2021 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE OCTOBER 26-27, 2021

Maximize PSS Reuse with Unified Test Realization Layer Across Verification Environments

Simranjit Singh, Ashwani Aggarwal, Suman Kumar Reddy Mekala, Arun K.R.
Samsung Semi-Conductors India R&D, Bangalore, India
simranjit.s@samsung.com, ashwani.a@samsung.com, suman.reddy@samsung.com, arun.kr1@samsung.com

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

Gnaneshwara Tatuskar
Cadence Design Systems, Bangalore, India
tatuskar@cadence.com

***Abstract*—Accellera's Portable Test and Stimulus Standard (PSS) addresses one of the key verification challenges of test reuse across various verification targets – from IP to system, from simulation to silicon. The PSS provides specification to represent the test intent in an abstract model that is agnostic to the verification target. Using PSS tools, the models are used to create and generate complex test cases. However, to realize or execute a test, it requires verification target specific set of methods or API, increasing portability effort significantly. This paper describes an approach to have a unified realization layer to maximize reuse with PSS.**

Keywords—Accellera, PSS, Verification, Reuse, Simulation, Emulation, SystemVerilog

I. Introduction

With growing complexity of SoCs, the traditional verification approach is becoming more and more challenging to meet the ever-shrinking time-to-market windows. It requires significant effort to develop verification environments to create tests at various verification levels like at IP, sub-systems, system and post-silicon. With little to no reuse, the effort goes in creating tests with same verification intent for each of the verification targets. In addition, highly complex system-level scenarios require more sophisticated tools and methodologies.

The Accellera's Portable Test and Stimulus Standard [1][2] (PSS) reduces the verification effort by enabling reuse across various verification levels. With PSS, the test intent is decoupled from the test implementation by using a high-level model to specify the verification intent. Using a PSS tool, the tests are generated for the verification target. To be able to realize or execute the test in the target verification environment, it requires the user to develop the implementation methods or APIs. These implementation APIs are used to extract the Design-Under-Test (DUT) configuration from the generated test and configure it using the verification environment. It makes the APIs verification target specific. For example, for a SystemVerilog (SV) verification environment, the implementation APIs are developed usually in SV whereas for emulator, Virtual Prototyping [3] (VP) and Silicon, 'C' is used. The Figure 1 shows the test reuse with PSS methodology.

Using PSS, the overall verification effort is reduced greatly. However, the test realization requires significant effort to develop implementation APIs that are verification target specific. It makes the reuse more challenging. The IP level verification environment are often developed using SV and hence need implementation APIs in SV. The PSS model and the SV APIs can be reused at the sub-system level testing with SV verification environment. For the system level testing using emulator, Virtual Prototype (VP) or Silicon, the SV APIs cannot be reused. Hence, it requires the APIs to be developed again and importantly, to be qualified again. With SV APIs, there is high level of confidence that the APIs are correct, as they have been used in IP and sub-system level verification. But, with the new APIs developed for system level testing, there is always doubt that whether the issue is in the DUT or in the APIs. It could lead to significant effort in debugging failures due to bugs in the APIs. In addition, with difference in the APIs, it is not easy to reproduce the issues from one verification environment to another which affects the portability of the tests. Moreover, it requires the teams to maintain two set of APIs and update for each SoC.

Due to this gap in reuse of implementation APIs, adopting PSS methodology becomes a challenge and often teams prefer to continue with the traditional approach, as the initial effort is too great.

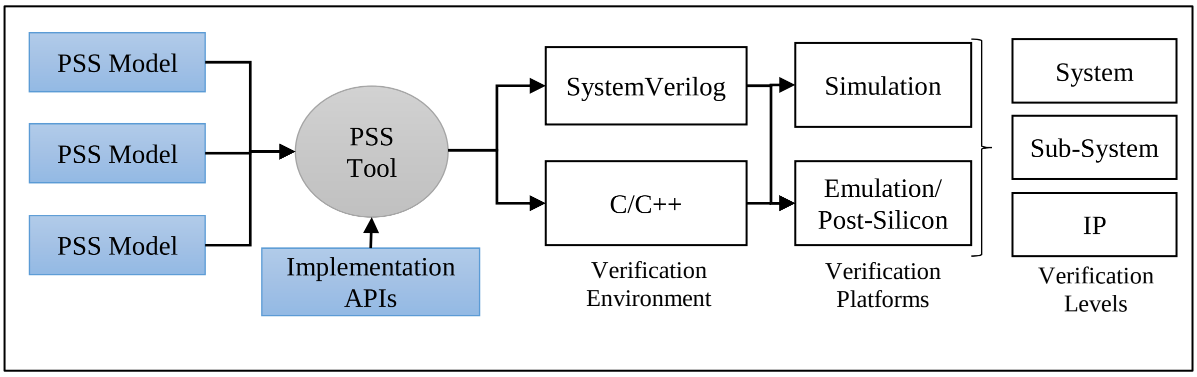

Figure 1 Reuse with PSS

To make the adoption more beneficial for all verification teams involved, it is important to address this hurdle in reuse of the implementation APIs. This paper describes an approach to have a unified test realization layer to support various verification targets.

II. PSS TEST REALIZATION

The PSS test realization has mainly two aspects – 1) test generation from the PSS tool and 2) implementation APIs.

To enable test code generation from PSS tools, the PSS specification provides construct called "exec" blocks in which users can specify the code to perform the operations for the corresponding "action". The Figure 2 shows the code of exec block of an action. The example shows the exec block code of an action called display, which simply calls a method "configure_dut" with an argument, which is an attribute in the PSS model.

```
extend display {
  exec body {
    configure_dut(a);
  }
}
```

Figure 2 PSS exec block

Using the actions in a given scenario, the PSS tools generate the test code by embedding the corresponding exec body code in it. For the example in Figure 2, it will simply generate code with call to the method "configure_dut".

The target language for the generated test code is controlled via options to the PSS tools. For examples, Cadence Perspec[4], target language can be selected as C or SV.

In the example in Figure 2, the exec body is not specific to any target language. In this case, the tool will generate the exec body code in test for both, C or SV, target languages. However, if the target language is specified with the exec body, C or SV, then tool will generate code only for the exec block specific to the selected target language.

The other aspect, implementation APIs, provides the implementation of the methods called from the exec blocks. For the example in Figure 2, the implementation of the "configure_dut" method has to be provided by the implementation APIs. This completes the flow of scenario creation from PSS to realization in the target verification environment.

For the unified test realization, it is crucial that the exec body is not specific to a target language. As PSS tools can generate code for the specified target test language, it allows users to maintain only one exec block for multiple verification environments. In addition, it ensures the test behaviour is consistent in all verification environments that makes it easy to reproduce issues. The following section describes how this unified exec blocks approach is used with implementation APIs to achieve reuse.

III. UNIFIED TEST REALIZATION APIS

The unified test realization approach is based on the idea of using single test language for the implementation

APIs for all verification targets. The best candidate to achieve this goal is the ‘C’ language as it can be used in all the verification environments. The following sections describe the usage of ‘C’ APIs for various verification targets.

A. Emulator and Silicon

In emulator and silicon verification environment, the tests are usually developed in ‘C’ to verify sub-systems or system level scenarios. These tests are cross-compiled for the target CPU using the corresponding tool-chain. Hence, the ideal candidate for the PSS test implementation APIs is ‘C’. With PSS, it requires the PSS tool to generate the test in ‘C’. The Figure 3 shows the flow with the PSS generated tests using ‘C’.

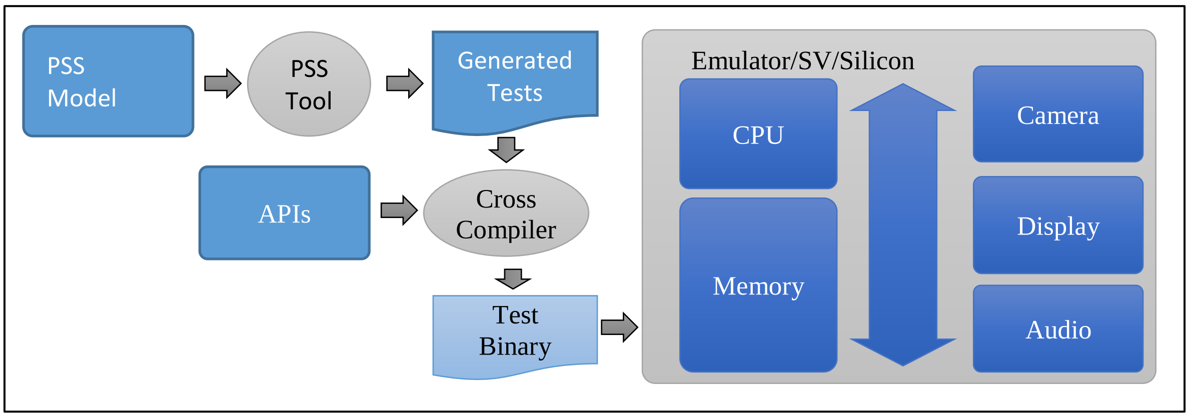

Figure 3 C-APIs with CPU driven system level testing

B. RTL Simulation

With RTL simulation, the verification environments are developed using High-level Verification Languages (HVL) like SV or specman-e. At a higher level, a verification environment has two major aspects:

1) Stimulus: Includes various methods to generate DUT configuration and input vectors as per the test internt and to program the DUT and test-bench components accordingly.

2) Checking: Includes test-components like monitors, checker and scoreboard to verify the tests.

With PSS, the configuration generation part of the stimulus aspect is done using the attributes in PSS model. The methods to configure the DUT and other test-bench components are reused. However, as discussed earlier, these test sequences cannot be reused for other verification environments.

With the unified test realization approach, the C-APIs are reused for simulation environment instead of the SV methods. This is achieved by using the C-SV Direct Programming Interface [5] (DPI), which allows the SV tasks to be called from C-APIs. This mechanism is discussed in the following sections.

C. Virtual Prototype Verification

Virtual Prototyping (VP) is a proven methodology, which is widely used to enable early software development. A VP is an abstract functionally accurate simulation model of the hardware developed using C++/SystemC. Due to its higher abstraction than RTL, it is easy to model and is available earlier than hardware. The VP enables software teams to develop and verify software, much before the hardware is available. Once the hardware is available, the software being-up on hardware is usually done within few days as software is already verified on VP. To ensure that the software can be ported to the hardware, VP needs to be functionally accurate to the hardware, which requires significant effort in VP verification, both at the unit level and at the system level. As this verification is very close to hardware verification, VP also is a candidate for reusing PSS based verification. For VP as well, C is the natural choice for the implementation APIs. For the system level testing with CPU based environment, the C-APIs can be used similar to the emulation environment as shown in Figure 3. For the unit-level testing, the C-APIs can be compiled with the VP environment as VP models are also in C++.

As shown with the above verification targets, by using C as language for implementation APIs, the effort of developing verification target specific APIs can be avoided. Hence, with C-APIs the unified test realization can be achieved.

IV. Unified Test Realization in SV

The flow with C-APIs reuse in a SV verification environment is shown in the Figure 4. In this flow, the PSS tool is used to generate the test code that invokes the C-APIs.

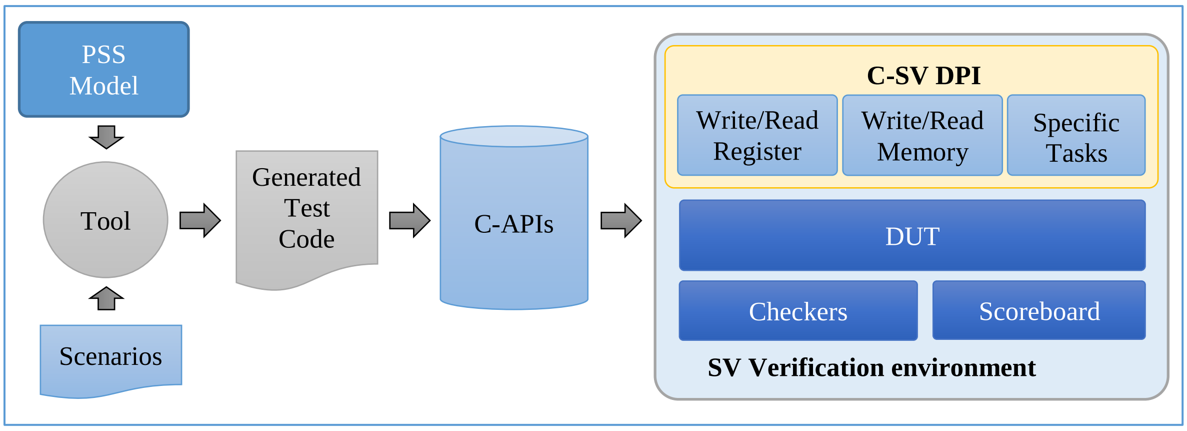

Figure 4 C-API reuse in SV environment using DPI

As discussed above, the stimulus aspect of the SV verification environment is replaced with the C-APIs. The C-APIs mostly remain unchanged for the SV verification environment – process the configuration generated by the PSS test and program the DUT or verification components accordingly. The major change happens at the low-level operations where all the register and memory access operations are mapped to corresponding SV methods using DPI. In addition, the C-APIs can call SV methods to perform specific operations e.g. call a method to load a data binary file to the DUT memory. Importantly, the verification components like checkers and scoreboard remain unchanged.

A. *SV Code Generation*

The important part of this flow is the test generation support from the PSS tool. Although the C-APIs are reused, the generated test needs to be integrated with the SV verification environment. Hence, the tool must provide the support to generate SV test code required to integrate with the verification environment and handle the DPI calls to reuse the C-APIs. For the unified test realization, the Cadence Perspec tool was used to address these requirements.

With SV as the test language, the Perspec tool generates a “perspec_top” SV module and C test code where the PSS exec block codes are embedded. This C test code is similar to what is generated when C is selected as the test language. The “perspec_top” module is integrated with the existing SV verification environment as shown in Figure 5.

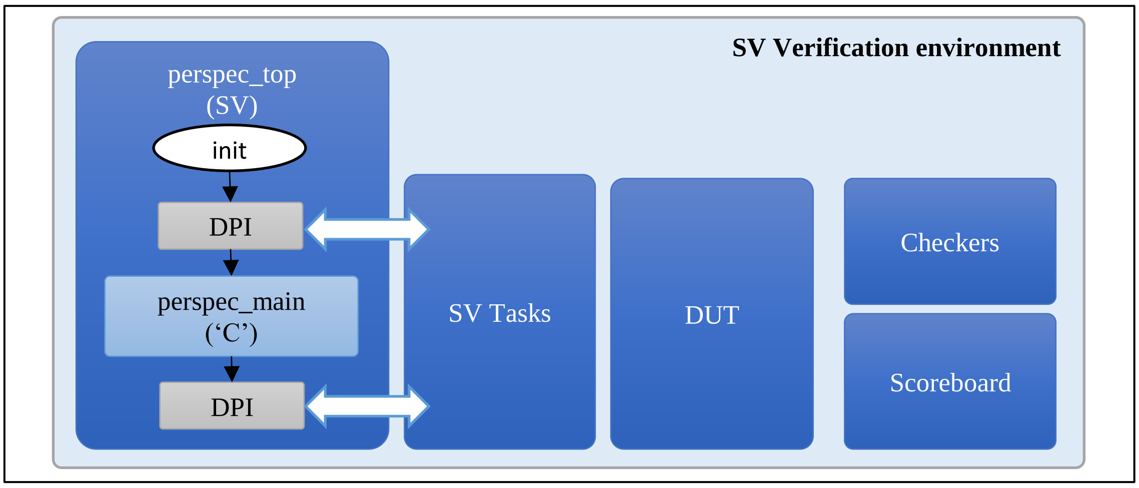

Figure 5 Perspec SV Test Generation

In the initial block of the “perspec_top”, it invokes the “perspec_main” which is the C test code. The tool automatically generates the DPI implementation needed to invoke the “perspec_main” from “perspec_top”. The Figure 6 shows an example code snippet of the generated “perspec_top”. In addition to a call to the “perspec_main”, it contains declaration of a task “sv_configure_dut” which is the top level SV task defined in the SV verification environment.

```
task sv_configure_dut()
perspec_top perspec_top();

initial begin
   … // initialize DUT
   perspec_top.perspec_main();
```

Figure 6 Generated perspec_top SV module

In the “perspec_main”, it calls the methods as specified by the user for the exec blocks of actions in the test scenario. For the example shown in Figure 2, it will contain a call to the “configure_dut” method. As the “sv_configure_dut” is the test implementation method, the “configure_dut” would need to call the “sv_configure_dut” task internally. It is achieved by generating an additional SV file from the PSS model. The Figure 7 shows an example implementation of such exec block. It defines the “configure_dut” as an SV task which internally invokes the “sv_configure_dut’ task. Using this exec block, the tool generates the “perspec_exports.sv” file, which contains the SV implementation of “configure_dut”. In addition, the DPI implementation needed to bind C and SV implementation of “configure_dut” is generated in “perspec_top” module.

```
import SV function configure_dut;

exec file “perspec_exports.sv”
   task automatic configure_dut()
      sv_configure_dut();
   endtask
```

Figure 7 PSS exec block to map C methods to SV

As shown with above examples, the required DPI implemenation to interface C and SV code can be generated. This utility is very useful with the unified test realization implementation.

B. C-API Reuse in SV

As shown in Figure 4, the C-APIs can be reused with the SV by mapping the register and memory access operations in C to SV. It requires the changes in the PSS exec block for “perspec_exports.sv” as shown in Figure 8.

```
import SV function sv_read_register;
import SV function sv_write_register;

exec file “perspec_exports.sv”
   task automatic sv_read_register (
            input int unsigned addr,
            output int unsigned data)
      data = vip_read_register(addr);
   endtask

   task automatic sv_write_register (
             input int unsigned addr,
             input int unsigned data)
      vip_write_register(addr,data);
   endtask
```

Figure 8 PSS exec block to map register access methods

In this approach, SV tasks to perform register read and write operations are defined. Internally, it uses the corresponding SV tasks from the verification components to perform the operations. In the example below, the “sv_write_register” is calling an existing SV task “vip_write_register”.

C. C-APIs Updates

To reuse the C-APIs in SV environment, it requires change in the way the register or memory access operations are done in the C-APIs. The Figure 9 shows the changes in the code to use the SV write register task when its used in SV environment. The “configure_dut” method calls a generic “write_register” method to configure the DUT. In the “write_register” method, depending on a compiler directive, it decides whether to call SV write register task or perform the operation. Similarly, the read register method is updated to invoke corresponding SV task. The same approach is used for the memory operations where SV tasks are reused. In addition, using compiler directives, calls to various SV tasks can be embedded in the C-APIs e.g. load input data files at the given addresses, program SV specific verification components, invoking certain SV tasks to perform additional checks for the given scenario.

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

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

Figure 9 C-APIs with SV tasks

V. Application of Unified Test Realization Approach

The unified test realization approach was applied for creating PSS models and implementation APIs for camera and display sub-systems.

1) PSS models: Exec blocks independent of the target language.
2) Implemantation APIs: Developed in C.

With this approach, the PSS model generated tests with the C-APIs were used for various verification targets.

A. Virtual Platform

The PSS model and the implementation C-APIs development started with using VP as the verification environment. Due to early availability and faster simulation, VP helped in quickly identifying major bugs in the PSS model and the C-APIs. The C-APIs were used in CPU-based environment, where the PSS generated C test code and C-APIs were cross-compiled for the target CPU. In addition, the tests and the C-APIs were reused for unit-level testing for few components. The unit-test environment did not use CPU model and the C-APIs were compiled natively along with the VP models. For the unit-testing environment, an approach similar to as shown in Figure 9 was followed, where the C-APIs were updated to map the register and memory accesses to the VP functions. Hence, the C-APIs were used in both, CPU based and non-CPU based VP environments.

B. Emulation

Like the VP environment, the PSS flow was used to generate tests in C. The generated C tests and C-APIs were cross-compiled for the target CPU in emulation environment. The C-APIs did not require any change for emulation environment as the register and memory access operations remained same as in VP.

C. For Simulation

The PSS model was used to generate tests in SV language. The C-APIs were reused by mapping the register and memory access operations to the corresponding SV methods. In addition, the following cases needed special handling.

- Interrupts: In emulation and VP environments, the generated tests were executed on the CPU. For DUT interrupts testing, the interrupts were enabled and corresponding Interrupts Service Routines (ISR) were added in the C-APIs. The ISRs were registered with the system interrupt controller and a call to an ISR would indicate that the interrupt has been received and is working as expected. The ISRs also clear the interrupts using the clear registers. This way, the C-APIs ensured the interrupts worked correctly. But, in SV environment, CPU was not used and hence, the ISRs approach could not be used. To address this, a SV task was created for each interrupt and was made sensitive to the corresponding interrupt signal in the RTL. Internally, on interrupt signal getting assert in RTL, the task invokes the corresponding ISR from the C-APIs. After the ISR execution, the control returns back to the task, which performs check on the interrupt signal to ensure that the ISR cleared the interrupt. With this approach, the ISRs defined in C-APIs were used for all environments.

- Delays: In emulation and VP environments, for consuming delays, the C-APIs used simple "for" loops with a variable being incremented for the given count. Each increment operation corresponds to a CPU clock cycle. But in SV, as there was no CPU used, the for loops will be executed in zero time. To address this issue, the C-APIs were updated to call a SV task with the count as a parameter, where the SV task would consume the required delay.

The C-APIs were also updated to handle additional SV specific configuration or tests by directly using existing SV tasks.

- Image Loading: For display sub-system, it required input images to be loaded in the memory for testing. The PSS model was updated to capture the image file to be used for the test and through C-APIs, the SV method to load the image data was invoked.
- Frequency Change Support: The C-APIs did not support changing frequency, as it was not a requirement for the emulation and VP environment. For SV environment, the existing SV task to control the frequency was reused. The PSS model was enhanced to add attributes to control frequency configuration and the C-APIs were updated to invoke the corresponding SV task.
- SV Specific Checks: In SV environment, additional checks are performed on the RTL signals to assert correct functionality of the DUT. In C-APIs, these RTL signals cannot be probed directly. To handle such scenarios, the C-APIs were updated to use existing SV tasks to perform these checks. Using the PSS generated test, the C-APIs could determine the DUT configuration and invoke the SV tasks to perform the checks relevant to the configuration.

VI. Results

With unified test realization layer, the effort to develop the SV APIs was avoided. By reusing C-APIs, it ensured that the test implementation remained consistent across verification environment. It made it easy to reproduce issues from one environment to another. For example, issues discovered with SV environment were reproduced with emulator seamlessly. It helped in identifying whether issue is in DUT or PSS model and APIs, which saved significant effort in debugging false failures. In addition, due to emulator's faster execution speed and ease of debugging with tools like Trace32, issues could be fixed faster than in RTL simulation environment.

Table 1 PSS for Display Sub-System

Effort	VP	Simulation	Emulation
PSS model	15 days	-	-
C-APIs	20 days	3 days	-
Test executed	2000	70	85

Hence, with unified test realization approach, the PSS methodology truly enables test reuse across verification targets. This reduced the effort of PSS development and encouraged teams to adopt PSS.

Acknowledgment

We would like to thank Mr. Balajee Sowrirajan, Corporate Vice-President, SSIR, Dr. Manoj Choudhary, Sr. Director, SSIR, Mr. Seonil Brian Choi, Master and Mr. Woojoo Space Kim, Principal Engineer for their continuous support in adopting PSS methodology. We would also like to thank Mr. Gnaneshwar Tatuskar, Cadence Design Systems India for helping us with the PSS methodology and the models development.

References

[1] Accellera Portable Test and Stimulus Standard Version 1.0a Language Reference Manual

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

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

[4] Cadence Perspec Tool User Guide
[5] Direct Programming Interface - https://www.doulos.com/knowhow/systemverilog/systemverilog-tutorials/systemverilog-dpi-tutorial/