Getting Generic with Test Intent: Separating Test Intent from Design Details with Portable Stimulus

by Matthew Ballance—Mentor, A Siemens Business

It's pretty typical to think about writing tests for a specific design. However, as the number of SoCs and SoC variants that a verification team is responsible for grows, creating tests that are specific to the design is becoming impractical. There has been a fair amount of innovation in this space recently. Some organizations are using design-configuration details to customize parameterized tests suites. Some have even gone as far as generating both the design and the test suite from the same description.

The emerging Accellera Portable Stimulus Standard (PSS) provides features that enable test writers to maintain a strong separation between test intent (the high-level rules bounding the test scenario to produce) and the design-specific tests that are run against a specific design. This article shows how Accellera PSS can be used to develop generic test intent for generating memory traffic in an SoC, and how that generic test intent is targeted to a specific design.

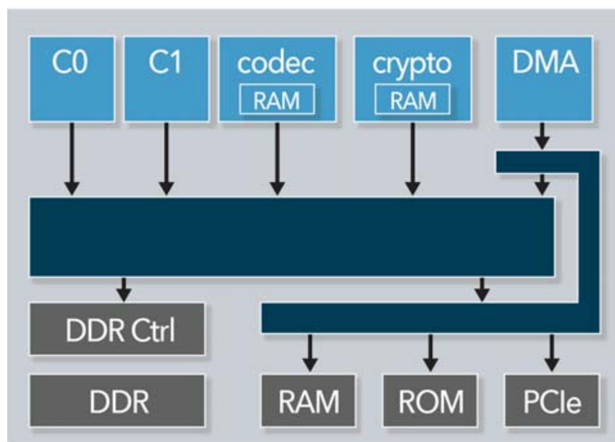


Figure 1: Example Multi-Core SoC

SoCs have complex memory subsystems with cache coherent interconnects for cores and accelerators, multiple levels of interconnects and bridges, and many initiators and targets – often with limited accessibility. While it's certainly important to verify connectivity between all initiators and targets, it is much more important to generate traffic between the various initiators and targets to validate performance. The goal here is to stress the interconnect bandwidth by generating multi-hop traffic and parallel transfers.

There are, of course, multiple approaches to generating traffic across an SoC memory subsystem. SystemVerilog constrained-random tests can be used, as could manually-coded directed tests. While both of these approaches are tried and tested, both also have drawbacks. Constrained-random tests are generally limited to simulation and (maybe) emulation, but we often want to run these traffic tests across all the engines – simulation, emulation, and prototype. Directed tests, while portable, are incredibly laborious to create, and often miss critical corner cases. A bigger challenge to both approaches, though, is the impact that a design change has on the test suite. Because test intent (what to test) is so enmeshed with the test realization (how to implement the test behavior), a change to the design often results in the need to manually review hundreds (or more) tests to identify needed updates to account for the new things that need to be tested in the design variant, and the things that are currently being tested that no longer exist in the new design variant.

If we take a step back from our memory subsystem traffic-generation problem, our test intent is actually quite simple: generate traffic between available initiators and available targets. Accellera PSS allows us to generalize a surprising amount of our

overall test intent and test scenarios without knowing anything about the specific resources present in the design. Accellera PSS also allows design details to be specified such that they are quite separate from the generic test intent and scenarios, making these easily reusable.

GENERIC TEST INTENT INFRASTRUCTURE

As previously stated, memory-subsystem traffic generation involves using initiators to transfer data from memory to memory. We start capturing generic test intent by characterizing a transfer. Specifically, where the data is stored and what initiator is performing the transfer.

```
package data_mover_types_pkg {

  buffer data_xfer_b {
    rand mem_region_e region;    // Memory region
    bit[63:0]         addr;
    rand bit[31:0]    size;
    rand data_mover_e data_mover; // Data-mover transferring the data

    rand bit[31:0]    num_hops;// Total hops this data has taken
  }

  enum data_mover_e {
    // Initially empty
  }

  enum mem_region_e {
    // Initially empty
  }

  import bit[63:0] alloc(
    mem_region_e region,
    bit[31:0]    sz);

}
```

Figure 2: Generic Description of a Transfer

Figure 2 shows the Accellera PSS description of a memory transfer and related types. A *buffer* in PSS is a data type that specifies that its producer must complete execution before its consumer can execute. The *data_xfer_b* type shown above captures the region in which the data is stored, the address and size of the data, what initiator transferred the data, and how many "hops" the data took in getting to its current location.

Note that empty enumerated types have been defined to capture the initiator moving the data (*data_mover_e*) and the memory region where the data is stored (*mem_region_e*). The specific enumerators that compose these types will be specified by the system configuration.

GENERIC TEST INTENT PRIMITIVES

Accellera PSS uses the *action* construct to specify the high-level test intent behavior. The *component* construct is used to collect resources and the actions that use those resources.

```
component data_mover_c {
  import data_mover_types_pkg::*;

  abstract action move_data_a {
    input data_xfer_b   in;
    output data_xfer_b  out;

    constraint in_out_c {
      out.size == in.size;
    }
  }
}
```

Figure 3: Generic Data Mover Action

Figure 3 shows the PSS description for a generic component action that transfers data. Note that the *move_data_a* action is abstract, which means that it cannot be used on its own (it's far too generic). However, this generic outline will be used as the basis for design-specific actions that represent the actual initiators in the system.

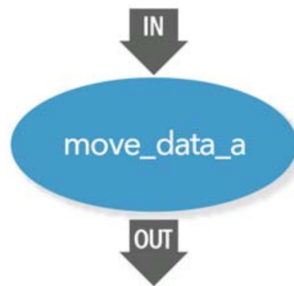


Figure 4: Generic Data Mover Action Diagram

The *move_data_a* action has an input and an output of type *data_xfer_b*. This specifies that some action must run before an action that inherits from *move_data_a*, and that this action produces a data transfer descriptor that can be used by another action. Figure 4 shows a diagram of the *move_data_a* action with its input and output buffers.

```
action data_src_a {
  output data_xfer_b out;

  constraint out_c {
    out.num_hops == 0;
  }
  exec pre_solve {
    out.addr = alloc(out.region, out.size);
  }
  exec body {
    fill(out.addr, out.size);
  }
}
```

```
action data_sink_a {
  input data_xfer_b in;

  constraint in_c {
    in.num_hops > 0;
  }
}
```

Figure 5: Data Initialization and Sink Action

It's also helpful (and generic) to provide generic actions that produce an initialized data-transfer descriptor and one that accepts and terminates a series of transfers. Basic versions of these can be provided (as shown in Figure 5 above), and more environment-specific versions provided for specific designs.

GENERIC TEST INTENT

With just the infrastructure and primitives we've defined thus far, we can already get started specifying test intent. Our first test scenario is shown in Figure 6 below.

```
coverspec point2point_cs(data_xfer_b dst) {
  initiator_cp : coverpoint dst.mover;

  target_cp : coverpoint dst.region;

  initiatorXtarget : cross initiator_cp, target_cp;
}

action mem2mem_point2point_a {
  data_src_a   src;
  data_sink_a  sink;

  activity {
    src;
    sink with out.num_hops == 1;
  }

  point2point_cs cov(sink.in);
}
```

Figure 6: Point-to-point Scenario

We've created a top-level action (*mem2mem_point2point_a*) that instantiates the data_src_a and data_sink_a actions, and traverses them in an activity block with the stipulation that one action come between (num_hops==1).

Figure 7 (below) shows a graphical representation of our point-to-point scenario. Note that, while we have required an action to exist between src and sink, we haven't specified what it is – just that it must accept an input data buffer and produce an output data buffer. Likewise, our coverage goals are specified in terms of the set of target memory regions and initiators, despite the fact that we don't know which initiators and targets our design will eventually contain.

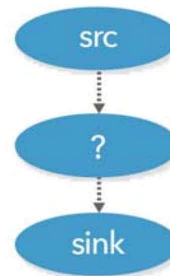


Figure 7: Graphical Representation of Point-to-point

This ability of a PSS description to capture generic test intent is one thing that makes it such a powerful way to specify test scenarios.

CAPTURING SYSTEM SPECIFICS

Of course, we do have to capture the specifics of our actual design before we can generate interesting and legal tests. Our example system contains:

- Two processor cores
- One DMA engine with 32 channels
- Two accelerators with private memory
- A DDR controller
- An internal RAM and ROM

We need to provide actions to describe the behavior of the initiators in our system that we want to test.

```
component cpu_c : data_mover_c {
  resource cpu_r {}

  pool cpu_r core;
  bind core *;

  import void memcpy(bit[63:0] dst,
    bit[63:0] src, bit[31:0] sz);

  action memcpy_a : data_mover_a {
    lock cpu_r core;

    exec pre_solve {
      out.addr = alloc(out.size);
    }
    exec body {
      memcpy(out.addr, in.addr, in.size);
    }
  }
}
```

Figure 8: CPU Component and Action

Figure 8 shows a PSS component and action that represents one of the processors in our system. PSS groups actions and the resources they require inside a component. In this case, we have a resource type to represent the CPU being used (*cpu_r*), and a pool of that resource type (*core*) to ensure that only one CPU operation can occur at one time.

Note that our cpu_c component inherits from the data_mover_c component, and the memcpy_a action inherits from the data_mover_a action.

```
component dma_c : data_mover_c {
  import data_mover_types_pkg::*;

  import void dma_xfer_mem2mem(
    bit[15:0] channel,
    bit[63:0] src, bit[63:0] dst, bit[31:0]sz);

  resource dma_channel_r {}

  pool dma_channel_r channels[32];
  bind channels *;

  action dma_mem2mem_a : move_data_a {
    lock dma_channel_r channel;

    exec pre_solve {
      out.addr = alloc(out.region, out.size);
    }
    exec body {
      dma_xfer_mem2mem(
        channel.instance_id,
        in.addr, out.addr, out.size);
    }
  }
}
```

Figure 9: DMA Component

As a consequence, it will have the same data buffer input and output that the data_mover_a action has.

Figure 9 below shows a description of the DMA component. Just like with our CPU component, we use a resource to describe how many parallel operations can run on an instance of the DMA component. Because we have 32 DMA channels, we create a pool of 32 *dma_channel_r* resources.

CAPTURING SYSTEM RESOURCES

Thus far, we have captured information about blocks within the design. These components and actions may well have been created by the teams responsible for verifying the IP, and reused at SoC level. Now, though, we need to capture the complete view of the resources and actions available in our SoC.

```
component mem_subsystem_c {

  cpu_c C0;
  cpu_c C1;

  codec_c  codec;
  crypto_c crypto;

  dma_c    DMA;

}
```

Figure 10: Mem Subsystem Resources

Figure 10 shows a top-level component with a component instance to capture each available resource in the design. We have two instances of the *cpu_c* component to represent the two processors, an instance of the *dma_c* component to represent the DMA engine, and component instances to represent the accelerators.

CAPTURING SYSTEM-LEVEL CONSTRAINTS

Now that we've captured the available resources, we need to capture the system-level constraints.

Figure 11 (above right) shows the system-level constraints. Note that we use type extension (the *extend* statement) to layer our system-level constraints into the existing enumerated types and base action. Like many powerful programming constructs, type extension is very useful when used

```
component mem_subsystem_c {
  // ...
  extend enum data_mover_types_pkg::mem_region_e {
    MEM_codec,
    MEM_crypto,
    MEM_DDR,
    MEM_RAM,
    MEM_ROM
  }

  extend enum data_mover_types_pkg::data_mover_e {
    DM_C0,
    DM_Cl,
    DM_codec,
    DM_crypto,
    DM_DMA
  }
```

```
extend action data_mover_c::move_data_a {
  constraint {
    if (comp == C0) { out.data_mover == DM_C0; }
    else if (comp == C1) { out.data_mover == DM_C1; }
    else if (comp == codec) { out.data_mover == DM_codec; }
    else if (comp == crypto) { out.data_mover == DM_crypto; }
    else if (comp == dma_c) { out.data_mover == DM_DMA; }
  }

  // Only accelerators can access local memories
  constraint {
    (out.region == MEM_codec) -> out.data_mover == DM_codec;
    (in.region == MEM_codec)  -> in.data_mover == DM_codec;
    (out.region == MEM_crypto) -> out.data_mover == DM_crypto;
    (in.region == MEM_crypto)  -> in.data_mover == DM_crypto;
  }
}
```

Figure 11: System-Level Constraints

judiciously, though overuse can easily lead to spaghetti code.

In addition to capturing the available memory regions (MEM_codec, MEM_crypto, etc), we also capture restrictions on which initiators can access which targets. Note that we've captured the restriction that only accelerators can access their local memories by stating that if either the source of destination memory is the accelerator-local memories, then the initiator must be the corresponding accelerator.

So, all in all, a fairly simple process to capture system capabilities and constraints.

BRINGING TEST INTENT AND SYSTEM SPECIFICS TOGETHER

Now that we have both generic test intent and system specifics available, we can bring them together and start generating specific tests.

```
component mem_subsystem_m2m_tests_c : mem2mem_test_c {
  mem_subsystem_c mem_subsys;

  // All actions below can share data buffers
  pool data_xfer_b data_xfer_pool;
  bind data_xfer_pool *;
}
```

Figure 12: System-Targeted Test Intent

Figure 12 shows how we can customize our generic test intent (*mem2mem_test_c*) with the capabilities of our specific system. Our specific test component extends from the generic test scenario we previously described. By instantiating the *mem_subsystem_c* component and connecting all the actions to the same pool of buffers, we make our system-specific actions and resources available to our generic test scenario.

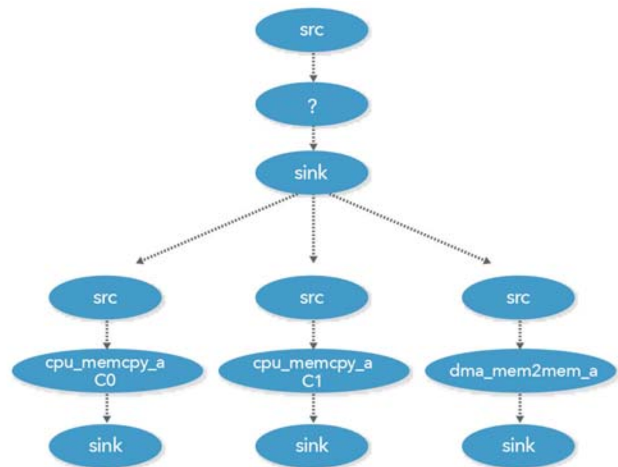


Figure 13: Example Specific Scenarios

Figure 13 shows a few specific scenarios that could result from our point-to-point scenario combined with our system description. One important thing about a portable stimulus description is that it is declarative and statically analyzable. This means that we can use analysis tools to discover exactly how many legal solutions there are to our point-to-point scenario in the presence of the available system resources. In our cases, there are a total of 72 legal point-to-point scenarios.

EXTENDING THE SCENARIO

We can easily expand our set of tests by using the system resource and constraint description we already have, and just altering our original test scenario a bit.

```
action mem2mem_multi_hop_test_a {
  data_src_a src;
  data_sink_a sink;

  activity {
    src;
    sink with out.num_hops == 2;
  }
}
```

Figure 14: Expanding Test Scenario

For example, we can alter the number of 'hops' our data takes moving from source to sink, as shown in Figure 14. If we increase the number of transfers to 2, there are 864 possible test scenarios. Expanding the number of hops to 4 results in an incredible 124,416 legal test scenarios. Not bad for just a few extra lines of PSS description!

We can just as easily extend the scenario to account for parallel transfers. In this case, we reuse our two-hop scenario and run two instances in parallel (Figure 15).

```
action mem2mem_multi_hop_parallel_test_a {
  activity {
    parallel {
      do mem2mem_multi_hop_test_a;
      do mem2mem_multi_hop_test_a;
    }
  }
}
```

Figure 15: Generating Parallel Transfers

The resulting transfers will be parallel back-to-back transfers, an example of which is shown in Figure 16. Because we've captured the available resources and their restrictions, our PSS processing tool will ensure that only legal sets of parallel transfers are generated.

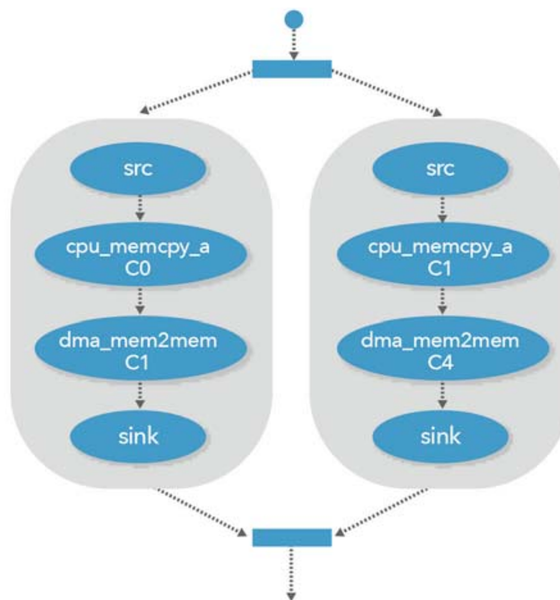


Figure 16: Example Back-to-Back Parallel Transfers

CHANGING THE DESIGN

Updating a test suite when the SoC changes, or trying to reuse a test suite for an existing SoC on a variant, is laborious and challenging. Just for a start, the set of available resources is different and the memory map is different.

The process is entirely different with a PSS-based test suite. Let's assume we have an SoC variant that doesn't have a codec, but does have an additional local RAM (Figure 17).

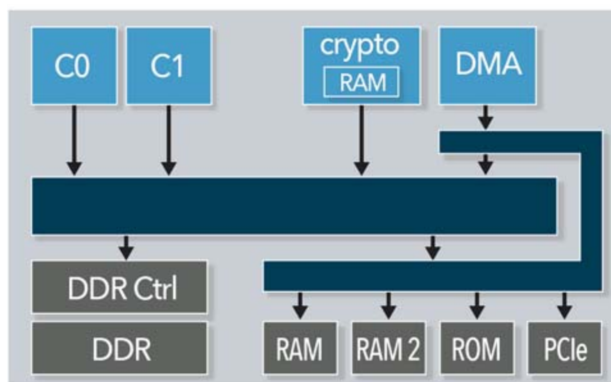


Figure 17: SoC Variant

```
component mem_subsystem_c {

  cpu_c C0;
  cpu_c Cl;

 codec_c  codec;
  crypto_c crypto;

  dma_c DMA;

  extend enum data_mover_types_pkg::mem_region_e {
    MEM_codec,
    MEM_crypto,
    MEM_DDR,
    MEM_RAM,
    MEM_RAM2,
    MEM_ROM
}
```

Figure 18: System Description Changes

The only change we need to make is to our description of the system resources. In this case, we need to remove the *codec* component instance and add another RAM to the *memory_region_e* enumeration, as shown in Figure 18 above.

With only these minor changes, a PSS processing tool can re-generate specific tests from our high-level test intent that match the new system. In this case, making these design changes expands the number of transfers described by our original point-to-point transfer test from 72 to 128.

SUMMARY

As we've seen from this simple example, the capabilities of Accellera PSS go far beyond the simple ability to target the same test intent to various verification platforms. PSS allows us to dramatically raise the abstraction level at which test intent is described, allowing us to easily capture generic test intent and test scenarios independent of the design details. Modeling available design resources and constraints and using these to shape test intent is straightforward. Finally, PSS test intent easily adapts to design changes, preserving the effort invested in capturing test intent. Combined, all of these capabilities dramatically boost verification productivity!

VERIFICATION ACADEMY

The Most Comprehensive Resource for Verification Training

32 Video Courses Available Covering

- UVM Debug
- Portable Stimulus Basics
- 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 8250 topics

Verification Patterns Library

www.verificationacademy.com

Mentor®
A Siemens Business

Mentor®
A Siemens Business
www.mentor.com

Editor:
Tom Fitzpatrick

Program Manager:
Rebecca Granquist

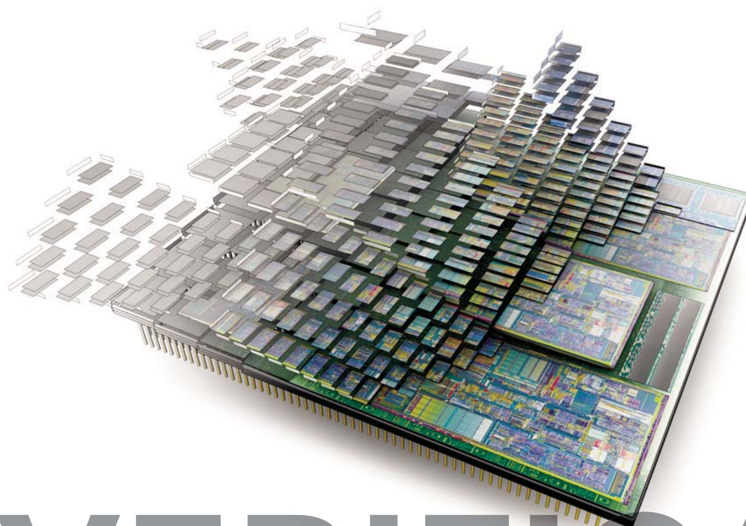
Mentor, A Siemens Business
Worldwide Headquarters
8005 SW Boeckman Rd.
Wilsonville, OR 97070-7777

Phone: 503-685-7000

To subscribe visit:
www.mentor.com/horizons

To view our blog visit:
VERIFICATIONHORIZONSBLOG.COM

Verification Horizons is a publication
of Mentor, A Siemens Business
©2017, All rights reserved.



VERIFICATION HORIZONS

VH