cadence®

Three Things You Need to Know to Use the Accellera PSS

Sharon Rosenberg, Senior Solutions Architect, Cadence

Three primary considerations for adopting the Accellera Portable Stimulus Standard (PSS) are understanding the following: the value and relevance of this standard; the fundamental concepts of PSS modeling, including building blocks, process, and mindset; and PSS portability and how these scenarios can be applied to a specific platform. In this paper, we explore these three topics.

Contents

Introduction

My favorite quote on the portable stimuli approach was made in 2012 by a manager who had the responsibility for the verification and validation of a large mobile chip. In a post-evaluation session, the manager stood in front of his management and said the following: "This solution is like solving world hunger." That's a big claim!

To balance this perspective, it should be emphasized that PSS does not apply to just any verification/validation task, and its deployment requires qualification for relevance and return on investment. It's important to understand the intended applications of PSS to properly assess whether it matches your needs.

Portable Stimulus Application and Value

Assume that you are an experienced validation engineer who has joined a new SoC verification team, and your role is to develop system tests. Your tests are likely to be coded in C with one main entry point function for each CPU core, and a lot of complex logic to coordinate concurrent execution. You are given a header file with the declarations of C firmware routines that activate the various system engines and I/O devices. The desired tests cannot just call these routines in arbitrary order with random parameters. There are complex rules on how to set up and activate the device engines with a specific use case in mind.

These would be bare-metal tests without operating system services such as mutexes and semaphores. There are a few factors to consider when crafting such tests:

- Code the logic to synchronize execution threads, without introducing data races or deadlocks or resource conflicts.

  For example, you may be asked to stress the interconnect by using all the available CPU cores and DMA-enabled devices. You must ensure that a CPU does not start reading a buffer before it was written by establishing synchronization points between the cores. In addition, do not assign the same DMA channel to perform two tasks at the same time.

- Be aware of state-machines and operation modes.

  If your goal is low-power scenarios, you may need to visit different low-power states, but not all the state transitions are legal. You may also need to coordinate specific traffic and timing to the appropriate power states; in other words, do not drive traffic to a display if its power domain is off.

- Carefully select the configuration and traffic for your multiple subsystems or IPs use cases.

  If you are assigned to validate a multimedia subsystem with a pipeline of engines that can process a video or images, each engine may support different video formats and may allow limited parallel processing. The challenge is to select a legal pipeline configuration and use appropriate video data that will match all the engine supported formats.

- Consider memory region attributes and devices accessibility rules.

  For example, I/O and cache coherency scenarios in which multiple CPU cores and I/O devices need to have parallel access to specific cacheable regions to stress the system, all while enabling self-checking.

These are just a few examples of why implementing test scenarios at the system level is a challenge. For every test, you must carefully plan and implement the scenario—an effort that can span hours, days, or weeks, even for experienced engineers.

PSS Modeling Concepts

PSS addresses the system verification challenges listed above by defining a language to express dependencies between targeted operations using the right set of abstractions. This enables a test writer to describe desired use cases while leaving the hard work of solving for legal scenarios and the effort to implement them to a tool.

In the PSS era, using an industrial-strength tool consists of two main steps.

- Step one: Model the system units of behaviors and their composition rules. These will typically be reused across projects and in subsystem context.
- Step two: In the specific project, test writers leverage the modeled behaviors to compose use cases with constraints on control flow, data flow,and scheduling.

These descriptions are the common high-level language for communicating with teams and stakeholders, and reasoning about application use cases and test scenarios.

Note that PSS is exposed to users as a domain-specific language (DSL) and as a C++ library. For education purposes, we will start with C++ code and then show the DSL input format. Both input formats are equivalent, and users can adopt their preferred style. The choice reflects a tradeoff between more concise and readable code in the DSL versus more seamless inline integration with procedural computation at specific points in the flow in C++.

Step One: Modeling the system units of behaviors and their composition rules.

In this section, we introduce the following PSS terms: *actions, inputs, outputs, flow-objects, resources,* and *states* to capture the action composition rules.

PSS incorporates the concept of a unit of behavior that is called an action. Actions can represent both device or validation environment operations. Examples of actions are the read or write operations that a CPU core performs, a transfer operation of a DMA, a TLP write of a PCI Express® (PCI®) transactor or even a power transition that is performed by the power management unit.

```
// memory buffer declaration
struct data_buff_s : public buffer {
   PSS_CTOR(data_buff_s, buffer);
   rand_attr<int> size {"size", range<>(4,1024)};
   rand_attr<bit> addr {"addr", width(64)};
};
type_decl<data_buff_s> data_buff_s_decl;

// CPU core resource declaration
struct core_s : public resource {
   PSS_CTOR(core_s, resource);
};
type_decl<core_s> core_s_decl;

class cpu_c : public component {
public:
   PSS_CTOR(cpu_c, component);

   class write_data : public action {
   public:
      PSS_CTOR(write_data, action);

      output<data_buff_s> dst {"dst"};
      lock<core_s> core {"core"};
   };
   type_decl<write_data> write_data_decl;

   class read_data : public action {
   public:
      PSS_CTOR(read_data, action);

      input<data_buff_s> src {"src"};
      lock<core_s> core {"core"};
   };
   type_decl<read_data> read_data_decl;
};
type_decl<cpu_c> cpu_c_decl;

class dma_c : public component {
public:
   PSS_CTOR(dma_c, component);

   struct channel_s : public resource {
      PSS_CTOR(channel_s, resource);
   };
   type_decl<channel_s> channel_s_decl;

   class mem2mem_xfer : public action {
   public:
      PSS_CTOR(mem2mem_xfer, action);

      input<data_buff_s> src {"src"};
      output<data_buff_s> dst {"dst"};
      constraint c { src->seg->size == dst->seg->size };

      lock<channel_s> channel {"channel"};
      share<core_s> core {"core"};
   };
   type_decl<mem2mem_xfer> mem2mem_xfer_decl;

   pool<channel_s>  channel_p {"channel_p", 16}; // 16 DMA channels per engine
   bind channel_bind { channel_p };
};
type_decl<dma_c> dma_c_decl;
```

Code Sample 1: PSS C++ code

Code Sample 1 and Figure 1 show code and graphical representations of several PSS actions. Note that actions associate behaviors with their composition rules—instead of keeping these rules in your head, you essentially teach the PSS tool what the legal conditions are and which attribute values are used to activate each system behavior.

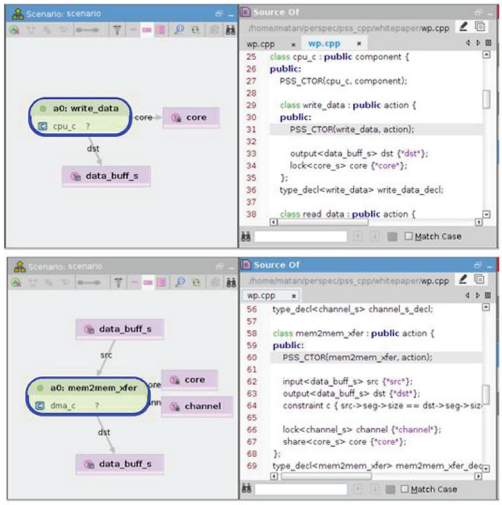

Figure 1: Action samples

The CPU component has `read_data` and `write_data` actions. The `read_data` action has an *input* buffer declaring that a pre-requisite to the `read` action is the availability of a buffer in memory. You also declare that one core will be exclusively locked for the `read_data` action duration using the lock statement.

The `write_data` action locks one of the cores but also has an output buffer, which means that after a write action completes, a memory buffer becomes available. The buffer that is one action's *input* is another action's *output*. It is one kind of *flow-object*.

The DMA action `mem2mem_xfer` is a slightly more elaborate example showing that executing a transfer requires input for the source buffer, the output of the transfer destination buffer, locking one of the DMA channels, and sharing one of the CPU cores.

PSS provides built-in semantics to capture resource requirements, configuration requirements, *state* and operation-mode dependencies, and more. Algebraic constraints can be used to tune the attributes and parameters of these operations, such as buffer size, specific configuration, and operation modes.

The following code snippet shows the top project instantiation that leveraged the reusable components and actions:

```
// top level project specific instantiation
class pss_top : public component {
public:
   PSS_CTOR(pss_top, component);
   comp_inst<cpu_c> cpu {"cpu"};
   comp_inst<dma_c> dma {"dma"};

   // 4 CPU cores in this configuration
   pool<core_s>  core_p {"core_p", 4};
   bind core_bind { core_p };

   // system memory pool available to components
   pool<data_buff_s> data_buff_p {"data_buff_p" };
   bind data_buff_bind { data_buff_p };
};
type_decl<pss_top> pss_top_d;
```

Code Sample 2: pss_top instantiation

Note that the read and write actions are well encapsulated and do not assume the number or structure of the core resources. Under the built-in `pss_top` component, the CPU and DMA components are instantiated with the available memory and core resources. The *bind* directive is used to connect the reusable CPU and DMA instances with our project-specific cores and buffer pools.

Step Two: Use the modeled behaviors to efficiently compose use cases that include control and data-flow and scheduling.

The related PSS terms in this section are compound *action, activity, parallel, sequence, schedule, binding*.

Now that the test legality rules have been captured, they can be used to create sophisticated high-quality content using a simple scenario specification language. Consider the following use case in the verification plan: "Initialize a buffer and create a random schedule of the DMA mem2mem transfers, each assigned a random channel and programmed by a randomly selected core". While this seems like a simple request, those who know bare-metal parallel programming understand the difficulties it presents. If the action is not in the same thread, every producer must notify the consumer of data availability, and at all times the test cannot exceed the overall number of DMA channels.

Code Sample 3 illustrates how the PSS scenario specification for such use case.

```
class parallel_transfers : public action {
public:
   PSS_CTOR(parallel_transfers, action);

   action_handle<cpu_c::write_data> wd {"wd"};
   action_handle<dma_c::mem2mem_xfer> xfer1 {"xfer1"};
   action_handle<dma_c::mem2mem_xfer> xfer2 {"xfer2"};
   rand_attr<int> count {"count", range<>(2,20)}; // randomize count

   activity a {
      sequence {
         wd,  // initialize buffer by a random core
         repeat { count,
            parallel { // do parallel transfers
               xfer1,
               xfer2
            }
         }
      }
   };
};
type_decl<parallel_transfers> parallel_transfers_decl;
```

Code Sample 3: Parallel legal random transfers scenario specification in DSL

The scenario specification is an intuitive translation of the verification plan request, allowing a tool to legally complete the unspecified attribute values. In the *activity block*, the user describes the desired scenario in terms of control flow and data flow. Much like in C or UVM sequences, the user can execute scenarios sequentially, in parallel or even in a random timing using the `schedule` statement. All these use-case directives are solved while considering the legality rules.

The activity diagram in Figure 2 captures a possible solution to the scenario specification shown in Code Sample 3.

In this case, a PSS tool solved the partial scenario specifications, resulting in multiple legal scenario instances. Each scenario instance distributes the CPU cores and DMA channels to achieve a different—yet legal—scenario solution. Note that the user can request the tool to randomize each DMA transfer at gen-time (the top diagram in Figure 2) or to defer the execution of the loop to run time (the bottom diagram in Figure 2).

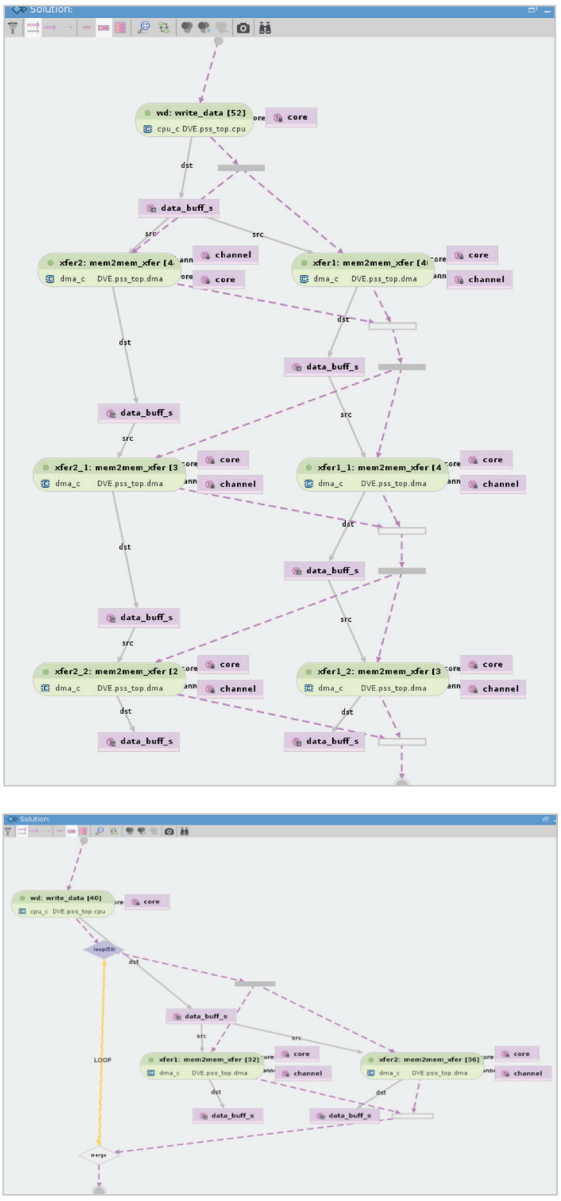

Figure 2: Parallel legal random transfer scenario instance screenshots

Actions can also be hierarchically composed of other lower-level action to achieve a more complex scenario. Code Sample 4 shows an example of such an action that uses the previously defined `parallel_transfers` actions.

```
action write_read_loop {
   activity {
      parallel {
         repeat(20) {
            do write_data;
            do read_data;
         };
         do parallel_transfers;
      };
   };
};
```

Code Sample 4: Nesting actions using the DSL input format

It takes time to adopt the PSS modeling concepts. Not all test writers need to model environments and many test writers can leverage the reusable models that are created by a few. At the same time, this approach will improve your productivity dramatically over writing the tests manually.

PSS Realization and Portability

PSS allows executing the same test intent in different target platforms, as shown in Figure 3.

The standard allows users to teach a PSS tool about their system and verification environment APIs. You can achieve portability by teaching the tool more than a single API style and requesting the tool to realize tests in the required target platform. For example, to activate a sub system in a bare-metal embedded execution, you may capture a C firmware routine, and the SystemVerilog (SV) testbench may call the corresponding SV register sequence.

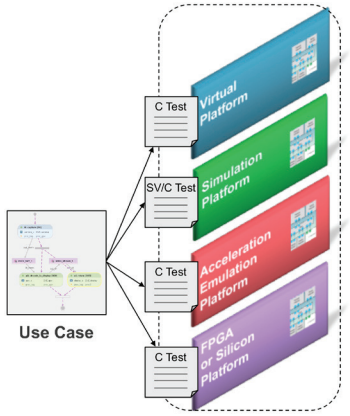

Figure 3: Retargeting use cases

If your DMA has the following C APIs to program and initiate a transfer, consider Code Sample 5.

```
void dma_program(int chan_num, int size, int src_buff, int dst_buff);
void dma_start(int chan_num);
void dma_wait_for_completion(int chan_num);
```

The user can define the template of generated code for an action:

```
extend mem2mem_xfer {
   exec body C = """
   dma_program({{channel.instance_id}}, {{src_data.seg.size}},...);
   dma_start({{channel.instance_id}});
   dma_wait_for_completion({{channel.instance_id}});
   """;
};
```

Code Sample 5: Using code template in exec body C

Note that PSS supports aspect-oriented programming (AOP), which has proven to be an essential feature for hardware verification. In Code Sample 5, we extend the original `mem2mem_xfer` definition and add the needed target code templates. If the action `mem2mem_xfer` was randomized to be executed as part of a use case, the PSS tool inserts the template code under the desired core entry function, embeds the randomized attribute values within the mustache signs, and adds sync points between the parallel threads to match the desired randomized timing. This technique allows defining new types and global objects that are needed frequently in software-driven tests, and leveraging the UVM factory on hardware testbenches.

```
extend mem2mem_xfer {
   exec body SV = """
   dma_program_sv({{channel.instance_id}}, {{src_data.seg.size}},...);
   dma_start_sv({{channel.instance_id}});
   dma_wait_for_completion_sv({{channel.instance_id}});
   """;
};
```

Code Sample 6: Using code template in exec body SV

If SV code is required, you can create an `exec body SV` block and call the needed tasks or UVM sequences, as shown in Code Sample 6. With the template style, you can generate a new class type, constraints, and factory calls to emulate a hand-written SV test. You can achieve portability by translating the same use case to either embedded C code or SV.

If the validation environment uses only a procedural interface, the user can realize an action with native execs. This style lets the user declare function signatures as import functions, and call them within an exec body block, as shown in Code Sample 7.

```
import void dma_program(int chan_num, int size,...);
import void dma_start(int chan_num);
import void dma_wait_for_completion(int chan_num);

action mem2mem_xfer {
   input data_buff_s src_data;
   output data_buff_s dst_data;
   ...
       exec body {
      dma_program(channel.instance_id,src_data.seg.size,...);
      dma_start(channel.instance_id);
      dma_wait_for_completion(channel.instance_id);
   };
};
```

Code Sample 7: Using import functions in exec body

This technique is less flexible, as it assumes strict procedural interfaces, but allows gen-time type checking with no duplication of the templates. More importantly, using native execs is the way to feed data back from the DUT/environment to the PSS model, for reactive test behavior.

Summary

When considering the use of PSS, users need to understand three main topics:

- The value of PSS
- Basic modeling concepts
- PSS realization and portability

We showed some generic examples at the SoC-level, and demonstrated the main steps required in PSS modeling.

Further Information

For more information, including how PSS can address your specific needs (low-power, coherency, memory virtualization, multi-IP scenarios, or interconnect testing), contact pss_info@cadence.com.