WHAT IS PORTABLE STIMULUS?
Over the past few years, lots of energy has been invested in improving the productivity and quality-of-results of design verification. The bulk of this effort has focused on techniques that are most applicable at the block level. These techniques – such as constrained-random transaction generation, functional coverage, and the UVM – have had a dramatic positive improvement on verification quality and productivity. However, while these techniques have been successful at the block level, verification continues to be increasingly challenging at the subsystem and SoC levels, and thus a new approach is called for.

Both commercial and in-house tools have been developed to improve the productivity and efficiency of verification. Mentor’s Questa® inFact™ is one example of a commercial tool that raises the level of abstraction (boosting productivity), increases test-generation efficiency, and can be applied across a wide variety of verification environments.

As interest in bringing automated tests to environments beyond transaction-oriented block-level environments has increased, so has interest in having a standardized input-specification language with which to specify these tests. In response, Accellera launched a working group, titled the Portable Stimulus Working Group (PSWG), to collect requirements, garner technology contributions, and specify a standardized input language that can be used to specify test intent that can be targeted to a variety of verification platforms. Mentor has been participating and driving the activity in the PSWG, and we’ve contributed our technology and expertise to the standardization process.

The goals of portable stimulus are illustrated by the figure below. Specifically, the concept is to have a single description of test intent (the portable stimulus description) that can be targeted to IP-level, subsystem-level, and SoC-level verification and be able to implement that test intent in a way that is appropriate to the verification engine used for that type of verification.

In addition to describing the goals of portable stimulus, it’s also important to emphasize what it is not. A portable stimulus description doesn’t force all descriptions to be at a single level of abstraction or force all test
intent to be done in a single, limited way. As we’ll see, there are multiple elements to the portable stimulus specification currently being developed by the PSWG, and users have the flexibility to describe their test intent in the way that is most natural to their verification task. It’s also important to note that it is the creation of highly-efficient automated tests that the PSWG seeks to make portable. Portable stimulus is not just a collection of “lowest common denominator” techniques that can easily be supported across all verification engines. Further, the Accellera Portable Stimulus Specification (PSS) is not intended to be a replacement for existing procedural languages, such as C/C++ or SystemVerilog. Since reuse of code in these existing languages is critical, Accellera PSS provides mechanisms to reuse behavior described in these languages and others.

PORTABLE STIMULUS FUNDAMENTALS
Portable stimulus seeks to raise the level of abstraction and enable users to automate testing of the complex scenarios that emerge in subsystem- and SoC-level verification. However, the PSS under development by the Accellera PSWG builds on the base of constraint-based, transaction-level verification, which is already well-understood and widely deployed today. On top of these fundamental features, the Accellera PSS provides features squarely targeted at enabling complex SoC-level scenarios to be productively captured by the user and efficiently realized.

Thus, the Accellera PSS supports random and non-random data fields and structures, familiar SystemVerilog constraints, and inheritance patterns familiar from object-oriented languages.

Building up scenarios in SystemVerilog is done by mixing constrained-random generation with procedural code. This introduces limitations, in terms of being able to reuse scenarios and customize them without changing the original code. Accellera PSS provides an action as a primitive element of behavior, as well as a way to encapsulate complex behaviors in a way that can easily be reused and customized. Within complex actions, sequential and parallel execution of sub-actions, as well as repetitions over sub-actions, can be specified. The behavior within an action is specified in a declarative manner that enables a high degree of automation and static analysis.

Accellera PSS provides dedicated constructs for modeling the resource requirements of actions as well as data exchanges between actions in a scenario. This enables a user to describe the rules that bound legal scenarios, and allows a tool to automatically create complex legal scenarios based on those rules – much as data constraints specify the bounds of a legal transaction, enabling a constraint solver to automate generation of many legal transactions.

PORTABLE STIMULUS AT THE BLOCK LEVEL
There is an enormous benefit to applying portable stimulus in block-level verification environments. Portable stimulus tools require very efficient and targeted test generation, because of the requirement to efficiently generate tests for SoC-level environments. In a block-level environment, efficient test generation achieves functional coverage goals more quickly and finds bugs earlier in the verification cycle. Users of Mentor’s Questa inFact, for example, have typically found that
the tool is 10–100x more efficient than random generation at achieving coverage goals, enabling them to find bugs more quickly and extend the scope of their coverage without increasing simulation resources.

The example used in this article is a multi-channel DMA engine. As is typical with DMA engines of this type, memory-transfer operations are characterized by a transfer descriptor that captures the transfer size, source and destination addresses, address increment settings, and detailed transfer options. At the block level, we want to comprehensively exercise combinations of these transfer-descriptor fields in order to comprehensively verify the DMA implementation.

A simplified view of the UVM testbench surrounding this IP is shown below. The DMA engine is exercised using a UVM sequence that programs registers within the DMA engine according to a DMA descriptor class.

### REUSING SV CONSTRAINTS

The DMA descriptor class contains fields and constraints that define a valid DMA transfer. The ability to leverage this existing description from a portable stimulus description is important, since an engineer has invested time to correctly capture the constraints, and since the rest of the environment is driven by this class. Fortunately, the transaction-level subset of Accellera PSS overlaps with the SystemVerilog constraint subset to an extent that many SystemVerilog constraint-based descriptions can be converted to PSS descriptions.

<table>
<thead>
<tr>
<th>SystemVerilog</th>
<th>PSS</th>
</tr>
</thead>
<tbody>
<tr>
<td>class <code>wb_dma_descriptor</code></td>
<td>struct <code>wb_dma_descriptor</code></td>
</tr>
<tr>
<td>extends uvm_sequence item;</td>
<td>extends uvm_sequence item;</td>
</tr>
<tr>
<td>`uvm_object_utils</td>
<td>`uvm_object_utils</td>
</tr>
<tr>
<td><code>(wb_dma_descriptor)</code></td>
<td></td>
</tr>
<tr>
<td>rand bit[5:0] channel;</td>
<td>rand bit[5:0] channel;</td>
</tr>
<tr>
<td>rand bit mode;</td>
<td>rand bit mode;</td>
</tr>
<tr>
<td>rand bit inc_src;</td>
<td>rand bit inc_src;</td>
</tr>
<tr>
<td>rand bit inc_dst;</td>
<td>rand bit inc_dst;</td>
</tr>
<tr>
<td>rand bit src_sel;</td>
<td>rand bit src_sel;</td>
</tr>
<tr>
<td>rand bit dst_sel;</td>
<td>rand bit dst_sel;</td>
</tr>
<tr>
<td>rand bit[31:0] src_addr;</td>
<td>rand bit[31:0] src_addr;</td>
</tr>
<tr>
<td>rand bit[31:0] dst_addr;</td>
<td>rand bit[31:0] dst_addr;</td>
</tr>
<tr>
<td>rand bit[8:0] chk_sz;</td>
<td>rand bit[8:0] chk_sz;</td>
</tr>
<tr>
<td>constraint <code>channel_c</code> {</td>
<td>constraint <code>channel_c</code> {</td>
</tr>
<tr>
<td>channel inside [0..30];</td>
<td>channel inside [0..30];</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
<tr>
<td>constraint <code>tot_sz_c</code> {</td>
<td>constraint <code>tot_sz_c</code> {</td>
</tr>
<tr>
<td>tot_sz &gt; 0;</td>
<td>tot_sz &gt; 0;</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
<tr>
<td>constraint <code>chk_sz_c</code> {</td>
<td>constraint <code>chk_sz_c</code> {</td>
</tr>
<tr>
<td>chk_sz &gt; 0;</td>
<td>chk_sz &gt; 0;</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
<tr>
<td>endclass</td>
<td>endclass</td>
</tr>
</tbody>
</table>
Questa inFact provides an import tool for exactly this purpose. A comparison of the original SystemVerilog class and PSS struct is shown above. Importing the SystemVerilog description and making it available inside a PSS description leverages the effort invested in creating the sequence-level description in SystemVerilog, makes getting started with PSS easier, and ensures that the PSS description stays in sync with any changes made to the sequence item on the SystemVerilog side.

**SPECIFYING PRIMITIVE OPERATIONS**

Now we will describe the most basic DMA operation: a DMA transfer. In a portable stimulus description, the data and behavior of an operation is encapsulated in an action.

As shown above, an action is declared within a component, which encapsulates resources shared by multiple actions. At this basic block-level of verification, we don’t need anything special in our wb_dma_c component. Our do_dma action simply captures a random wb_dma_descriptor struct field. We’ll fill in the implementation details later.

**DESCRIBING SCENARIOS**

From a test perspective, one of the first things we might want to do is simply generate a series of single DMA transfers. We describe our testing scenarios inside actions, just like our primitive operations. Since our test scenarios are themselves composed of actions, we add an Activity Graph (keyword: activity) to specify the relationships between sub-actions.

```
component wb_dma_block_c{
    wb_dma_c      dma;
    action simple_xfer{
        do_dma    xfer;
        activity{
            repeat (256) { xfer; }
        }
    }
}
```

Note that we declare our simple_xfer action within a component. This component contains an instance of the wb_dma_c component that declares the do_dma action. Our simple_xfer action simply runs 256 repetitions of the do_dma action.

```
action back2back_xfer{
    do_dma    xfer1, xfer2;
    constraint { xfer1.desc.channel != xfer2.desc.channel; }
    activity{
        repeat (256) { xfer1;
            xfer2;
        }
    }
}
```

We might want to extend our testing a bit to perform two back-to-back DMA transfers, with the constraint that the channel used by the two transfers are different. This should provoke more-interesting activity within the DMA controller. Note how we can constrain the random fields of an action instance from above – something that is challenging to do with a directed-random sequence.
SPECIFYING THE ENVIRONMENT INTERFACE

Thus far, we haven’t worried much about how our actions will connect to the UVM testbench environment. The type extension capability provided by PSS makes it easy to layer in our interface to the environment without needing to change any of the actions or components we’ve already described.

In our UVM testbench, stimulus is driven by a UVM sequence that generates `wb_dma_descriptor` sequence items. We’ll want to integrate our PSS description inside a UVM sequence and also have it generate `wb_dma_descriptor` sequence items — but with the field values selected by our portable stimulus tool instead of using regular SystemVerilog constrained-random.

PSS packages provide a great way to encapsulate environment specifics, and we use a package here to contain the specifics of how our `do_dma` action will integrate with our UVM sequence. Specifically, we assume our sequence provides a task named `do_item` that accepts and executes a `wb_dma_descriptor` sequence item. The `import` statement specifies the signature of this external method.

Next, we need to specify how the `do_dma` action uses this imported method. PSS provides exec blocks to specify the relationship between PSS entities and external code. The `body` type of an exec block specifies execution-time behavior (much as the UVM sequence body task does). In this case, we specify that the execution-time behavior of the `do_dma` action is to pass the `wb_dma_descriptor` field to the `do_item` task.

And with that, we’re done! Our new PSS-drive UVM sequence can now drive the UVM testbench, with the advantage that we can much more efficiently exercise the DMA transfer modes.

PORTABLE STIMULUS AT SUBSYSTEM AND SOC LEVEL

At the subsystem and SoC levels, both what is verified and how it is verified change. Now instead of focusing on verifying the implementation of the DMA engine, we’re more interested in how the DMA engine is integrated with the other blocks in the subsystem or SoC. What’s also different, especially at the SoC level, is that we have an embedded processor, and we will want to drive at least some test activity with code running on that processor.
For a subsystem-level environment, we might start with a block diagram similar to what is shown below.

The DMA engine is now in the context of a subsystem that includes a processor (stubbed out with a bus functional model) and other IP.

Bringing our PSS description forward into this subsystem/SoC environment can be done in two steps:

1. Model the requirements of our scenario-level testing
2. Specify the new environment integration

As mentioned before, our goal in this environment is to verify the integration with the other IP in the subsystem. To do so, we will run multiple, parallel DMA transfers. The first thing we will do is extend our `dma_c` component to specify the resources available – in this case, 31 DMA channels. Also, we will create a new action type that consumes a DMA channel and specifies its data-flow requirements.

Our updated DMA component and action now specify:

- The DMA has 31 channel resources (using the pool of resources)
- Each DMA operation takes a source memory buffer and produces a destination memory buffer
- Each `do_mem2mem_dma` operation (which inherits from `do_dma`) requires access to a DMA channel (using the `lock` field)
- The channel specified in the DMA descriptor must be the same as the channel assigned to the DMA operation
- The source and destination addresses used for the DMA operation must match the source and destination memory buffer

```plaintext
component aes_c{
    resource struct aes_s{
        pool[1] aes_s aes;
        bind aes(*);
    }

    action do_encrypt{
        input membuf_s in_data;
        lock aes_s aes;
    }
}
```
Filling in a bit more detail, we create an `aes_c` component to model operations on the AES block. Note that the `do_encrypt` action takes a memory buffer and that we’ve forced the address of input data to be the buffer address of the AES block. Constraints on the `membuf_s` input are bi-directional, so this constraint forces the DMA to target the AES device when a `do_mem2mem_dma` action sends data to a `do_encrypt` action. We also use a resource pool in the `aes_c` component to specify that only a single operation can occur on the AES block at a given time.

Finally, we specify a component to represent our system that specifies the available resources (DMA and AES blocks), and we specify a top-level action to perform parallel DMA transfers. Note that we’ve only captured the fact that we want to perform four parallel DMA operations. This is a partial specification: we don’t specify where the data should come from, or where it should go. The PSS processing tool will infer and connect the appropriate actions to ensure legal scenarios are generated. Specifically:

- Each of the four parallel transfers occurs on a different DMA channel
- Only one operation can target the AES block at a time

Partial specification is an incredibly powerful technique for generating complex test scenarios from a simple, concise specification.

**SOC-LEVEL INTEGRATION**

In our subsystem-level environment, where DMA transfers are still driven by a sequence, we can reuse the same style of integration with the UVM environment that we did in the block-level environment. At the SoC level, our test will use utility functions written in C to program the DMA. In many cases, these utility functions will be the beginning of driver routines to later be used within an OS driver. Having our integration tests call these same utility routines provides an additional level of confidence in these utility routines, as well as exercising the integration of hardware IP (see code snippit on the following page).
Just as with our block-level environment, we can extend our core PSS description to layer in the environment specifics. In this case, we describe the C API that we will call (\texttt{wb_dma_drv_single_xfer}) and we provide the definition of an \texttt{exec} block for the \texttt{do_dma} action that calls this API and passes values from the DMA descriptor.

```verilog
define packagewb_dma_soc_pkg {
    import void wb_dma_drv_single_xfer(
        bit[31:0] channel,
        bit[31:0] src,
        bit[31:0] inc_src,
        bit[31:0] dest,
        bit[31:0] inc_dst,
        bit[31:0] sz);

define extend action do_dma {
    exec body {
        wb_dma_drv_single_xfer(
            desc.channel, desc.src_addr, desc.inc_src,
            desc.dst_addr, desc.inc_dst,
            desc.tot_sz);
    }
}
}
```

Just as with our block-level environment, we can extend our core PSS description to layer in the environment specifics. In this case, we describe the C API that we will call (\texttt{wb_dma_drv_single_xfer}), and we provide the definition of an \texttt{exec} block for the \texttt{do_dma} action that calls this API and passes values from the DMA descriptor.

**BOOST PRODUCTIVITY WITH PORTABLE STIMULUS**

Portable stimulus tools help to raise the level of test description and enable modeling of scenarios that would be very challenging to create with directed and transaction-level constrained-random tests. As a result, they enable automated creation of more unique tests. As we’ve seen through the example used in this article, features of the Accellera PSS input specification enable test intent to be retargeted to different environments, while the core of the description remains environment independent. We’ve also seen that random fields and constraints can be easily brought in from existing SystemVerilog descriptions and that key components of the standard can be adopted incrementally, making it easy to get started.

So the next time you face a verification task that exceeds the capabilities of directed or constrained-random tests, think about applying portable stimulus.
The Most Comprehensive Resource for Verification Training

27 Video Courses Available Covering
• SystemVerilog OOP
• Formal Verification
• Intelligent Testbench Automation
• Metrics in SoC Verification
• Verification Planning
• Introductory, Basic, and Advanced UVM
• Assertion-Based Verification
• FPGA Verification
• Testbench Acceleration
• PowerAware Verification
• Analog Mixed-Signal Verification

UVM and Coverage Online Methodology Cookbooks
Discussion Forum with more than 7850 topics
Verification Patterns Library

www.verificationacademy.com