089481**
AUTHOR 3	**Name: Nitish Swamy** **Organization: Analog Devices** **Job Title: Senior Design Verification Engineer** **Email ID: Nitish.Swamy@analog.com** **Mobile no: 7411393620**

ABSTRACT

Today across the semiconductor industry, different languages and techniques are used for functional verification depending on the scope of verification. When verifying RTL block and subsystem, UVM based verification environment or other custom SystemVerilog based verification environments are preferred, which provide an elaborate support to constrained-random stimulus. At system level, embedded software is frequently used to exercise the design, resulting in use of languages and platforms with limited constrained-random capabilities. Several challenges result from different languages and techniques being used for block and subsystem-level verification. Additional complexities are introduced in tests for designs with multiple processor cores with the need of manually added synchronization handshakes in complex test scenarios. For every target application, the specifications need to be revisited and new tests must be developed at different layers of testing.

To avoid duplication of test creation effort, ensure better reuse of test suite, facilitate better constrained-random stability, aim for a unified test generation framework and for early coverage closure by discarding redundant tests, Portable Stimulus is a natural answer. The solution described in this paper aims at highlighting the various capabilities of PSS, unravelling the practical advantages achieved in modeling complex test behaviors and the path one must take to fully utilize its potential.

BACKGROUND

Creating sufficient tests to verify today's complex designs is a key verification challenge, and this challenge is present from IP block-level verification all the way to SoC validation. In cases of multi core designs, there is an added complexity. The tests must be multi-threaded to handle the various core operations as well as drive stimulus to the UVM based testbenches. This is traditionally done by manually creating tests which have the right synchronization handshakes between the multiple threads. In certain scenarios it is needed to switch between having core and coreless architectures. For instance, in block level tests, it may be sufficient to use a BFM model with a coreless design. The tests created for such an environment cannot be ported directly to a subsystem or system level verification with the actual cores in place. Also, coverage closure being the key metric driving the verification effort, it is required to create meaningful tests that add to the overall coverage and make sure that these are not redundant tests.

Looking for a solution to all such challenges in the traditional verification strategy, Portable Stimulus is the industry's attempt to reign in the exploding cost of doing verification caused by the duplication of efforts across the different engineering roles throughout a project's development life cycle. This paper captures the portability, ease of generating multi-threaded tests, ease of switching between core and coreless architectures and techniques to help in early coverage closure.

DESCRIPTION

The Portable Stimulus process includes writing a model in a standard language which captures the verification intent, independent of the verification/validation platform and can be used to derive tests for a given target application. This will allow improved collaboration between engineering teams, vertical stimulus re-use from block to system level and horizontal re-use across project platforms (Virtual platform, simulation, emulation, FPGA and post-silicon) and different projects.

Test development process with Portable Stimulus

As described in the below figure 1, test development process is based on the PSS foundational framework, with models described for each IP, capturing the nature of test scenarios intended to be run.

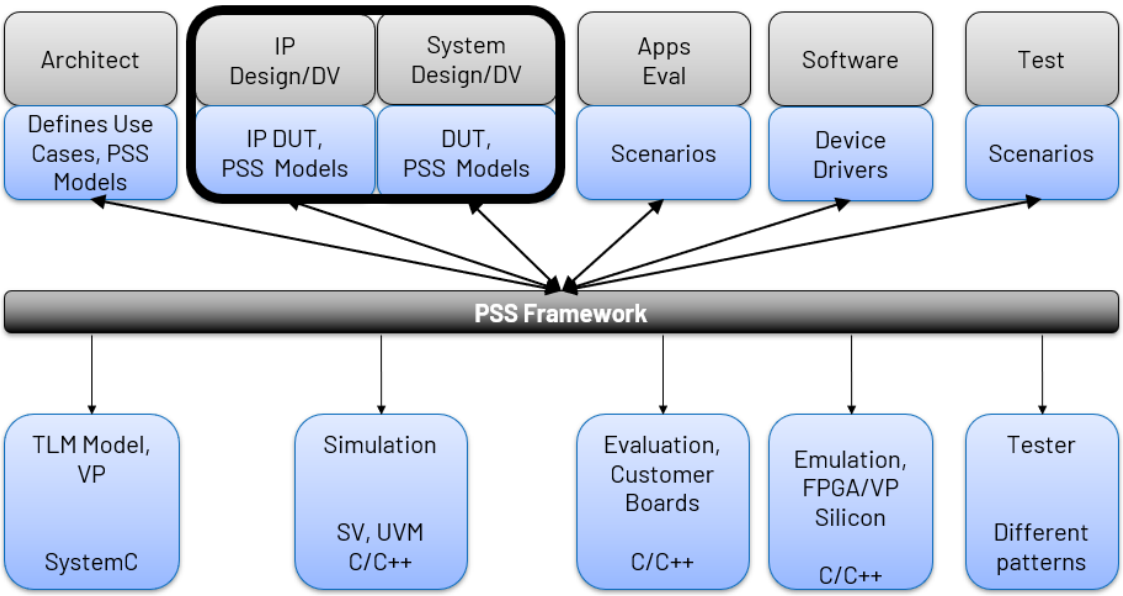

Figure 1: Test development process in ADI with PSS

IP/block level verification, along with subsystem level to some extent, rely on a UVM based simulation environment, aided by emulation in the faster platforms. As these tests cannot be directly carry forwarded to the system level in the traditional flow, PSS enables separation of "test intent" from the "implementation". The models defined in PSS can carry the verification intent at a much higher level of abstraction and be independent from the implementation platforms. This enables reuse of test scenarios defined in ip/subsystem level across system level too, by modifying the underlying integration hooks to map to the right implementation platforms. Figure 2 showcases the separation of test intent from the implementation for a sample SPI transmit scenario.

```
action SpiTx_a {
        input config_state_s spi_state_in;
        adi_spi_xfrData xfer_data;
        input sml_data_buff in_buff;
        share sml_core_r cores;
        constraint cores.tag == cpu0;
        addr_handle_t buff_addr_t;
        int test_addr_tx;

        exec post_solve {
                buff_addr_t = make_handle_from_claim(in_buff.mem_seg);
        }
```

Test intent captured in **SpiTx_a** action

```
import SV function pss_spi_Tx;
extend action spi_c::SpiTx_a {
        exec body {
                xfer_data.tx_buffer = buff_addr_t;
                xfer_data.tx_buffer_sz = in_buff.mem_seg.size;
                comp.pss_spi_Tx(spi_state_in.spi_id, xfer_data);

        };
}
```

Test implementation specified in **exec body**

Figure 2: Separation of test intent from implementation

Prerequisites for Portable Stimulus Deployment

All designs may not be suitable for PSS framework. To deploy the PSS flow in any design verification, certain prerequisites must be met. Captured below is a checklist of steps to be carried out to consider the feasibility of portable stimulus. Buy-in from all stake holders including DV, emulation, firmware development, Eval and Post-Si teams is required to ensure a unified verification flow with PSS.

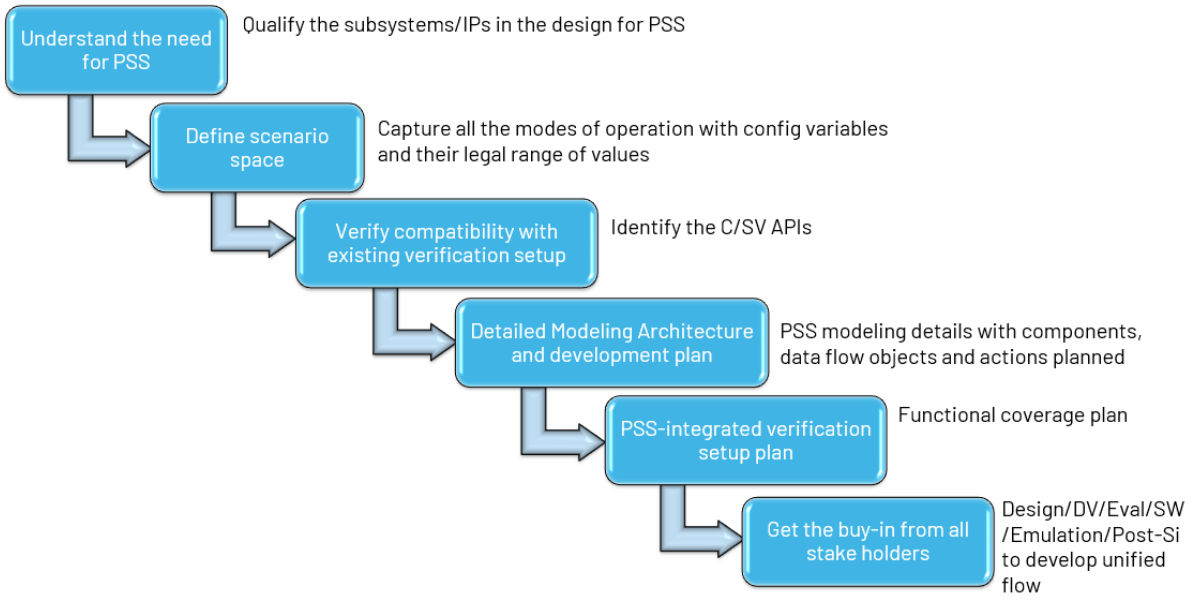

Figure 3: Prerequisites for PSS deployment

PSS Usage Flow

The process of deploying PSS framework involves model creation and test generation. Model creation comprises of building PSS file sets which depict the various design components and the interaction between them. Test generation involves "solving" the model along with the various constraint-random configurations specified in the model and to create scenarios that are defined in the verification plan.

Model writers must be well versed with PSS and capturing the design elements in the right way which includes components, actions, data structures, data flow elements and the integrations for various target platforms. Test writers can have a limited understanding of PSS and need to be able to create complex scenarios by using the building block actions defined in the model. This involves handling native operators in PSS (sequence, parallel, repeat) and creating meaningful scenarios.

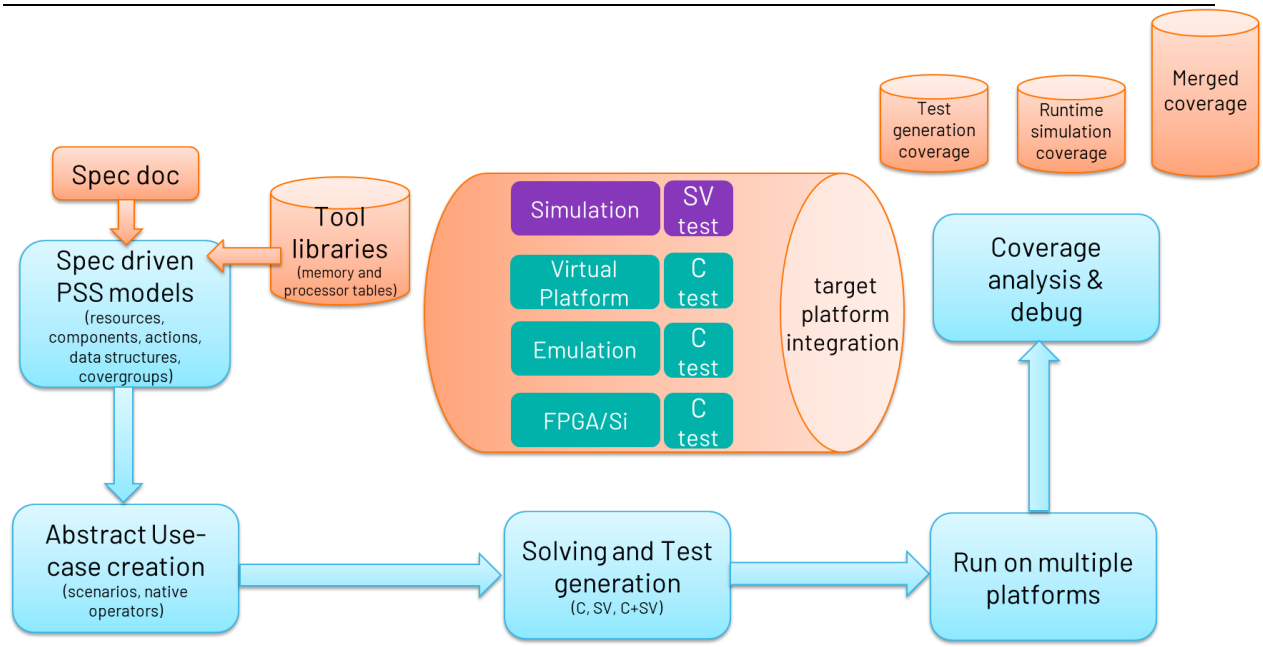

Figure 4: PSS Usage Flow

Modeling

Modeling involves the usage of PSS constructs to define the DUT blocks, VIP blocks and describe the associated tasks to be performed on each of these blocks, coded in an aspect-oriented approach called the DSL (Domain Specific Language). Few PSS terminologies used in the following sections are described below:

- Component: Represent the design elements that make up the hierarchy. Eg: spi, i2c, uart, spi_vip
- Action: Elements which capture test behaviors in a PSS model. Eg:spi_config, uart_tx, spi_tx
- Flow objects: Represents incoming or outgoing data/control flow for actions. Eg: buffer, state, stream
- Resources: Hardware design resources that are needed to perform the test functionalities. Eg: dma channels, processor cores.
- Activity(scenarios): A set of action instances, flow objects and scheduling constraints which makes up testcases.

Modeling an IP in PSS requires an elaborate effort from the model-writer to effectively capture the design configurations and constraints in order to meet the verification strategy. This can be seen as a one-time effort and the same model of the IP can be re-used across different projects with very minimal changes to suit the custom needs of the design. The steps involved in effective modeling strategy is listed below.

1. **Defining Scenario Space**

   The first step of modeling an IP would be to define its scenario space. Scenario space of that IP comprises of all the possible legal configurations in which the IP can be configured to run.

This should effectively capture the variables which are randomized and the associated constraints. It is important to correctly capture all essential fields as this would be an input to the model writer. Any missed constraints/fields would result in verification holes.
SPI peripheral DUT would have the configuration fields as below:

- State: initiator or target.
- Mode of operation: half duplex, full duplex
- Tx/Rx DMA: enabled or disabled
- Continuous chip-select: enabled or disabled
- Various interrupt controlling fields
- Tx/Rx transaction count

2. Modeling the components and actions

This would be the most important aspect of model writing. It involves the usage of PSS constructs to define the blocks which make up the model. On a top level, the dut block and its VIP counterpart are declared as "components". The tasks to perform at a higher abstraction level are declared as "actions". Actions combine to form the scenarios that represent the verification intent. These actions are embedded into their respective components.
To model the data flow elements between the actions, we make use of "state", "buffer" and "stream" objects. These components and their actions are all residing in a top-level default component called "pss_top".

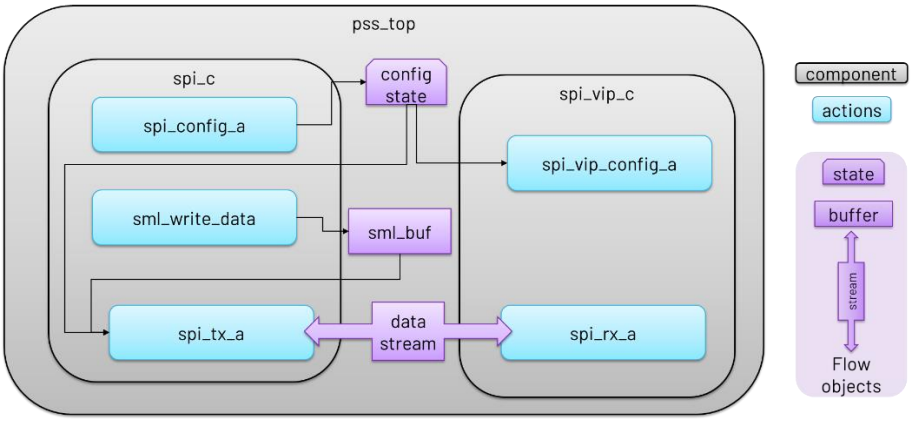

Figure 5: Modeling of SPI

Shown in the figure above is the modeling aspects of a typical peripheral IP like SPI, along with the PSS structures defined.

PSS-integrated verification setup

With the modeling aspects covered, it is important to consider the integration of the model into various target platforms. The implementation specific functions and tasks, corresponding to each action are to be mapped to the right target platforms. This would ensure that the flow of control and data between the abstract action/scenarios to the leaf level execution bodies

is as expected. The implementation tasks/functions which are used by the leaf level actions need to be provided in the model as imported functions. Example is shown in figure 2.

For designs with more than one execution threads, it is important to identify them, including the SV-UVM execution flows run on the host machine.

Tool specific libraries that help in handling memories, processor cores and various IPs in the design can be made use of. Mailbox enables the mechanism for inter-executable synchronization between C and SV threads in the generated tests. Regions of a common memory must be identified and declared as part of the integration PSS files.

Shown below is a typical config.csv file which is read by the solver tool to populate the processor information required to solve the model. It contains the various cores in the design along with their attributes and mapping schemes.

Table 1: Processor info table

@package: sml_pkg	**@size_const: NUM_OF_CORES**	**@struct: sml_processor_info_s**				
#tag	#kind	#cluster	#cluster_id	#core_id	#mapping_scheme	#enabled
core0	M33	CLUSTER1	1	0	EMBEDDED_C	TRUE
core1	M33	CLUSTER2	2	1	EMBEDDED_C	TRUE
core2	M33	CLUSTER3	3	2	EMBEDDED_C	TRUE
host	NONE	NONE	NONE	NONE	SV_UVM	TRUE

Table 2 contains the memory information which can be used to read and write into the design memories using pre-defined actions.

Table 2: Memory Info Table

@package: sml_pkg	**@size_const: NUM_OF_MEM_BLOCKS**		**@struct: sml_memory_info_s**				
#mem_block	#base_addr	#end_addr	#alignment	#backdoor_enabled	#ctype	#sec	#enabled
SRAM0	0x20000000	0x2001FFFC	4	TRUE	WriteBack	0	TRUE
SRAM1	0x20040000	0x2005FFFC	4	TRUE	None	1	TRUE
SRAM2	0x20080000	0x2009FFFC	4	TRUE	WriteThrough	2	TRUE
SRAM3	0x200C0000	0x200DFFFC	4	TRUE	WriteThrough	2	TRUE

Design Topology can be captured in user-defined tables and parsed with the tool libraries. These can contain the various instances of the design along with any interesting attributes required to be captured by the user. The model is written to be design change agnostic in terms of the number of instances and their configurations.

Table 3: Design Topology Table: SPI and DMA

@struct: spi_info_s	@size_const: NUM_OF_SPI	@package: config_pkg
#spi_inst		
spi_inst0		
spi_inst1		
@struct: dma_info_s	@size_const: NUM_OF_DMA	@package: config_pkg
#dma_inst		
dma_inst0		
dma_inst1		

3. Coverage

The coverage targets specified by the covergroup construct are more directly related to the test scenario being created. PSS allows defining of covergroups and coverpoints similar to systemverilog and can be used to collect the pre-run coverage metrics, allowing the users the flexibility to either retain the tests or discard them based on the value they add to the overall coverage.

PSS enables coverage specification on use cases and on attributes of actions and flow objects. The coverage metrics can be collected either in generation time or in runtime, and on all executable platforms including post-silicon validation platforms.

- Generation time coverage helps in regression planning and to prioritize the execution of tests on target platforms.
- Run time coverage helps in validating the generation time coverage results, and to perform coverage analysis on scenarios and attributes that cannot be determined at generation time. .

```
covergroup {
                cs_cp   : coverpoint cs_ctl;
                lsb_cp  : coverpoint ctl.lsb;
                state_cp : coverpoint spi_state;
                mode_cs_x : cross cs_cp, lsb_cp, state_cp;
        } cp1;
```

Figure 6: Covergroup in PSS

PSS ON EMULATION

As design size and complexity increases, there is a need to accelerate the runtimes of simulations to achieve faster coverage targets and gain confidence on the design. With our project, it was necessary to cover long running simulations that would span over days together. These scenarios are not feasible on simulation and hence were run on **Emulator.**

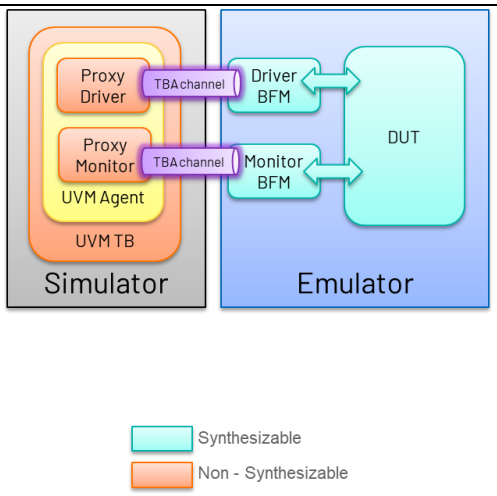

Figure 7: Simulation Acceleration using emulation

The challenge was to generate test cases that would run on the Emulator without any hassle of manual tweaks to the tests. The capabilities of PSS were truly tested and with minor updates to the existing model, the tests scenarios running on simulations were also run-on emulator, observing a very high degree of runtime improvements (upto 100x speedup).

The setup involved declaring the drive and collect tasks, which form the key components of communication between Testbench and DUT, as interface tasks. This limits the number of synchronization points between the simulator and emulator, thereby resulting in faster runtimes.

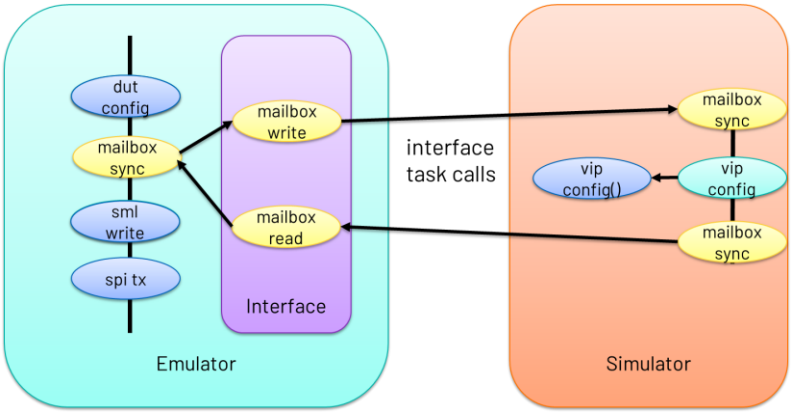

Figure 8: PSS on Emulation

PSS ON VIRTUAL PROTOTYPING PLATFORM

Porting PSS to enable test generation on System C models was seamlessly done. This involved the binding of relevant target functions which can be run on these models. The VP platform being on Windows required the generated target directory to be copied over to the windows setup.

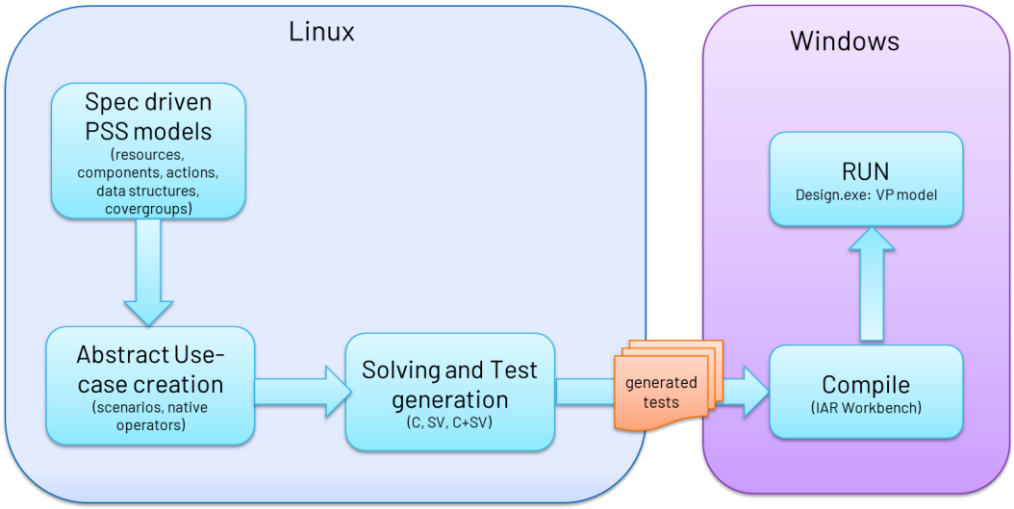

Figure 9: PSS on VP

PRELIMINARY RESULTS

- PSS vastly reduces the bring-up time of test suites for complex processor subsystems with multiple cores.
- Coverage closure can be tightly monitored at test generation time, rather than post simulation.
- Self-checkers(scoreboards) in PSS removes the need of checkers in all target platforms. This saved enormous time for Virtual Protoyping platform, Emulation platform and on custom FPGA.
- Multi platform and cross project test intent reuse enables better utilization of resources.

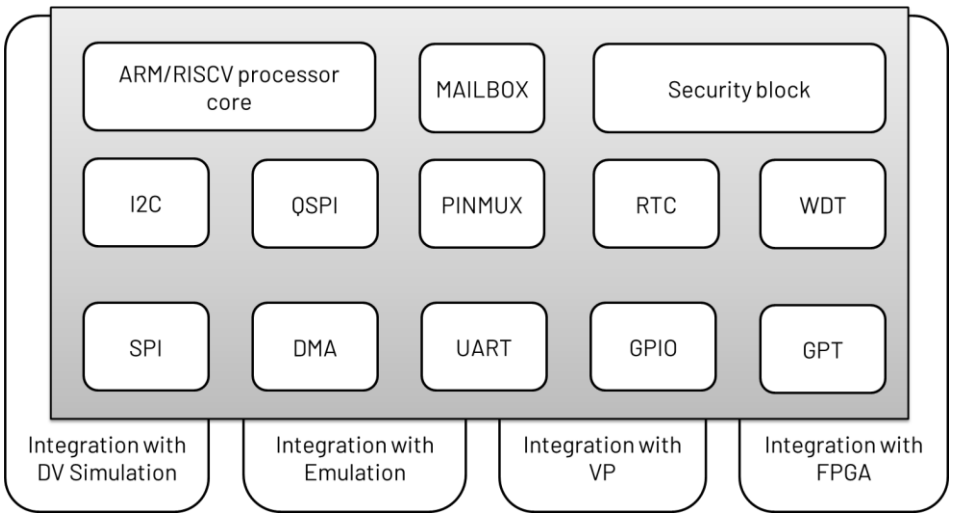

Figure 7: Live-project processor subsystem shown with the various target platforms

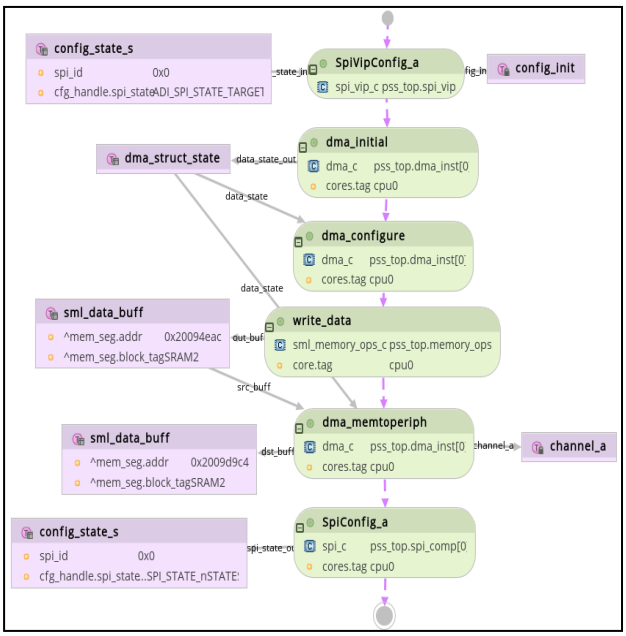

Figure 8: PSS generated test scenario: SPI-DMA

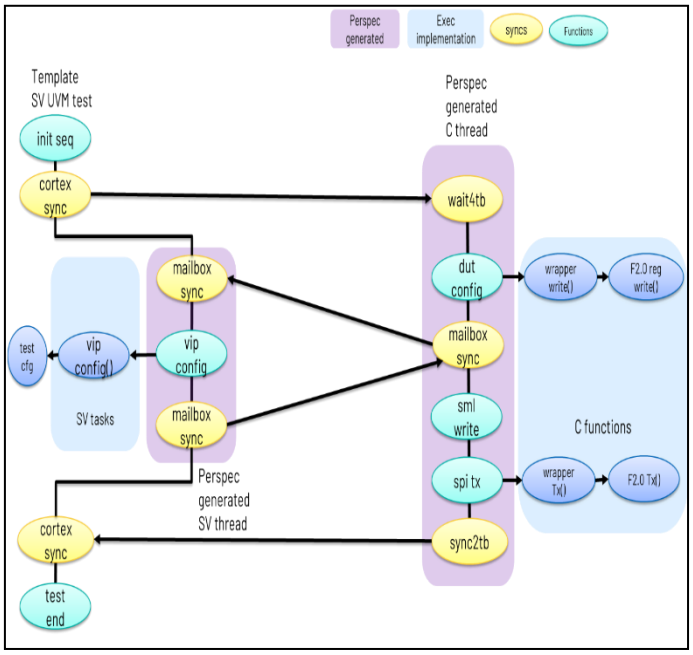

Figure 9: Thread view of SPI-DMA test scenario

CONCLUSION

- PSS has helped us shorten the design verification cycle, with an estimated reduction of 50% of time spent in test generation for VP, FPGA and Emulation platforms.
- Coverage with PSS enables users to check the metrics **before** running a simulation and significantly reduces time spent by optimizing the test regression suite prior to running them.
- Duplication of test creation effort has been nullified and creating complex test scenarios in designs with multiple cores has been made simple.
- Randomization of attributes is offloaded to the solver engine, thereby reducing the effort and time on the run time simulator.

ACKNOWLEDGEMENTS

- Rajiv Nadig, Paul Drum, Jatin Nagpal, Sonal Patil, Akshata Kulkarni, Shriyanshi Kapoor and Kajal Majalatti for the continuous effort in modeling and bring-up.
- Gnaneshwara Tatuskar and Sandeep Katti for their constant tool and methodology support.