There and Back Again: Simulation-to-Silicon Scenario Reuse with PSS

Tom Fitzpatrick
Big Fish EDA Consulting
tom@bigfisheda.com

***Abstract*- The Portable Test and Stimulus Standard (PSS)[1] enables verification engineers to separate the description of their verification intent from the platform-specific implementation, allowing identical scenarios to execute across multiple target environments. This paper demonstrates how to organize PSS models to maximize reuse across UVM simulation and various C-based implementations, including Linux executables and embedded bare-metal implementations.**

We present a comprehensive methodology for deploying PSS-generated scenarios throughout the verification flow—from simulation and emulation to lab testing and actual silicon validation. The paper illustrates how a single PSS model can generate both UVM sequences and multiple flavors of C code tailored for different target configurations. The generated code can execute in UVM environments using either C-embedded sequences or C-capable UVM components, while the embedded C implementations can be deployed on RTL processor models, emulated hardware, or actual silicon.

Key contributions include practical guidance on PSS model organization for maximum portability, demonstration of cross-platform scenario consistency, and implementation strategies for both threaded Linux environments and bare metal embedded systems. We showcase the complete flow using publicly available tools, though the methodology applies to any PSS-compliant toolchain. The approach enables verification teams to develop scenarios once and deploy them seamlessly across the entire hardware-software verification continuum.

I. Introduction

As SoC designs grow in complexity, verification teams face mounting challenges in creating portable, reusable test content across hardware and software domains. A critical requirement in modern hardware-software co-development is executing identical scenarios seamlessly in both simulation environments (using UVM) and lab environments (using C-based tests on actual hardware). While SystemVerilog and UVM provide modular, reusable verification environments, they fall short in enabling cross-domain scenario portability. UVM cannot inherently implement the same complex scenarios for both UVM-based simulation and C-based lab environments. The limitations of UVM sequences in specifying system-level scenarios, coupled with the difficulty of porting UVM sequences to C-based tests, create a fundamental verification gap.

PSS addresses these challenges through its two-layer architecture that separates abstract modeling from platform-specific realization. At the *abstract model* layer, PSS enables verification engineers to define test scenarios, data flows, and behavioral constraints in a platform-agnostic manner. The realization layer handles platform-specific translation and execution details, mapping abstract models to concrete implementations for UVM-based simulation or C-based software for execution on multiple target platforms, including simulation, emulation and post-silicon. This separation eliminates the need to maintain redundant versions of the same test scenario across platforms while enabling true test portability. Verification teams can develop comprehensive abstract test models once, then leverage automated tools to generate appropriate implementations for each target environment.

II. Modelling in PSS

Consider a simple hardware system (described in more detail in [1]) consisting of a DMA-capable memory and an accompanying peripheral device. As shown in Figure 1, each device supports a data/programming interface to load or dump data to/from each device or program the appropriate registers, as well as a handshake-driven data channel to transfer data directly between them. The shaded blocks in the figure indicate UVM components in a typical UVM verification environment, while the white blocks indicate the RTL design components.

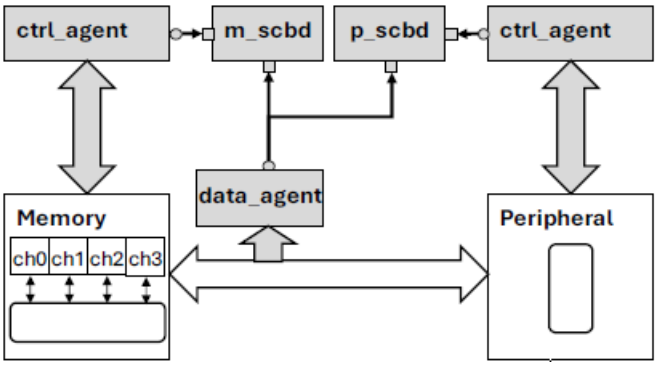


Figure 1. Memory-Peripheral Design and UVM Environment

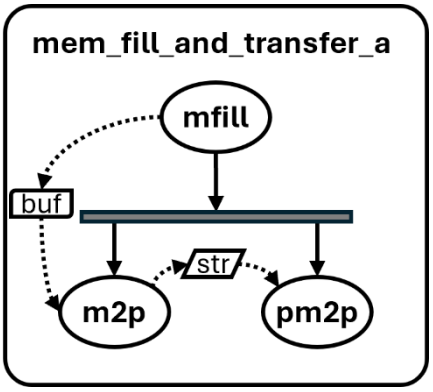
There are many scenarios we could create for this model, but for the purposes of this paper, we will focus on a simple test scenario where we will fill a buffer in the memory with random data and then transfer the data to the peripheral.

PSS Abstract Scenario

PSS provides several language constructs to model scenarios at different levels of abstraction. As in SystemVerilog and UVM, PSS models can be written as fully directed tests, directed-random tests, or fully random. Let's begin with a directed-random test that will load the memory and then transfer that data to the peripheral. Consider the following code snippet:

```
action mem_fill_and_transfer_a {
  mem_c::mem_fill mfill;
  mem_c::mem2periph m2p;
  periph_c::mem2periph p_m2p;

    activity {
        mfill;
        parallel {
            m2p;
            p_m2p;
        }
        bind mfill.obuf m2p.ibuf;
        bind p_m2p.istr m2p.ostr;
    }
}
```



Code 1: PSS Directed Test

In this example, the *activity* defines the specific steps of the scenario, starting with **`mem_fill`**, which loads data of a desired size into the memory starting at a particular address, followed by both the memory and peripheral performing complementary **`mem2periph`** operations in parallel to accomplish the handshake between them and transfer data of the given size from a source address in the memory to the FIFO in the peripheral. The two **`bind`** statements ensure that the data buffer, which models the address and size of the data created by the fill action is sent to the peripheral.

However, the real productivity of PSS comes from its ability to *infer* additional actions to support a *partial specification* of verification intent. Let's suppose that what we really want to verify is the ability for the memory to send data to the peripheral. If that is truly our intent, we could simplify the activity as follows

```
action mem_fill_and_transfer_a {
  mem_c::mem2periph m2p;

    activity {
        m2p;
    }
}
```

Code 2: PSS Critical Intent Test

The **mem2periph** action is defined as

```
action mem2periph {
    input cb_mbuf ibuf;
    output cb_pstr ostr;
    ...
}
```

Code 3: PSS Action Definition

The PSS rules state that the input buffer, **ibuf**, must be provided by another action, so the tool may *infer* any available action that produces an output buffer of the correct type. In our example, the buffer could be provided, as shown above, by the **mem_fill** action, but it could also be provided by a **mem2mem** action (not shown) that moves a buffer from one memory location to another, or by a **periph2mem** action (also not shown) that takes data from the peripheral and puts it into the memory.

The other operative rule in PSS is that any stream object produced by an action (in this case, the **ostr** object of type **cb_pstr**) must be bound to another action that inputs a stream object of the same type. In our system, the only action that inputs a compatible stream object is the **periph_c::mem2periph** action, so that must be inferred in parallel with **m2p**.

So, for this scenario, we will definitely do the two **mem2periph** actions in parallel but a PSS tool may infer one of several actions to provide the **m2p** input.

Guiding PSS Inferencing

We can ensure that the solver will infer a **mem_fill** action by adding a useful field to the **cb_mbuf** buffer type. Consider:

```
buffer cb_mbuf {
    rand bit [32] addr;
    rand bit [32] size; // number of bytes
    rand bit[8] step;

    constraint c1 {step >= 0;}
}
```

Code 4: PSS Buffer Definition

We use the **step** field to help guide the solver to create the desired scenario by tracking how many data transfers will be done in the scenario. Since the **mem_fill** action will start any scenario, we will add a constraint to the action definition:

```
action mem_fill {
    output cb_mbuf obuf;
    constraint c5 {obuf.step == 0;}
}
```

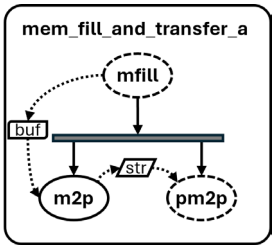
Code 5: PSS Action with Constraint

If we want to ensure that the **mem_fill_and_transfer_a** action above will always infer the **mem_fill** action, we can add a constraint:

```
action mem_fill_and_transfer_a {
  mem_c::mem2periph m2p;

  constraint m2p.ibuf.step == 0;

    activity {
        m2p;
    }
}
```



Code 6: Critical Test Intent Modified with Constraint

This requires that the buffer input to **m2p** has a **step** value of 0, and the only action that can produce a buffer with **step==0** is **mem_fill**, so the solver must infer **mem_fill** to supply **m2p.ibuf**. The inferred actions are shown in the diagram accompanying Code 6 with dashed lines. To increase reuse and flexibility, the constraint can alternatively be defined in an extension of the **mem_fill_and_transfer_a** action as

```
extend action mem_fill_and_transfer_a {
    constraint m2p.ibuf.step == 0;
}
```

Code 7: Test Modification via Extension

Then the extension could be put in a file and included in the command line when invoking the PSS solver to guide the inferred scenario. If the extension is not included, then the solver is free to randomly infer other valid scenarios. Note that the **step** field is not part of the resulting scenario. It is only used by the solver to determine the legal scenario to be generated.

It is true that, if we only wanted to define the specific directed scenario shown in Code 1, we could have coded that scenario directly. The inclusion and constraining of the **step** variable is shown to indicate how a model can be created that would, by default, produce a wide variety of scenarios, but can be easily constrained to create a specific directed scenario as desired. The number of scenarios that could be created for the partial specification in Code 6 obviously depends on the value of the **step** variable. We show this technique to illustrate the power of including generation-specific fields in the model that do not appear explicitly in the realization but allow us to tune the generated scenarios as needed.

III. PSS Realization Layer

To create the implementation of a scenario, each leaf-level action in a PSS model includes one or more *exec* blocks that define the target-specific behavior of the action. The most common type is the **exec body** block, which defines the specific implementation for the action. For example, a memory fill action that performs incrementing writes can be implemented in SystemVerilog or C as:

```
virtual task automatic do_mem_fill(bit[31:0] dest, size);
  bit[31:0] data;
  for (int i = 0; i < size*4; i = i + 4) begin
    bit stat = std::randomize(data);
    ctrl_if_api.write32(dest+i, data);
  end
endtask
```

Code 8: SystemVerilog Implementation of Memory Fill Action

```
void do_mem_fill(int dest, int size) {
  for (uint32_t i = 0; i < (uint32_t)size; i++) {
    uint32_t write_addr = (uint32_t)dest + (i * 4);
    uint32_t write_data = rand();  // Generate pseudo-random data
    *(volatile uint32_t *)(uintptr_t)write_addr = write_data;
#ifdef ARM
    SAFE_YIELD_EVERY(8); // See footnote 1
#endif
  }
}
```

Code 9: C Implementation of Memory Fill Action

The PSS model can map the desired target implementation to the action in several ways. The easiest way is to import the **do_mem_fill** method in a PSS package and extend the **action** to add the **exec body** block to call the imported method as shown here:

[1] As we will see later, the **ifdef** in the C implementation allows the method to be compatible with an explicit yield/resume mechanism needed for a bare-metal threading implementation to run as ARM embedded software. Without the **ifdef**, the method is, by default, compatible with pthreads. It is up to the user whether to use **#ifdef** in a single implementation or whether to provide separate ARM- and pthreads-compatible versions and include the appropriate files to the execution tool flow.

```
package cb_pss_mem_vip_pkg {
    import cb_params_pkg::*;
    function void do_mem_fill(bit [32] dest, bit [32] size);
    import target SV function do_mem_fill;

      extend action cb_mem_fill {
        exec body {
          do_mem_fill(obuf.addr, obuf.size);
        }
      }
  }
```

Code 10: PSS Imported Method and Action Extension with exec Block for Memory Fill Action

This allows the method call for the action to be inserted in the generated code according to the prototype provided, with the data types translated appropriately to the target language, as defined in the **`exec body`** block.

III. PSS Test Generation

A PSS tool generates test implementations that coordinate action execution according to the schedule and constraints defined within the PSS activity model. Actions within the generated test can execute either sequentially or in parallel, where multiple actions run concurrently to achieve the desired system behavior. The implementation of parallel action execution is inherently dependent on the target language.

So, a PSS tool must do two things to generate a target implementation of a test scenario. It must convert the specified method call in the **`exec body`** block to the target language, and it must generate a framework in the target language to invoke the action method according to the schedule defined by the **`activity`**, including any constraints and inferred actions.

SystemVerilog Target Implementation

In UVM/SystemVerilog environments, PSS tools leverage native **`fork/join`** constructs for parallel execution. A straightforward tool flow is shown in Figure 2, where a PSS model is compiled directly into a UVM virtual sequence that is run from a test. In this case, the RTL design has been enhanced from Figure 1 to include AXI4 interfaces for the memory and peripheral devices. The UVM environment is similarly enhanced to use standard AXI4 VIP components to connect to these interfaces. The action realizations use the API provided by the VIP, just as originally the realizations relied on the underlying API of the **`cb_agent`** components in the UVM environment, which are now made passive to monitor the data paths.

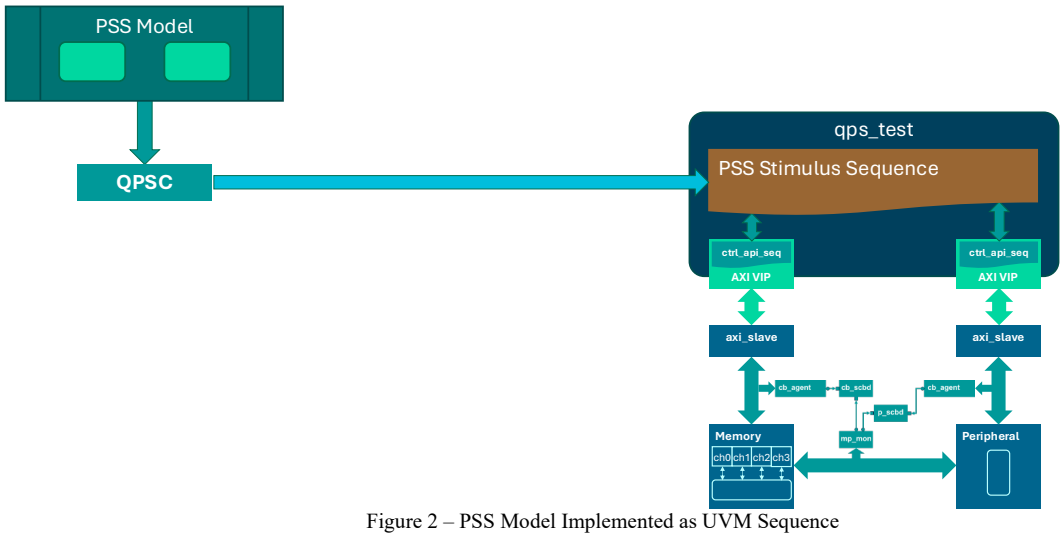


Figure 2 – PSS Model Implemented as UVM Sequence

In Figure 2, the box labelled "QPSC" represents the PSS tool that compiles and solves the PSS model to generate the target-specific implementation. The author's experiments were done using Questa® One, which includes a Portable Stimulus Compiler, but the methodology here discussed should apply to any PSS-compliant tool.

A simple UVM sequence to do a memory fill operation then transfer the data to the peripheral would look something like:

```
task body();
  do_mem_fill('h17d400, 'h10);
  fork
    do_mem_m2p(0, 'h17d400, 'h10); // use DMA channel 0
    do_periph_m2p('h10);
  join
endtask
```

Code 11: SystemVerilog Sequence to Execute Memory-to-Peripheral Transfer

The **do_mem_fill** task would be implemented in SystemVerilog as

```
virtual task automatic do_mem_fill(bit[31:0] dest, size);
      bit[31:0] data;
      for (int i = 0; i < size*4; i = i + 4) begin
         // Set up Write Transaction
         bit stat = std::randomize(data);
         ctrl_if_api.write32(dest+i, data);
      end
endtask
```

Code 12: SystemVerilog Task Implementation

where **ctrl_if_api** is a pointer to the sequence running on the underlying VIP component to provide the **write32()** method. With proper planning, this pointer can be set to whatever VIP component may be used, whether a standard protocol such as AXI or as a basic UVM agent (see [2]). Similarly the **do_mem_m2p** and **do_periph_m2p** tasks will provide the implementations of the **mem_c::mem2periph** and **periph_c::mem2periph** actions from Code 1.

Additional Exec Block Types

In a typical SystemVerilog-based target environment, there is no need to include anything in the PSS model other than **exec body** blocks to specify the realization. However, it may be necessary to include some target-specific information in the PSS model to help the code generator create valid code. For example, in order to call **do_mem_fill** from the generated sequence's **body** method as shown above, we must have a declaration of the task to be able to compile the generated sequence. It may be the case that the PSS tool will recognize the method and automatically insert the declaration.

```
class pss_gen_seq extends my_uvm_base_seq;
  `uvm_object_utils(pss_gen_seq)

  virtual task automatic do_mem_fill(input bit[31:0] dest, size);
  endtask
  ...
task body();
  do_mem_fill('h17d400, 'h10);
  fork
    do_mem_m2p(0, 'h17d400, 'h10); // use DMA channel 0
    do_periph_m2p('h10);
  join
endtask
endclass
```

Code 13: Sample Generated PSS Sequence

Then the task implementation can be user-supplied in a sequence derived from the generated sequence:

```
class my_pss_seq extends pss_gen_seq;
  `uvm_object_utils(my_pss_seq)

  virtual task automatic do_mem_fill(input bit[31:0] dest, size);
    for (int i = 0; i < size*4; i = i + 4) begin
      bit stat = std::randomize(data);
      ctrl_if_api.write32(dest+i, data);
    end
  endtask
  ...
endclass
```

Code 14: User-Defined Sequence to Provide Task Implementation[2]

If the PSS tool does not automatically insert the declaration, then it must be included in the PSS model, usually via an extension.

```
extend component cb_vip_ex_c {
    extend action mem_fill_and_transfer_a {
        exec declaration SV = """
            virtual task do_cb_fill(bit[31:0] dest, size);
            endtask
        """;
    }
```

Code 15: Exec Declaration Block in Action Extension

Note that the syntax for the **`exec declaration`** block uses what PSS refers to as a *target template exec block* and indicates the target language, in this case, SystemVerilog. It is also worth mentioning that, since PSS currently (the Working Group is working on addressing this in an upcoming release) allows **`exec declaration`** blocks on actions and not components, the preferred way to provide an **`exec declaration`** block to multiple generated sequences is to derive the desired root test actions from a common abstract base action and put the **`exec declaration`** in the base action.

```
component cb_vip_ex_c {
  abstract action test_base_a {}

  action mem_fill_and_transfer_a : test_base_a {
    ...// see declaration in Code 6 above
  }
}

extend component cb_vip_ex_c {
    extend action test_base_a {
        exec declaration SV = """
            virtual task do_cb_fill(bit[31:0] dest, size);
            endtask
        """;
    }
```

Code 16: Using Abstract Base Action for Exec Declaration Block

C Target Implementation

C-based implementations require a different set of exec declaration blocks

[2] Note that Code 14 is nearly identical to Code 8 but adds the additional context of showing the task definition within the **`my_pss_seq`** class.

```
  extend component cb_vip_ex_c {
      extend action test_base_a {
          exec declaration C = """
extern void do_cb_fill(int dest, int size);
...
          """;
      }
```

Code 17: C-Language Target Template Exec Declaration Block

but C test generation also requires a threading mechanism to support parallel execution. PSS tools can employ two approaches: generating platform-specific threading code directly, or utilizing a generic threading API that provides abstraction from platform-specific details. For example, the C implementation could be compiled using a Linux pthreads implementation (Figure 3) or an embedded implementation for a specific target platform (Figure 4). In either case, the generated code and associated infrastructure will be compiled into an executable form for the target platform. Note that Questa One includes QEMU-compatible UVM VIP that allows the simulation to be driven by the compiled executable on the customized OS-specific QEMU kernel. A similar approach would apply to any similar VIP library.

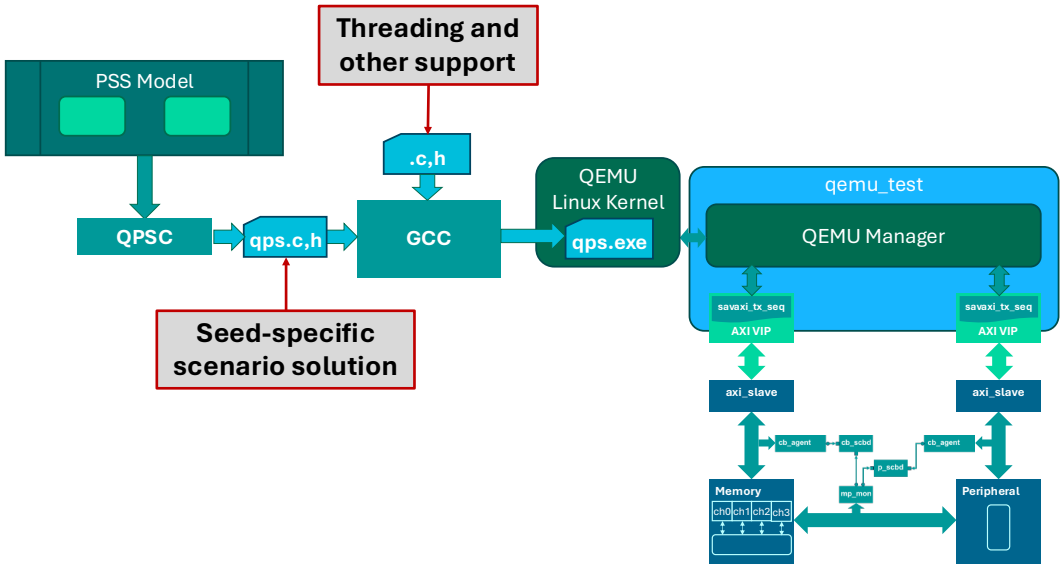


Figure 3 – PSS Model Implemented in Linux with Pthreads

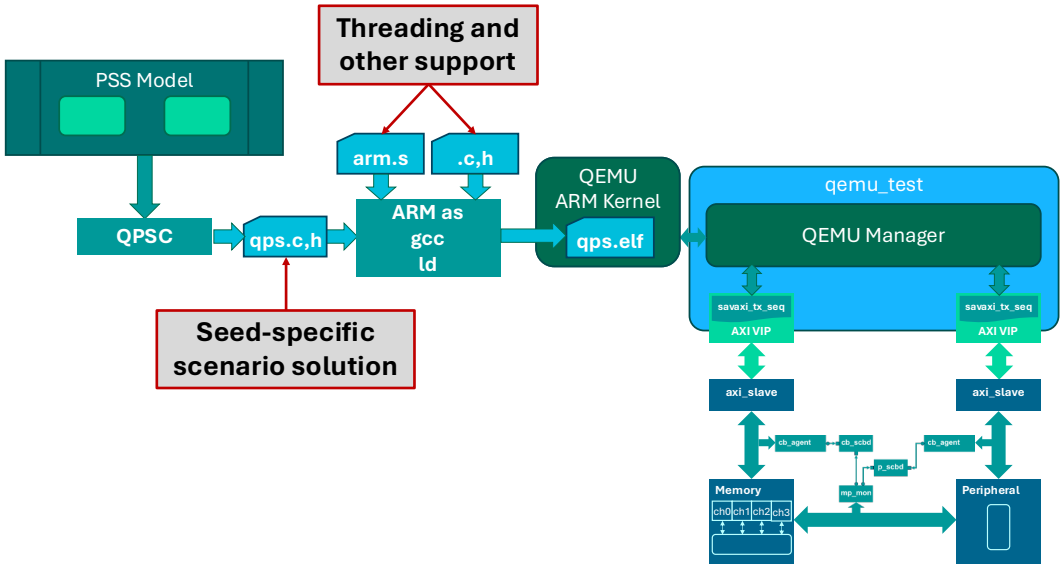


Figure 4 – PSS Model Implemented as Embedded ARM Code

Note that, in the ARM-based embedded SW setup, the .elf file could also have been loaded into a program memory for an ARM CPU model to execute or even in a post-silicon environment for an actual processor to run. It is presumed that any PSS-compliant tool chain can support similar execution environments for generated C code.

The API-based approach to threading offers considerable advantages in terms of greater portability and maintainability. In Code 18, the implementation of the **`pss_thread*`** methods is OS-specific. In the case of Linux generation, the tool may or may not default to using the standard Pthread library. If it does, then there is no additional requirement on the PSS code.

```
void mem_fill_and_transfer_a(void) {
  do_mem_fill(0, 0x17d400, 0x10);
  {
    pss_thread_t branch_1 = pss_thread_create(&thread_func_1);
    pss_thread_t branch_2 = pss_thread_create(&thread_func_2);
    pss_thread_join(branch_1);
    pss_thread_join(branch_2);
  }
}
void thread_func_1(void) {
  do_cb_m2p(0, 0x17d400, 0x10);
}
void thread_func_2(void) {
  do_cb_periph_m2p(0x10);
}
```

Code 18: C Test Implementation

However, it may require the user to ensure that the Pthread library is used by the generated code by specifying an **`exec header`** block[3] in the PSS code (again, as an extension):

[3] In PSS, an **`exec header`** block is used to specify declarations, such as **`#include`** directives, presupposed by subsequent code blocks, as opposed to **`exec declaration`** blocks, which specify declarative statements used to define entities that are used by subsequent code blocks, such as the definition of global variables or functions.

```
  extend component cb_vip_ex_c {
      extend action test_base_a {
          exec header C = """
#include <pthread.h>
...
          """;
      }
```

Code 19: Exec Header Target Template Block

In the case of ARM-based embedded code generation, we can use the same threading API, but now we have to account for a few more things. In addition to the **thread_create** and **thread_join** calls, we must also ensure that the threading mechanism is initialized (**pss_scheduler_init()**) and started (**pss_scheduler_start()**) before the generated sequence executes, and we must be able to clean things up at the end (**semihost_exit(0)**). This is accomplished with **exec run_start** and **exec run_end** blocks, which would be declared as

```
  extend component cb_vip_ex_c {
      extend action test_base_a {
          exec header C = """
#include "arm_pss_threads.h"
...
          """;
exec run_start C = """
pss_scheduler_init();
pss_scheduler_start();
""";

exec run_end C = """
semihost_exit(0);
""";

            exec declaration C = """
  static inline void semihost_exit(int code){
               register int op asm("r0") = 0x18;
               register int arg asm("r1") = code;
               asm volatile("hlt 0xf000" : : "r"(op),"r"(arg) : "memory");
  }
""";
      }
   }
```

Code 20: Adding Exec Run_start and Run_end Blocks

IV. Error Checking

The UVM environment shown in Figure 1 uses the standard architecture of relying on agents on each interface to drive and monitor the behaviors on those interfaces and also write appropriate transactions to a set of scoreboards to ensure that the data read on any particular interface matches what was expected. Thus, when data is loaded into the memory at a given address and transferred to the peripheral, the data read back from the peripheral can be checked against the data originally loaded into the memory to ensure correctness. In this sense, the scoreboards can be considered "reference models" used for error checking. This approach makes perfect sense to UVM users. Unfortunately, reference models implemented in the UVM scoreboards are also not easily portable to C code. Instead, PSS provides a mechanism to embed the reference model in the generated code to simplify error checking regardless of the target implementation.

Consider the scenario shown in Code 13 and Code 18. The specific values used in the scenario represent a valid solution to the original PSS model. In this case, both the UVM in Code 13 and C in Code 18 rely on the do_mem_fill operations to also randomize the data that gets written to the memory. The discerning reader will recognize that there is thus no specific error checking in the generated C code, unless we can somehow include error checking in the do_cb_periph_m2p function. However, the only way to do that is to include the expected data somehow in the function.

Fortunately, PSS lets us do this. Recall the scenario shown in the diagram accompanying Code 1. As solved in Code 13 and Code 18, the buffer output from **mfill** would have an address of **0x17d400** and a size of **0x10**. The buffer

is passed to **mem_c::mem2periph**, which passes a stream object to **periph_c::mem2periph**. So the PSS model includes a data path we can use to provide the expected data to the **do_cb_periph_m2p** method, which we can do by adding a data field to the buffer declaration from Code 4:

```
buffer cb_mbuf {
    rand bit [32] addr;
    rand bit [32] size; // number of bytes
    rand bit[8] step;
    list<bit[32]> expected_data;

    constraint c1 {step >= 0;}
}
```

Code 21: PSS Buffer Definition with data field

A *list* in PSS is analogous to a queue in SystemVerilog in that it is a variable-sized ordered list of values of the given type, in this case, 32-bit data values. Then, instead of relying on UVM or C to randomize the data values, we include the randomization in the realization of the **mem_fill** action as shown in Code 22.

```
extend action cb_mem_fill {
  list<bit[32]> data;

  exec post_solve {
    repeat(obuf.size) {
      data.push_back(0);
    }
    randomize data;
    obuf.expected_data = data;
  }
}
```

Code 21: PSS Buffer Definition with data field

The **exec post_solve** block allows the PSS solver to determine fields like **size** as part of the normal constraint solving to determine the scenario and then use these fields to finalize details, like the random data to be written. By including the data in the buffer object, it necessarily gets passed to the **mem_c::mem2periph** action as well.

```
extend action mem2periph {
  constraint ostr.expected_data == ref_model(ibuf.expected_data);
}
```

Code 22: Additional constraint to pass expected data to stream output

The call to **ref_model()** allows the data to be manipulated as appropriate to ensure the correct data values get passed along. In the degenerate case, the **ref_model()** would simply return the input data. Then, in the **periph_c::mem2periph action**, we can do the actual checking:

```
extend action mem2periph {
  exec body {
    check_data(istr,expected_data);
  }
}
```

Code 23: Checking the data in the peripheral

Note that the realization of the **mem2periph** action will perform the appropriate target operation to receive the data, and will also check that the received data matches the data passed in through the PSS model.

V. File Organization

In order to take full advantage of the reuse and portability benefits available with PSS, some forethought must be used in organizing the code between packages and files. In general, our recommendation is to keep a single component or package in its own file. That way, the user can control which specific implementation of a given element is used

by specifying the desired file on the command line for the PSS tool and/or simulator with the generated code. Consider the following:

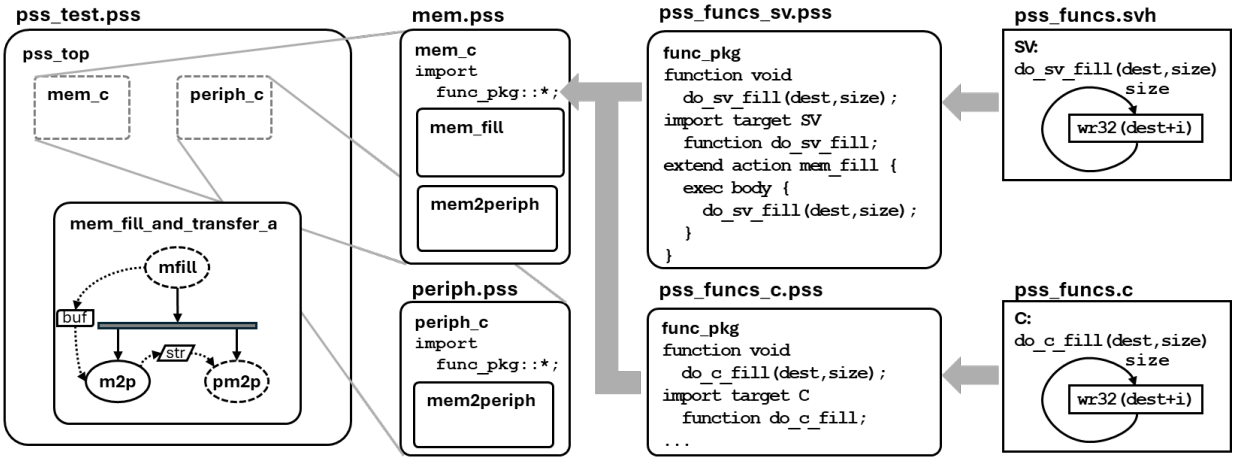


Figure 5 – Organizing PSS Files

The top-level PSS component, **`pss_top`**, defined in **`pss_test.pss`**, instantiates the **`mem_c`** and **`periph_c`** components and defines the **`mem_fill_and_transfer_a`** action. The **`mem_c`** component, defined in **`mem.pss`**, defines the relevant memory actions such as **`mem_fill`** and **`mem2periph`** referenced above. It imports the **`func_pkg`** package for which, in this example, we provide two implementations. The **`pss_funcs_sv.pss`** file imports the SystemVerilog version of the behavior, shown here as **`do_sv_fill`**, and extends the **`mem_fill`** action to call this method in the **`exec body`**. The **`pss_funcs_c.pss`** file imports the C version of the method and will similarly extend the action to call this version. When the generated file is compiled, the language-appropriate source file is also compiled to provide the target-specific implementation of the method.

VI. PSS Model Results

Proper organization of PSS models with these exec blocks ensures maximum reuse and portability. The resulting code can reproduce identical scenarios across all target environments, whether UVM simulation, Linux-based C environments, or embedded bare metal implementations.

For the code snippets shown above, the solved PSS scenario can be visualized as shown in the following diagram:

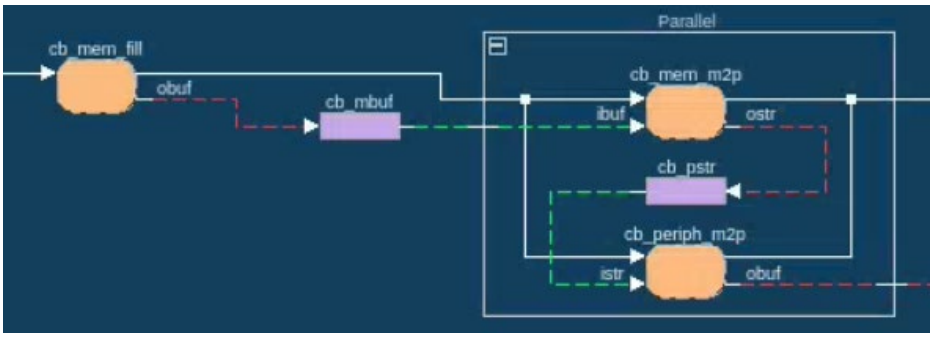


Fig. 4 – Solved PSS Scenario for Memory-to-Peripheral Transfer

When the generated sequence is executed in a UVM environment, the results can be observed as shown in the screenshot in Figure 5, where MFILL represents the **`mem_fill`** action, M2P represents **`mem_c::mem2periph`** and PM2P represents **`periph_c::mem2periph`**.

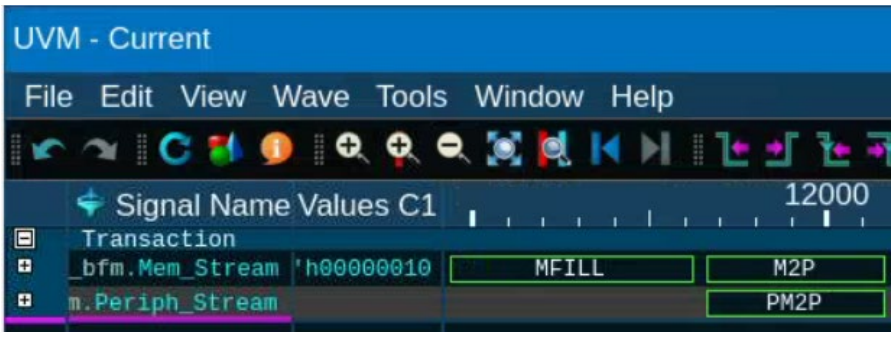


Figure 5 – Simulated UVM Scenario for Memory-to-Peripheral Transfer

The generated implementation using embedded ARM32 bare metal C code produces equivalent results as shown in the QEMU interface screenshot in Figure 6.

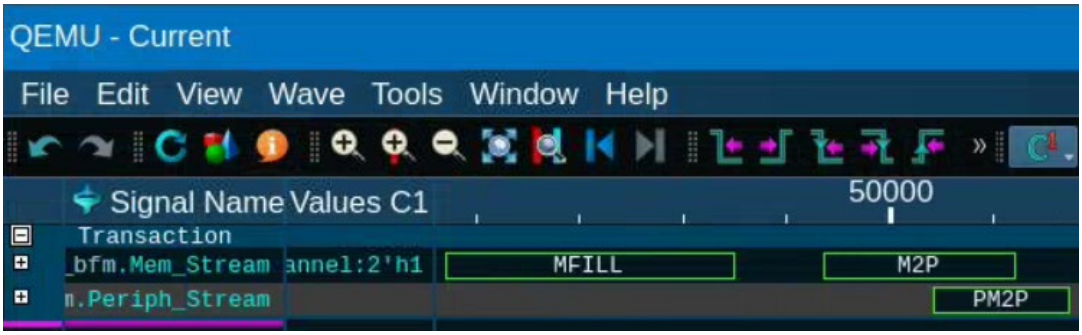


Figure 6 – Simulated UVM Scenario for Memory-to-Peripheral Transfer

In this particular case, the bare metal threading package was implemented via a manually generated (AI-assisted) polling mechanism, which accounts for the timing differences between the two waveforms shown. In Figure 5, the SystemVerilog fork/join semantics allow the **`M2P`** and **`PM2P`** boxes to appear to start at the same time, while the bare metal package allows the **`M2P`** operation to start and then yield at some point before completion before the **`PM2P`** can start. Both diagrams represent valid implementations of the scenario defined by the PSS model in that **`MFILL`** completes before either **`M2P`** or **`PM2P`** starts, and they both complete before anything else can start.

VII. Conclusion

This paper demonstrates a practical methodology for achieving true scenario portability across the entire verification flow using PSS. By properly organizing PSS models with appropriate exec blocks and leveraging the standard's two-layer architecture, verification teams can develop scenarios once and deploy them across UVM simulation, Linux-based C environments, and embedded bare metal implementations. The approach presented enables seamless transition between verification environments throughout the development cycle, from early simulation through final silicon validation. The intelligent randomization capabilities of PSS, combined with its platform-agnostic abstract modeling layer, offer substantial benefits in development efficiency and verification coverage. This approach can easily save weeks to months that would otherwise be spent re-implementing UVM-based simulation tests as C-based tests to run in the lab or post-silicon.

While we demonstrated the approach using a vendor-specific tool chain, the methodology is generally applicable to any PSS user, regardless of the PSS tool being used, assuming it implements the PSS standard.

Acknowledgements

The author wishes to thank TziYang Shao of Siemens EDA, who provided essential assistance in tailoring the QEMU kernel to support embedded ARM code generation, and in linking the QEMU kernel to the simulation.

The author also wishes to thank Sergey Khaikin of Cadence Design Systems, for showing him the embedded error checking approach discussed in Section IV.

References

[1] Accellera Systems Initiative, *Portable Test and Stimulus Standard Version 3.0*, August, 2024. [Online]. Available: https://accellera.org/downloads/standards/portable-stimulus

[2] T. Fitzpatrick, B. Oden, A. Abd-Allah, and B. Graham, “PSS and Protocol VIP: Like a Hand in a Glove,” in Proc. DVCon U.S., San Jose, CA, USA, Mar. 2025.