Automating sequence creation from a Microarchitecture specification

Subramoni Parameswaran, Ravi Ram

Xilinx, Inc.

Ph: 408 879 2702

2101 Logic Drive

San Jose CA 95124

Email: subramo@xilinx.com, rram@xilinx.com

Abstract – Specification changes and bugs discovered late in the project tend to cluster around the specific timing of important signals and require significant modifications of all underlying sequences, tests and testbench code. These disruptive, late-stage changes can add significant delays, especially when the signal relationships and timing are coded into an external vendor VIP. To address these issues, we consider some approaches and describe a UVM configuration database solution for architecting sequence class creation at the very inception of the VIP/project to enable coverage driven and test driven automated stimulus generation and coverage, eliminating manual and error-prone modifications to the underlying sequences during the course of the project.

I. INTRODUCTION

One of the most common challenges in verification both at the block level and full chip level is the creation of sequences that can drive the input signals to the DUT in accordance with the microarchitecture specification. This effort is made harder because of a few practical issues:

1. The microarchitecture specification is often incomplete or incorrect with regards to specific signal behavior, especially at the beginning of the project when the testbench code is usually created. Architects and Designers usually do not describe the specific boundaries of signal timing as part of the architectural specification (as in the case of approximately timed or loosely timed models) in terms of minimum and maximum delays for various signal transitions
2. The desired behavior/timing of generated signals often changes at later stages in the project because of the discovery of bugs found later in the project, especially with timing analysis of the chip
3. The discovery of coverage holes during coverage analysis necessitates "tweaking" the sequences to hit corner cases manually, which is an error prone effort that can lead to further coverage holes because sequences are shared by multiple tests

As a result, the sequences that generate input signals end up changing constantly throughout the course of the project. Worse, the modification of the sequences is an error prone, iterative process which often introduces other problems. Furthermore, everyone on the project uses their own coding guidelines and coding styles for generating stimulus in their sequences, which makes code difficult to share and maintain.

How do we minimize sequence creation while enabling targeted random tests that improve coverage? How do we drive input signals in a way that allows for future changes without modifications to the underlying sequence code? How do we standardize the stimulus creation of critical signals to ensure that we can drive to full coverage closure in a quick manner?

II. PREREQUISITES

In order to get the most out of this paper, you should already be familiar with basic Object Oriented Programming (OOP) concepts and the UVM configuration database. Using basic OOP concepts and the configuration database, it is possible to define signal configuration classes that can be adaptively used by the test writer to control stimulus generation.

III. THE INITIAL MICROARCHITECTURE SPECIFICATION

Assume that we are interested in creating a sequence that generates a single read request, which requires the *enable1*, *enable2*, *read_addr* and *read_req* to be driven in that order to the DUT (Figure 1). In other words, the initial specification calls for *enable1* to be high at or before *enable2*, *enable2* to be high at or before *read_addr* and *read_addr* to be high at or before *read_req*.

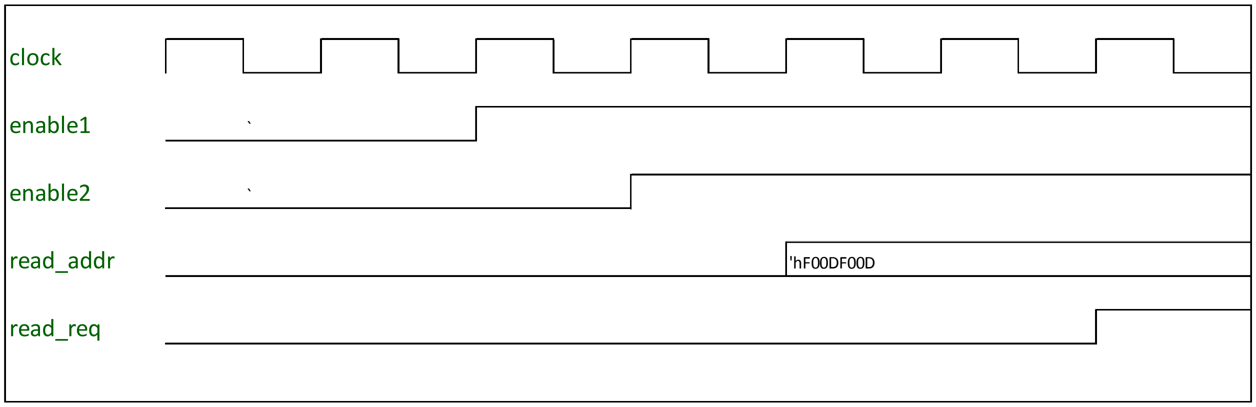

Figure 1 - Initial Microarchitecture Specification

IV. SOLUTION 1 – HARDCODING DELAYS INTO THE SEQUENCES

There are multiple possible approaches to driving the signals shown above. One approach is to send a single transaction from the sequence to the driver and have the driver manage the timing of all signal transitions for a read request. Another approach is to have the sequence do the heavy lifting by defining the timing and transition behavior of the signals in the sequence. We show the latter approach, but the solutions proposed in this paper are applicable regardless of the approach chosen.

A first cut attempt at coding the read sequence might look like Figure 2.

```
class read_seq extends uvm_sequence #(read_txfer);
…………………
virtual task body():
  `uvm_do_with(req, {enable1 == 1'b1;});
  `uvm_do_with(req, {enable1 == 1'b1; enable2 == 1'b1;});
  repeat (2) `uvm_do_with(req, {enable1 == 1'b1; enable2 == 1'b1; read_addr == 'hF00DF00D;});
  `uvm_do_with(req, {enable1 == 1'b1; enable2 == 1'b1; read_addr == 'hF00DF00D; read_req == 1'b1;});
endtask
```

Figure 2 - Sequence with hardcoded delays

The problem here is that the specific timing relationships between *enable1*, *enable2*, *read_addr* and *read_req* have been hard coded in the sequence. In the snippet above, *enable1* is high for one cycle before *enable2* and *enable2* is high for one cycle before *read_addr*, which in turn is high for 2 cycles before *read_req*. There is no randomization for the delays between *enable1*, *enable2*, *read_addr* and *read_req*.

If a bug were discovered that required *enable1* and *enable2* to be high in the same cycle, a new sequence (Figure 3) would have to be created to ensure *enable1* and *enable2* went high at the same time.

```
class read2_seq extends uvm_sequence #(read_txfer);
…………………
virtual task body():
  repeat (2) `uvm_do_with(req, {enable1 == 1'b1; enable2 == 1'b1;});
  repeat (2) `uvm_do_with(req, {enable1 == 1'b1; enable2 == 1'b1; read_addr == 'hF00DF00D;});
  `uvm_do_with(req, {enable1 == 1'b1; enable2 == 1'b1; read_addr == 'hF00DF00D; read_req == 1'b1;});
endtask
…………………

endclass
```

Figure 3 - Second sequence with hardcoded delays

But the problem with this sequence in turn is that it might not cover bugs in other critical scenarios, for example where *enable1*, *enable2* and *read_addr* all simultaneously go high in the same cycle.

V. SOLUTION 2 – IMPLEMENTING RANDOMIZATION IN THE SEQUENCE

A better way would be to build randomization into the sequence where the delays between the various signals are randomized within minimum and maximum parameter bounds. This involves defining the cycle delay parameter bounds for the signal pairs within the sequence (e.g. minimum and maximum cycles delay between *enable1*, *enable2*) and creating a constrained random variable which is randomized within the parameter bounds. The sequence would then use the constrained random variable to determine how many cycles to wait after *enable1* before driving *enable2*.

Consider these parameter bounds overlaid on the microarchitecture spec in Figure 4 and the corresponding sequence in Figure 5.

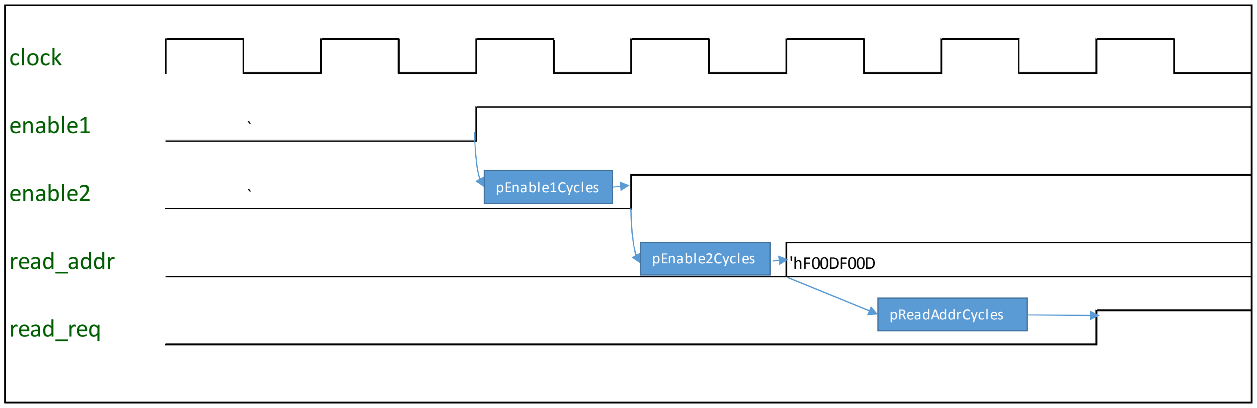

Figure 4 - Microarchitecture specification with delay parameters

```
class read_seq extends uvm_sequence #(read_txfer);
…………………
rand uint pEnable1Cycles;                              // Specific number of cycles where only enable1 will be high
uint pEnable1CyclesMin = 0, pEnable1CyclesMax = 13;    // Range of cycles where only enable1 can be high
…………………
constraint cEnable1Cycles {
  pEnable1Cycles >= pEnable1CyclesMin;
  pEnable1Cycles <= pEnable1CyclesMax;
}
// All other parameters like pEnable2Cycles, pReadAddrCycles etc will have similar definitions/constraints

virtual task body():
  repeat(pEnable1Cycles)     `uvm_do_with(req, {enable1 == 1'b1;});
  repeat(pEnable2Cycles)     `uvm_do_with(req, {enable1 == 1'b1; enable2 == 1'b1;});
  repeat(pReadAddrCycles) `uvm_do_with(req, {enable1 == 1'b1; enable2 == 1'b1; read_addr == 'hF00DF00D;});
  `uvm_do_with(req, {enable1 == 1'b1; enable2 == 1'b1; read_addr == 'hF00DF00D; read_req == 1'b1;});
endtask
…………………
endclass
```

Figure 5 - Sequence with built in random delays

The delay parameters such as ***pEnable1CyclesMin*** and ***pEnable1CyclesMax*** are present within the sequence class itself and the sequence class uses these parameters to generate the specific delay values within the sequence. This can be shown pictorially as Figure 6.

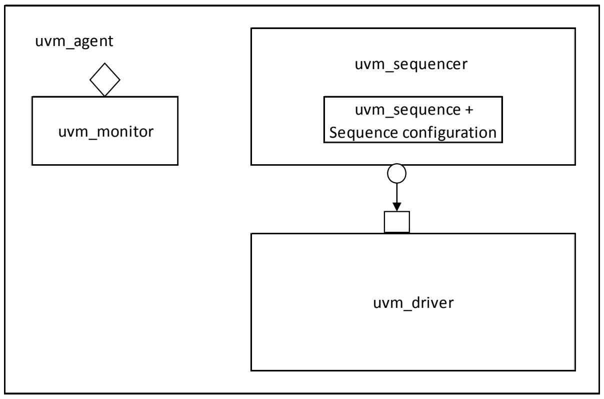

Figure 6 - Class diagrams for sequence with built in sequence configuration for random delays

With this approach, we have introduced randomization into the generation of delays between the various signals, significantly improving our coverage. Still, three problems remain.

1. First, because the delays for the signals are coded into the sequence itself, creating targeted random test cases by limiting/constraining the delay parameter ranges requires the test writer to create new sequence classes, which is a heavy burden for changing just two parameters.
2. Second, if the specification of the minimum and maximum changes, we have to edit all our sequences.
3. Third, it would be ideal to provide the test writer with the ability to modify the sequence delay behavior at run time. That way the underlying sequence can remain stable, and the test writer can create targeted random test scenarios based on the specific coverage holes or bugs that are discovered during the course of the project.

VI. SOLUTION 3 – STABLE AND FLEXIBLE SEQUENCE, RANDOMIZATION DRIVEN BY TEST

We achieve the objective of a stable and flexible sequence in three steps. First, we move the configuration parameters out of the sequence into a central signal configuration class that can be shared by multiple sequences.

Notice that the signal configuration class as shown in Figure 7 is built, it has a default minimum and maximum delay which can be overridden by the test writer at run time.

```
class read_seq_cfg extends uvm_object;
rand uint pEnable1Cycles;                        // Specific number of cycles where only enable1 will be high
uint pEnable1CyclesMin, pEnable1CyclesMax;     // Range of cycles where only enable1 will be high
……
function new(string name="read_seq_cfg",
  int unsigned pEnable1CyclesMin = 0,                // Set default minimum, override at run time if needed
  int unsigned pEnable1CyclesMax = 13,             // Set default maximum, override at run time if needed
  ….
  );
  this.pEnable1CyclesMin = pEnable1CyclesMin;
  this.pEnable1CyclesMax = pEnable1CyclesMax;
endfunction

constraint cEnable1Cycles {
  pEnable1Cycles >= pEnable1CyclesMin;
  pEnable1Cycles <= pEnable1CyclesMax;
}
endclass
```

Figure 7 – Signal configuration class for sequence

Next, we have specific tests create an object of this signal configuration class and override any parameters of interest. Figure 8 below shows an example of a specific targeted test that is interested in achieving delays close to the maximum between *enable1* and *enable2*. The test writer sets the delay parameter range to [12:13] cycles between (*enable1*, *enable2*) in this test.

```
class read_long_enables_test extends uvm_test;
……
rand read_seq_cfg m_read_seq_cfg;
virtual function void build_phase(uvm_phase phase):
…..
  m_read_seq_cfg = read_seq_cfg::type_id::create("read_seq_cfg");
  if(!m_read_seq_cfg.randomize()) `uvm_error(get_type_name(), "Unable to randomize read_seq_cfg");
  read_seq_cfg.pEnable1CyclesMin = 12;
  read_seq_cfg.pEnable1CyclesMax = 13;
  uvm_config_object::set(this, "*", "read_seq_cfg", m_read_seq_cfg);

endfunction
endclass
```

Figure 8 - Test that sets up sequence configuration

Finally, we have the sequence shown below in Figure 9 use the configuration parameters provided by the test to determine the timing relationships between *enable1* and *enable2* in the sequence.

```
class read_seq extends uvm_sequence #(read_txfer);
…………………
virtual task body():
 parent = get_sequencer();                    // Get the uvm_component to eventually get linked config object
 if (parent == null) `uvm_fatal("nullParent", …);
 if (m_read_seq_cfg == null) begin
      if (!uvm_config_db#(read_seq_cfg)::get(parent, "", "read_seq_cfg",
                                               m_read_seq_cfg))   // Get the configuration for the sequence
           `uvm_fatal("nullReadReqSeqCfg", ….);
 end

 repeat(m_read_seq_cfg.pEnable1Cycles)     `uvm_do_with(req, {enable1 == 1'b1;});
 repeat(m_read_seq_cfg.pEnable2Cycles)     `uvm_do_with(req, {enable1 == 1'b1; enable2 == 1'b1;});
 repeat(m_read_seq_cfg.pReadAddrCycles) `uvm_do_with(req, {enable1 == 1'b1; enable2 == 1'b1;
                                                             read_addr == 'hF00DF00D;});
 `uvm_do_with(req, {enable1 == 1'b1; enable2 == 1'b1; read_addr == 'hF00DF00D; read_req == 1'b1;});
endtask
……
endclass
```

Figure 9 - Sequence that uses the configuration object provided by the test for delay randomization

With this approach, we have allowed a targeted test to be created by manipulating the sequence configuration object passed to the sequence at run time without requiring the test writer to define an additional sequence class at compile time that further constrains the delay parameter between (*enable1*, *enable2*). The adaptable sequence as shown in Figure 9 gets the configuration object created by the test writer as shown in Figure 8 and drives the signal transitions for *enable1* and *enable2* by looking into the delay parameter values populated in the signal configuration object created by the test writer.

An explanation may be in order for the need to get a pointer to the parent sequencer in Figure 9. Because the sequence is an *uvm_object*, it cannot use a pointer to itself as the first parameter in the *uvm_config_db::get()* call which it makes to get its configuration values. The first parameter to the *uvm_config_db::get()* call needs to be an *uvm_component*. As a result, the sequence first gets a pointer to its parent sequencer (which is derived from *uvm_component*) and then passes the parent sequencer pointer as the first parameter in the *uvm_config_db::get()* call to get its configuration object.

Pictorially, this approach is shown in Figure 10.

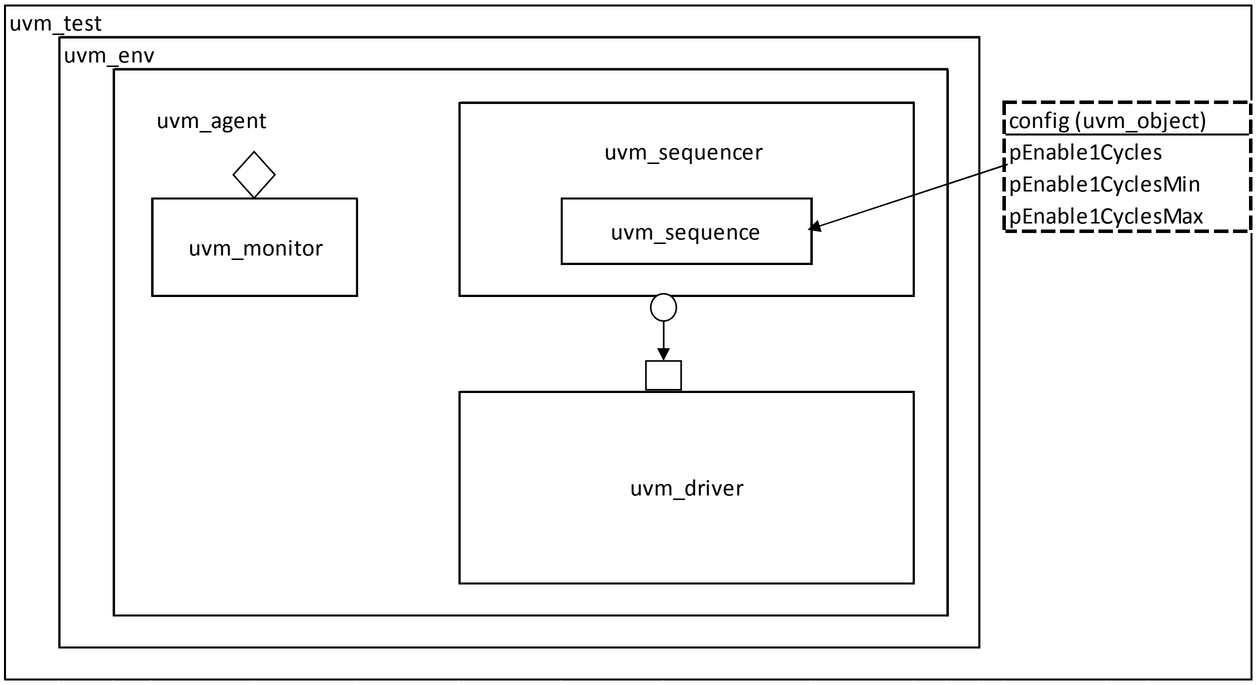

Figure 10 - Class Diagrams for sequence that uses external configuration for delay randomization

VII. SOLUTION 4 - PROVIDING THE ULTIMATE FLEXIBILITY TO THE TEST WRITER

The previous approach coded the read sequence in a way that came close to guaranteeing that the underlying sequence would not have to be changed to create different tests. However, there still remains one situation where the underlying sequence would have to be changed. Assume that the microarchitecture specification changed - Now *enable2* is allowed to be asserted before, on, or after *enable1* (Recall that previously *enable2* could only be asserted at or after *enable1*). The snippet of sequence code in Figure 9 presumes that *enable1* has to be asserted at or before *enable2*. Therefore, would have to modify the sequence as a result of the microarchitecture change, which is not ideal. Can we come up with a solution that provides ultimate flexibility to the test writer even in the face of microarchitecture specification changes? To do this, all we have to do is to define the signal configuration parameters for the sequence differently. Instead of defining the signal configuration parameters relative to the previous signal transition/edge, we define them relative to the start point of the sequence as shown in Figure 11.

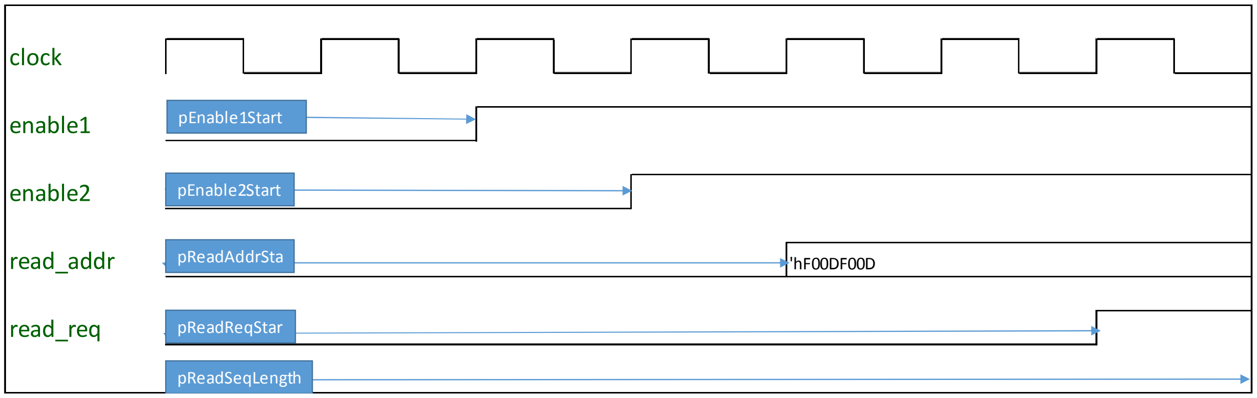

Figure 11 - Redefining delay parameter bounds for ultimate flexibility

Now if we wanted *enable2* to be driven before *enable1*, the test writer would simply constrain ***pEnable2Start*** to be smaller than ***pEnable1Start*** when they create the configuration object. The sequence code is shown in Figure 12. In the interests of brevity, the modified test code (similar to Figure 8) and the modified configuration object (similar to Figure 7) are not reproduced here. To enable more code sharing, we could store the "properties"/"attributes" of each of the signals (values and delays) in a signal generator class (*sig_gen* – See Figure 13) and re-use the class to generate the values for the signals on a per cycle basis. When the signal generator object is created (build time) using the signal generator class, it is passed information on the initial value, the initial length, the final value and the final length. The signal generator object can then be queried to obtain the cycle-by-cycle values of the signals.

```
class read_seq extends uvm_sequence #(read_txfer);
…………………
virtual task body():
 parent = get_sequencer();                    // Get the uvm_component to eventually get linked config object
 if (parent == null) `uvm_fatal("nullParent", …);
 if (m_read_seq_cfg == null) begin
      if (!uvm_config_db#(read_seq_cfg)::get(parent, "", "read_seq_cfg",
                                                             m_read_seq_cfg))   // Get the configuration for the sequence
           `uvm_fatal("nullReadReqSeqCfg", ….);
 end

 // local variables (l_ variables) to compute the cycle-by-cycle values of the signals that we want to drive
 logic l_enable1, l_enable2, l_read_req;
 logic l_read_addr;

 for (int i = 0; i < m_read_seq_cfg.pReadSeqLength; i++) begin
        if (i == m_read_seq_cfg.pEnable1Start)     l_enable1 = 1'b1;
        if (i == m_read_seq_cfg.pEnable2Start)     l_enable2 = 1'b1;
        if (i == m_read_seq_cfg.pReadAddrStart) l_read_addr = 8'hf00df00d;
        if (i == m_read_seq_cfg.pReadReqStart)   l_read_req = 1'b1;

      `uvm_do_with(req, {enable1 == l_enable1; enable2 == l_enable2;
                                   read_addr == l_read_addr; read_req = l_read_req;});
 end
endtask
……
endclass
```

Figure 12 - Modified sequence with complete flexibility

```
class sig_gen #(type T=bit);
  // Type of signal we are dealing with (for example single bit, multi-bit etc)
  rand T sigVal;

  // Cycle duration for the initial phase of the signal
  rand int unsigned initialLength;
  // Value to be returned in the initial phase of the signal
  rand T          initialVal;

  // Cycle duration for the final phase of the signal
  rand int unsigned finalLength;
  // Value to be returned in the final phase of the signal
  rand T finalVal;
    int count = 0;
  …….
  function new(string       name,
              int unsigned initialLength,
              T          initialVal = 'd0,
              int unsigned finalLength,
              T          finalVal  = ~initialVal);
      this.name         = name;
      this.initialLength = initialLength;
      this.initialVal    = initialVal;
      this.finalLength   = finalLength;
      this.finalVal      = finalVal;
  endfunction // new

  function T getVal();
      T returnVal;
        if (count < initialLength) return initialVal;
        else if ((count >initialLength) && (count < finalLength)) return finalVal;
        else (… ERROR ….);
        count++;
  endfunction
endclass
```

Figure 13 – Signal generator class to store signal attributes and retrieve signal values

It should be noted that the parameter definitions as provided in Figure 11 can be translated into an executable microarchitecture specification as shown in Table 1. Furthermore, the executable microarchitecture specification bound values can be used to define the specific delay parameters in the signal configuration object which in turn will control the behavior of the signal generator object.

To elaborate further, a change in the microarchitecture specification as represented in Table 1 would simply lead to different parameter bound values in the signal configuration object (similar to the one shown in Figure 7). The changes in the signal configuration object can either be done in the default parameter values of the *new()* (constructor) function of the signal configuration class or by changing all tests to override the default parameter behavior of the *new()* (constructor) function of the signal configuration class. The signal configuration object thus created would then define the bound values that would be passed into the signal generator objects *new()* (constructor) function shown in Table 1.Furthermore, the executable microarchitecture specification bound values can be used to define the specific delay parameters in the signal configuration object which in turn will control the behavior of the signal generator object.

Table 1 - Executable Microarchitecture specification

Signal parameter	Minimum bound from start of sequence	Maximum bound from start of sequence
pEnable1Start	0	2
pEnable2Start	0	3
pReadAddrStart	0	4
pReadReqStart	5	6
pReadSeqLength	7	9

Table 1 can be easily extended to cover cases where each signal undergoes more than one transition as part of the sequence. In this case, we would need one signal parameter per transition of the signal that indicates the distance from the start of the sequence until the specific signal transition.

However, the problem becomes more interesting when the multiple transitions of one signal are somewhat correlated to the transitions of other signals. For example, consider the case when the protocol allows two *read_req* "burst" requests for a single assertion of *read_addr*. In this case, there are two choices in sequence coding/creation:

1. We can specify each of the *read_req* burst transitions as a separate entry in Table 1. Thus the test writer can specify and control the timing of the burst transitions. The test writer has to take care to ensure that the *read_req* transitions are ordered correctly with respect to the *read_addr* transitions
2. We can provide a sequence configuration class parameter to indicate the number of *read_req* burst transitions and provide delay parameters for the distance between burst 1 and burst 2. In this case, the sequence would hard code the relationship between *read_req* and *read_addr* and the test writer loses some control over the relative order of the burst signal transitions (*read_req* will always be preceded by *read_addr*). With this approach the test writer has to specify less parameters to achieve the intended effect. This option is easier for the test writer, but less resilient in the face of microarchitecture changes.

The modified sequence code is shown in Figure 14. It uses the *sig_gen* class defined in Figure 13 to compute the values of the various signals on a cycle by cycle basis and populates the sequence object that is handed off to the driver.

```
class read_seq extends uvm_sequence #(read_txfer);
…………………
virtual task body():
  …….
  // local variables (l_ variables) to compute the cycle-by-cycle values of the signals that we want to drive
  sig_gen#(bit) l_enable1, l_enable2, l_read_req;
  sig_gen#(byte) l_read_addr;

  l_enable1 = new("l_enable1", .initialLength(m_read_seq_cfg.pEnable1Start),
                               .initialValue(1'b0), .finalValue(1'b1),
                               .finalLength(m_read_seq_cfg.pReadSeqLength - m_read_seq_cfg.pEnable1Start));
  l_enable2 = new("l_enable2", .initialLength(m_read_seq_cfg.pEnable2Start),
                               .initialValue(1'b0), .finalValue(1'b1),
                               .finalLength(m_read_seq_cfg.pReadSeqLength - m_read_seq_cfg.pEnable2Start));
  …. // similar definitions for l_read_req and l_read_addr

  for (int i = 0; i < m_read_seq_cfg.pReadSeqLength; i++) begin
      `uvm_do_with(req, {enable1 == l_enable1.getVal(); enable2 == l_enable2.getVal();
                                   read_addr == l_read_addr.getVal(); read_req == l_read_req.getVal();});
  end
endtask
……
endclass
```

Figure 14 - Sequence that uses the underlying signal class and external configuration parameters for maximum flexibility

VIII. IMPLEMENTATION NOTE

The final approach has been used successfully and has formed the basis of a VIP that is shared by multiple verification environments. One of the successes of the implementation is that the VIP has been used to simulate many different signal behaviors and what-if scenarios on the testbench signals across multiple architectural generations without requiring any modification of the underlying sequences.

IX. FUTURE WORK

Using scripting languages and the underlying signal generator class, the executable microarchitecture specification can be used to automatically generate the sequences. The underlying signal generator class can be enhanced to provide different kinds of non-uniform randomization distributions like bathtub distributions across minimum and maximum parameter bounds, leading to better coverage. In order to do this effectively, the minimum and maximum delay parameter bounds would be passed into the *sig_gen* class along with the desired distributions of the derived delay values. The *sig_gen* class would then use the delay parameter bounds and the desired distribution as inputs to compute the specific delay values and use them to return the cycle by cycle values of each signal.

X. SUMMARY

We explored the problem of controlling stimulus generation with minimal additions/modifications to underlying testbench code. Using the UVM configuration database, by defining configuration objects that are used in the sequences, and performing test driven sequence generation, we show a few solutions to achieving this goal. Using these approaches, randomized sequences can be automatically generated from the microarchitecture specification. Specification changes to signal timing can be seamlessly absorbed by the test writers without requiring modifications to the testbench. Multiple directed tests with different coverage intents can be created using the same underlying sequences.