To Infinity And Beyond - Streaming Data Sequences in UVM

Mark Litterick, Jeff Vance, Jeff Montesano
Verilab GmbH[1], Munich, Germany. Verilab Inc[1], Austin, TX, USA.
Email: mark.litterick@verilab.com, jeff.vance@verilab.com, jeff.montesano@verilab.com

***Abstract*- This paper introduces the concept of autonomous stimulus generation using streaming data techniques and discusses its application in the verification of complex sensor style devices. With a focus on digital simulation of SoCs containing real-number models for the analog sub-components, we demonstrate how to implement autonomous analog and digital data streaming patterns using UVM sequences and drivers.**

I. INTRODUCTION

Controlling continuous streams of stimulus values over periods of time is often required in design verification. Examples include streaming data patterns to digital interfaces and real number stimulus for analog models (e.g. antenna receivers, internal sensors or analog-to-digital converters). In these cases we need fine grained control over the input patterns, a flexible and powerful user interface, and sufficient encapsulation to ensure straightforward scenario development for the user. In this paper we show how to generate autonomous data stream stimulus using SystemVerilog [1] in the context of a UVM [2] testbench, while providing the user with a sequence API consistent with standard UVM practice. In addition, we show how this approach works with designs that require real number streaming, optional forcing of internal nets, and file-based input for real-life application-specific data streams.

A. What Are Streaming Data Sequences

In the context of this paper we use the term *streaming* to identify a stimulus pattern where a single sequence item is used to define repetitive and potentially infinite streams of data values. The corresponding driver operation is *autonomous* since the derived low-level stimulus values are generated independently and driven indefinitely without further intervention from the sequence layers. The test sequences can interrupt and redirect the ongoing driver operation at any time, just by sending a new sequence item that describes the subsequent stimulus. The sequences typically do not need to know exactly where the driver is in the pattern, and complicated handshakes and synchronization demands are not normally required.

Within a UVM testbench, the sequence layers generate an item using standard constrained-random techniques and pass the item to the driver which generates the actual derived stimulus data patterns on it's own, based completely on the sequence item content and if necessary relevant configuration fields. Key characteristics of effective streaming include:

- sequences fully control the autonomous behavior
- sequence-driver handshake must be non-blocking
- operation can be interrupted by a new request

This paper demonstrates how to implement streaming data stimulus in a manner that is compliant with standard UVM practices, even though some of the mechanisms behind the scenes have changed. This allows test developers to easily execute complex real-number analog and digital data patterns such as those shown in Figure 1, by executing a single sequence call for each.

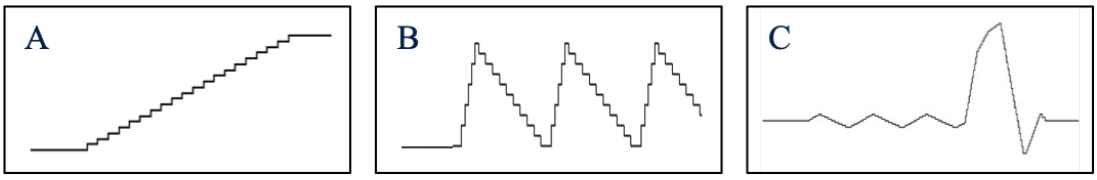

Figure 1 - Example Streaming Waveforms

[1] www.verilab.com

B. *Primary Applications*

Despite the apparent focus on efficient analog stimulus development, the techniques presented in this paper are primarily targeted at *digital simulation* of complex sensors that contain real-number models of the analog sub-components. Such System-on-Chip (SoC) devices typically contain external analog inputs (e.g. radio-frequency or microwave antennas, optical receivers) or embedded sensors (e.g. temperature, pressure, acceleration, current and voltage sensors). Figure 2 shows a typical block diagram of such a sensor SoC.

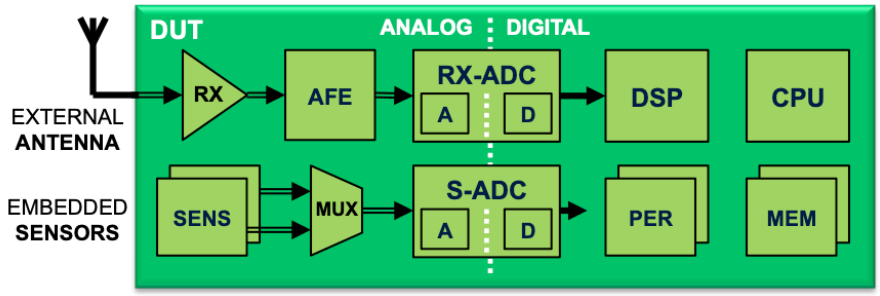

Figure 2 - Generic Complex Sensor SoC

II. Streaming Data Stimulus

The streaming data stimulus techniques presented in this paper are extensions of the standard UVM constrained-random sequence-based stimulus paradigm discussed in [3][4], and not an alternative bus-functional-model architecture. Our goal is to preserve all the higher-level testbench and test sequence layers while providing an autonomous data streaming solution that fits nicely with the stimulus structure of a standard UVC and its enclosing environment as shown in Figure 3.

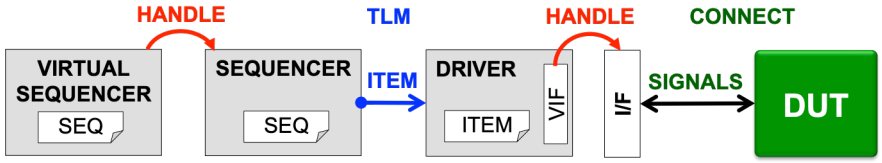

Figure 3 - Standard UVM Stimulus Architecture

For effective data streaming it is essential that the driver runs autonomously in the background and that the operation can be interrupted at any time with a new request with alternative stimulus. This capability allows the user to easily compose complex stimulus patterns and also to regain control over the autonomous driver whenever it is required. In order to achieve this in UVM we need to modify the sequencer-driver handshake mechanism to be non-blocking.

The normal blocking handshake in UVM can be represented by the following code snippet from the driver *run_phase* task:

```
forever begin
  seq_item_port.get_next_item(item);
  drive_item(item); // blocking drive
  seq_item_port.item_done();
end
```

The modified non-blocking sequencer-driver handshake is shown in the following code:

```
forever begin
  seq_item_port.get_next_item(item);
  seq_item_port.item_done(); // pass control back to sequencer
  fork
    drive_item(item); // blocking drive (possibly forever)
    seq_item_port.peek(); // blocking peek -> wait for new item
  join_any
  disable fork;
end
```

Notice that *item_done* is now called before *drive_item* in order to return control to the sequence layer. In addition the *drive_item* task will be aborted due to the *disable fork* statement if a new sequence item appears on the TLM port while the task is still running. This non-blocking handshake mechanism allows multiple sequences to be combined in order to provide the required stimulus patterns without the need for complicated fork-join loops or flow control in the higher-level sequences layers.

Having established the basic technique for interacting with the driver from the sequence layer, we can now look in detail at how this is used to develop real-number analog stimulus, low-level digital data streams and file-based stimulus for applications with alternative data sources.

C. Real Number Analog Stimulus

Our primary goal for the real-number analog stimulus is to define and execute stimulus patterns such as those shown in Figure 4, each with a single sequence call (i.e. *`uvm_do* or *seq.start*) for patterns A and B, or several consecutive separated sequence calls in the case of pattern C.

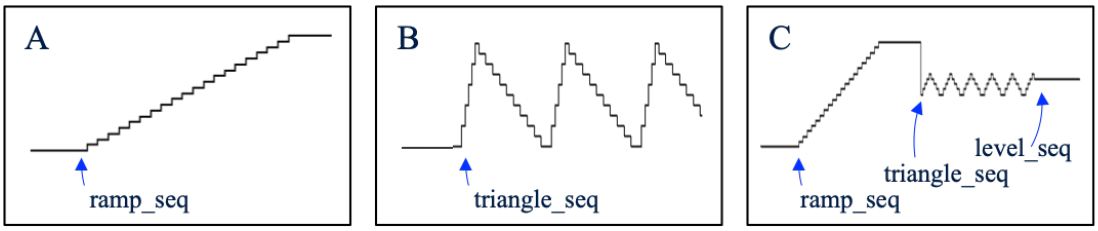

Figure 4 - Example Real-Number Analog Stimulus Patterns

It is essential that the sequences describe the stimulus in a way that fully controls the analog pattern generation. However, the derived fine-grain stimulus values for each consecutive analog step could be calculated within the sequences themselves, or delegated to the driver. In this paper we demonstrate the mechanism where the driver calculates the derived values autonomously, since we have found this an effective solution that allows the sequence layer to focus on higher-level scenarios without being burdened by low-level details, and it provides better performance in applications with very high-speed stimulus or where the number of derived values is very large. Note that it is still possible to benefit from some aspects of the autonomous driving capability, even if you prefer to do all the calculations for derived values in the sequence layer, for example the ability to interrupt a never-ending pattern with alternative stimulus.

In order to illustrate the mechanism, we will first consider a simple sequence describing a ramp and demonstrate how the driver can derive the low-level data values and drive the corresponding signal interface. Then we extend the concept such that the driver can generate more complex, repetitive and infinite patterns. Note that the enhanced driver provides all the required driving capability without the need for class extensions or overrides. The basic ramp can be described as shown in the following sequence:

```
class ramp_seq extends uvm_sequence #(analog_seq_item);
  rand real start; // start value
  rand real stop;  // stop value
  rand real step;  // step size
  rand time rate;  // duration of each step
  ...
    `uvm_field_real(step, UVM_ALL_ON | UVM_REAL_DEC)
    `uvm_field_int(rate, UVM_ALL_ON | UVM_TIME)
  ...
  constraint default_c {
    soft start inside {[-1.5:1.5]};   soft stop inside  {[-1.5:1.5]};
    soft step  inside {-0.01,0.01};   soft rate inside  {[1ps:10ns]};
  }
  constraint ramp_c {
    (step == 0.0) -> start == stop; // level
    (step > 0.0)  -> stop > start;  // ramp up
    (step < 0.0)  -> stop < start;  // ramp down
  }
  ...
  task body();
    `uvm_do_with(req, {m_start==start; m_stop==stop; m_step==step; m_rate==rate;})
```

Notice that we are using control knobs of type *real* and *time*, with corresponding *field util* macros and constraints[2]. In order to support this the basic sequence item would provide similar fields as shown below:

```
class analog_seq_item extends uvm_sequence_item;
  rand real m_start; // start value
  rand real m_stop;  // stop value
  rand real m_step;  // step size
  rand time m_rate;  // duration of each step
```

The driver receives this item from the sequencer and can generate all the intermediate incremental steps for the ramp based on item fields as shown in the following code:

```
task analog_driver::drive_item(analog_seq_item item);
  real value = item.m_start;
  forever begin
    drive_value(value);   // drive signals
    #(item.m_rate);       // wait for step delay
    value += item.m_step; // increment or decrement
    if (item.m_step == 0.0) break;                              // level
    if ((item.m_step > 0.0) && (value >= item.m_stop)) break; // up
    if ((item.m_step < 0.0) && (value <= item.m_stop)) break; // down
  end
```

For analog signals we have an added complication of whether the value should be driven differentially (relative to a common mode reference) or single-ended (relative to ground). We recommend using single-value *real* numbers for all sequence layers including the sequence item fields, and controlling the driving mode using a configuration field. The driver can then take the configured mode into account in the final *drive_value* method as shown below (where the corresponding interface signals are also of type *real*):

```
function void analog_driver:: drive_value(real value);
  if (cfg.m_differential) begin
    vif.data_p <= vif.cm_ref + value/2;
    vif.data_n <= vif.cm_ref - value/2;
  end else begin // single-ended
    vif.data_p <= value;
    vif.data_n <= 0.0;
  end
```

Having described the basic mechanism for autonomously driving the signal interfaces based on a simple ramp description, we will now consider how to extend the concept to handle more complex patterns and repetition. First we replace the original analog sequence item with the following structure:

```
class analog_seq_item extends uvm_sequence_item;
  rand int unsigned   m_repetition; // repetition (0=infinite)
  rand analog_shape   m_shape;      // shape kind (LINEAR,...)
  rand analog_segment m_segment[];  // segment descriptor
```

Where the *analog_shape* enumerated type describes the stimulus pattern and the *analog_segment* descriptor array provides details of each segment in the shape. For a shape composed of *linear* segments, the descriptor object fields would be similar to the previous ramp for each segment as shown:

```
class analog_segment extends uvm_object;
  rand real m_start; // start value
  rand real m_stop;  // stop value
  rand real m_step;  // step size
  rand time m_rate;  // duration of each step
  rand time m_pause; // duration after segment completion
```

[2] If the *real* number UVM extensions are not supported by your simulation tool or license variant, then an alternative solution is to work with integer values in the sequence layers and provide a configuration field for the driver precision, e.g.: `vif.data_p <= value * cfg.m_precision; // m_precision = 0.001`

The new driver can be designed to iterate on the segment structure for the required number of repetitions (which can be infinite), unless of course a new sequence item is received in the meantime. Now the driver has a lot of autonomous work to do, but it is still fully under the control of the received sequence item as shown in the following code (note that saturation at stop value is not shown due to space limitations):

```
task analog_driver::drive_item(analog_seq_item item);
  int count = 0;
  real value;
  do begin
    case (item.m_shape)
      LINEAR: foreach (item.m_segment[i]) begin
        value = item.m_segment[i].m_start;
        forever begin
          drive_value(value);                // drive signals
          #(item.m_segment[i].m_rate);       // wait for step delay
          value += item.m_segment[i].m_step; // increment or decrement
          if (item.m_segment[i].m_step == 0.0) break;           // level
          if ((...m_step > 0.0) && (value >= ...m_stop)) break; // up
          if ((...m_step < 0.0) && (value <= ...m_stop)) break; // down
        end
        // if value>stop (up) or value<stop (down) drive saturation value=stop
        #(item.m_segment[i].m_pause); // wait for segment delay
      end
      ... // potentially other shape patterns
    endcase
    count++;
  end
  while ((count < item.m_repetition) || (item.m_repetition == 0));
```

Note that the shape kind can be enhanced to include non-linear patterns such as *SINUSOID*; in this case the descriptor fields would include (or be interpreted as) amplitude, phase angle and angular step. The corresponding SystemVerilog math functions such as *$sin* could then be used in the driver implementation.

With all the hard work being off-loaded to the driver, we can now provide the user with a comprehensive set of sequences, such as those shown below:

```
class analog_level_seq;     // set specified level
class analog_ramp_seq;      // ramp and stop
class analog_sawtooth_seq;  // sawtooth (ramp and repeat)
class analog_triangle_seq;  // triangle (ramp up/down and repeat)
class analog_trapezoid_seq; // trapezoid (ramp up/down with pauses)
class analog_sinusoid_seq;  // sinusoid (sine and repeat)
```

These sequences encapsulate any low-level conversion to the slightly more complex sequence item fields, in order to provide the user with a nice clean API, as shown in the following code example for a repetitive triangle sequence (other sequences are not shown due to space limitations):

```
class analog_triangle_seq extends uvm_sequence #(analog_seq_item);
  rand real hi_value;           // high value
  rand real lo_value;           // low value
  rand real step_up;            // step for up ramp
  rand real step_down;          // step for down ramp
  rand time rate_up;            // duration of step for up ramp
  rand time rate_down;          // duration of step for down ramp
  rand int unsigned repetition; // repetition (0=infinite)
  ...
  virtual task body();
    `uvm_do_with(req, {
      m_shape == LINEAR;  m_repetition == repetition; m_segment.size() == 2;
      m_segment[0].m_start == lo_value;   m_segment[1].m_start == hi_value;
      m_segment[0].m_stop  == hi_value;   m_segment[1].m_stop  == lo_value;
      m_segment[0].m_step  == step_up;    m_segment[1].m_step  == step_down;
      m_segment[0].m_rate  == rate_up;    m_segment[1].m_rate  == rate_down;
      m_segment[0].m_pause == 0ps;        m_segment[1].m_pause == 0ps;
    })
```

The following code example demonstrates how we can call a few of these sequences from an enclosing test sequence in order to generate the complex real-number analog pattern shown previously in Figure 4-C. Note that in this example we fully specify the triangle and level sequences, but only partially specify the ramp sequence allowing constrained random values to be selected for the other fields.

```
class example_test_seq extends uvm_sequence;
  ...
  virtual task body();
    `uvm_do_with(ramp_seq, {start==0.0; stop==1.5;})
    ...  // explicit delay or other stimulus
    `uvm_do_with(triangle_seq, {repetition==0; // repeat indefinitely
       step_up==0.01;   rate_up==20ps;   hi_value==0.85;
       step_down==0.02; rate_down==15ps; lo_value==0.75;})
    ...  // explicit delay or other stimulus (then abort ongoing triangle seq)
    `uvm_do_with(level_seq, {value==0.25;})
```

D. Digital Stimulus Patterns

In some applications the autonomous data streaming mechanism can also be used to generate low-level digital stimulus patterns. Note that this is in addition to any high-level transaction-based digital streaming protocol that could be running at a sequence level to general digital packets and frames. For many applications transaction-based streaming is also required, but the topic is outside the scope of this paper.

For the generic complex sensor SoC presented here, one obvious use-case is to generate data for the internal ADC outputs using the driving mechanism described in Section III. This data can be regular shape patterns similar to those already discussed for the analog stimulus (i.e. ramp, triangle, sinusoid, etc.), or application specific data extracted from files as discussed in Section II-E. The handshake mechanism between sequencer and driver is the same as for the analog agent and the sequence layers are constructed in a similar manner. For the digital interface however, the stimulus must typically be driven in response to internal control signals, for example the ADC end-of-conversion signal, as shown in the following code:

```
task digital_driver::drive_item(digital_seq_item item);
  int count = 0;  bit[11:0] value;
  do begin
    foreach (item.m_segment[i]) begin
      forever begin
        ... // calculate or get value
        wait (vif.start_of_conversion);
        @(posedge vif.end_of_conversion);
        vif.drive_result <= value;
        ... // break on segment end
      end
    end
    count++;
  end
  while ((count < item.m_repetition) || (item.m_repetition == 0));
```

E. File-Based Stimulus

It is often useful to supplement the pattern sequences with the capability to stream input data values read from a file. This is particularly useful for applications with a lot of analog or digital signal processing. The data could be captured from the real-world application or generated by an appropriate mathematical tool. Several implementations are possible here, but we will demonstrate a simple autonomous driver-based solution using the established framework, which allows us to drive complex application-specific data pattern such as that shown in Figure 1-C, from within the UVM sequence setup.

In this case we can extend the shape kind enumerated type to include a *FILE* literal and add an additional *analog_file_seq* sequence to our portfolio. Once again we have an option of generating the low-level data values (in this case by reading from a file) in either the sequence or the driver, however only the driver-based solution provides the option to read one value at a time, which can result in better performance if the data files are very large. Since the file-name string cannot be random, it is more flexible to define this via the configuration object using straightforward helper methods (this allows the sequence to be called with *uvm_do* or *start*). Note that the sequence API is similar to the pattern sequences: for the analog sequence we just supply repetition and rate, for the ADC digital sequence the rate is not required since the data is streamed reactively on end-of-conversion as discussed in Section II-D.

```
class analog_file_seq extends uvm_sequence #(analog_seq_item);
  rand int unsigned repetition; // repetition (0=infinite)
  rand time         rate;       // duration for each data value
  ...
    `uvm_do_with(req, {
      m_shape == FILE;        m_repetition == repetition;
      m_segment.size() == 1;  m_segment[0].m_rate == rate;
    })
```

The *case* statement in the *drive_item* task shown previously can be enhanced to read the required data values from the comma-separated-value file and drive them at the required rate, as shown below. Note this also supports finite and infinite repetition as well as abort when a new sequence item disables the task.

```
case (item.m_shape)
  ... // LINEAR: ... etc.
  FILE: begin
    string separator;
    int fd = $fopen(cfg.m_filename,"r");
    if (fd == 0) begin
      `uvm_fatal(...,$sformatf("file not found (%s)",cfg.m_filename))
    end else begin
      while (!$feof(fd)) begin
        if ($fscanf(fd,"%f",value) == 1) begin // extract real-number value
          drive_value(value);                  // drive signals
          #(item.m_segment[0].m_rate);         // delay for each value
        end
        if ($fscanf(fd,"%s",separator) == 0)   // read separator characters
          break;                               // break on white-space
      end
      $fclose(fd);
    end
  end
endcase
```

The corresponding test sequence only has to set the filename for the comma-separated-value data file, and call the file sequence with the required repetition and rate, then the driver will process the file content and drive the values until the pattern is complete or interrupted by alternative stimulus.

```
class example_test_seq extends uvm_sequence;
  ...
  cfg.set_analog_filename("path_to_file/analog_data.csv");
  `uvm_do_with(file_seq, {repetition==1; rate==50ns;})
  ... // explicit delay or other stimulus
```

This technique can also be used to extract data from an external model (e.g. *C*, *MatLab* or *Python*) via *DPI* function calls. In this case the data could be provided on-the-fly rather than saved in a pre-generated file, which would allow for adaptive data generation and avoid maintenance of many data files. To implement this we would add a *DPI* literal to our *shape* enumerated type and add a corresponding *analog_dpi_seq* sequence for the user. In the driver we would modify the case statement in the *drive_item* task to call the *DPI* function when new data was required (not shown due to space limitations).

III. Internal Driving

An effective verification technique that we recommend for complex SoC devices with a significant amount of analog model content is to provide the user with the capability to optionally drive internal nets in the design, as well as the ability to stimulate the full path with all models integrated into the top-level SoC. This enables tests to be developed independently from the analog models: initial tests can be developed prior to analog model availability, and a sub-set of tests can retain independence throughout the lifetime of the project. Furthermore these tests have more control of the internal analog and digital data paths without compromising the essential full path verification goals. Since the analog models are often fully integrated only at the SoC level, this means we have a multi-purpose SoC level testbench with stimulus connections to the external ports, primary sensors and internal analog and ADC layers as shown in Figure 5.

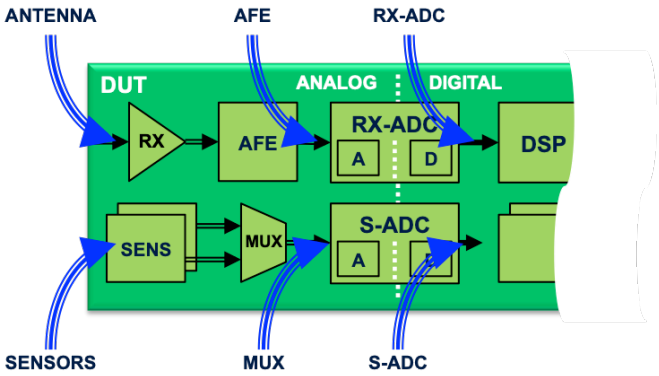

Figure 5 - Internal Driving of Analog and ADC Paths

Since the same testbench is used for both full-path (e.g. external *ANTENNA* and embedded *SENSORS*) and internal stimulus (e.g. *AFE*, *MUX, RX-DIG* or *SADC-DIG*) scenarios, it is important that we do not interfere with the full-path operation of the device when we are *not* running an active sequence on an agent that targets an internal signal. Note however that we do not require a dynamic mechanism to release the internal nets after sequence-based driving, since we are not using this setup to dynamically switch between full-path and internal driving - instead we are providing alternative stimulus paths for the test scenarios running on the same underlying environment. With this simplification in mind, a straightforward mechanism for optionally forcing the internal nets can be achieved by using *always* blocks sensitive to interface signal changes in the top-level testbench module connections, and drive the internal signals using a *SystemVerilog force*[3] procedural continuous assignment statement as shown below. Note that if a sequence is not executed, the driver will not update the corresponding interface signal, and the *always* block will not *force* the internal net. This code could also be absorbed into the interface file if preferred.

```
module testbench();
  ...
  always @(analog_if.data_p)
    force path.to.internal.net.p = analog_if.data_p;
```

IV. Conclusion

This paper presented the concept of using sequences to control autonomous analog and digital data streaming as an elegant solution to a specific set of verification challenges that are common in digital simulation of complex sensor SoC devices. The sequence-based techniques demonstrated have evolved over time during the verification of many diverse applications for different clients, and represent a pragmatic and scalable solution that is consistent with standard UVM practices.

Although the focus for this paper was digital simulation of SoC devices with mixed analog and digital content, the real-number stimulus techniques are directly transferrable to analog-mixed-signal (AMS) simulation using the UVM, and data streaming in general can be valuable for pure digital verification challenges. The solutions demonstrated in the paper are easy to implement, easy to use and portable - allowing the verification team to focus on developing interesting and relevant stimulus scenarios that expose more bugs with less effort.

References

[1] SystemVerilog, *IEEE Std 1800-2012*
[2] Accelera, *Universal Verification Methodology*, v1.2
[3] Vance, Montesano and Litterick, *Use the Sequence, Luke - Guidelines to Reach the Full Potential of UVM Sequences*, SNUG Austin 2018
[4] Litterick, Vance and Montesano, *Be a Sequence Pro to Avoid Bad Con Sequences*, DVCon EU 2019, Tutorial

[3] If your simulation tool does not support *force* for the target net type, or if the net is in a *VHDL* block, then you may have to implement the force using vendor-specific system functions, such as *$hdl_xmr_force* (*Synopsys*), *$signal_force* (*Mentor*) or *$xm_force* (*Cadence)*, e.g.:

```
$xm_force("path.to.internal.net.p", $sformatf("%g", analog_if.data_p));
```