2024 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE
MUNICH, GERMANY
OCTOBER 15-16, 2024

A Novel Approach for HW/SW Co-Verification: Leveraging PSS to Orchestrate UVM and C Tests

Tom Fitzpatrick, Siemens EDA, Groton, MA, USA (tom.fitzpatrick@siemens.com)

Vishal Baskar, Siemens EDA, Fremont, CA, USA (vishal.baskar@siemens.com)

Wael Abdelaziz Mahmoud, Siemens EDA, Cairo, Egypt (wael.mahmoud@siemens.com)

Mohamed Nafea, Siemens EDA, Cairo, Egypt (mohamed.nafea@siemens.com)

***Abstract*—The complexity of System on Chips (SoCs) continues to grow rapidly. Accordingly, new standards and methodologies are introduced to overcome these verification challenges. The Portable Test and Stimulus Standard (PSS) from Accellera is one of the standard examples used to pursue such challenges. In this paper we will show a methodology to use PSS to orchestrate the process of HW/SW co-verification by driving UVM and C tests and controlling the interaction between them.**

***Keywords*— Portable Stimulus; HW/SW Co-Verification; UVM; Constrained-Random**

Introduction

The complexity of System on Chips (SoCs) continues to grow rapidly with the integration of more functionality onto a single chip. As a result, traditional verification methodologies struggle to keep pace with the growing complexities, leading to longer development cycles and increased risk of design errors. In response to this challenge, hardware and software co-verification emerges as a pivotal technique in validating SoC designs. However, there is much testing that needs to be done at the block- and subsystem-level before software-driven system-level testing can begin. Traditionally, this early testing is done in simulation using the Universal Verification Methodology (UVM) to create a set of modular reusable verification components assembled into "testbenches" (UVM environments), including stimulus sequences to exercise specific transactions across the different interfaces of the device-under-test (DUT). When the full SoC design is assembled, it usually includes an embedded processor and requires software, usually written in C, to exercise the same functionality that was previously modeled as UVM transactions driven by an agent executing a sequence.

This need to duplicate the verification implementation, first in UVM and then later in C, is a substantial bottleneck that negatively impacts the verification productivity of many projects. The Portable Test and Stimulus Standard (PSS) from Accellera [1] was architected specifically to address this need to reuse verification intent throughout a project across multiple, possibly heterogeneous, implementations. By definition, a PSS test defines a set of actions that represent the verification behaviors required to exercise desired functionality of the DUT. These actions themselves can be defined at multiple levels of abstraction, from basic bus read/write operations to higher-level actions, such as DMA transfers, message passing, or other behaviors. An action that may be driven by a UVM agent at the block level may easily represent a C-level function that may be called from a software driver running in the embedded processor at the SoC level.

A PSS test is composed of the following:

1. A set of actions that define the set of behaviors to be executed.
2. An activity that defines the critical actions and their relative scheduling constraints
3. Data flow requirements between actions
4. Additional data and scheduling constraints
5. Target-specific implementation(s) of each action

From these elements, a PSS tool can generate a target-specific implementation of the overall test scenario, check Figure 1: PSS orchestrating UVM & C testsFigure 1. To the extent that the schedule and constraints are flexible enough to permit multiple scenarios to satisfy the specified verification intent, a PSS tool can effectively create a constrained-random implementation of the desired scenario. This scenario-level randomization makes it easier to create complex test scenarios from easy-to-describe tests. This approach also provides the secondary benefit of allowing the block-level verification team to create a library of actions that can be reused at the subsystem and system level.

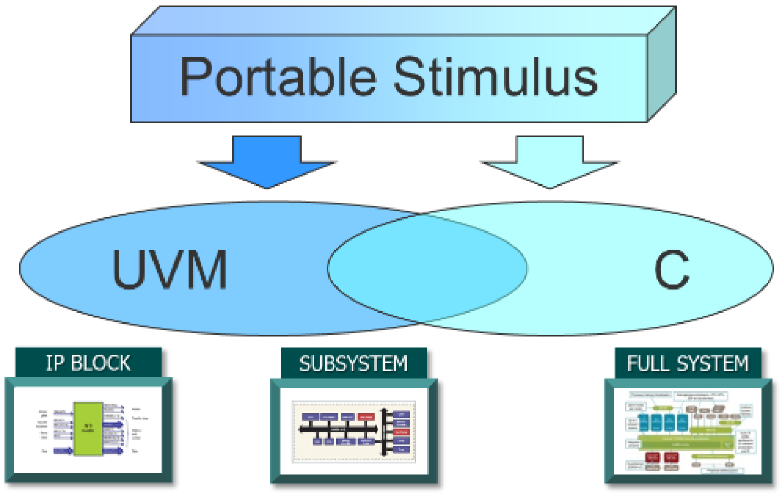


Figure 1: PSS orchestrating UVM & C tests.

When targeting a UVM environment, the PSS test scenario can be realized as a UVM virtual sequence that implements the schedule of actions as defined in the activity in PSS. Each action may itself be realized as the invocation of a UVM transaction-level sequence on a UVM agent. As multiple blocks are assembled into a subsystem, a PSS model may be created that schedules the actions defined for each sub-block into an activity that defines the subsystem-level test. If this test is also targeted at a UVM environment, the resulting realization will be a subsystem-level UVM virtual sequence that will still ultimately invoke UVM transaction sequences on the agents, but it is much easier to coordinate these behaviors across blocks and interfaces from the PSS model than to write UVM sequences manually to try and create the desired scenarios.

When we reach the system level, as mentioned above, the design will include an embedded processor model that will represent part of the system. In a fully-UVM environment, this processor will be modeled by a UVM agent that will drive transactions on the bus to make the other blocks do the desired operations. Often this UVM transactor will be replaced by a high-level C model and eventually it will be replaced by an RTL model of the processor that will run C code. Using PSS, we can continue to model the verification intent at the SoC level by merging the sets of actions from the lower levels into the desired scenarios. The "P" in "PSS" lets us replace the UVM transaction-based implementation of the processor's actions with C implementations of the same actions. If the overall test environment is still UVM-based, which is often the case while simulation is still being used, then the paradigm of creating a UVM virtual sequence to coordinate the behaviors can be preserved, as long as the C implementations of the processor's actions can be controlled from the sequence.

Whether the embedded software on the processor (or modeled as such) performs bus operations to make the other blocks perform, or implements specific functionality required by the application, the behaviors for which the processor is responsible are still represented as PSS actions. Each specific action implementation may be realized as part of the driver software that will ultimately run on the final chip, but partitioning the task via the PSS model allows the development of these driver pieces, and the overall driver itself, to begin much earlier in the process. This is what hardware-software co-verification is all about.

This paper will present a PSS-based framework to improve hardware and software co-verification techniques in the verification process of SoCs. We will explore how an SoC-level test can be composed from block- and subsystem-level PSS models and how a generated UVM virtual sequence can be used to schedule the action realizations, whether in UVM or in C, to implement the schedule defined in the test. By relying on the PSS solver to randomly create scenarios that satisfy the verification intent, it is possible to create a set of directed tests, one for each valid solution of the scenario space, that together comprise a test suite with all the advantages that runtime constrained randomization has provided for many years.

We will see how a PSS-compliant tool can generate both UVM and C implementations of desired actions and create the required infrastructure using DPI-C to support the execution of these actions in a UVM-based heterogeneous verification environment [2] and explore how to configure a tool flow to ease the integration of separate C tests in a UVM environment to allow true hardware-software co-verification. We will also explore how a fully-PSS-based verification flow can make it easier to reach this goal.

PRELIMINARIES AND RELATED WORK

Traditional co-design methodologies, where hardware and software are developed independently after early partitioning decisions, are no longer suitable due to the lack of unified representation, simulation, and synthesis frameworks [3]. Functional verification involves running detailed simulations to ensure hardware designs meet specifications, employing EDA tools like QuestaSIM, while emulation accelerates the process using hardware emulators. Methodologies such as Universal Verification Methodology (UVM) and Portable Stimulus Standard (PSS) facilitate a reusable, scalable, and automated verification environment [1]. Recently, a C-based approach to hardware modeling has become necessary since C and C++ are dominant languages among chip architects and software engineers. This approach allows for more natural partitioning of functionality between hardware and software, enabling thorough verification during early design stages, reducing risk, boosting productivity, and improving confidence in design performance [4].

The co-verification of System-on-Chip (SoC) designs involves several challenges that stem from the inherent complexity and interdependence of hardware and software components. As embedded systems grow more sophisticated, addressing these challenges becomes crucial for ensuring reliable and efficient system performance.

- The simultaneous verification of hardware and software components increases the complexity of the verification process. Ensuring that both domains interact correctly requires detailed models and comprehensive test scenarios.
- Effective co-verification necessitates the integration of various tools and frameworks used for hardware and software verification. Developing robust interfaces and protocols for tool integration is essential to facilitate smooth co-verification processes.
- HW/SW co-verification is resource-intensive, requiring significant computational power and skilled personnel. The need for extensive simulation, emulation, and debugging can strain available resources.
- Identifying and resolving issues in co-verification is inherently more complex than in traditional verification. Problems that arise can span both hardware and software domains, necessitating expertise in both areas.
- Ensuring proper timing and synchronization between hardware and software components is a critical challenge. Timing mismatches can lead to functional errors and performance issues.
- Scaling the co-verification process to accommodate large and complex SoC designs is challenging. As the size and complexity of SoCs increases, so does the difficulty in managing and verifying all interactions and states.

Overcoming these challenges necessitates the use of sophisticated verification technologies, strong integration frameworks, and highly skilled engineering teams. As HW/SW co-verification progresses, continuous research and development are crucial to improving the efficiency and effectiveness of SoC verification processes. This ensures that increasingly complex embedded systems achieve the necessary performance and reliability standards.

In addition to traditional methodologies, the industry is increasingly adopting hybrid co-verification approaches to address the growing complexity of SoCs. Hardware-assisted verification, combining emulation and FPGA prototyping, offers significant speed advantages and early software validation, as highlighted in [5]. Virtual platforms, such as QEMU, allow for high-level software testing and debugging before hardware availability, reducing development time and cost. Transaction-level modeling (TLM) using Specman and SystemC with TLM ports enables fast simulation speeds by abstracting away low-level details, facilitating early design space exploration and performance analysis [6]. Additionally, in the past, authors have worked on a C/C++-based design environment for hardware/software co-verification, where they describe both hardware and software throughout the design flow using C/C++ [2]. The adoption of these advanced co-verification methodologies ensures comprehensive validation, improves efficiency, and accelerates time-to-market for complex SoC designs.

In this paper, we introduce the combination of the PSS, UVM, and C++ wherein it offers superior advantages for HW/SW co-verification in SoC designs. PSS enables the creation of reusable and adaptable verification scenarios across multiple platforms, enhancing productivity and reducing verification time. UVM provides a robust and standardized framework for creating modular and reusable verification components, promoting consistency, and reducing duplication of effort. The use of C++ in conjunction with PSS and UVM allows for high-performance and flexible testbench development, leveraging object-oriented programming to manage complexity and improve maintainability.

PROBLEM FORMULATION

As we have seen in the above section, HW/SV co-verification is becoming incredibly challenging. We need to find a way to cross the boundaries between UVM-based tests and C-tests in an organized and seamless way. PSS is the best language and methodology to orchestrate the execution of HW/SV co-verification, by defining the actions' abstract scenario, which can be solved in a random fashion. This PSS model will manage the design configurations, attributes, resources, and memory management. Also, it will define a realization model which can drive synchronization of UVM-based tests and C-tests. The next sections will discuss some of the building blocks using PSS to help achieve HW/SV co-verification seamless integration as well as a few case studies using this proposed methodology.

INTRO TO PSS CONSTRUCTS

The Portable Test and Stimulus Standard defines a declarative language to model the critical verification intent at an abstract level as well as target-specific implementations of the abstract model to allow the generation of target-specific tests that implement the critical verification intent in the desired target test environment. The abstract model is comprised of a set of constructs to

- encapsulate specific behaviors that must be executed,
- define the relative schedule of execution of these behaviors,
- specify the data/communication between them,
- organize the behaviors relative to the target components they represent, and
- constrain the scheduling and interaction of behaviors based on the resources available in the target system.

Since PSS was designed from the start to facilitate generation of test implementations in multiple target languages, there is a definite separation between the abstract model, which is by itself target-language-agnostic, and the realization layer, which specifies the target-specific implementation for relevant parts of the abstract model.

We will not provide a language tutorial here, but it is important for the reader to understand these concepts and recognize the PSS language constructs used to represent them to follow the examples provided in the paper.

PSS Action

A PSS *action* defines a specific behavior that is to be exercised. An action may be *atomic*, in which case its realization is defined by the inclusion of an *exec block*, or it may be *compound*, in which case it includes an *activity* that defines the schedule of sub-actions to be executed. All scenarios in PSS begin with a *root action* which is typically a compound action that defines an activity to schedule the execution of other actions to compose the scenario. Eventually each of these actions resolves down to a hierarchical activity schedule of atomic actions, each of which is represented by an exec block to define the target-specific implementation of the scenario.

PSS Activity

As mentioned, an *activity* describes the schedule of sub-actions that make up a compound action. Sub-actions are *traversed* according to the schedule, including actions being traversed sequentially (the default) or in parallel, where the parallel actions share common predecessor actions, which must all complete before the parallel actions may be traversed, and common successor actions, which will not be traversed until all parallel action traversals complete. Action traversal statements in an activity may also include in-line constraints on the traversal of a sub-action.

PSS Flow Objects

Data flow between actions is defined by *data flow objects* in PSS, which are specialized data *struct*s that include additional semantics to govern the scheduling of actions that are bound to them. Actions may declare data flow objects as either *input* or *output* ports. *Buffer* flow objects require that the producer action, which outputs the

buffer, completes its traversal before the consumer action, which inputs the buffer, may begin its traversal. Thus, buffer objects model storage in a PSS abstract model.
Stream objects require both the producer and consumer to operate in parallel. That is, they represent transient data in a PSS model.
Data flow objects may include specific data fields and may define constraints on these fields. Actions that input or output data flow objects may also define constraints on the data fields of the flow object ports. The constraint solver shall generate random values for the flow object fields that satisfy all constraints on the field.

PSS Components

A PSS *component* models a feature of the target environment, that will be responsible for implementing a give set of actions. Components act both as hierarchical containers for action and object instantiation, as well as namespaces for their declarations. All PSS models must start with a top-level component which is names *pss_top* by default.

PSS Packages

PSS also supports *packages*, which serve as namespaces for item declarations, and may be imported into other elements of a PSS model. PSS packages are often used to encapsulate sets of extensions for components, actions and flow objects to define the target-specific exec blocks and other elements to refine an abstract model to a specific target implementation.

PSS Resources

A *resource* is PSS is used to represent a specific element of the target environment required to successfully traverse an action. The action may *lock* the resource for its exclusive use, or it may *share* the resource, in which case other actions are also allowed to share the same resource. For example, an action that models a DMA transfer, may lock a channel resource. While the DMA transfer action is being traversed, a specific instance of the channel resource will be assigned to it and no other action may lock the same channel instance during that time.

PSS Inferencing

Because PSS is a declarative language, the specification of components, actions, activities, flow objects and other model elements allows a scenario to be solved from a partial specification of the critical verification intent required. The solver will use the definition of flow object bindings among actions, constraints on shared fields, resource allocation, and other considerations to *infer* the traversal of additional actions beyond those explicitly traversed in the model to ensure the legality of the desired scenario. For example, if a DMA transfer action requires a data buffer object as an input, the solver would have to infer the traversal of another action that can supply a data buffer of the required type with compatible constraints.

PSS Target Template Exec Block

An exec block in PSS may specify a target-language-specific implementation of the behavior using a *target-template* block, which defines a string literal containing target-language code, optionally embedding references to fields in the PSS description. Target templates are defined for SystemVerilog and C code in PSS models. Thus, to define separate implementations for a given atomic action, a different exec block is required for each desired target implementation.

PSS Procedural Interface

PSS also allows tasks or functions in SystemVerilog or C to be *imported* into the PSS model. The exec block using the procedural interface will call the imported method and pass references to PSS fields as arguments to the method. In this case, a single PSS exec block can be mapped to separate target implementations by swapping in the desired package to define exactly which method implementation is to be used.

PSS Procedural Layer

PSS exec blocks may also include the specification of abstract procedural constructs such as loop, if-else and other statements that allow the specification of complex control flow algorithms in a single exec block. These types of ‘exec’ blocks often wind up calling simplified target methods, like register read or write operations of other relatively simple methods. The control flow is reflected in the target implementation language when the test scenario is generated, with the target-specific implementation of the underlying methods called appropriately.

PROPOSED METHODOLOGY

This section covers the proposed methodology to tackle this problem using PSS. Figure 2 shows a flow chart for the proposed methodology in this paper, which can be summarized in the following steps:

1. Explore testbench environment and develop PSS model.
2. Debug PSS (Abstract & Solved) models using Siemens's PSS-compliant UI (User Interface) Debugger tool.
3. Implement the realization model by adding the contents of actions' exec blocks for both C and UVM tests.
4. Solve PSS module and generate corresponding tests (C & UVM) using Siemens's PSS-compliant tool.
5. Integrate generated UVM/C tests into UVM test bench using SV interfaces and DPI-C (details will be provided)
6. Compile and simulate the TB (running tests on target platforms)

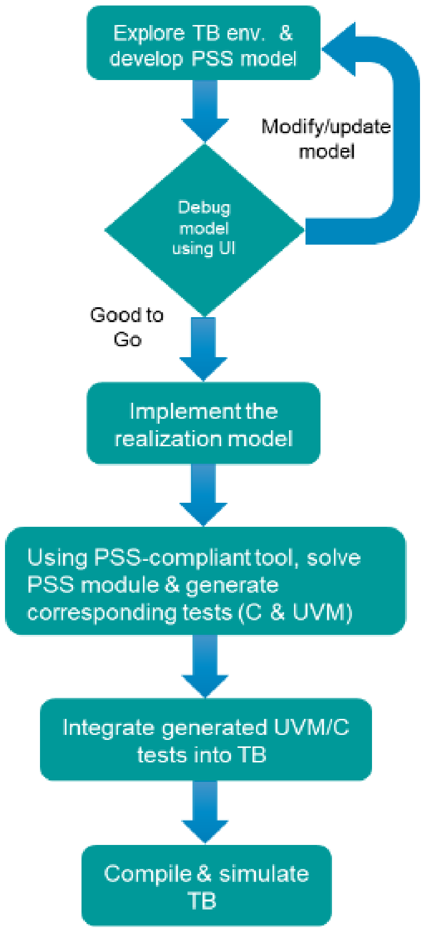

Explore TB env. & develop PSS model
Modify/update model
Debug model using UI
Good to Go
Implement the realization model
Using PSS-compliant tool, solve PSS module & generate corresponding tests (C & UVM)
Integrate generated UVM/C tests into TB
Compile & simulate TB

Figure 2: Proposed Methodology

CASE STUDIES

In this paper, two different use cases are implemented to demonstrate the effectiveness of the proposed methodology, utilizing the PSS for HW/SW co-verification by coordinating the UVM and C tests. The two use cases are SPI Testbench and DMA VIP use cases. In the following section, a detailed examination of each use case is provided.

Use Case 1: SPI Testbench

Numerous hardware modules are designed to interact with software through memory-mapped registers. In the final implementation, the system-level software, operating on a CPU, performs read and write operations on these registers via a bus interface within the hardware module. Utilizing UVM sequence-based stimulus, access to these registers is conducted through a bus agent. These accesses may be executed in a directed way that simulates software interactions or through constrained random stimulus.

In this use case, an updated version of the SPI testbench is used as one of the Verification Academy's cookbook examples [7]. Figure 3 illustrates the architecture of this testbench, which shows how to use a C routine to test the DUT.

We've built a PSS model to verify part of this testbench which can be used to illustrate and validate the proposed methodology.

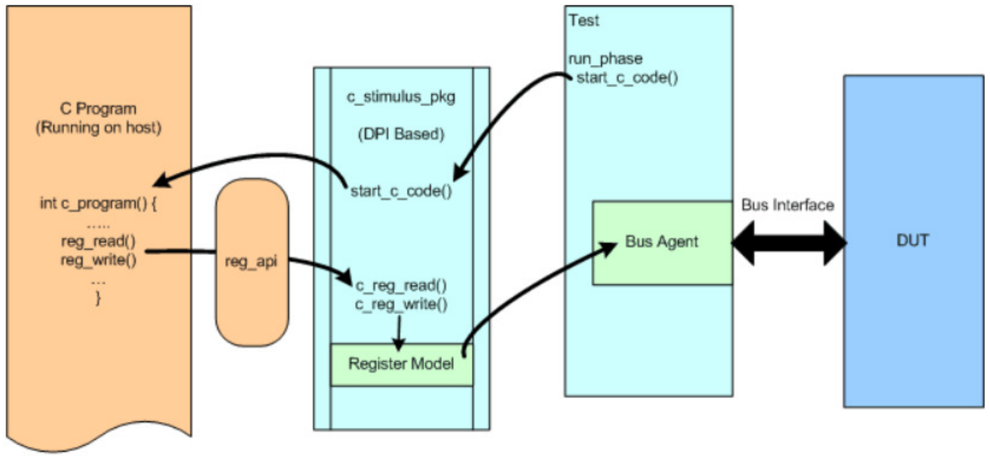


Figure 3: UVM C Registers Architecture

Example 1 shows the UVM 'run_phase' of "spi_c_poll_test" UVM test, which is used to explain the proposed methodology. This test runs different C-DPI calls to execute C-Tests during the execution of SV UVM test, for example "spi_c_poll_test_routine()" is used to do some Registers read and write with different arguments, as shown in Example 2.

```
task spi_c_poll_test::run_phase(uvm_phase phase);
  spi_tfer_seq spi_seq = spi_tfer_seq::type_id::create("spi_seq");
  uvm_reg_data_t reg_data;
  phase.raise_objection(this, "Test Started");
  `uvm_info("run_phase", "starting c code", UVM_LOW)
  set_c_stimulus_register_block(spi_rm);
  fork
    spi_c_poll_test_routine();
    // Respond to SPI transfers:
    begin
      forever begin
        spi_seq.BITS = spi_rm.ctrl_reg.char_len.get();
        spi_seq.rx_edge = spi_rm.ctrl_reg.rx_neg.get();
        spi_seq.start(m_env.m_spi_agent.m_sequencer);
      end
    end
  join_any
  `uvm_info("run_phase", "c code finished", UVM_LOW)
  phase.drop_objection(this, "Test Finished");
endtask: run_phase
```

Example 1: UVM run_phase of spi_c_poll_test UVM test

```
#include "spi_regs.h" // Defines for register offsets etc
#include "reg_api.h"  // DPI Register Hardware access layer API

int spi_c_poll_test_routine() {
  int no_chars = 1;
  int format = 0;
  int divisor = 2;
  int slave_select = 1;
  int control = 0;
  …
  register_thread(); // To register this thread with the c_stimulus_pkg
DPI context
  reg_write(DIVIDER, divisor);
  while(i < 10) {
    control = no_chars + (format << 9) + 0x3000;
    reg_write(CTRL, control);
    …
    reg_write(CTRL, control);
    …
    i++;
  }
  return 0;
}
```

Example 2: spi_c_poll_test_routine

The original intent of this UVM test is to perform some operations on the C-side and some other SV UVM sequences in parallel. Accordingly, a PSS model is implemented to model the required functionality. Example 3 shows part of a PSS model used to model the scenario of the C-Test.

```
// Component for actions contributing to C-Test
component spi_c_poll_comp {
    import C_funcs::*;
    action write_reg_a {
        rand address_t address;
        rand data_t data;
        …
        exec body C = """
            reg_write({{address}},{{data}});
        """;
    }
    action read_reg_a {
        rand address_t address;
       rand read_data_t data;
        …
        exec body C = """
```

```
             {{data}} = reg_read({{address}});
         """;
     }
     …
     action spi_c_poll_test_a {
         activity {
                 do write_reg_a with {address == CTRL && data == control;};
                 …
                 do read_reg_a with {address == CTRL && data == control;};
                 …
         }
     }
}
```

Example 3: Part of PSS code to model scenario for C-Test

The **spi_c_poll_comp** component is the top-level component for this test and defines the **write_reg_a** and **read_reg_a** atomic actions, each of which includes a target-template exec body block to specify it's C-code implementation. The "moustache" notation ("{{}}") passes in the **address** and **data** field references from the PSS model. The **spi_c_poll_test_a** action defines the activity that will traverse the **write_reg_a** and **read_reg_a** actions to implement the test. Thus, the desired test write a given value to the CRTL register and then read back from the same register.
For this paper's purposes, Siemens' PSS-compliant tool is used to generate and traverse C and UVM tests explained in this paper.
Example 4 shows the PSS model and the top activity used to traverse PSS actions responsible for executing the pre-generated C-Test and UVM-Sequence, which are both used to perform HW/SW co-verification proposed methodology.

```
package sv_funcs {
    function void spi_c_poll_test_wrapper_task();
    import target function spi_c_poll_test_wrapper_task;
    function void spi_tfer_seq_wrapper_task(bit [7] bits, bit rx_edge);
    import target function spi_tfer_seq_wrapper_task;
}
// Component for actions contributing to SV UVM-Test
component spi_sv_poll_comp {
    import sv_funcs::*;
    action spi_c_poll_test_act {
        exec body {
            spi_c_poll_test_wrapper_task();
        }
    }
    action spi_tfer_seq_act {
        rand bit [7] bits;
        rand bit rx_edge;
        exec body {
            spi_tfer_seq_wrapper_task(bits, rx_edge);
        }
    }
    action spi_sv_poll_test_a {
        activity {
           repeat (10) {
                parallel {
                    do spi_c_poll_test_act;
                    do spi_tfer_seq_act with {bits == 1; rx_edge == 0;};
                }
            }
        }
    }
}
```

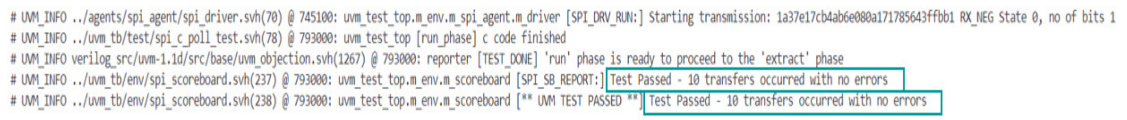
Example 4: Part of PSS code to model scenario top-level actions for HW/SW co-verification

To execute both C-Test as well as UVM-Sequence in the same PSS model, a wrapper SV task is created to wrap the call to "spi_c_poll_test_routine()" C-DPI function call, as shown in Example 5. In the same example, another SV task is used to implement SV code required to start the UVM sequence. SV tasks are traversed on-the-fly to execute top-level PSS action (spi_sv_poll_test_a) during the simulation of the SPI testbench.

```
// DPI Imports for c based routines:
import "DPI-C" context task spi_c_poll_test_routine();
…
task spi_c_poll_test_wrapper_task;
    spi_c_poll_test_routine();
endtask
task spi_tfer_seq_wrapper_task (byte unsigned bits, byte unsigned rx_edge);
    spi_tfer_seq spi_seq = spi_tfer_seq::type_id::create("spi_seq");
    spi_seq.BITS = bits;
    spi_seq.rx_edge = rx_edge;
    spi_seq.start(my_seqr);
endtask
…
```

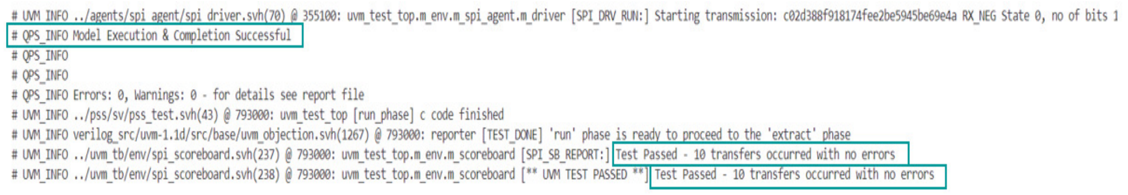
Example 5: Part of SV implementation of SV tasks

Both "spi_c_poll_test_act" and "spi_tfer_seq_act" actions are running for 10 transfers (i.e., repeat for 10 iteration) in parallel to model the original behavior in SPI testbench.
Simulation results after simulating the above PSS model are matching the original SPI UVM testbench, check Figure 4, for original SPI TB, and Figure 5 after deriving the same TB using PSS.



```
# UVM_INFO ../agents/spi_agent/spi_driver.svh(70) @ 745100: uvm_test_top.m_env.m_spi_agent.m_driver [SPI_DRV_RUN:] Starting transmission: 1a37e17cb4ab6e080a171785643ffbb1 RX_NEG State 0, no of bits 1
# UVM_INFO ../uvm_tb/test/spi_c_poll_test.svh(78) @ 793000: uvm_test_top [run_phase] c code finished
# UVM_INFO verilog_src/uvm-1.1d/src/base/uvm_objection.svh(1267) @ 793000: reporter [TEST_DONE] 'run' phase is ready to proceed to the 'extract' phase
# UVM_INFO ../uvm_tb/env/spi_scoreboard.svh(237) @ 793000: uvm_test_top.m_env.m_scoreboard [SPI_SB_REPORT:] Test Passed - 10 transfers occurred with no errors
# UVM_INFO ../uvm_tb/env/spi_scoreboard.svh(238) @ 793000: uvm_test_top.m_env.m_scoreboard [** UVM TEST PASSED **] Test Passed - 10 transfers occurred with no errors
```

Figure 4: Simulation transcript output of original SPI TB



```
# UVM_INFO ../agents/spi_agent/spi_driver.svh(70) @ 355100: uvm_test_top.m_env.m_spi_agent.m_driver [SPI_DRV_RUN:] Starting transmission: c02d388f918174fee2be5945be69e4a RX_NEG State 0, no of bits 1
# QPS_INFO Model Execution & Completion Successful
# QPS_INFO
# QPS_INFO
# QPS_INFO Errors: 0, Warnings: 0 - for details see report file
# UVM_INFO ../pss/sv/pss_test.svh(43) @ 793000: uvm_test_top [run_phase] c code finished
# UVM_INFO verilog_src/uvm-1.1d/src/base/uvm_objection.svh(1267) @ 793000: reporter [TEST_DONE] 'run' phase is ready to proceed to the 'extract' phase
# UVM_INFO ../uvm_tb/env/spi_scoreboard.svh(237) @ 793000: uvm_test_top.m_env.m_scoreboard [SPI_SB_REPORT:] Test Passed - 10 transfers occurred with no errors
# UVM_INFO ../uvm_tb/env/spi_scoreboard.svh(238) @ 793000: uvm_test_top.m_env.m_scoreboard [** UVM TEST PASSED **] Test Passed - 10 transfers occurred with no errors
```

Figure 5: Simulation transcript output of modified SPI TB with PSS

Use Case 2: DMA VIP

Direct Memory Access (DMA) is a hardware mechanism that enables peripheral components to transfer their I/O data directly to and from the main memory without involving the system processor. In the use case presented in this paper, the system includes a DMA-capable multi-channel memory that supports direct read and write operations, facilitating concurrent DMA transfers across its four channels. Additionally, this memory can handle data transfers to and from the peripheral, which also supports direct read and write operations. For data transfers between the memory and the peripheral, both devices must be programmed to handshake with each other and operate concurrently to ensure smooth data transfer.

From a testbench perspective, the Siemens AXI VIP component is instantiated as the "axi_manager" connected to the memory component. In this case, the axi_manager acts as the "internal VIP" mimicking the processor to program the memory. Meanwhile, the "axi_manager" connected to the peripheral component serves as the "external VIP," modeling the "real world" to communicate with the DUT via the peripheral. This is analogous to a UART or other I/O device found in a typical system. Figure 6**Error! Reference source not found.** illustrates this architecture, highlighting the roles of the internal and external VIPs within the use case.

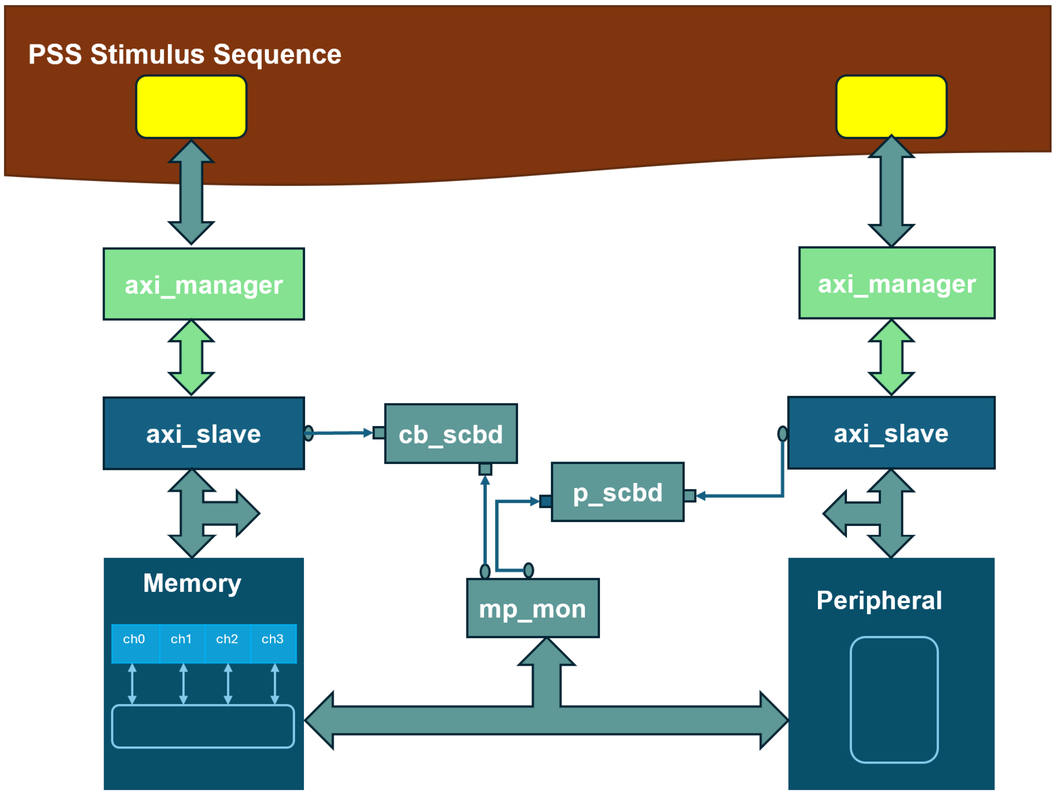


Figure 6: DMA VIP Architecture

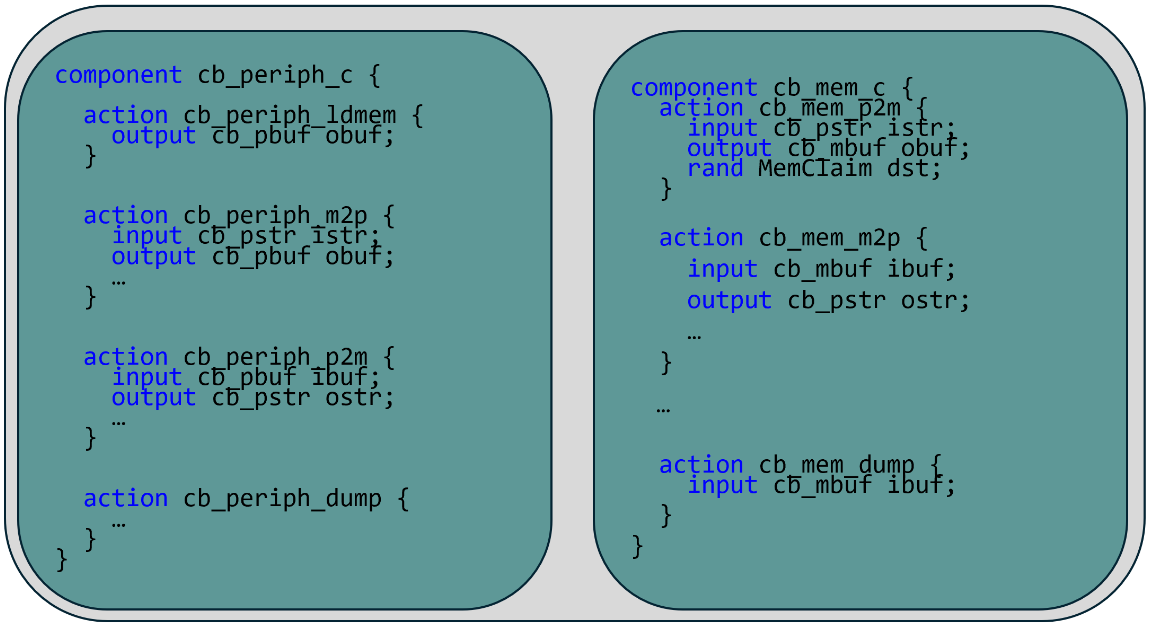
The PSS implementation for the use case focuses on testing various scenarios that may occur in the system, aligning with the concept of the PSS language, which targets scenario-level verification. To achieve this, a set of actions is defined, with each action covering operations performed by components such as the peripheral and memory. Table 1 summarizes part of the actions modeled in PSS to verify the system.

Table 1: PSS Model Actions

Action Name	**System component**	**Action Description**
cb_periph_ldmem	Peripheral	This action is responsible for reading data from the peripheral and loading the memory with this data.

cb_periph_m2p	Peripheral	This action is responsible for transferring data from memory block to peripheral block.
cb_periph_p2m	Peripheral	This action is responsible for transferring data from peripheral block to memory block.
cb_mem_fill	Memory	This action is responsible for filling the memory block.
cb_mem_m2m	Memory	This action is responsible for transferring data between two memory locations
cb_mem_m2p	Memory	This action is responsible for transferring data from memory block to peripheral block.

Figure 7: Snippet of PSS code showing modelling for Memory and Peripheral Figure 7 shows part of the PSS modeling code, illustrating the implementation of these actions in the verification environment.



```
component cb_periph_c {
  action cb_periph_ldmem {
    output cb_pbuf obuf;
  }

  action cb_periph_m2p {
    input cb_pstr istr;
    output cb_pbuf obuf;
    …
  }

  action cb_periph_p2m {
    input cb_pbuf ibuf;
    output cb_pstr ostr;
    …
  }

  action cb_periph_dump {
    …
  }
}
```

```
component cb_mem_c {
  action cb_mem_p2m {
    input cb_pstr istr;
    output cb_mbuf obuf;
    rand MemClaim dst;
  }

  action cb_mem_m2p {
    input cb_mbuf ibuf;
    output cb_pstr ostr;
    …
  }

  …

  action cb_mem_dump {
    input cb_mbuf ibuf;
  }
}
```

Figure 7: Snippet of PSS code showing modelling for Memory and Peripheral component.

To further illustrate the proposed methodology, the same PSS model defines two top-level actions: one responsible for generating the UVM virtual sequence, and the other for generating the C test. The generated virtual sequence calls UVM tasks, which in turn invoke the C test generated from the PSS model using the DPI-C technique.

Figure 8 shows the top-level PSS action responsible for generating the UVM test, covering the intended behavior that needs to be verified. According to PSS modeling semantics, action inferencing will take place, inferring the appropriate actions to complete the model and thereby enhancing randomization.

```
action cb_vip_nobind_ex_a {
  cb_mem_c::cb_mem_fill mfill;
  cb_mem_c::cb_mem_dump mdump;
  cb_mem_c::cb_mem_m2m m2m;
  cb_mem_c::cb_mem_m2p m2p;
  cb_mem_c::cb_mem_p2m p2m;
  cb_periph_c::cb_periph_m2p pm2p;
  constraint mdump.ibuf.step == 2;
  activity {
    repeat (4) {
      select {
        p2m with {obuf == mdump.ibuf;};
        m2m with {obuf == mdump.ibuf;};
        }
        mdump;
    }
  }
}
```

Figure 8: Part of PSS code to model scenario top-level actions for HW/SW co-verification.

Figure 9 shows a snippet of the developed PSS top-level scenario used to generate the C test. Figure 10 shows the Siemens PSS-compliant generated C test. This C test will be called by one of the UVM tasks that will be called during simulation.

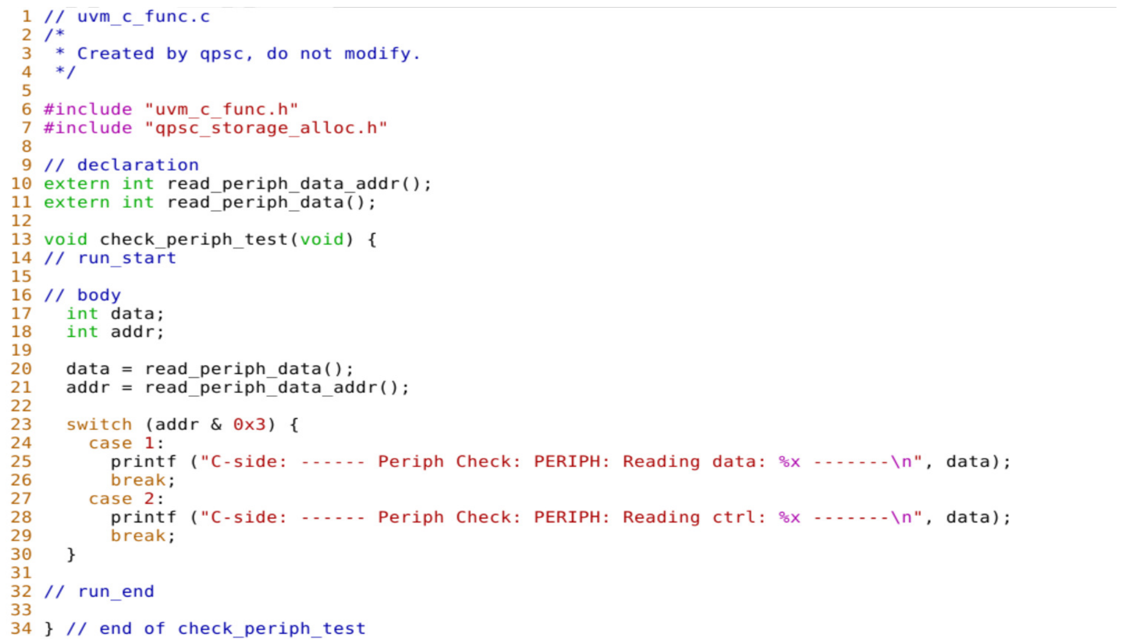
```
action cb_periph_dump {
  exec declaration C = """
    extern int read_periph_data_addr();
    extern int read_periph_data();
  """;

  exec body C = """
    int data;
    int addr;
    data = read_periph_data();
    addr = read_periph_data_addr();

    switch (addr & 0x3) {
      case 1:
        printf ("C-side: ------ Periph Check: PERIPH: Reading data: %x -------
\n", data);
        break;
      case 2:
        printf ("C-side: ------ Periph Check: PERIPH: Reading ctrl: %x -------
\n", data);
        break;
    }
  """;
}

action top_periph_dump {
  activity {
    do cb_periph_c::cb_periph_dump;
  }
}
```

Figure 9: PSS code to model scenario top-level actions for C test



```c
// uvm_c_func.c
/*
 * Created by qpsc, do not modify.
 */

#include "uvm_c_func.h"
#include "qpsc_storage_alloc.h"

// declaration
extern int read_periph_data_addr();
extern int read_periph_data();

void check_periph_test(void) {
// run_start

// body
  int data;
  int addr;

  data = read_periph_data();
  addr = read_periph_data_addr();

  switch (addr & 0x3) {
    case 1:
      printf ("C-side: ------ Periph Check: PERIPH: Reading data: %x -------\n", data);
      break;
    case 2:
      printf ("C-side: ------ Periph Check: PERIPH: Reading ctrl: %x -------\n", data);
      break;
  }

// run_end

} // end of check_periph_test
```

Figure 10: Siemens PSS-compliant generated C test.

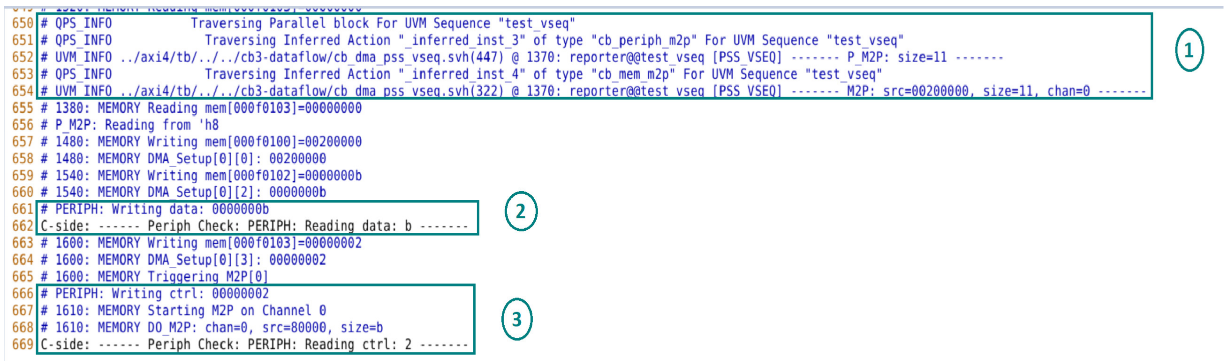
Figure 11 shows part of the user defined UVM task where the body of this task calls the generated C test. The UVM task is defined in the UVM sequence generated as well from the Siemens PSS-compliant tool.

```systemverilog
import "DPI-C" context function void check_periph_test();

virtual task automatic do_cb_periph_m2p(input int signed id, input int unsigned
size);
      logic[DATA_WIDTH-1:0] m2p_active='b010;
      // Set up the Recording Stream
      int rec_tr, read_tr, write_tr;
      `uvm_info("PSS_VSEQ",$sformatf("------- P_M2P: size=%0d -------
",size),UVM_MEDIUM);
      rec_tr = $begin_transaction(Bus_Stream[id], "PM2P");
      $add_attribute(rec_tr,size, "size");
      …
      write_tr = $begin_transaction(Bus_Stream[id], "Wr32",,rec_tr);
      axi_seq[id].write32('h4,size);
      $add_attribute(write_tr,'h4,"addr");
      $add_attribute(write_tr,size,"data");
      $end_transaction(write_tr);
      // Call C test generated from Siemens PSS-compliant tool
      check_periph_test();
     …
   endtask
```

Figure 11: Part of the UVM task calling generated C test

Figure 12 shows the simulation transcript that proves the proposed methodology, with Block 1 showing the logging information generated by the Siemens PSS-compliant tool during simulation. Blocks 2 and 3 display the debugging messages printed from the UVM task and the generated C tests, respectively.



```
# QPS_INFO           Traversing Parallel block For UVM Sequence "test_vseq"
# QPS_INFO             Traversing Inferred Action "_inferred_inst_3" of type "cb_periph_m2p" For UVM Sequence "test_vseq"
# UVM_INFO ../axi4/tb/../../cb3-dataflow/cb_dma_pss_vseq.svh(447) @ 1370: reporter@@test_vseq [PSS_VSEQ] ------- P_M2P: size=11 -------
# QPS_INFO             Traversing Inferred Action "_inferred_inst_4" of type "cb_mem_m2p" For UVM Sequence "test_vseq"
# UVM_INFO ../axi4/tb/../../cb3-dataflow/cb_dma_pss_vseq.svh(322) @ 1370: reporter@@test_vseq [PSS_VSEQ] ------- M2P: src=00200000, size=11, chan=0 -------
# 1380: MEMORY Reading mem[000f0103]=00000000
# P_M2P: Reading from 'h8
# 1480: MEMORY Writing mem[000f0100]=00200000
# 1480: MEMORY DMA_Setup[0][0]: 00200000
# 1540: MEMORY Writing mem[000f0102]=0000000b
# 1540: MEMORY DMA_Setup[0][2]: 0000000b
# PERIPH: Writing data: 0000000b
C-side: ------ Periph Check: PERIPH: Reading data: b -------
# 1600: MEMORY Writing mem[000f0103]=00000002
# 1600: MEMORY DMA_Setup[0][3]: 00000002
# 1600: MEMORY Triggering M2P[0]
# PERIPH: Writing ctrl: 00000002
# 1610: MEMORY Starting M2P on Channel 0
# 1610: MEMORY DO_M2P: chan=0, src=80000, size=b
C-side: ------ Periph Check: PERIPH: Reading ctrl: 2 -------
```

Figure 12: Simulation transcript output of the DMA VIP Use Case

CONCLUSION

As mentioned in the above sections, HW/SW co-verification is a challenging process, traditional methodologies may not scale to the recent increase of SoC complexities. PSS can be used as a unified representation for both hardware and software co-verification.

In this paper, a complete framework has been introduced to ease HW/SW co-verification. Two use models have been introduced to go through the proposed methodology, which will help verification engineers and teams to improve SoC verification, by writing one PSS model which can execute C-Tests as well as UVM sequences to cover the different aspects of HW/SW co-verification.

Finally, these methodologies facilitate a more thorough and efficient verification process by enabling higher levels of automation and abstraction. The integration of PSS, UVM, and C++ in HW/SW co-verification not only streamlines the verification process but also enhances the quality and reliability of SoC designs.

REFERENCES

[1] Accellera, "Portable Test and Stimulus Standard Version 2.1", https://www.accellera.org/downloads/standards/portable-stimulus

[2] R. Edelman, "UVM and C – Perfect Together", DVConUS 2018

[3] https://www.eetimes.com/transaction-based-methodology-supports-hw-sw-co-verification/

[4] L. Semeria and A. Ghosh, "Methodology for hardware/software co-verification in C/C++," Proceedings 2000. Design Automation Conference. (IEEE Cat. No.00CH37106), Yokohama, Japan, 2000, pp. 405-408, doi: 10.1109/ASPDAC.2000.835134.

[5] A. S. Mahdi, C. Archonta, G. Tzimas and A. El-Kady, "A SoC-ZYNQ-Based SW-HW Co-Simulation and Verification Method," 2019 Panhellenic Conference on Electronics & Telecommunications (PACET), Volos, Greece, 2019, pp. 1-6, doi: 10.1109/PACET48583.2019.8956264.

[6] https://dvcon-proceedings.org/wp-content/uploads/hardware-software-co-verification-using-specman-and-systemc-with-tlm-ports.pdf

[7] Verification Academy, https://verificationacademy.com/cookbook/uvm-universal-verification-methodology/c-based-stimulus