Adopting Accellera's Portable Stimulus Standard: Early Development and Validation using Virtual Prototyping

Simranjit Singh, Ashwani Aggarwal, Harshita Prabha, Vishnu Ramadas
Samsung Semi-Conductors India R&D, Bangalore, India
simranjit.s@samsung.com, ashwani.a@samsung.com, harshita.p@samsung.com, v.ramadaska@samsung.com

Seonil Brian Choi, Woojoo Space Kim
Samsung Electronics, 1-1, Samsungjeonja-ro, Hwaseong-si, Gyeonggi-do, Korea
seonilb.choi@samsung.com, space.kim@samsung.com

***Abstract* - With increasing complexity of the SoCs, traditional verification approach is not feasible to meet the strict time-to-market windows. Accellera's Portable Stimulus Standard (PSS) provides the methodology to abstract the test intent to make the tests easily portable to various verification levels like at IP, sub-system, post-silicon etc. However, adopting PSS methodology poses its own challenges regarding development and validation of PSS models and implementation APIs. In addition, the availability of the qualified PSS models at early stage of the verification effort is a challenge. In this paper, we discuss how we used Virtual prototyping (VP) for early development and validation of PSS models for faster and productive PSS adoption.**

I. Introduction

During last couple of years, there has been significant traction around Accellera's Portable Test and Stimulus Standard[1][2] (PSS). Although, official standard was released in 2018, many companies started developing the tools around PSS methodology, which was still at draft stage. This helped in commercial tools being available around the same time as standard was released.

PSS is nothing less than a revolution, which is addressing two of the most challenging problems, "verification" and "reuse", in modern SoCs. With the growing complexity of Systems and the ever-shrinking time-to-market windows, the verification challenge is growing day by day – which puts additional burden on verification engineers at different stages of SoC life cycle. A significant effort goes into creating tests with same verification intent at various levels, like IP, sub-system, system, post-silicon, with little to no reuse. Additionally, porting of these tests from one SoC to another SoC is time-consuming and error-prone. With PSS methodology, the test intent is abstracted from the implementation, which make it easy to reuse across various verification levels and SoCs. The Figure 1 captures PSS flow and its reuse across verification levels and platforms.

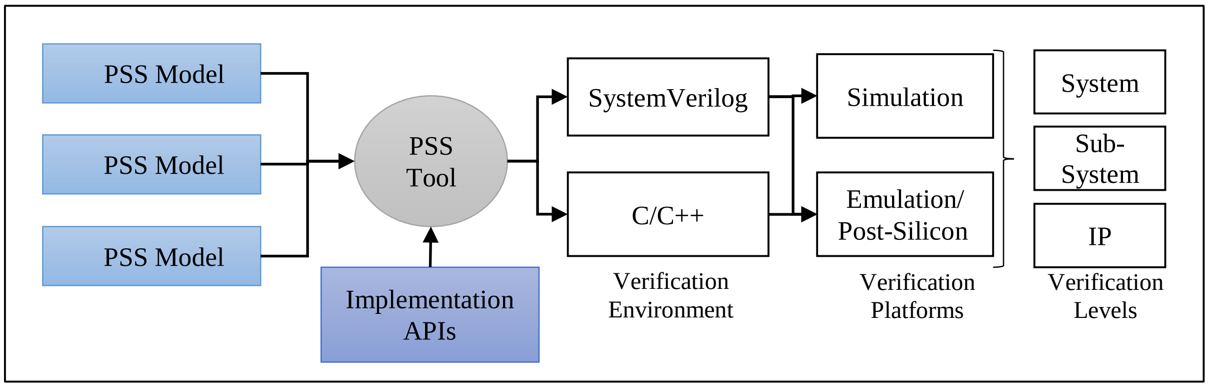

Figure 1 Reuse with PSS

However, adopting PSS methodology is not simple. It takes significant effort to adopt the PSS methodology – development of PSS models for the required components and especially, validation of these models. In this paper, we discuss about PSS development, various challenges in its deployment and using Virtual Prototyping [3] (VP) to overcome those challenges.

II. PSS Methodology And Challenges

The PSS methodology provides an approach to specify test intent at a higher level of abstraction, which allows the test to be portable to various verification targets. In this flow, a test case is specified using an abstract model, which captures the test configuration and the data flow. This abstract model is agnostic to the verification platform and the verification level i.e. it has no dependency on how the test is realized in any given verification environment. With this decoupling of test intent and test realization, the tests can be generated for various targets using PSS tools.

However, it is not easy to get fast benefits out of PSS. The main challenge in deploying PSS is that companies do not have existing PSS models available for their IPs. The standard EDA solutions cannot help here, as most of the IPs are proprietary and confidential. Thus, the verification teams develop the PSS models, which includes identifying abstract attributes to capture the test intent and developing the implementation APIs to program the Design-Under-Test (DUT) and test. The Figure 2 depicts the flow, with the PSS artifacts to be developed highlighted in blue.

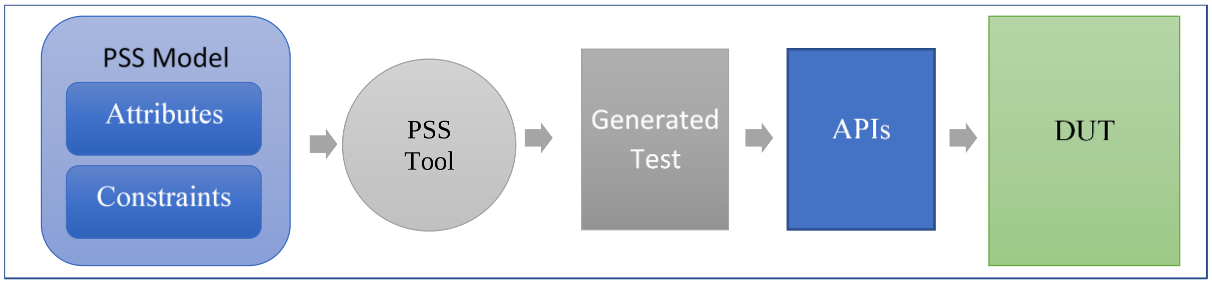

Figure 2 PSS Development

The PSS model comprises of the following major artifacts.

1) Attributes: To specify the configuration parameters of the DUT. These attributes may cover the complete range of control parameters or only very high-level parameters. The PSS tools randomize the attributes to generate configuration, thus providing ability to generate numerous configuration for the same test.

2) Constraints: To ensure that randomized configuration is correct, PSS provides mechanism to specify constraints between the attributes. As part of the PSS model, the teams need to model the constraints – covering all possible attribute values to ensure valid attribute configuration is generated.

3) APIs: The implementation APIs are not directly part of the PSS model but are needed to program the DUT as per the configuration generated in the test. These APIs extract the configuration from the PSS attributes and programs the DUT registers and load data to the memory and therefore, are specific to a given verification target. For example, in SystemVerilog (SV) simulation environment, these are often developed using SV whereas for emulation or post-silicon validation, these are developed in C/C++. In addition to test realization, these APIs may also perform verification checks e.g. verify that the relevant interrupts are raised, verify DUT output etc.

Using these PSS models of various components, verification teams can create any complex scenarios in very short time.

The crucial aspect of the PSS model development is verification of the model itself. If the constraints in the PSS model are incorrect, the generated test configuration will be invalid and the test will fail. If the APIs are not configuring the DUT correctly, the tests will fail. It may lead to very long debug cycles, as the verification team would not know whether the issue is in the DUT or with the generated test. Unless the PSS model is validated, there is always doubt in the mind of verification engineers whether the tests generated by the PSS model are correct or not. Hence, we need to ensure the validity of the PSS model before deploying it for verification across various teams.

Another challenge for PSS model developers is “how to qualify the models fast” to make it deployable for verification. At the IP or the sub-system-level, verification environments are simulation based (SV/UVM), which has very slow execution speed for even simpler test cases. In addition, the debugging is hard and the iterations to fix bugs and validate is time-consuming. Hence, using simulation for validating the PSS model is not optimal.

The other way to validate PSS models is using emulation. In emulation environment, the execution speed is high and turn-around times are shorter. However, there are two critical bottlenecks to validate PSS model on emulator. The first issue is the emulator is available only at a relatively later stage of the SoC development, which means the PSS model could not be validated in required time. The second issue is that the emulator is a very expensive resource, being shared by many teams and not freely available.

Due to these challenges, validating the PSS model is a tedious task, which defeats the whole purpose of PSS. Many teams decide not to use PSS precisely due to this reason.

At Samsung, we recognized this challenge very early. To mitigate it and to get faster results, we used a different approach – use Virtual Prototyping methodology to qualify and mature the PSS model. In the following section, it discusses how VP methodology helps in getting PSS models ready early and fast.

III. PSS Development Flow with Virtual Prototyping

A. *What is Virtual Prototyping?*

Virtual Prototyping (VP) is a proven methodology, which is widely used to enable early software development. Due to complexity of the systems, hardware design and verification takes significant time and effort for the hardware to be ready. Traditionally, the software teams wait for the hardware board to be available to bring-up the software. As it is the first time the software and hardware come together, software teams face lot of issues. It leads to significant effort to debug and fix the issues and teams often struggle to meet the deadlines.

With the VP methodology, an abstract and functionally accurate model of the system hardware is developed at the start of the project. Due to its abstract nature, it is easier to develop and is available much earlier than the hardware. A virtual prototype of a SoC includes all major components: CPU models, which are generally Instruction Set Simulators (ISS), transaction-level bus model with memory-map, memory sub-systems and various hardware accelerators and peripheral models. Using the VP, software teams develop and verify software before hardware is available. To ensure that the software developed using VP is portable to the hardware; VP models need to have bit-accurate implementation of the functionality, register models and interrupt signals. With VP, the software bring-up could be achieved within days once the hardware is available.

In addition, the models are developed using C++/SystemC, which makes it easy to debug and is available to large set of users unlike simulation and emulation platforms. Moreover, due to its abstraction, the simulation speed of VP is much higher than the RTL simulation – usually more than 100x. Hence, due to its early availability and accuracy to the hardware, VP is the ideal platform to validate the PSS models.

B. *PSS Development with VP*

As mentioned above, the PSS model, attributes and constraints, is an abstract representation of the test intent and is agnostic to the implementation and target platform. Thus, using VP for validation has no impact on the way the PSS models are created. However, the implementation APIs are target specific and hence need to be considered carefully. To ensure maximum reuse across various verification targets, the implementation APIs would be best suitable to be in ‘C’. It has following advantages:

1) For System Testing: C-APIs can be cross-compiled and executed on any platform with target CPU – VP, emulation and post silicon validation etc. Interestingly the target CPU can be different in each environment.

2) For Simulation: C-APIs can be used in SV based simulation testing using C-SV DPI (Direct Programming Interface). The Figure 3 shows the flow with reuse of C-APIs using the DPI support. The SV test code generated by the PSS tool invokes the C-APIs, which in turn invoke SV tasks to perform any register or memory access. The Figure 4 shows a PSS example with a component and its corresponding exec block. The exec block contains call to a high-level user function, “configure_dut”, which takes an argument of “struct” type. The Figure 5 shows the “configure_dut” code snippet along with the change in the C-APIs to enable reuse with SystemVerilog. The C-APIs are updated to call the SystemVerilog (SV) tasks to perform the register or memory accesses. The example shows the C-API method “configure_dut” that is called from the SV test, generated by a PSS tool, to configure the DUT. For performing register writes, it uses a method “write_register”, which can be used to invoke the corresponding SV task, “sv_write_register”, by using a compiler directive “WITH_SV_SIM”. With this approach, the C-APIs can be reused easily.

3) For VP Unit-Testing: C-APIs can be used for IP level testing where it is compiled as part of the VP model test-bench. This is useful in the verification environments without the CPU models. The Figure 5 shows the updates in C-APIs to achieve it. Similar to reuse for SV, the “write_register” method can be used to invoke a VP method, “tlm_write_register”, by using a compiler directive “NATIVE_BUILD”. The “tlm_write_register” invokes VP test-bench function to perform the register write.

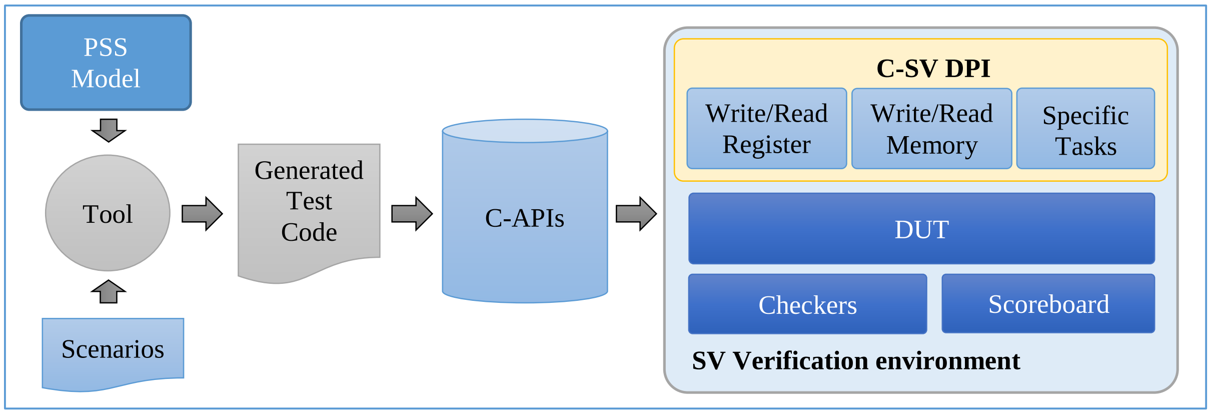

Figure 3 C-APIs reuse with SystemVerilog using DPI

```
component dut_c {
    action do_operation {
        //input and outputs
       dut_conf_s dut_conf; //dut configuration struct
    }
}


extend action dut_c::do_operation {
   exec body C = """
   #message(NONE, "Executing: modem_c::receive");
   configure_dut(dut_conf);
   check_dut();
   """;
}
```

Figure 4 PSS component and exec block example

```
void configure_dut (volatile dut_conf_s * a_conf)
{
    write_register (DUT_REG_ADDR, a_conf->val);
    …….
}

void write_register (unsigned int a_address, unsigned int a_data)
{
#ifdef  WITH_SV_SIM
    sv_write_register (a_address, a_data);
#elif  NATIVE_BUILD
        tlm_write_register (a_address, a_data);
#else
    *(a_address) =  data;
#endif
}
```

Figure 5 C-APIs changes to reuse SystemVerilog tasks

With the ‘C’ for implementation APIs, it is very easy to use the generated tests across various verification targets.

C. Executing PSS Tests on VP

The Figure 6 shows a scenario using the component example shown in Figure 3. The scenario uses “FILL” operation for attributes “input_format”, “output_format”, “operation” and “size”. The PSS tools use the FILL operation to generate test for each value of the attribute or combination of attributes. For the given example, the attribute “input_format” has twelve possible values, attribute “output format” has four possible values, attribute “operation” has eleven possible values and attribute “size” has 4 possible values, then tool will generate more than twenty-one hundred (2112) tests for this scenario.

```
extend component pss_top {
    action dut_scenario {
        activity {
            a0: do display_c::show with {
                dut_conf.input_format == FILL;
                dut_conf.output_format == FILL;
                dut_conf.operation == FILL;
                dut_conf.size == FILL;
            };
        };
    };
};
```

Figure 6 PSS scenario

The Figure 7 shows the pseudo C test code generated from the scenario shown above.

```
int main() {
    // top configuration object instantiation
    dut_conf_s dut_conf;
    //initialize the configuration object with values generated from the scenario
    dut_conf.input_format = IN_FMT0;
    dut_conf.output_format = OUT_FMT0;
    dut_conf.operation == OP_X;
    dut_conf.size == QVGA;
    ///////
    //exec block code of the component
    configure_dut(&dut_conf);
    check_dut();
}
```

Figure 7 Generated C test pseudo code

The Figure 8 shows the flow of executing tests on VP. The generated tests are cross-compiled for the target CPU and are executed on VP as if running on hardware. Due to its high simulation speed, thousands of tests can be executed in short time.

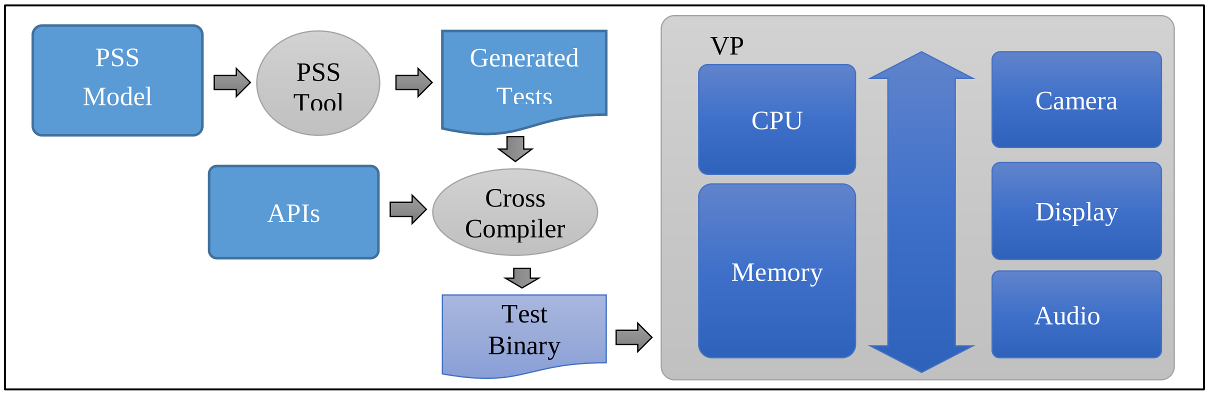

Figure 8 Execution of PSS tests on VP

D. Automated Checking of PSS Tests with VP

The PSS methodology covers mainly the specification to represent the test intent. The EDA PSS tools use this specification and generate the test that covers configuring and driving the stimulus to the DUT. The key part of asserting whether the test has passed or failed is to be decided by the user. By using ‘C’, we ensured that the implementation APIs are reused across verification platforms and hence are consistent by design. The similar approach was used to ensure the test pass criteria could be reused and is consistent across verification platforms. For this, the C-APIs were enhanced to perform the following checks.

1) Interrupt Integrity: This checks that all the required interrupts are triggered. It assumes that all the interrupts will be enabled in the test. If there is no interrupt, the test is declared as failed. This check is applicable and realized with other verification platforms – simulation and emulation. In addition, it also flags if an unexpected interrupt or an error-condition interrupt occurs.

2) Data Integrity: This check is used to ensure that the DUT output is as expected. Often, verification environments use output from a functional model as golden data. With C-APIs, the golden data is loaded in the memory at an address generated using PSS. At the end of the test, the C-APIs, compares the output data with the golden data to declare test as passed or failed. Some components also provide the CRC of the output data. In such cases, the comparison is made between just the CRC values from DUT with the golden CRC. With this approach, the same data integrity check can be performed for all verification platforms. The limitation of this approach is that the check is performed only at the DUT boundary. In case of failure, the data at various internal stages will have to be checked against golden data. Here again, VP is a better candidate as the VP model can be debugged very easily to find the stage which is causing the mismatch.

```
unsigned int dut_end_interrupt, dut_golden_crc;

//invoked from PSS exec block
void configure_dut (volatile dut_conf* a_conf)
{
   update_dut_golden_data();    //user function which updates the golden crc for the test
   register_interrupt_handlers(); //registers interrupt handlers with interrupt controller
   …….
}

//interrupt handler for end interrupt
void handle_dut_end_interrupt ()
{
   dut_end_interrupt = 1;
}

//invoked from PSS exec block after “configure_dut”
int check_dut()
{
   int status = 0;
   while(!dut_end_interrupt);            //wait for end interrupt to occur
   status = check_dut_interrupts() ;     //check all relevant interrupts are done
   status += check_dut_crc();            //compare DUT crc with the golden crc
   return status;
}
```

Figure 9 C-API code for interrupt and data integrity checks

The Figure 9 shows pseudo code for interrupt and data integrity checks in the C-APIs. Due to C-API reuse for DUT configuration and verification checks, it is easy to reproduce the issue across verification environments. For example a failure reported in the RTL simulation, can be easily reproduced in the VP. It makes it easy to identify whether the issue is in the PSS, verification environment or in the DUT. It saves a lot of time in reproducing the issues and debugging.

E. Capturing Golden Data with PSS

A reference model is a key part of any verification environment. These reference models are the bit-accurate functional models the DUT and hence called as functional models as well. Often, these models are created in 'C' and are reused for creating the VP models. These functional models take the configuration and stimulus in form of input files and dump the output in files. Along with the output at the DUT's boundary, outputs from various internal stages are also dumped. It makes it easy to locate the root-cause if there is any output mismatch with DUT. With PSS, these functional models can be reused to dump the golden data for data integrity checks as shown in Figure 10.

The PSS specification allows users to add additional code blocks to generate various files using the PSS tools. Using this feature, the PSS model is updated to generate the input configuration file as needed by the functional model. The values from various PSS attributes are used to generate the configuration file. It ensures that both the functional model and the DUT use same configuration, which avoids data mismatches due to configuration differences. The PSS tools generates the configuration file along with the test code. The function model is executed with this configuration file along with the input stimulus files, for example input images for a display sub-system. This gives the golden data which is used by the C-APIs during the test execution. As the VP models are reusing the functional model code, the data integrity checks can be performed successfully on VP. In case of mismatches, which may occur due to incorrect configuration file generated from PSS models or incorrect configuration by C-APIs, the VP can be used to debug the issue in short time.

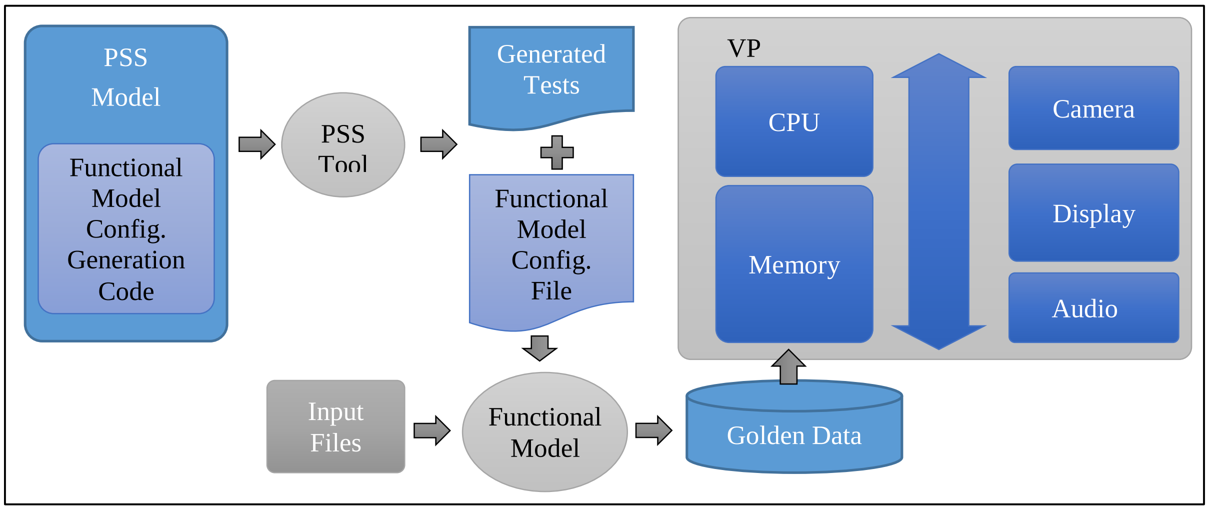

Figure 10 Golden Data Flow with PSS

F. Debugging PSS Tests with VP

As discussed earlier, one of the key problem areas of validating PSS models is debugging issues.

With RTL simulation, the slow simulation speed is the bottleneck – for each change to try to fix the issues iteration, it takes significant simulation time and causes delays. With large test cases, which may take more than 12 hours, it is not feasible to debug with RTL simulation.

With emulation, there is no support to debug the RTL code or to analyze the waveforms while the test is running. It requires user to specify the signals to be dumped in a waveform for analysis, which considerably slows down the execution speed. Moreover, user has to be careful while selecting the time range for which the signals are to be captured. If the event of interest happens outside the selected time range, it has to be done all over again.

With VP, a simple C++ debugger like gdb can be used to debug the issues. For ease of simulation and debugging, there are various tools available with additional debugging capabilities. For example,

1) Register Trace: Captures all the read-write operations performed in the simulation. It makes it easy to analyze and to identify issues in the DUT configuration. The configuration issues could be due to incorrect constraints in the PSS models or incorrect C-APIs. It is also useful to check the error conditions reported by the DUT.

2) ISS Debugging: Source code debuggers like gdb or Lauterbach's Trace32 can be attached to the CPU ISS model to debug the test code being executed by the CPU. It allows user to put analyze the values of the variables and CPU registers. It also provides the support, during test execution, to change the values of variables or CPU registers to trigger specific behaviors. In addition, it allows users to set break-points in the API code to debug issues.

3) Memory View and Dump: It allows user to check the value in the memory and dump it into a file if needed.

4) VP Model Debugging: Source code debugger like gdb can be used to debug the VP models itself. It allows users to look inside the model functionality to identify what is causing issues.

With these debug features, the PSS model and APIs can be debugged easily and in very short time.

IV. RESULTS

The methodology of using VP for PSS development was applied on a display sub-system. It used a smaller VP with only the display sub-system along with Cortex-M0 as CPU. With VP, we could achieve the following.

1. For the display sub-system, over two thousand tests were generated using "FILL" on just four attributes, ("FILL" generates a test for each possible value of the given attribute(s)).

2. Executed all the two thousand tests in just under five hours.

3. Setup a regression framework for PSS models using VP for execution. It helped us identify any failures due to changes in the PSS models.

Due to quick turn-around times with VP, we could identify many issues in the PSS models and APIs in very short time. It helped us qualify the PSS model fast.

After qualification with VP, the PSS model was used for tests on other platforms.

1) Emulation: PSS model and C-APIs were reused without any change for generating tests for emulation. It needed the generated test code and the C-APIs to be compiled in the emulation build framework. The interrupt and data integrity checks helped us identify failing tests and were fixed.

2) *Simulation*: PSS model was reused without any change for simulation platform. The C-APIs were reused with SV using the DPI functions. For data integrity, the existing SV checkers were reused which also performed data integrity check at the internal stages.

In Table 1, the difference in test execution and debug support with various verification platforms is shown. Considering only the test execution time and the debugging effort, we can safely deduce that VP reduced the PSS development effort significantly.

Table 1 Test Execution on Verification Platforms

Activity		VP	SoC RTL Simulation	Emulation
PSS Test Creation		<5mins	<5mins	<5mins
Test Execution Time		<5mins	~7hrs	~2hrs Average license queue time; >10mins execution time
Debugging	Source Code	Yes	Yes	No
	Waveforms	Yes	Yes	Yes
	Register View	Yes	No	Yes
	Memory View	Yes	No	Yes

V. CONCLUSION

A Virtual Platform (VP) is an ideal candidate for adopting the PSS methodology due to its early availability, functional accuracy, faster simulation speed and ease of debugging. The VP is available to large number of users and is more cost-effective than other platforms. In addition, it is not necessary for virtual platform to be complete for the full SoC; partial virtual platform is good enough to develop and qualify PSS model for the IPs available in virtual platform. The PSS models qualified using VP are more stable and available early for deployment across various verification platforms.

ACKNOWLEDGMENT

We would like to thank Mr. Balajee Sowrirajan, Corporate Vice-President, SSIR, Dr. Manoj Choudhary, Sr. Director, SSIR, for their continuous support in adopting PSS methodology. We would also like to thank the Cadence India team – Mr. Gnaneshwar Tatuskar and Mr. Harish Peta for helping us with the PSS methodology and the models development.

REFERENCES

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

[2] https://www.accellera.org/images/downloads/standards/pss/Portable_Test_Stimulus_Standard_v10a.pdf

[3] https://www.design-reuse.com/articles/15326/fast-virtual-prototyping-for-early-software-design-and-verification.html