Make Your Constraints More Dynamic with Portable Stimulus

by Matthew Ballance – Mentor, A Siemens Business

INTRODUCTION

If you work in functional verification, you've likely become quite familiar with random constraints from functional verification languages such as SystemVerilog. Using a constraint solver to automate stimulus generation is key to quickly generating lots of stimulus that hits cases that weren't envisioned by the test writer. When using constrained-random generation, constraints are the mechanism by which we customize what is legal and interesting in the stimulus space.

Accellera's Portable Stimulus Standard (PSS) introduces some new constraint capabilities, in addition to supporting the capabilities that we've become familiar with in SystemVerilog. This article provides a guided tour of one of these new constraint features, along with examples that highlight their benefits.

CONSTRAINT FUNDAMENTALS

If you've used SystemVerilog, you're likely very familiar with the constraint construct; random constraints declared within a class along with random fields. When an instance of the class is randomized, the constraints limit the available range of values.

Let's say we are generating IPV4 traffic, and have a data structure that represents an IPV4 header. Figure 1 shows a SystemVerilog class and the corresponding PSS struct that we might use to represent this collection of random data.

Note that we also have a very basic constraint on the *length* field, since the total length of all packets must be at least 20 bytes. Also note just how similar the SystemVerilog and the PSS description of this IPV4 header is. So, if you can write SystemVerilog data structures and constraints, you can just as easily write PSS descriptions of data structures and constraints.

```
class ipv4_header;
  rand bit[7:0]  ihl;
  rand bit[7:0]  tos;
  rand bit[15:0] length;
  rand bit[15:0] identification;
  rand bit[15:0] flags_offset;
  rand bit[7:0]  ttl;
  rand bit[7:0]  protocol;
  rand bit[7:0]  options[8];

  constraint length_c {
    length >= 20;
  }
endclass
```

SystemVerilog

```
struct ipv4_header {
  rand bit[7:0]  ihl;
  rand bit[7:0]  tos;
  rand bit[15:0] length;
  rand bit[15:0] identification;
  rand bit[15:0] flags_offset;
  rand bit[7:0]  ttl;
  rand bit[7:0]  protocol;
  rand bit[7:0]  options[8];

  constraint length_c {
    length >= 20;
  }
}
```

Accellera PSS

Figure 1: IPV4 Header in SystemVerilog and PSS

CUSTOMIZING RANDOMIZATION

In both SystemVerilog and PSS, we can customize pure-data randomization in a couple of ways. The simplest way, of course, is to add more constraints. We can add more constraints by declaring a new data structure that inherits from the base data structure and adds more constraints, as shown in Figure 2.

Here again, both the constructs and syntax are remarkably similar between SystemVerilog and Accellera PSS.

Constraints can also be added 'in-line' when an instance of a data structure is randomized. When using a methodology such as UVM in SystemVerilog, randomization is likely to occur in a UVM sequence with the data structure instance subsequently being sent to the rest of the testbench. In a PSS model, actions roughly play the same role as a sequence, both selecting values for data structure fields and specifying what behavior in the environment should be performed.

In both cases, additional constraints can be added in-line with the call to

```
class ipv4_small_header extends   ipv4_header;

  constraint small_length_c {
    length <= 128;
  }

endclass
```

SystemVerilog

```
struct ipv4_small_header : ipv4_header {

  constraint small_length_c {
    length <= 128;
  }

}
```

Accellera PSS

Figure 2: Small IPV4 Header Data Structure

```
class send_small_ipv4_headers_seq extends uvm_sequence;

  virtual task body();
    ipv4_header hdr = ipv4_header::type_id::create();
    forever begin
      assert(hdr.randomize() with {length < 128; });
    end
    // . . .
  endtask
endclass
```

SystemVerilog

```
action create_small_ipv4_headers {
  action ipv4_header hdr;

  activity {
    repeat {
      hdr with  { length < 128; };
      // . . .
    }
  }
}
```

Accellera PSS

Figure 3: Adding In-Line Constraints

randomize values of the data structure fields. Figure 3 shows a UVM sequence and a PSS action that create a series of small IPV4 headers by adding an in-line constraint.

While in-line constraints are very handy, the fact that we've hard-coded values and relationships directly within the constraint blocks makes our SystemVerilog and PSS descriptions more brittle. What if my definition of a small header changes one day? I'll need to find any place in my testbench where I've added a constraint like this and update it. What if I want to constrain another field temporarily any time a small packet is being created? Here again, I would need to make updates across my entire test description.

WHAT ARE DYNAMIC CONSTRAINTS?

PSS adds a new construct called a dynamic constraint that is remarkably helpful in addressing the limitations of hard-coded inline constraints. The data structure-level constraints that we've looked at thus far in both SystemVerilog and PSS are considered static. Specifically, once a constraint is declared in a class or struct it is applied every time an instance of the class is randomized. Accellera PSS supports static constraints inside struct and action types, but also introduces a new type of constraint: a dynamic constraint. A dynamic constraint is almost exactly the mirror image of a static constraint. While a static constraint always applies, a dynamic constraint only applies once the user activates it. Initially, this might seem like a fairly useless construct. It's anything but!

In PSS, I might declare my IPV4 header struct anticipating that I would want to create some specific specializations of the struct. Figure 4 shows two dynamic constraints I might apply to enable creation of small headers and large headers.

```
struct ipv4_header {
  rand bit[7:0]   ihl;
  rand bit[7:0]   tos;
  rand bit[15:0]  length;
  rand bit[15:0]  identification;
  rand bit[15:0]  flags_offset;
  rand bit[7:0]   ttl;
  rand bit[7:0]   protocol;
  rand bit[7:0]   options[8];

  constraint length_c {
    length >= 20;
  }

  dynamic constraint small_header_c {
    length <= 128;
  }

  dynamic constraint large_header_c {
    length >= 32768;
  }
}
```

Figure 4: PSS IPV4 Header Struct with Dynamic Constraints

Note that these two constraints conflict. However, because dynamic constraints don't apply until the user activates them, that doesn't create any problems.

We can use a dynamic constraint like any other constraint expression, including inside an inline constraint. Figure 5 shows an updated version of my *create_small_ipv4_headers* action that uses the new dynamic constraint.

```
action create_small_ipv4_headers {
  action ipv4_header hdr;

  activity {
    repeat {
      hdr with  { small_header_c; };
      // . . .
    }
  }
}
```

Figure 5: Inline Randomization with a Dynamic Constraint

Simply by replacing a literal constraint (length <= 128) with a symbolic one (*small_header_c*) the code already conveys more of the author's intent. This description is also less brittle. If we decide that a small header needs to be defined differently, we can simply update the original dynamic constraint definition, and all uses of that constraint will automatically use the new definition.

As you can start to see, dynamic constraints allow a constraint API to be developed such that test writers can symbolically constrain objects instead of directly referring to fields and constant values.

COMPOSING DYNAMIC CONSTRAINTS

Dynamic constraints provide value beyond just making code easier to understand and easier to update. Dynamic constraints are boolean constraints, which means we can use them in a conditional constraint. The value of a dynamic constraint is 'true' if it is applied and 'false' if not. This property of dynamic constraints allows us to compose more-interesting relationships.

What if we wanted to generate a series of headers that were either *large* or *small*? Using the knowledge that dynamic constraints are boolean constraints, we can state our intent quite simply, as shown in Figure 6.

```
action create_small_or_large_ipv4_headers {
  action ipv4_header hdr;

  activity {
    repeat {
      hdr with  { small_header_c || large_header_c; };
      // . . .
    }
  }
}
```

Figure 6: Composing Inline Constraints with Dynamic Constraints

The use of dynamic constraints isn't limited to inline constraint blocks. We also can use them inside the data structures along static constraints to encapsulate common constraints and make our constraints more modular and easier to understand.

```
ruct ipv4_header {

// . . .

constraint length_c {
  length >= 20;
}

// High priority packets should aways be small
constraint small_high_priority_packets_c {
  if (dscp_cs2_c) {
    length <= 256;
  }
}

// . . .

dynamic constraint dscp_cs0_c {
  tos[5:0] == 0;
}

dynamic constraint dscp_cs1_c {
  tos[5:0] in [8, 10, 12, 14];
}

dynamic constraint dscp_cs2_c {
  tos[5:0] in [16, 18, 20, 22];
}
```

Figure 7: Using Dynamic Constraints Inside Static Constraints

Figure 7 shows an example of using dynamic constraints inside static constraints. For the purposes of this example, we have decided that, for our application, packets with immediate priority (DSCP level of CS1) must be less-equal to 256 bytes in size. Using dynamic constraints to associate a meaningful name with the constraint expression makes our code easier to read and maintain, just as it did in the case of inline constraints.

USING VIRTUAL DYNAMIC CONSTRAINTS

Dynamic constraints, just like static constraints, are virtual. This means that we can change the meaning of a dynamic constraint (and, thus, the generated stimulus) using inheritance and factory-style type overrides.

What if we wanted to run a set of tests in which the definition of a small header is different from the default definition? Clearly it's undesirable to actually modify the test scenarios themselves. Using dynamic constraints, and the fact that they are virtual, allows us to define a new struct where the definition of a small header is different, as shown in Figure 8.

```
struct ipv4_header_larger_small_headers : ipv4_header {
  // Change the definition of small headers
  dynamic constraint small_header_c {
    length <= 256;
  }
}
```

Figure 8: Overriding a Dynamic Constraint

We create a new header struct that inherits from the existing *ipv4_header* struct and create a new definition of the *small_header_c* constraint. Just as with a static constraint, this version of the constraint will be used for all instances of the *ipv4_header_larger_small_headers* struct. But, how do we cause this struct to be used instead of the *ipv4_header* struct that is used in our test scenario (Figure 9).

```
component packet_gen_c {

  action create_small_ipv4_headers {
    action ipv4_header hdr;

    activity {
      repeat {
        hdr with  { small_header_c; };
        // . . .
      }
    }
  }
}
```

Figure 9: Small Headers Scenario

Accellera PSS provides us with a very useful notion of 'override', which is effectively a UVM Factory built into the language. Just like the UVM Factory, the PSS override construct provides a way to replace instances of a given type with another derived type. The PSS type extension construct provides an easy way to inject these overrides without modifying the original scenario.

Figure 10 shows how type extension and the override construct are combined to cause the *ipv4_header_larger_small_headers* to be used by our *create_small_ipv4_headers* scenarios. This will cause our scenario to use the new definition of a small header without us needing to modify any code.

```
extend component packet_gen_c {
  override {
    type ipv4_header with ipv4_header_larger_small_headers;
  }
}
```

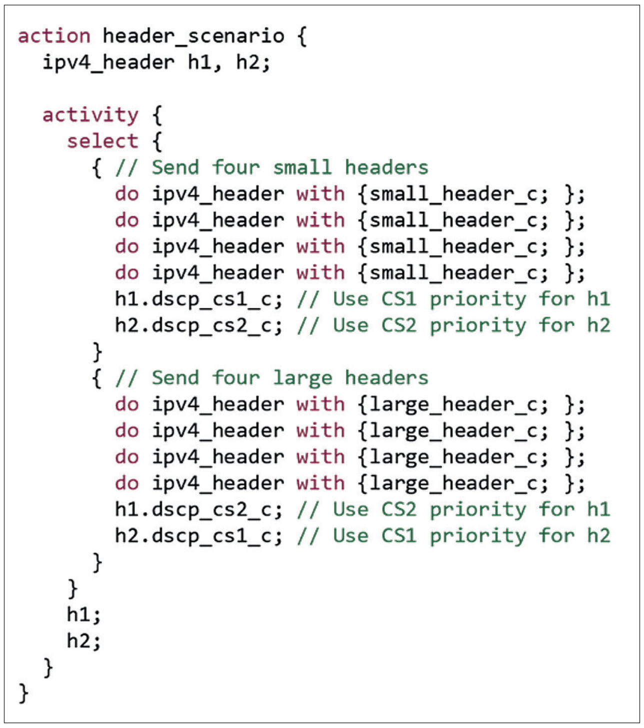
Figure 10 : Injecting an Override Statement

USING DYNAMIC CONSTRAINTS WITH ACTIVITIES

Thus far, we've focused on applications for dynamic constraints that are fairly data-centric and restricted to a single data structure. These capabilities of dynamic constraints only increase when applied in the context of a PSS activity. If you've attended or watched one of the

Accellera PSS tutorials, you've learned a bit about Activities. An activity is a declaratively-defined behavior that can be statically analyzed. An activity is closer to a set of random variables and constraints than it is to imperative code in SystemVerilog. Dynamic constraints effectively enable functional programming within an activity.

In SystemVerilog, we can only pass values between calls to randomize. For example, without a UVM sequence we could select a header size to be small, medium, or large, then constrain the packet size to this pre-selected size. However, the only way to pass forward the notion that a future header should be 'small' independent of a specific value is to add more variables to encode that intent. Dynamic constraints provide exactly this capability in Accellera PSS.



```
action header_scenario {
  ipv4_header h1, h2;

  activity {
    select {
      { // Send four small headers
        do ipv4_header with {small_header_c; };
        do ipv4_header with {small_header_c; };
        do ipv4_header with {small_header_c; };
        do ipv4_header with {small_header_c; };
        h1.dscp_cs1_c; // Use CS1 priority for h1
        h2.dscp_cs2_c; // Use CS2 priority for h2
      }
      { // Send four large headers
        do ipv4_header with {large_header_c; };
        do ipv4_header with {large_header_c; };
        do ipv4_header with {large_header_c; };
        do ipv4_header with {large_header_c; };
        h1.dscp_cs2_c; // Use CS2 priority for h1
        h2.dscp_cs1_c; // Use CS1 priority for h2
      }
    }
    h1;
    h2;
  }
}
```

Figure 11: Using Dynamic Constraints in an Activity

Figure 11 shows the use of dynamic constraints in an activity. In this case, the select statement chooses between sending four small-header packets and sending four large-header packets. Then, two normally-constrained headers are sent. We use dynamic constraints to easily control these two final headers based on the select branch taken. If we select the branch that sends the four small headers, we cause h1 to be sent with priority CS1 and cause h2 to be sent with priority CS2.

CONCLUSION

As you've hopefully seen from the preceding article, dynamic constraints provide significant new capabilities above and beyond those provided by the constraints that we've become familiar with. Dynamic constraints are present in all versions of the Accellera Portable Stimulus Standard, and are supported by Mentor's Questa® inFact portable stimulus tool.

Dynamic constraints are yet another example of how Accellera's Portable Stimulus Standard is enabling greater abstraction and productivity in capturing test intent, in addition to enabling that test intent to easily be made portable across a variety of target platforms. If you're interested in contributing to the evolution of features for productively capturing test intent, I'd encourage you to get involved with the Accellera Portable Stimulus Working Group!

VERIFICATION ACADEMY

The Most Comprehensive Resource for Verification Training

31 Video Courses Available Covering

- UVM Debug
- Portable Stimulus Basics
- SystemVerilog OOP
- Formal Verification
- 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

UVM

Mentor®
A Siemens Business
www.mentor.com

Editor:
Tom Fitzpatrick

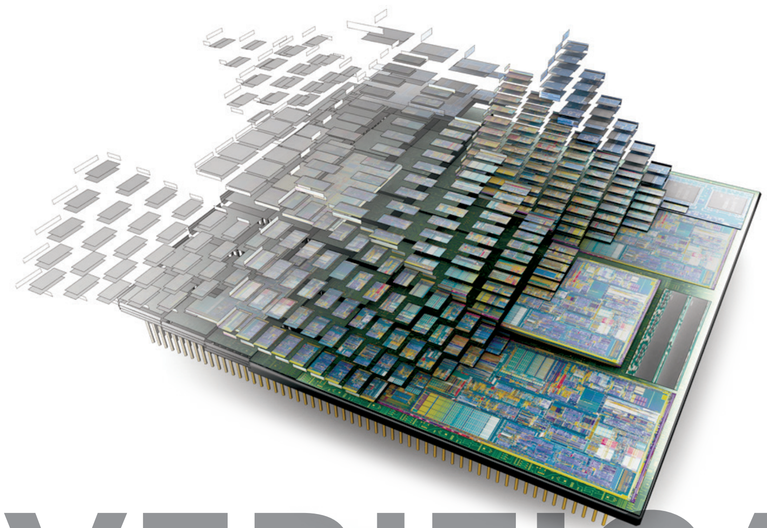
Program Manager:
Rebecca Granquist

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

VH