Sequence, Sequence on the Wall – Who's the Fairest of Them All?

Using SystemVerilog UVM Sequences for Fun and Profit

Rich Edelman
Mentor Graphics
Fremont, CA, US

Raghu Ardeishar
Mentor Graphics
McLean, VA, US

***Abstract*— The reader of this paper is interested to use UVM sequences to achieve his test writing goals. Examples of UVM sequences will be used to demonstrate basic and advanced techniques for creating interesting, reusable sequences and tests.**

Keywords—Functional verification; SystemVerilog; UVM; UVM Sequences; Transactions; Pipelined Driver; Out-of-order completion;

I. INTRODUCTION

Sequences have become the basic test writing technique as UVM [2] development is continuing. When a functional verification engineer writes a test using the SystemVerilog [1] UVM, he will be writing a UVM Sequence.

A sequence is simply a SystemVerilog task call which can consume time, and which has one of two jobs. It can either cause other sequences to be started, or it can generate a transaction to be passed to a driver.

A sequence is implemented as a SystemVerilog task named 'body()' in a class derived from a uvm_sequence. In programming terms a sequence is a functor[5][6]. The implementation of the body is free to perform any function it desires, from creating a single transaction and sending to the driver, to creating multiple transactions, to creating and starting sub-sequences. It could create no transaction but simply print a message. A sequence is most useful when it is creating a related sequence of information or causing other activity on the driver and interface that represents traffic or tests.

II. A UVM TESTBENCH

A UVM testbench has many parts (agent, driver, monitor, sequencer, sequences and virtual interfaces) as shown in Figure 1 below. This paper will focus on the transactions and sequences with some discussion about building drivers to support various kinds of stimulus.

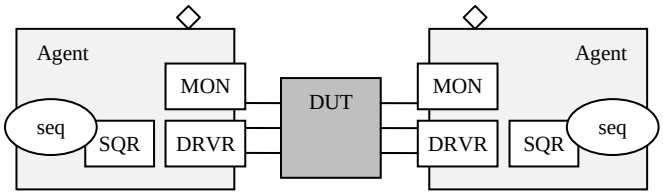


Figure 1 - UVM Testbench

In a UVM testbench a sequence is started that "runs" on a sequencer. The sequences may cause transactions, also known as sequence_items or just items; to be sent to the driver. If multiple sequences wish to send transactions to the driver at the same time, the sequencer will arbitrate who gets to go first. The default arbitration is first-come-first-served.

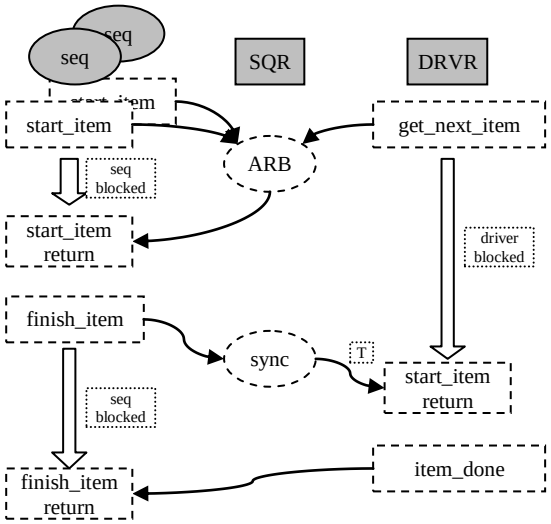


Figure 2 - Sequence/Sequencer/Driver

A. *Transactions*

A transaction is a collection of information which is to be sent from one point to another. In use with a sequence a transaction is called a "sequence_item". A sequence may create a transaction, filling in the required information, and then send

that transaction to a driver. The driver will receive the transaction and act upon it. A transaction can be thought of as a command for the driver to execute.

A simple transaction with three fields is described below. It has three random values, a bit named 'rw', and two 32 bit vectors, 'data' and 'addr'. This transaction represents a READ or a WRITE transaction with an address and data.

```
class trans_c extends uvm_sequence_item;
  `uvm_object_utils(trans_c)
  rand bit rw;
  rand bit [31:0] data;
  rand bit [31:0] addr;
endclass
```

The slightly more complicated transaction below represents the roll of a die. The value of the die is represented by the class member variable named 'value', which is declared as an enumerated type.

```
typedef enum {
  DICE1 = 1, DICE2, DICE3, DICE4, DICE5, DICE6
} dice_t;

class dice_item extends uvm_sequence_item;
  `uvm_object_utils(dice_item)
  rand dice_t value;
  ...
endclass
```

The packet transaction below contains one class member variable for each field in the packet, including 'som', 'addr', 'payload', 'checksum' and 'eom'. The fields are all randomized, except for checksum, which is calculated. This transaction contains a constraint for the som and eom and payload fields. The checksum is calculated using the built-in function 'post_randomize'.

```
class packet extends uvm_sequence_item;
  `uvm_object_utils(packet)

  rand bit[7:0] som;
  rand bit[7:0] addr;
  rand bit[7:0] payload [8];
       bit[7:0] checksum;
  rand bit[7:0] eom;

  constraint val {
    som == '0;
    eom == '1;
    foreach (payload[i])
      payload[i] inside {[0:100]};
  }

  function void post_randomize();
    calc_checksum();
  endfunction

  virtual function void calc_checksum();
    checksum = 0;
    foreach(payload[i])
      checksum = f(checksum, payload[i]);
  endfunction
  ...
endclass
```

The 'big_packet' transaction below is a simple extension of the packet transaction above. Big_packet extends packet, thereby inheriting all the member variables, functions and tasks of packet, while adding the io_type, and replacing the payload field with a larger payload. Normally, replacing class member variables using inheritance can lead to confusion in use, but for in this example special care is being taken to avoid confusion. The calc_checksum is implemented in the extended class to allow the correct payload access. The details of this virtual function and inheritance are outside the scope of this paper.

```
class big_packet extends packet;
  `uvm_object_utils(big_packet)

  rand bit[7:0] payload [32]; // Size change
       bit[7:0] io_type;      // New attribute

  virtual function void calc_checksum();
    checksum = 0;
    foreach(payload[i])
      checksum = f(checksum, payload[i]);
  endfunction

  ...
endclass
```

B. Sequences

Once a transaction is defined, a sequence of transactions can be created and sent to the driver and on to the device under test. In the UVM, the most convenient way to achieve this is with a uvm_sequence.

A sequence of transactions is created below, specifically, a sequence of dice rolls.

```
class roll_sequence extends
    uvm_sequence#(dice_item);
  `uvm_object_utils(roll_sequence)

  rand int how_many;
  constraint val { how_many > 70;
                   how_many < 100; }

  task body();
    dice_item t;

    for(int i = 0; i < how_many; i++) begin
      t = dice_item::type_id::create(
        $sformatf("t%0d", i));
      if (!t.randomize()) begin
        `uvm_fatal("SEQ", "Randomize failed")
      end
      start_item(t);
      finish_item(t);
      #2;
    end
  endtask
endclass
```

In the code above, a sequence named 'roll_sequence' is defined which has a random variable named 'how_many' constrained to take on the values of 71 to 99. The body() task is built simply. It is a for-loop which counts from zero to 'how_many'. Each time through the loop a new dice object is created and randomized. Then start_item() and finish_item() are called to first ask the sequencer for permission to send and then to actually send the transaction.

In this code if how_many had been 75, then 75 dice objects would have been created, randomized and sent to the driver and device under test.

C. Sending Transactions to the Driver

The sequence, sequencer and driver in a UVM testbench have a special way they communicate [9]. They work together to send arbitrated requests from the sequence to the driver, and optionally send responses from the driver to the sequence.

The sequence creates transactions and sends them to the driver. A sequence usually contains code which causes interesting sequences to be created; for example a 'fibonacci' sequence and a 'value_item' transaction are combined below to cause a new value in the sequence to be sent to the driver each time start_item() and finish_item() are called.

```
class value_item extends uvm_sequence_item;
  `uvm_object_utils(value_item)
  rand int value;
  ...
endclass

class fibonacci_sequence
    extends uvm_sequence#(value_item);
  `uvm_object_utils(fibonacci_sequence)

  rand int how_many;
  constraint val { how_many > 10; how_many < 30; }

  ...
  task body();
    int previous_value1;
    int previous_value2;

    value_item t;

    // First value.
    t = value_item::type_id::create(
      $sformatf("t%0d", 0));
    t.value = 0;
    start_item(t);
    finish_item(t);
    previous_value1 = previous_value2;
    previous_value2 = t.value;

    // Second value.
    t = value_item::type_id::create(
      $sformatf("t%0d", 1));
    t.value = 1;
    start_item(t);
    finish_item(t);
    previous_value1 = previous_value2;
    previous_value2 = t.value;

    // Iterate.
    for(int i = 2; i < how_many; i++) begin
      t = value_item::type_id::create(
        $sformatf("t%0d", i));
      if (!t.randomize() with {
        value == previous_value1 +
                 previous_value2;
      } ) begin
        `uvm_fatal("SEQ", "Randomize failed")
      end
      start_item(t);
      finish_item(t);
      previous_value1 = previous_value2;
      previous_value2 = t.value;
    end
  endtask
```

In the Fibonacci sequence body code two initial value transactions are sent (0 and 1). Each iteration through the for-loop calculates the new value and sends that value to the driver. This calculation of 'next-value' is simple in this case, but could be quite complex and involve many state variables from other places in the testbench or DUT. For the fibonacci sequence, the next value in the sequence is a simple calculation. In a typical sequence the next value computation may involve multiple random variables and constraints.

Once a sequence has been constructed it can be randomized, or data fields can be set. Once the sequence is ready to be executed, it is started "on" a sequencer. The sequence is not actually running on the sequencer, but the sequencer is acting as the required arbitration mechanism for the sequence to send transactions to a driver.

The main job of the sequencer is to arbitrate multiple sequence requests to send a transaction to the driver. It also manages responses if needed. A sequencer manages a single driver. When a sequence wants to send the driver a transaction, it asks for permission from the sequencer using start_item(). Once granted the sequence can send transactions to the driver using finish_item().

```
class seq extends uvm_sequence#(trans_c);
  …
  task body();
    for(int i=0; i<10; i++) begin
      trans = trans_c::type_id::create("t");
      start_item(trans);
      if (!trans.randomize())
        `uvm_fatal("Randomize failed ...")
      finish_item(trans);
    end
  endtask
endclass

class driver extends uvm_driver#(trans_c);
  `uvm_component_utils(driver)

  bit [7:0] mem [256] = '{default:0};
  ...
  task run_phase(uvm_phase phase);
    trans_c t;
    forever begin
      seq_item_port.get_next_item(t);

      if (t.rw == 0) // WRITE
        mem[t.addr] = t.data;
      else           // READ
        t.data = mem[t.addr];
     #10;
     seq_item_port.item_done();
    end
  endtask
endclass
```

In the sequence above, start_item() and finish_item() communicate with the sequencer. They are misnamed. Start_item() should be thought of as 'request_for_grant()' and finish_item() should be thought of as 'execute_item()'. Start_item() asks permission from the sequencer to send a transaction to the driver. Once start_item() returns, permission has been granted and the sequence can send a transaction to the sequencer (and driver) – by calling finish_item(). See Figure 2 - Sequence/Sequencer/Driver.

Once finish_item() has been called, the transaction is sent to the driver for the driver to "execute". The driver eventually calls item_done() which will cause finish_item() to return in the sequence.

D. Getting Responses from the Driver to the Sequence

A traditional solution to getting responses from the driver is to have the sequence "wait" for a response using the API get_response(), and to have the driver create a new object, a response, then have the driver use the API put_reponse() to send the newly created object back to the sequence. This solution is not recommended for speed and efficiency reasons. Instead it is recommended to embed the response in the request. This type of response avoids creating a new object, and avoids needing to use an additional sequencer/driver connection (rsp_export and rsp_port), and avoids using the get_response() and put_response() APIs. The various examples in this paper avoid the use of get_response() and put_response(), instead preferring to embed any response in the actual request itself.

Instead of having a dedicated response object, use the request object to contain the response. For example, a READ transaction could have two fields – the address and data. The sequence sets the address and issues the request to the driver. The driver executes the request, and then when it receives the read data response, it fills that read data response into the request object data field. A READ transaction now contains an address and a data – it is a completed transaction.

Instead of being request or responses objects there is just a single transaction object that is used to describe the information being transferred.

A sequence is code that runs; a program; a SystemVerilog task which executes. When it executes a sequence can either create downstream sequences or downstream sequence items (transactions). UVM sequences are modeled as functors, but can be more simply thought of a test or program which will execute, eventually causing activity on a bus or interface.

E. Writing a test – starting a sequence

A test is a program which causes the device under test to be tested. In this paper, a test is a sequence or a collection of sequences that perform the stimulus generation and possibly checking functionality of functional verification. A test can be written by constructing a sequence and starting it. The sequence will cause other sequences to be constructed and started or will cause transactions to be generated.

A sequence is a dynamic SystemVerilog object and must be constructed by calling new(). Usually new() is not called directly, but rather the UVM factory is used.

```
class test extends uvm_test;
  `uvm_component_utils(test)

  env e;

  ...

  function void build_phase(uvm_phase phase);
    e = env::type_id::create("e", this);
  endfunction

  task run_phase(uvm_phase phase);
    roll_sequence seq;
    phase.raise_objection(this);
    seq = roll_sequence::type_id::create(
      "seq", , get_full_name());
    if (!seq.randomize())
      `uvm_fatal("TEST",
        "Randomization failed for seq")
    seq.start(e.sqr);
    phase.drop_objection(this);
  endtask
endclass
```

In the test above, the sequence is started by calling seq.start(e.sqr). The start routine is part of the sequence base class. It will do a variety of things, including transaction recording before it does the most important thing – it calls the sequence 'body()' task.

The implementation of the sequence start task in the file uvm_sequence_base.svh in uvm-1.1c is 116 lines long. There are forks and #0s and callbacks. In order to keep a sequence simple, avoid using any of the available callbacks.

Think of the seq.start() as a way to have a transaction automatically started before body() is called, then body() is called, and after body() completes the automatic transaction is ended. Calling seq.start() is really just like calling seq.body(). There are a few UVM related bookkeeping related tasks that start performs.

F. Sequence API

The sequence and sequencer have many API controls. Grab, lock and sequence_item priority are some of the most useful. A sequence can call 'grab', which will cause the next arbitration of the sequencer to be held by this sequence until it ungrabs. A sequence can call 'lock', which will cause the lock request to go into a queue. The next time the sequencer processes the queue, and chooses the lock, the sequencer will be locked for this sequence until it is unlocked. Priority is used when a sequence_item is started using start_item(item, priority). The sequencer arbitration mode must be SEQ_ARB_STRICT_FIFO for priority to be used. The default priority is 100. Higher numbers mean higher priority.

To use the priority, use the start_item priority argument.

```
start_item(t, packet_priority);
finish_item(t);
```

The seq sequence below has the ability to be a grabbing sequence, a locking sequence or a sequence with a priority. It generates N sequence_items, first doing a WRITE, then a READ to the same address. The data read is compared with the data written – using the data returned in the transaction request. Additionally, this sequence will not repeat an address. It will pick addresses that have not yet been picked, each time through the loop – using the 'addr_used' array, and the '!(t.addr inside addr_used)' constraint.

```
class seq extends uvm_sequence#(trans);
  `uvm_object_utils(seq)

  int grabbing = 0;
  int locking = 0;
  int qos = 100; // default is 100 in the UVM

  rand int n;

  constraint val { n > 10; n < 30; }

  bit [7:0] addr_used[bit[7:0]];
```

```
...
task body();
  bit [7:0] expected_data;
  trans t;

  if (grabbing) grab();
  if (locking) lock();


  for (int i=0; i<n; i++) begin
    t = trans::type_id::create(
      "t",, get_full_name());

    start_item(t, qos);

    if (!t.randomize() with {
        t.rw == 0; // Always do a WRITE first.
        !(t.addr inside { addr_used });
        } )
        `uvm_fatal("seq", "Randomize failed")

    finish_item(t);

    // Remember this address. Collect stats
    addr_used[t.addr] = t.addr;
    expected_data = t.data;
    t.data = -1;

    // Now do a READ.
    t.rw = 1;
    start_item(t, qos);
    `uvm_info("seq", $sformatf("Sending R %s",
      t.convert2string()), UVM_MEDIUM)
    finish_item(t);

    // Now CHECK.
    if (t.data != expected_data)
      `uvm_error("seq", $sformatf(
        "Mismatch: expected %3d, got %3d",
        expected_data, t.data))
  end

  if (locking) unlock();
  if (grabbing) ungrab();

  endtask
endclass
```

If four instances of these sequences are trying to gain access to the sequencer, and they are using default priority, grab, lock, and priority=1000 respectively, their requests will be granted in the following order: grab, lock, priority=1000 and finally default priority.

In order to use priority, the sequencer must be in the proper mode. The arbitration mode of the sequencer can be set using the set_arbitration() API.

```
function void build_phase(uvm_phase phase);
  d = driver::type_id::create("d", this);
  seqr = uvm_sequencer#(trans)::
    type_id:: create( "seqr", this);
  seqr.set_arbitration(SEQ_ARB_STRICT_FIFO);
endfunction
```

The set_arbitration() call on the sequencer object can take any of the values listed below. The default (SEQ_ARB_FIFO) is first in, first out, and priority is ignored. In order to use priority and first in, first out, set the arbitration mode to SEQ_ARB_STRICT_FIFO. The other arbitration modes are left as an exercise for the reader.

Arbitration Mode	Definition
SEQ_ARB_FIFO	Requests are granted in FIFO order (default)
SEQ_ARB_WEIGHTED	Requests are granted randomly by weight
SEQ_ARB_RANDOM	Requests are granted randomly
SEQ_ARB_STRICT_FIFO	Requests at highest priority granted in fifo order
SEQ_ARB_STRICT_RANDOM	Requests at highest priority granted in randomly
SEQ_ARB_USER	Arbitration is delegated to the user-defined function

Table 1 - Sequencer Arbitration Modes

III. SEQUENCES WITH CONSTRAINTS

A. Generating Random Stimulus

The sequences below work together to generate a random 'data' value. The base sequence 'my_sequence' creates a transaction, then does a start_item and randomize. The transaction payload – the 'data' field is assigned from the sequence value – which was constrained or set from an upper level sequence. This base class sequence has a simple job – create a transaction, fill in the data value, and send the transaction to the driver.

```
typedef enum bit { CLR = 0, ROT13 = 1 } mode_t;

class transaction extends uvm_sequence_item;
  `uvm_object_utils(transaction)

  rand mode_t mode; // CLR or ROT13
  rand byte   data[];
       string secret_data;
  ...
endclass

virtual class my_sequence extends
    uvm_sequence#(transaction);
  `uvm_object_utils(my_sequence)

  string data;
  ...
  task body();
    transaction t;
    t = transaction::type_id::create("t");
    start_item(t);
    if (!t.randomize())
      `uvm_fatal("SEQ", "Randomization failed")
    t.data = data;
    finish_item(t);
  endtask
endclass
```

The short_word_seq extends my_sequence, adding two constraints. The length of the array of data (the string) is constrained to be between 3 and 8, and the data array is constrained to only have the values 'a' to 'z'.

```
class short_word_seq extends my_sequence;
  `uvm_object_utils(short_word_seq)

  rand int length;
  constraint val_length {
    length >= 3; length <= 8; }

  constraint val_data_length {
```

```
    data.size() == length; }
  constraint val_data {
    foreach (data[i]) {
      data[i] inside {["a":"z"]};
    }
  }
endclass
```

When the sequence short_word_seq is run it will cause a transaction to be created, randomized and sent to the driver. The transaction payload – the string value, implemented as an array of bytes will contain any lowercase letter, and be a string of length 3 to 8.

B. Randomization without Repeating

The SystemVerilog constraint language contains a keyword called 'randc'. Randc is similar to 'rand', in that it defines this variable to be randomizable. Additionally, randc means that when the variable is randomized, that no value should repeat until all values have been used. This is a very handy way to exercise all possible values in a random order.

The use of randc requires that the values being used stay in existence throughout the lifetime of the randomization – the values are consumed one by one. If a class is created with a randc variable, then that class goes out of scope and another is created, the new version of the class doesn't know any of the previous class history – it must start over with all the possible values.

The sequence above generates a random string of length 3 to 8. The four_letter_words sequence below will require, a constraint to generate all possible four-letter words. One solution is to use 'randc'. There are four letters that must be generated. A simple set of loops could be created to simple iterate through each value in turn. This solution would not present the words in a random order. The solution could generate these names and randomize the list – but this is effectively what we can ask the randomization engine to do.

In order to reduce the constraint complexity we can view generating 4 letter words using 'a' to 'z' in the same way as generating a random number from 0 to 456976, where 'a' is represented by 0 and 'z' is represented by 26. The four letter word is simply the product of each of the letters in the word.

Once we have generated an integer between 0 and 456976, we can perform modulo-26 arithmetic to divide that integer down into 4 values. SystemVerilog constraints provide a post_randomize function which is called after the class has been randomized. In the code below, post_randomize performs the necessary modulo math, and creates a four-letter string from the results.

```
class four_letter_words_randc;
  string data;
  randc int value;
  constraint val { value >= 0; value < 456976; }

  function void post_randomize();
    int v;
    data = "    "; // Four spaces.
    v = value;
    for(int i = 0; i < 4; i++) begin
      data[i] = (v%26) + "a";
      v = v/26;
    end
  endfunction
endclass
```

Each time 'four_letter_words_randc' is randomized a new random solution will be picked for the 'value' variable. No solution will be repeated until all possible solutions are used.

The four_letter_words_randc class is not a UVM object – it is a simple container class that can be used to allow randc solutions to be generated. Once the solution is generated it can be used as needed. In the sequence four_letter_words below four_letter_words_randc is created once. Each time four_letter_words is randomized, pre_randomize is called, causing 'r' to be randomized. In four_letter_words, the post_randomize function retrieves the random solution generated in the four_letter_words_randc container.

```
class four_letter_words extends my_sequence;
  `uvm_object_utils(four_letter_words)

  four_letter_words_randc r;

  function new(string name = "four_letter_words");
    super.new(name);
    r = new();
  endfunction

  function void pre_randomize();
    if (!(r.randomize()))
      `uvm_fatal("R",
        "Randomize failed for sub-randomize")
  endfunction

  function void post_randomize();
    data = r.data;
  endfunction
endclass
```

C. Generating Distribution based random stimulus

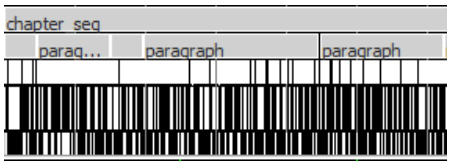
In the short_word_seq above, a distribution of letters can be added – the statistical distribution of letters in the English language; one distribution for leading characters, another distribution for the remainder of the word. Now when short_word_seq is used, the random distribution of characters should match a typical English word.

```
class short_word_seq extends my_sequence;
  `uvm_object_utils(short_word_seq)

  rand int length;
  constraint val_length {
    length >= 3; length <= 8; }

  constraint val_data_length {
    data.size() == length; }

  constraint val_data {
    foreach (data[i]) {
      data[i] inside {["a":"z"]}; // 'a' to 'z'

      // en.wikipedia.org/wiki/Letter_frequency
      if ( i == 0) // First letter of the word
      data[i] dist {
        "a" := 116,
        "b" := 47,
        "c" := 35,
        "d" := 27,
        "e" := 20,
        "f" := 38,
        "g" := 20,
```

```
        "h" := 72,
        "i" := 63,
        "j" :=  6,
        "k" :=  6,
        "l" := 27,
        "m" := 44,
        "n" := 24,
        "o" := 63,
        "p" := 25,
        "q" :=  2,
        "r" := 17,
        "s" := 78,
        "t" := 167,
        "u" := 15,
        "v" :=  6,
        "w" := 68,
        "x" :=  1,
        "y" := 16,
        "z" := 01
      };
      else // Remaining letters of the word
      data[i] dist {
        "a" := 81,
        "b" := 15,
        "c" := 28,
        "d" := 42,
        "e" := 127,
        "f" := 22,
        "g" := 20,
        "h" := 60,
        "i" := 70,
        "j" := 01,
        "k" := 08,
        "l" := 40,
        "m" := 24,
        "n" := 67,
        "o" := 75,
        "p" := 19,
        "q" := 01,
        "r" := 60,
        "s" := 63,
        "t" := 90,
        "u" := 28,
        "v" := 10,
        "w" := 23,
        "x" := 02,
        "y" := 20,
        "z" := 01
      };
    }
  }
  ...
endclass
```

D. *Hierachy of sequences*

Given the two sequences that can generate a word, short_word_seq and the four_letter_word_seq, new sequences to generate sentences, paragraphs and chapters can be created easily as layers of sequences.



The ‘chapter’ sequence creates, randomizes and starts many paragraph sequences. The ‘paragraph’ sequence creates, randomizes and starts many sentence sequences. The ‘sentence’ sequence creates, randomizes and starts the appropriate word sequences. At each level in the sequence hierarchy constraints can be added. In the example below, there is a constraint for how many items to create and for how long to wait between items.

```
class sentence_seq
    extends uvm_sequence#(transaction);
  `uvm_object_utils(sentence_seq)

  rand int how_many;
  rand int gap;
  rand int max_gap = 5;
  rand bit use_four_letter_word = 0;

  constraint val_how_many {
    how_many > 0; how_many < 200;
  }

  constraint val_max_gap {
    max_gap >= 5; max_gap < 10;
  }

  constraint val_gap {
    gap >= 0; gap < max_gap;
  }

  virtual dut_if vif;

  task body();
    sequencer p_sequencer;
    uvm_sequence_base seq;

    $cast(p_sequencer, m_sequencer);
    vif = p_sequencer.vif;

    for(int i = 0; i < how_many; i++) begin
      if (use_four_letter_word)
        seq = four_letter_words::type_id::create(
          "four_letter_words");
      else
        seq = short_word_seq::type_id::create(
          "short_word_seq");

      if (!seq.randomize())
        `uvm_fatal("SEQ", "Randomize failed")

      seq.start(m_sequencer);

      vif.wait_for_clk(gap);
    end
  endtask
endclass

class paragraph_seq extends
     uvm_sequence#(transaction);
  `uvm_object_utils(paragraph_seq)

  ...
  task body();
    sentence_seq seq;

    for(int i = 0; i < how_many; i++) begin
      seq = sentence_seq::type_id::create(
        "sentence");

      if (!seq.randomize())
        `uvm_fatal("SEQ", "Randomize failed")

      seq.start(m_sequencer);
    end
  endtask
endclass

class chapter_seq extends
    uvm_sequence#(transaction);
  `uvm_object_utils(chapter_seq)
```

```
  ...
  task body();
    paragraph_seq seq;

    for(int i = 0; i < how_many; i++) begin
      seq = paragraph_seq::type_id::create(
        "paragraph");

      if (!seq.randomize())
        `uvm_fatal("SEQ", "Randomize failed")

      seq.start(m_sequencer);
    end
  endtask
endclass
```

IV. SEQUENCES

A. *Using Overrides*

The packet and big_packet transactions are registered with the factory and are constructed using the factory. By using the factory, they enable factory overrides. Factory overrides are used to replace a class instance with a derived class instance. In the case of packet and big_packet, a big_packet, derived from packet can replace an instance of packet.

There are two types of factory overrides; type overrides and specific instance overrides. A type override means that any time that type is requested from the factory, the override type should be returned instead. An instance specific override means that any time that specific instance name is requested from the factory, the override type should be returned instead.

The syntax for type overrides in the factory can be read as – when a packet is requested, please substitute a packet_with_randc_addr instead.

```
packet::type_id::set_type_override(
      packet_with_randc_addr::get_type());
```

The syntax for instance overrides in the factory can be read as – when a packet is requested that has the given instance name, please substitute a big_packet instead.

```
packet::type_id::set_inst_override(
      big_packet::get_type(),
      "uvm_test_top.e1.sequencer.seq1.packet");
```

In order to enable instance based transaction replacement the instance must be created with a context name. When a transaction is created in a sequence, simply provide the third argument to create with 'get_full_name()'. Leave the second argument empty.

```
 t = packet::type_id::create(
   "packet", , get_full_name());
```

When running simulation, the factory overrides can be debugged by using the factory print() command. Issuing the print command for this example will produce output similar to the text below. This text describes two overrides. An instance override that overrides packet with big_packet for the instance "uvm_test_top.e1.sequencer.seq1.packet" and a type override that overrides packet with packet_with_randc_addr.

```
factory.print();

#### Factory Configuration (*)
#
# Instance Overrides:
#
# Requested Type: packet
# Override  Path:
#           uvm_test_top.e1.sequencer.seq1.packet
# Override  Type: big_packet
#
# Type Overrides:
#
#   Requested Type  Override Type
#   --------------  -----------------------------
#   packet          packet_with_randc_addr
#
...
```

The output tells us how the factory is currently programmed. The type packet will be overridden with packet_with_randc_addr, and the instance 'uvm_test_top.e1.sequencer.seq1.packet' packet type will be overridden with big_packet.

B. *Virtual Sequences*

A virtual sequence is not really 'virtual'. It is a real sequence whose job is slightly different than a regular (non-virtual) sequence. A virtual sequence is responsible for starting other sequences. A regular sequence is responsible for sending transactions to a driver. In reality a sequence can do both things, but normally does one or the other, but not both.

In order to generate traffic on all three interfaces below a virtual sequence might start three sequences – one on each agent. Axx_basic_sequence below is a virtual sequence.

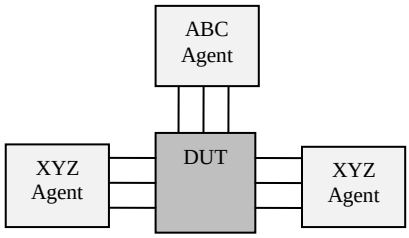


Figure 3 - DUT with 3 interfaces

```
class axx_basic_sequence extends
    axx_sequence_base;
  `uvm_object_utils(axx_basic_sequence)

  axx_env env;

  ...
  task body();
    abc_sequence seq1;
    xyz_sequence seq2, seq3;

    seq1 = abc_sequence::type_id::create("seq1");
    seq2 = xyz_sequence::type_id::create("seq2");
    seq3 = xyz_sequence::type_id::create("seq3");
    ...
    fork
      seq1.start(env.agent1.sequencer);
      seq2.start(env.agent2.sequencer);
      seq3.start(env.agent3.sequencer);
    join
  endtask
endclass
```

C. Simple Sequence in Tasks

Some verification teams resist constructing classes and using the sequence API to start sequences or the transaction API to start_items and finish_items. These teams can benefit from simple task based wrappers which hide the UVM details, and provide a more familiar task based environment. [4]

Given the two simple sequences below, read_seq and write_seq the tasks read and write below can be created.

```
class read_seq extends uvm_sequence#(trans);
   `uvm_object_utils(read_seq)
   bit [7:0] addr; // Input
   bit [7:0] data; // Output
   ...
   task body();
     trans t;
     t = trans::type_id::create(
       "t",, get_full_name());
     t.rw = 1;
     t.addr = addr;
     start_item(t);
     finish_item(t);
     data = t.data;
   endtask
endclass

class write_seq extends uvm_sequence#(trans);
   `uvm_object_utils(write_seq)
   bit [7:0] addr; // Input
   bit [7:0] data; // Input
   ...
   task body();
     trans t;
     t = trans::type_id::create(
       "t",, get_full_name());
     t.rw = 0;
     t.addr = addr;
     t.data = data;
     start_item(t);
     finish_item(t);
   endtask
endclass
```

The task read has an input and an output. The task creates a sequence, and assigns the address field. It then starts the sequence and when the sequence returns the task output argument is assigned from the sequence data field.

```
task read(input bit [7:0] addr,
          output bit [7:0] data);
  read_seq seq;
  seq = read_seq::type_id::create(
    "read",, get_full_name());
  seq.addr = addr;
  seq.start(seqr);
  data = seq.data;
endtask
```

The write task has two inputs. It creates a sequence, copies the addr and data in, and starts the sequence.

```
task write(input bit [7:0] addr,
           input bit [7:0] data);
  write_seq seq;
  seq = write_seq::type_id::create(
    "write",, get_full_name());
  seq.addr = addr;
  seq.data = data;
  seq.start(seqr);
endtask
```

The verification engineer can now write tests that are simple calls to write and read. These tasks must know what sequencer to run on, but that information can be provided in many ways.

The code below iterates through 256 addresses, issuing a WRITE followed by a READ. Then the read data is checked against the data written. This is a simple self-checking test.

```
for (int addr = 0; addr < 256; addr++) begin
  bit [7:0] data;
  bit [7:0] rdata;
  data = addr+1;
  write(addr,  data);
  read (addr, rdata);
  if (data != rdata)
    `uvm_fatal("TASK", $sformatf(
     "Compare failed wdata=%0x, rdata=%0x",
     data, rdata))
end
```

V. Controlling Sequences with Files

Many of the previous sequences use randomization to provide transactions to the DUT. The example below turns this idea on its head. It provides a file based mechanism to specify the sequence name to run along with a parameter for the sequence. These examples are simplistic and short for purposes of demonstration, but any file could be used to drive sequences and transactions – from sequence names to run to actual data that gets populated in the transactions.

The example below parses the simulation command line, looking for filenames. Those filenames are specified using +FILE=.

Usage…

```
<SIM CMD> +FILE=sequences.txt +FILE=sequences2.txt
```

The files sequences.txt and sequences2.txt contain formatted lines that can be easily read. Each line contains a sequence name, and an integer. The sequence name is the name of the sequence to start, and the integer is the count of how many transactions to send.

```
sequences.txt:
  fibonacci_sequence 10
  triangle_sequence  20
  bad_class 10

sequences2.txt:
  fibonacci_sequence 2
  triangle_sequence  4
```

The read_sequence_from_file sequence doesn't generate transactions directly, it reads a file, and starts a sequence as specified in the file.

```
class read_sequence_from_file extends
    uvm_sequence#(value_item);
  `uvm_object_utils(read_sequence_from_file)

  ...
  task body();
    int fd;
    string filenames[$];
    string sequence_name;
    int count;
    uvm_object obj;
    simple_value_base_class seq;
    int ret;
```

```
    uvm_cmdline_processor clp;
    clp = uvm_cmdline_processor::get_inst();
    if (clp.get_arg_values("+FILE=",
      filenames) == 0) begin
      `uvm_fatal(get_type_name(),
        "+FILE=<file> not specified")
    end

    foreach (filenames[n]) begin
      fd = $fopen(filenames[n], "r");
      while(!$feof(fd)) begin
        ret = $fscanf(fd, "%s %d",
          sequence_name, count);
        if ($feof(fd))
          break;

        obj = factory.create_object_by_name(
          sequence_name);

        if (obj == null) begin
          factory.print(1);
          `uvm_fatal(get_type_name(),$sformatf(
            "factory.create_ (%s) failed",
              sequence_name))
        end

        if (!$cast(seq, obj)) begin
          factory.print(1);
          `uvm_error(get_type_name(),$sformatf(
     "Sequence (%s) is not compatible with %s.",
            sequence_name,
            simple_value_base_class::type_name))
        end
        else begin
          if (!seq.randomize() with
              {seq.how_many == count;})
            `uvm_fatal(get_type_name(),
              "Randomization failed")

          seq.start(m_sequencer);
        end
      end // while !EOF
    end // foreach filenames
  endtask
endclass
```

The code first gets all the file names into a list – filenames. It iterates each filename, using $fopen() to open the file, and $fscanf to read one line at a time. The parsing is simple – a string followed by an integer. The factory routine create_object_by_name() is used to create a sequence. Some error checking is performed and the sequence is randomized using a 'with constraint' controlling the value of 'how_many'. Finally, the sequence is started.

Many other kinds of files could be used to drive verification. For example, for an encryption engine actual files to be encrypted could be used as input, with the encrypted file used as the expected check. For a video encoding system, the input could be video data. The reader is encouraged to explore using files as a way to connect legacy verification environments to new UVM based verification environments.

VI. SEQUENCES AS COLLECTIONS OF TESTS

Once a collection of sequences has been created, they represent the various programs or tests that can be run to exercise the device-under-test. With a little effort these sequences can be reused with each other – various combinations of the sequences can be run in various orders to simulate traffic variations or other combinations of legal input. Each sequence is a collection of legal stimulus.

With a given environment, sequences that are written to run on this environment can be picked from the command line or can be picked programmatically (randomly) to create more test cases.

For example, sequences of type "seq1" and "seq2" will be created from this command line:

```
<SIM CMD>  +SEQ=seq1 +SEQ=seq2 ...
```

In the environment, the run task is reduced to parsing the simulation command line using the UVM command line processor. For each of the +SEQ arguments on the command line, a sequence of that type name gets constructed, and started. Multiple +SEQ specifications execute one after the other, in order.

```
task run_phase(uvm_phase phase);
  seq_class seq;
  string list_of_sequences[$];
  uvm_cmdline_processor clp;

  clp = uvm_cmdline_processor::get_inst();
  if ( clp.get_arg_values("+SEQ=",
    list_of_sequences) == 0 ) begin
    `uvm_fatal(get_type_name(),
      "No sequences specified.")
  end

  phase.raise_objection(this);
  foreach (list_of_sequences[n]) begin
    $cast(seq, factory.create_object_by_name(
      list_of_sequences[n]));
    seq.start(null);
  end
  phase.drop_objection(this);
endtask
```

The command line language to specify sequences to run can be made much more complex; for example, supplying sequences to run in parallel. As the command line language grows, eventually, it is more efficient and less error prone to use the SystemVerilog constructs like fork/join to write complicated sequence invocations instead of using an ad-hoc command line description.

VII. INTERACTING WITH C CODE

There are a number of considerations when using C code with SystemVerilog. SystemVerilog is naturally threaded, C code is not. Using SystemVerilog with DPI-C [3] makes it easy to have threaded C code. Most C code will need some protection from threaded code issues.

Using C code with UVM Sequences is equivalent to using SystemVerilog DPI-C with SystemVerilog classes. Unfortunately DPI-C and classes don't get along well in SystemVerilog.

SystemVerilog DPI-C [1] defines both imports and exports. Imports are defined to be C entry points that are "imported" or made available for SystemVerilog to call. Exports are SystemVerilog entry points that are "exported" for C to call. Imports and exports must be declared from within a "static"

context; for example from within a module, an interface or a package scope. It is not legal to declare a DPI-C import or export in a class. This LRM restriction causes three cases to be considered for using DPI-C with classes.

The way that DPI-C is used from a class depends on what information is passed between C and SystemVerilog. Three cases are outlined below.

The examples below illustrate using DPI-C with a simple SystemVerilog class. This was done for simplicity to clearly show the concepts. The reader is encouraged to apply the ideas to a UVM based class, including UVM sequences.

DPI-C Usage	***Examples of functionality desired***
Simple import	C code is used by SystemVerilog to perform some calculation. This calculation is not specific to any instance, and the C code never calls back into the SystemVerilog code.
Simple import and export	Same as simple import, but with the addition that the C code may call back into the SystemVerilog class code.
C code bound in an interface	Flexible solution that allows a specific instance of a class to be bound to a specific instance of an interface. The interface can define a specific C interface to use.

Table 2 - DPI-C Functionality Desired

A. Simple import

In this case the SystemVerilog UVM sequence is used to call C code to calculate something. The C code has no notion of context or state. The SystemVerilog UVM sequence calls the C code with inputs and gets outputs. For example, calculating a checksum from an array of bytes or encoding bytes.

SystemVerilog Code:

```
import "DPI-C" context task c_calc(output int x);

class C;
  static int g_id;
  int id;

  function new();
    id = g_id++;
  endfunction

  task class_calc(output int x);
    c_calc(x);
  endtask

  task go();
    int x;
    for(int i = 0; i < 5; i++) begin
      class_calc(x);
      $display("x=%0d", x);
    end
  endtask
endclass
```

C code:

```
int x = 0;

int
c_calc(int *val) {
  *val = x++;
  return 0;
}
```

The C code ‘c_calc’ is called directly from the class based code.

B. Simple import and export

In this case the SystemVerilog UVM sequence is used to call C code to calculate something, and the C code may additionally call back into the SystemVerilog UVM sequence. In this case both DPI-C imports and DPI-C exports are used.

SystemVerilog Code:

```
typedef class C;

import "DPI-C" context task c_calc(int id,
                                   output int x);
export "DPI-C"         task calling_back;

C function_mapping[int];

function void dpi_register(C c, int id);
  function_mapping[id] = c;
endfunction

function C dpi_lookup(int id);
  return function_mapping[id];
endfunction

task calling_back(int id, output int x);
  C my_class_handle;
  my_class_handle = dpi_lookup(id);
  my_class_handle.class_calling_back(x);
endtask

class C;
  static int g_id;
  int id;

  function new();
    id = g_id++;
    dpi_register(this, id);
  endfunction

  task class_calc(output int x);
    c_calc(id, x);
  endtask

  task class_calling_back(output int x);
    x = id;
    #(10+id);
    $display("@%0t: classid=%0d", $time, id);
  endtask

  task go();
    int x;
    for(int i = 0; i < 5; i++) begin
      class_calc(x);
      $display("x=%0d", x);
    end
  endtask
endclass
```

C code:

```
#include "dpiheader.h"

int x = 0;

int
c_calc(int id, int *val) {
  int y;

  calling_back(id, &y);
  printf("Interface Id%0d -> y=%d", id, y);

  *val = x++;
  return 0;
}
```

The C code ‘c_calc’ is called directly from the class based code, just as the simple import code is above. But the C code also calls back into the SystemVerilog code, and calls a class based function. The C code does not have the ability to call the class function directly, instead calling a wrapper function which takes an index. The index is used to find the class handle, and forward the call to the class function. In this way the C code knows which class called it by keeping track of that index. [12]

C. C code bound in an interface

In this case the SystemVerilog UVM sequence is used to call C code, and the C code may have a state. The state represents the current condition in the C code model. For example, the C code may be modeling a state machine or a CPU with registers. Each C call from the UVM sequence causes the state to change.

The C code can interrogate the current scope using DPI-C functions such as svGetScope. Based on the current scope the C code can restore any necessary state. Additionally, svSetUserData and svGetUserData can be used to help manage scope specific state information.

SystemVerilog Code:

```
typedef class seq;

interface my_interface();
  import "DPI-C" context task c_calc(
                                output int x);
  export "DPI-C"         task intf_calling_back;
  export "DPI-C"         task tick;

  seq my_class_handle;

  function void init(seq s);
    s.vif = interface::self();
    my_class_handle = s;
  endfunction

  task tick(int tics_to_wait = 1);
    if (tics_to_wait <= 0)
      return;
    #(tics_to_wait);
  endtask

  task intf_calling_back(output int x);
    my_class_handle.class_calling_back(x);
  endtask
endinterface

class seq;
  static int g_id = 1;
  int id;
  virtual my_interface vif;

  function void init(virtual my_interface l_vif);
    id = g_id++;
    vif = l_vif;                // Assign VIF
    vif.my_class_handle = this; // VIF has a
                                //  handle to this
                                //  object.
  endfunction

  task class_calc(output int x);
    vif.c_calc(x);
  endtask

  task class_calling_back(output int x);
    x = id;
    #(10);
    $display("SV: @%0t: classid=%0d", $time, id);
  endtask

  task body();
    int x;
    for(int i = 0; i < 5; i++) begin
      class_calc(x);
      $display("SV: @%0t: x=%0d", $time, x);
    end
  endtask
endclass

module top();
  my_interface if0();
  my_interface if1();
  my_interface if2();

  initial begin
    automatic seq s = new();
    s.init(if0); // Connect
    s.body();
  end
  initial begin
    automatic seq s = new();
    s.init(if1); // Connect
    s.body();
  end
  initial begin
    automatic seq s = new();
    s.init(if2); // Connect
    s.body();
  end
endmodule
```

C Code:

```
#include "dpiheader.h"

int x = 0;

int
c_calc(int *val) {
  int id;
  svScope scope;

  scope = svGetScope();

  tick(10);

  // id is a variable stored one per scope.
  if ((id=(int)svGetUserData(scope, "id")) == 0) {
    intf_calling_back(&id);
    if (svPutUserData(scope, "id", id) != 0)
      printf("FATAL: svPutUserData() failed.\n");
  }

  printf("c_calc(%0d, %0d) called. Scope = %s\n",
    id, *val, svGetNameFromScope(svGetScope()));
```

```
  *val = x++;
  return 0;
}
```

SystemVerilog DPI-C has a notion of scope. It can be accessed using svGetScope(). This means that C code can find out what "scope" it is operating in. C code that is scope-aware can retrieve the scope specific data, perform updates or calculations and return these scope specific results.

SystemVerilog DPI-C also has a user data field that is available to store context or scope specific information. By using svGetUserData() and svSetUserData() scope specific information can be stored and retrieved.

In the example code above, a simple integer is stored per scope. When the code is called, it retrieves the instance specific integer using svGetUserData().

The example code above calls back into SystemVerilog using the example routine 'intf_calling_back()'. This is a function in the interface 'my_interface'. That function is a small wrapper that calls the class based version of the function.

The SystemVerilog interface is used as the DPI-C scope for a class instance. Each class instance will have its own instance of 'my_interface'. When the class is constructed, the interface and class swap handles to each other using the 'init()' routine. Initialization could be done in either the interface or in the class. This example does both as a demonstration.

VIII. ADVANCED SEQUENCES

Advanced sequences require more knowledge about the way the UVM sequencer, driver and sequences communicate with each other, and require detailed knowledge of the particular protocol being modeled. In the example below, no specific protocol is being modeled, but the basic handshake is simple READY/VALID signaling.

Interrupts, out-of-order completion and simple pipelined behavior will be described below. Pipelining here means two activities (like two READs outstanding at once). Both READs have been issued, but neither read has yet completed. This could have been accomplished with two sequences, each issuing a READ transaction.

A. Pipelined drivers

A pipelined driver must be able to service a first request, and then a second request before the first request is complete. A driver could receive the first request, and place it immediately on a work-list. After the first request is on the work-list, the driver could receive a second request, and place it on the work-list as well. This work-list will accumulate outstanding transactions. These accumulated transactions can be serviced with a simple process that checks to see if there is a transaction on the work-list. If there is a transaction on the work-list, then the transaction is removed from the work-list, and "executed" on the bus. The execution on the bus may require multiple clocks or multiple wire signaling. This execution is the low-level bus protocol wiggling pins. Whether in-order or out of order pipelining, the work-list is used to allow the driver to not block waiting for execution of the transaction. The driver remains active at all times, receiving all requests. The driver code may be written to choke the flow of requests at some point, restricting the number of active outstanding transactions.

An in-order protocol relies on the fact that each response or completion will be in the same order as the request are issued. The matching up of requests and responses is therefore implicit in the ordering of the requests.

An out-of-order protocol cannot rely on implicit ordering, and must find a way to associate the response with the originating request. Many out-of-order systems use a tag – a simple integer – which is used to mark the request and the response. Then the driver can maintain a list of outstanding transactions indexed by the tag. When a response is returned the index can be used to find the originating request.

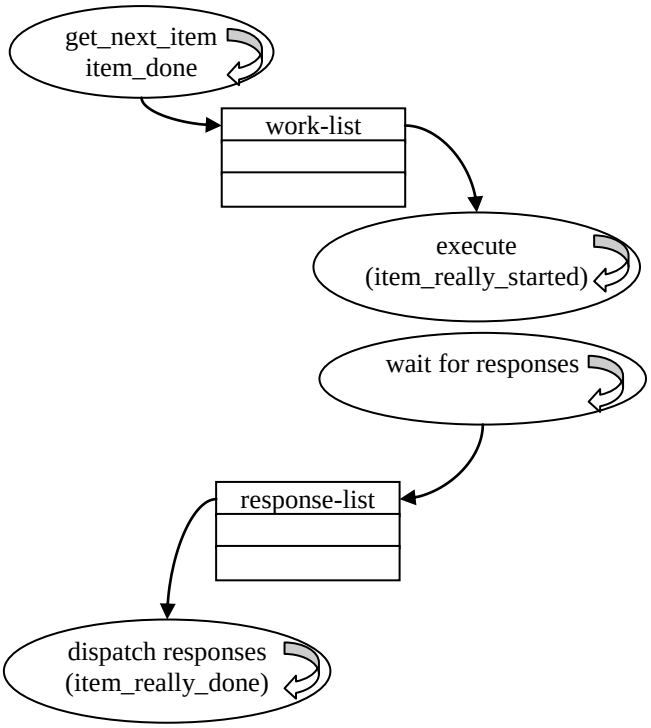


Figure 4 - Out-of-order request/response handling

A pipelined driver therefore may contain multiple processes. The first process fills the work-list, and the second process removes item from the work-list and performs the actual execution of the transactions.

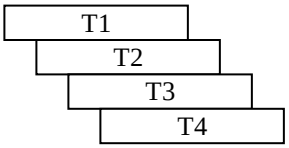


Figure 5 - Pipelined transaction - in-order completion

A pipelined driver expects that a first transaction is submitted, and before completion, a second transaction is submitted.

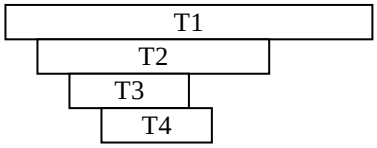


Figure 6 - Pipelined transactions - out-of-order completion

An out-of-order driver is a bit more complicated, since it cannot determine the order of responses, so it must manage a response list, and have processes which fill the response list and empty it.

B. Pipelined sequence

In order for pipelined stimulus to be created, the sequence must cause multiple start_item calls. Whether from a single sequence or multiple sequences, then net effect is that the driver sees many start_item calls. Each start_item call represents a transaction which may be pipelined.

The driver must signal the sequence immediately by calling item_done, so that the sequencer "releases" the current transaction, and another can be received. If item_done is not called, then the driver/sequencer communication will stop, and new pipelined transaction can be received.

```
class my_transaction_base extends
    uvm_sequence_item;
  `uvm_object_utils(my_transaction_base)

  bit item_really_started_e;
  bit item_really_done_e;

  function void item_really_started();
    item_really_started_e = 1;
  endfunction

  function void item_really_done();
    item_really_done_e = 1;
  endfunction
endclass
```

All transactions should extend the my_transaction above. It implements the item_really_done and item_really_started. Item_really_started is used to synchronize when a transaction has actually been presented to the device under test, not when the driver has received the transaction from the sequence.

Item_really_done is used to cause finish_item to wait until the transaction is really done, and not simply when the driver calls item_done.

All sequences should extend the my_sequence below. The finish_item task is re-implemented to provide proper transaction recording. Finish_item calls super.finish_item, which will cause the sequencer/driver communication to operate, but then it waits for item_really_started and records the transaction start. It then waits for item_really_done and records the transaction end. When compiling the UVM, the define `UVM_DISABLE_AUTO_ITEM_RECORDING` should be used.

```
class my_sequence_base #(type T = int)
    extends uvm_sequence#(T);
  `uvm_object_param_utils(my_sequence_base#(T))

  task finish_item(uvm_sequence_item item,
                   int set_priority = -1);
    uvm_sequencer_base sequencer;
    T trans_t;

    super.finish_item(item);

    if ($cast(trans_t, item)) begin
      // This makes pipelining work
      wait(trans_t.item_really_started_e == 1);

`ifdef UVM_DISABLE_AUTO_ITEM_RECORDING
      if(sequencer == null) sequencer =
        item.get_sequencer();
      if(sequencer == null) sequencer =
        get_sequencer();

      void'(sequencer.begin_child_tr(
        item, m_tr_handle,
        item.get_root_sequence_name()));
`endif

      // This makes pipelining work
      wait(trans_t.item_really_done_e == 1);
    end

`ifdef UVM_DISABLE_AUTO_ITEM_RECORDING
    sequencer = item.get_sequencer();
    sequencer.end_tr(item, $time);
`endif
  endtask
endclass
```

C. Out-of-order completion

Out-of-order completion is a natural extension of the pipelined mode. In the pipelined mode above, transactions are completed inorder. This allows for parallel operation, but does not allow maximum throughput when there are operations that complete in different amounts of time.

Out-of-order completion requires one addition to the pipelined implementation – a tag or id. Each transaction must be identified by a tag or id, so that requests and responses can be connected together. Request 1 and 2 are sent to the driver and on to the hardware. Response 2 comes back first, followed by response 1. The driver can connect the out-of-order responses with the requests using the tag or id.

```
class trans extends my_transaction_base;
  `uvm_object_utils(trans)

  static int g_id = 1;
  int        id;
  int        finished_id;
  bit [7:0]  output_value;

endclass
```

The transaction extends my_transaction_base, enabling item_really_started and item_really_done processing.

The sequence below extends my_sequence_base, enabling the enhanced finish_item.

```
class sequenceN extends my_sequence_base#(trans);
  `uvm_object_utils(sequenceN)

  rand int n;
  constraint val { n > 10; n < 100; }

  task body();
    for(int i = 0; i < n; i++) begin
      fork
        automatic int j = i;
        begin
          trans t;
          t = trans::type_id::create(
            $sformatf("t_%0d", i));
          t.set_name($sformatf("tr%0d", t.id));
          if (!t.randomize()) begin
            `uvm_fatal("SEQ1", "Randomize failed")
          end
          start_item(t);
          finish_item(t);
          if (t.id+1 != t.output_value)
            `uvm_fatal("SEQ", "DUT Failed")
          else
            `uvm_info("SEQ",
              $sformatf("tr%0d matches", t.id),
              UVM_MEDIUM)
        end
      join_none
    end
    // Don't finish until all the threads are done
    wait fork;
  endtask
endclass
```

The sequence above tries to generate all transactions inside a fork/join none. A more realistic example would generate only the transactions needed or that were possible to pipeline.

```
class driver extends uvm_driver#(trans);
  `uvm_component_utils(driver)

  virtual dut_if vif;

  // Mapping from tag to transaction handle.
  // Used to find the transaction handle for
  //   a tag response
  // 't_outstanding' is the list of outstanding
  //   transactions.
  trans               t_outstanding[256];

  // Requests ------------------------
  // List (mailbox) of tags that are to be
  //   processed.
  mailbox #(bit[7:0]) tags_to_be_executed_mb;


  // Responses -----------------------
  // List (mailbox) of tags that are completed.
  mailbox #(bit[7:0]) tags_that_have_completed_mb;

  function void build_phase(uvm_phase phase);
    // Infinite sized mailboxes.
    tags_that_have_completed_mb = new();
    tags_to_be_executed_mb      = new();
  endfunction
```

The run_phase starts three helper processes (execute_requests, service_interrupts and dispatch_responses). It also has its own process, which is a traditional forever loop that does a get_next_item and an item_done. The get_next_item receives a transaction and immediately puts it in a mailbox (tags_to_be_executed_mb).

```
  task run_phase(uvm_phase phase);
    fork
      execute_requests();
      service_interrupts();
      dispatch_responses();
    join_none

    forever begin
      trans t;
      bit [7:0] tag;

      seq_item_port.get_next_item(t);

      tag = t.id; // Truncates 32 bits to 8.
      t_outstanding[tag] = t;

      seq_item_port.item_done(); // Immediate
                                 // item_done!

      tags_to_be_executed_mb.put(tag);
    end
  endtask
```

The execute_requests process simply does a get from the tags_to_be_executed mailbox and then calls the BFM to execute the transaction.

```
  task execute_requests();
    trans t;
    bit [7:0] tag;

    forever begin
      // Get a tag to be executed.
      tags_to_be_executed_mb.get(tag);

      // What's the transaction handle?
      t = t_outstanding[tag];

      // Execute.
      t.item_really_started();
      vif.execute(t.id);
    end
  endtask
```

The service_interrupts process simply waits for an interrupt and then finds the tag and transaction handle that was the original tagged request. Once it finds the original request transaction it fills in any results that are available. This is the response for the transaction. Once the response is filled in, then the transaction goes on the tags_that_have_completed mailbox.

```
  task service_interrupts();
    bit [7:0] tag;
    bit [7:0] output_value;

    forever begin
      // Wait for an interrupt.
      vif.handle_interrupt(tag, output_value);

      // Figure out which trans this tag
      //   represents
      t = t_outstanding[tag];

      // Get the data from the interrupt /
      //   response. This is the response
      //   we'll send back.
      t.output_value = output_value;

      tags_that_have_completed_mb.put(tag);
    end
  endtask
```

The dispatch_responses process retrieves a completed tag from the tags_that_have_completed mailbox. The transactions'

item_really_done is called, finally releasing the finish_item call from the sequence.

```
  task dispatch_responses();
    trans t;
    bit [7:0] tag;

    forever begin
      // Get the "response tags"
      tags_that_have_completed_mb.get(tag);

      // Find out WHICH transaction this tag is.
      t = t_outstanding[tag];

      // Remove it from the table.
      t_outstanding[tag] = null;

      // Signal 'item_really_done()'
      t.item_really_done();
    end
  endtask
endclass
```

Depending on the protocol and the transaction types the threads, processes, mailboxes and handling will vary.

IX. SEQUENCES FOR AXI4

Stimulus for AXI4 [11] can be modeled using sequences. The AXI4 protocol is a complex protocol with many options and controls, and multiple layers of communication. A full discussion of AXI4 and the sequences, controls and options related to AXI4 is beyond the scope of this paper.

An AXI4 transfer consists of multiple transfers on various channels. A read transfer involves the read address channel and the read data channel. A write transfer involves the write address channel, the write data channel and the write response channel.

A simple way to view the AXI4 protocol with respect to sequences is to create transaction level sequences – read and write for example. The read and write transactions are made up of lower level channel (phase) transactions. A read transaction is implemented as a sequence which will start a read address sequence. Then it starts a read data sequence waiting for the read result. The write transaction is implemented as a sequence which starts a write address sequence and a write data sequence. Then is waits for the write response by using a write response sequence.

Each of the sequences involved above has options and controls. For example, AXI4 has burst mode and single transfer. It has burst length and byte enables. An AXI transaction has many controls and can be constrained in many interesting ways. Using these controls and contraints allows the entire protocol space to be explored. Any set of sequences created must be built to include all the possible combinations – legal and illegal. This sequence c

SUMMARY

Sequences are a powerful tool for stimulus generation and provide a consistent, extensible interface against which to write tests. Sequences can be randomized, they can be overridden in the factory and they can be used to create arbitrarily complex tests.

Any UVM test writer is really a UVM sequence writer. Sequence writing is the future in UVM testbench development.

ACKNOWLEDGMENT

R.E. thanks Dave Rich for his input and thoughts about the best ways to talk about sequences and Adam Rose for the discussion around item_really_done. Hopefully we'll talk soon about item_really_started.

REFERENCES

[1] SystemVerilog LRM - http://standards.ieee.org/findstds/standard/1800-2012.html

[2] UVM Accellera Standard - http://www.accellera.org/downloads/-standards/uvm

[3] Rich Edelman and Doug Warmke, "Using SystemVerilog DPI Now", DVCON 2005

[4] Rich Edelman and Alain Gonier, "Easier Sequences – SystemVerilog UVM Sequence and Task Equivalence", IPSOC 2012

[5] Function Objects. http://wikipedia.org/wiki/Function_object

[6] Functor. http://wikipedia.org/wiki/Functor

[7] Rich Edelman, "Sequences In SystemVerilog", DVCON 2008

[8] Rich Edelman, "You Are In a Maze of Twisty Little Sequences, All Alike – or Layering Sequences for Stimulus Abstraction", DVCON 2010

[9] Mark Peryer, "There's something wrong between Sally Sequencer and Dirk Driver – why UVM sequencers and drivers need some relationship counselling", DVCON 2012

[10] English Letter Frequency - http://wikipedia.org/wiki/Letter_frequency

[11] AMBA AXI4 – www.arm.com

[12] Mark Peryer, "Command Line Debug Using UVM Sequences", DVCON 2011

APPENDIX

Please contact the authors for source code of the examples.

A. *Using DPI-C with classes*

1) *Simple Import Only*

```
// ============================================
// File simple-import-only/t.sv
// ============================================
  import "DPI-C" context task c_calc(output int x);

class C;
  static int g_id;
  int id;

  function new();
    id = g_id++;
  endfunction

  task class_calc(output int x);
    c_calc(x);
  endtask

  task go();
        int x;
        for(int i = 0; i < 5; i++) begin
          class_calc(x);
          $display("x=%0d", x);
        end
  endtask
endclass

module top();
  initial begin
    automatic C c = new(); c.go();
  end
  initial begin
    automatic C c = new(); c.go();
  end
  initial begin
    automatic C c = new(); c.go();
  end
endmodule


// ============================================
// File simple-import-only/t.c
// ============================================
#include "dpiheader.h"

int x = 0;

int
c_calc(int *val) {
  *val = x++;
  return 0;
}
```

2) *Simple Import Export*

```
// ============================================
// File simple-import-export/t.sv
// ============================================
typedef class C;

  import "DPI-C" context task c_calc(int id,
                                     output int x);
  export "DPI-C"         task calling_back;

  C function_mapping[int];

  function void dpi_register(C c, int id);
    function_mapping[id] = c;
  endfunction

  function C dpi_lookup(int id);
    return function_mapping[id];
  endfunction

  task calling_back(int id, output int x);
        C my_class_handle;
        my_class_handle = dpi_lookup(id);
    my_class_handle.class_calling_back(x);
  endtask

class C;
  static int g_id;
  int id;

  function new();
    id = g_id++;
        dpi_register(this, id);
  endfunction

  task class_calc(output int x);
    c_calc(id, x);
  endtask

  task class_calling_back(output int x);
    x = id;
        #(10+id);
        $display("@%0t: classid=%0d", $time, id);
  endtask

  task go();
        int x;
        for(int i = 0; i < 5; i++) begin
          class_calc(x);
          $display("x=%0d", x);
        end
  endtask
endclass

module top();
  initial begin
    automatic C c = new(); c.go();
  end
  initial begin
    automatic C c = new(); c.go();
  end
  initial begin
    automatic C c = new(); c.go();
  end
endmodule


// ============================================
// File simple-import-export/t.c
// ============================================
#include "dpiheader.h"

int x = 0;

int
c_calc(int id, int *val) {
  int y;

  calling_back(id, &y);
  printf("Interface Id%0d -> y=%d", id, y);

  *val = x++;
  return 0;
}
```

3) *Bound to an Interface*

```
// ============================================
// File bound-to-an-interface/t.sv
// ============================================
typedef class seq;

interface my_interface();
  import "DPI-C" context task c_calc(output int x);
  export "DPI-C"         task intf_calling_back;
  export "DPI-C"         task tick;

  seq my_class_handle;

  function void init(seq s);
    s.vif = interface::self();
        my_class_handle = s;
  endfunction

  task tick(int tics_to_wait = 1);
        if (tics_to_wait <= 0)
          return;
    #(tics_to_wait);
  endtask

  task intf_calling_back(output int x);
        // virtual my_interface vif;
        // vif = interface::self();
    my_class_handle.class_calling_back(x);
```

```
  endtask
endinterface
class seq;
  static int g_id = 1;
  int id;
  virtual my_interface vif;

  function void init(virtual my_interface l_vif);
    id = g_id++;
    vif = l_vif;
        vif.my_class_handle = this;
  endfunction

  task class_calc(output int x);
    vif.c_calc(x);
  endtask

  task class_calling_back(output int x);
    x = id;
        #(10);
        $display("SV: @%0t: classid=%0d", $time, id);
  endtask

  task body();
        int x;
        for(int i = 0; i < 5; i++) begin
          class_calc(x);
          $display("SV: @%0t: x=%0d", $time, x);
        end
  endtask
endclass

module top();
  my_interface if0();
  my_interface if1();
  my_interface if2();

  initial begin
    automatic seq s = new();
    s.init(if0); /*or if0.init(s);*/ s.body();
  end
  initial begin
    automatic seq s = new();
    s.init(if1); /*or if1.init(s);*/ s.body();
  end
  initial begin
    automatic seq s = new();
    s.init(if2); /*or if2.init(s);*/ s.body();
  end
endmodule

// ================================================
// File bound-to-an-interface/t.c
// ================================================
#include "dpiheader.h"

int x = 0;

int
c_calc(int *val) {
  int id;
  svScope scope;

  scope = svGetScope();

  tick(10);

  if ((id=(int)svGetUserData(scope, "id")) == 0) {
    intf_calling_back(&id);
    if ((int)svPutUserData(scope, "id", (int)id) != 0)
          printf("FATAL: svPutUserData() failed.\n");
  }

  printf(" C: %s: id=%0d\n",
    svGetNameFromScope(scope), id);

  *val = x++;
  return 0;
}
```

B. *Grab/Lock and Arbitration*

```
import uvm_pkg::*;
`include "uvm_macros.svh"

class trans extends uvm_sequence_item;
  `uvm_object_utils(trans)
  rand bit rw;
  rand bit [7:0] addr;
  rand bit [7:0] data;
       int       qos;

  constraint addr_value {
    addr > 0;
    addr < 100;
    addr[1:0] == 0;
  }

  constraint data_value {
    data > 0;
    data < 100;
  }

  function new(string name = "test");
    super.new(name);
  endfunction

  function string convert2string();
    return $sformatf("%s(a=%3d, d=%3d) [qos=%0d]",
      (rw==1)?" READ":"WRITE",
      addr, data, qos);
  endfunction
endclass


class read_seq extends uvm_sequence#(trans);
  `uvm_object_utils(read_seq)
  bit [7:0] addr; // Input
  bit [7:0] data; // Output

  function new(string name = "test");
    super.new(name);
  endfunction

  task body();
    trans t;
    t = trans::type_id::create("t",, get_full_name());
    t.rw = 1;
    t.addr = addr;
    start_item(t);
    finish_item(t);
    data = t.data;
  endtask
endclass

class write_seq extends uvm_sequence#(trans);
  `uvm_object_utils(write_seq)
  bit [7:0] addr; // Input
  bit [7:0] data; // Input

  function new(string name = "test");
    super.new(name);
  endfunction

  task body();
    trans t;
    t = trans::type_id::create("t",, get_full_name());
    t.rw = 0;
    t.addr = addr;
    t.data = data;
    start_item(t);
    finish_item(t);
  endtask
endclass

class seq extends uvm_sequence#(trans);
  `uvm_object_utils(seq)

  int grabbing = 0;
  int locking = 0;
  int qos = 100; // default is 100 in the UVM

  rand int n;

  constraint val { n > 10; n < 30; }

  bit [7:0] addr_used[bit[7:0]];
```

```
  function new(string name = "test");
    super.new(name);
  endfunction

  task body();
    bit [7:0] expected_data;
    trans t;
    `uvm_info("seq", $sformatf(
      "Starting. n=%0d qos=%0d, grab=%0d **********",
      n, qos, grabbing), UVM_MEDIUM)

    if (grabbing) grab();
    if (locking) lock();

    for (int i=0; i<n; i++) begin
      t = trans::type_id::create(
        "t",, get_full_name());
      start_item(t, qos);
      if (!t.randomize() with {
          t.rw == 0; // Always do a WRITE first.
          !(t.addr inside { addr_used });
          } )
          `uvm_fatal("seq", "Randomize failed")
      t.qos = qos;
      `uvm_info("seq", $sformatf("Sending W %s",
        t.convert2string()), UVM_MEDIUM)
      finish_item(t);

      // Remember this address.
      addr_used[t.addr] = t.addr;
      expected_data = t.data;

      // Now do a READ.
      t.rw = 1;
      start_item(t, qos);
      `uvm_info("seq", $sformatf("Sending R %s",
        t.convert2string()), UVM_MEDIUM)
      finish_item(t);

      // Now CHECK.
      if (t.data != expected_data)
        `uvm_error("seq",
          $sformatf("Mismatch: expected %3d, got %3d",
          expected_data, t.data))
    end

    if (locking) unlock();
    if (grabbing) ungrab();

    `uvm_info("seq",
      $sformatf("Finished. n=%0d **********", n),
      UVM_MEDIUM)
  endtask
endclass

class driver extends uvm_driver#(trans);
  `uvm_component_utils(driver)

  bit [7:0] mem [256] = '{default:0};

  function void dump();
    $display("====================================");
    for(int addr = 0; addr < 256; addr+=8) begin
      $display("%3x: %3x %3x %3x %3x %3x %3x %3x %3x",
        addr,
        mem[addr],   mem[addr+1],
        mem[addr+2], mem[addr+3],
        mem[addr+4], mem[addr+5],
        mem[addr+6], mem[addr+7]);
    end
    $display("====================================");
  endfunction

  function new(string name = "test",
      uvm_component parent = null);
    super.new(name, parent);
  endfunction

  task run_phase(uvm_phase phase);
    trans t;
    forever begin
      seq_item_port.get_next_item(t);

      if (t.rw == 0) // WRITE
        mem[t.addr] = t.data;
      else           // READ
        t.data = mem[t.addr];

      `uvm_info("run",
        $sformatf("Got %s", t.convert2string()),
        UVM_MEDIUM)

      #10;

      seq_item_port.item_done();
    end
  endtask
endclass

class env extends uvm_env;
  `uvm_component_utils(env)

  driver d;
  uvm_sequencer#(trans) seqr;

  function new(string name = "test",
      uvm_component parent = null);
    super.new(name, parent);
  endfunction

  function void build_phase(uvm_phase phase);
    d   =driver              ::type_id::create("d",this);
    seqr=uvm_sequencer#(trans)::type_id::create(
      "seqr", this);
    seqr.set_arbitration(SEQ_ARB_STRICT_FIFO);
  endfunction

  function void connect_phase(uvm_phase phase);
    d.seq_item_port.connect(seqr.seq_item_export);
  endfunction

  task read(input bit [7:0] addr, output bit [7:0] data);
    read_seq seq;
    seq = read_seq::type_id::create(
      "read",, get_full_name());
    seq.addr = addr;
    seq.start(seqr);
    data = seq.data;
  endtask

  task write(input bit [7:0] addr, input bit [7:0] data);
    write_seq seq;
    seq=write_seq::type_id::create(
      "write",, get_full_name());
    seq.addr = addr;
    seq.data = data;
    seq.start(seqr);
  endtask
endclass

class test extends uvm_test;
  `uvm_component_utils(test)

  env e;

  function new(string name = "test",
      uvm_component parent = null);
    super.new(name, parent);
  endfunction

  function void build_phase(uvm_phase phase);
    e = env::type_id::create("e", this);
  endfunction

  task run_phase(uvm_phase phase);

    factory.print();
    phase.raise_objection(this);
    fork
      // ------------------------------------------------
      for (int i=0; i < 3; i++) begin // Normal
        seq s1;
        s1 = seq::type_id::create("s1",,
         get_full_name());
        if (!s1.randomize())
          `uvm_fatal("seq", "Randomize failed for s1")
        s1.start(e.seqr);
      end
      // ------------------------------------------------
      for (int i=0; i < 3; i++) begin // QOS == 1000
        seq s2;
        s2 = seq::type_id::create("s2",,
         get_full_name());
        s2.qos = 1000;
        if (!s2.randomize())
```

```
        `uvm_fatal("seq", "Randomize failed for s2")
      s2.start(e.seqr);
    end
    // ---------------------------------------------
    for (int i=0; i < 3; i++) begin // GRABBING
      seq s3_grabbing;
      s3_grabbing = seq::type_id::create(
        "s3_grabbing",, get_full_name());
      s3_grabbing.grabbing = 1;
      if (!s3_grabbing.randomize())
        `uvm_fatal("seq",
          "Randomize failed for s3_grabbing")
      s3_grabbing.start(e.seqr);
    end
    // ---------------------------------------------
    for (int i=0; i < 3; i++) begin // LOCKING
      seq s4_locking;
      s4_locking = seq::type_id::create(
        "s4_locking",, get_full_name());
      s4_locking.locking = 1;
      if (!s4_locking.randomize())
        `uvm_fatal("seq",
          "Randomize failed for s4_locking")
      s4_locking.start(e.seqr);
    end
  join

  `uvm_info(get_type_name(),
    "Starting task based tests", UVM_MEDIUM)

  for (int addr = 0; addr < 256; addr++) begin
    bit [7:0] data;
    bit [7:0] rdata;
    data = addr+1;
    e.write(addr,  data);
    e.read (addr, rdata);
    if (data != rdata)
      `uvm_fatal("TASK", $sformatf(
      "Addr=%0xx, Compare failed wdata=%0x, rdata=%0x",
        addr, data, rdata))
  end
  phase.drop_objection(this);
 endtask
endclass

module top();
 initial
   run_test("test");
endmodule
```

C. *Pipeling and Out-of-order transactions*

```
import uvm_pkg::*;
`include "uvm_macros.svh"

interface dut_if (input wire clk);

  bit       go;
  bit [7:0] itag;
  bit       interrupt;
  bit [7:0] otag;
  bit [7:0] output_value;

  // Keeps track of transactions that are running.
  // Two transactions with the same tag will
  // never be running at the same time.
  bit task_outstanding[256]; // Same size as the
                             //   tag has bits.

  task handle_interrupt(output bit [7:0] tag,
                        output bit [7:0] o_value);

    @(posedge interrupt);
    // Signal that this slot is open.
    // The task has completed.
    task_outstanding[otag] = 0;

    // Grab the response
    tag = otag;
    o_value = output_value;
    // Return -> this is an interrupt that
    //   should be serviced.
  endtask

  task execute(bit [7:0] tag);

    if (task_outstanding[tag] == 1) begin
       wait(task_outstanding[tag] == 0);
    end
    task_outstanding[tag] = 1;
    @(posedge clk);

    // Transfer 'tag' across the interface.
    itag = tag;

    go = 1;
    @(posedge clk);
    go = 0;
    @(posedge clk);
  endtask
endinterface


module dut( input bit      clk,
            input bit      go,
            input bit[7:0] itag,
           output bit      interrupt,
           output bit[7:0] otag,
           output bit[7:0] output_value);

  int s;

  initial
    // Creates a stream named /top/dut_i/DUT
    s = $create_transaction_stream("DUT", "kind");

  task automatic process(bit[7:0]tag);
    int tr;
    int dut_delay;
    int wait_time;

    dut_delay = $urandom_range(1000, 200);

    // Transaction Begin
    tr = $begin_transaction(s, $sformatf("tr%0d", tag));
    $add_attribute(tr, tag, "tag");
    $add_attribute(tr, $time, "start_time");
    $add_attribute(tr, dut_delay, "dut_delay");
    // -----------------

    #dut_delay;

    // Calculate the DUT function (+1)
    output_value = tag+1;

    // Put the calculated value on the interface
    otag = tag;
```

```
    // -----------------
    $add_attribute(tr, output_value, "output_value");
    $add_attribute(tr, $time, "end_time");
    $end_transaction(tr);
    // Transaction End

    // Notify the system, that this calculation is done
    wait_time = $time;
    tr = $begin_transaction(s,
      $sformatf("int_svc_wait_tr%0d", tag));
    $add_color(tr, "pink");

    // Cause an interrupt.
    wait(interrupt == 0);
    interrupt = 1;
    @(posedge clk);

    wait_time = $time - wait_time;
    $add_attribute(tr, wait_time, "wait_time");
    $end_transaction(tr);

    interrupt = 0;
    @(posedge clk);

  endtask

  always @(posedge clk) begin
    if (go == 1) begin
      fork
        automatic bit[7:0] my_tag = itag;
        process(my_tag);
      join_none
    end
  end
endmodule

// ----------------------------------------

class my_transaction_base extends uvm_sequence_item;
  `uvm_object_utils(my_transaction_base)

  bit item_really_started_e; // Set to 1, when we're
                             //  started.
                             // It's never set to 0,
                             // since once we're started,
                             // we're started

  bit item_really_done_e; // Set to 1, when we're done.
                          // It's never set to 0, since
                          // once we're done, we're done

  function void item_really_started();
    item_really_started_e = 1;
  endfunction

  function void item_really_done();
    item_really_done_e = 1;
  endfunction

  function new(string name = "my_transaction_base");
    super.new(name);
  endfunction
endclass

class my_sequence_base #(type T = int)
    extends uvm_sequence#(T);
  `uvm_object_param_utils(my_sequence_base#(T))

  function new(string name = "my_sequence_base#(T)");
    super.new(name);
  endfunction

  virtual task start_item (uvm_sequence_item item,
                           int set_priority = -1,
                           uvm_sequencer_base sequencer
                                              =null);
    super.start_item(item, set_priority, sequencer);
  endtask

  task finish_item(uvm_sequence_item item,
                                int set_priority = -1);
    uvm_sequencer_base sequencer;
    T trans_t;

    super.finish_item(item);

    if ($cast(trans_t, item)) begin
      // This makes pipelining work
      wait(trans_t.item_really_started_e == 1);

`ifdef UVM_DISABLE_AUTO_ITEM_RECORDING
      if(sequencer == null) sequencer =
        item.get_sequencer();
      if(sequencer == null) sequencer = get_sequencer();

      void'(sequencer.begin_child_tr(
        item, m_tr_handle,
        item.get_root_sequence_name()));
`endif

      // This makes pipelining work
      wait(trans_t.item_really_done_e == 1);
    end

`ifdef UVM_DISABLE_AUTO_ITEM_RECORDING
    sequencer = item.get_sequencer();
    sequencer.end_tr(item, $time);
`endif
  endtask
endclass

// ----------------------------------------

class trans extends my_transaction_base;
  `uvm_object_utils(trans)

  static int g_id = 1;
  int        id;
  int        finished_id;
  bit [7:0]  output_value;

  function new(string name = "trans");
    super.new(name);
    id = g_id++;
  endfunction

  function string convert2string();
    return $sformatf(
      "tr%0d id=%0d finished_id=%0d, output_value=%0d",
         id,id,    finished_id,     output_value);
  endfunction

  function void do_record(
      uvm_recorder recorder = uvm_default_recorder);

    $add_attribute(recorder.tr_handle, id, "id");
    $add_attribute(
      recorder.tr_handle, finished_id, "finished_id");
    $add_attribute(
      recorder.tr_handle, output_value, "output_value");
  endfunction
endclass


class sequenceN extends my_sequence_base#(trans);
  `uvm_object_utils(sequenceN)

  rand int n;

  constraint val { n > 10; n < 100; }

  function new(string name = "sequenceN");
    super.new(name);
  endfunction

  task body();
    for(int i = 0; i < n; i++) begin
      fork
        automatic int j = i;
        begin
          trans t;
          t = trans::type_id::create(
            $sformatf("t_%0d", i));
          t.set_name($sformatf("tr%0d", t.id));
          if (!t.randomize()) begin
            `uvm_fatal("SEQ1", "Randomize failed")
          end
          start_item(t);
          finish_item(t);
          if (t.id+1 != t.output_value)
            `uvm_fatal("SEQ", "DUT Failed")
          else
            `uvm_info("SEQ",
              $sformatf("tr%0d matches", t.id),
              UVM_MEDIUM)
```

```
      end
    join_none
  end
  // Don't finish until all the threads are done
  wait fork;
endtask

function void do_record(uvm_recorder recorder);
  super.do_record(recorder);
  $add_attribute(recorder.tr_handle, n, "n");
endfunction
endclass

// ----------------------------------------

class driver extends uvm_driver#(trans);
  `uvm_component_utils(driver)

  virtual dut_if vif;

  function new(string name = "driver",
      uvm_component parent = null);
    super.new(name, parent);
  endfunction

  // Mapping from tag to transaction handle.
  // Used to find the transaction handle for a
  //   tag response
  // 't_outstanding' is the list of outstanding
  //   transactions.
  trans               t_outstanding[256];

  // Requests ------------------------
  // List (queue) of tags that are to be processed.
  mailbox #(bit[7:0]) tags_to_be_executed_mb;


  // Responses ------------------------
  // List (queue) of tags that are completed.
  mailbox #(bit[7:0]) tags_that_have_completed_mb;

  function void build_phase(uvm_phase phase);
    tags_to_be_executed_mb      = new();
    tags_that_have_completed_mb = new();
  endfunction

  task run_phase(uvm_phase phase);
    fork
      execute_requests();
      service_interrupts();
      dispatch_responses();
    join_none

    forever begin
      trans t;
      bit [7:0] tag;

      seq_item_port.get_next_item(t);
      `uvm_info("DRVR",
        {" Got ", t.convert2string()}, UVM_MEDIUM)
      tag = t.id; // Truncates 32 bits to 8.
      t_outstanding[tag] = t;
      seq_item_port.item_done(); // Immediate item_done!

      tags_to_be_executed_mb.put(tag);
    end
  endtask

  task execute_requests();
    trans t;
    bit [7:0] tag;

    forever begin
      // Get a tag to be executed.
      tags_to_be_executed_mb.get(tag);

      // What's the transaction handle?
      t = t_outstanding[tag];

      if (t == null)
        `uvm_fatal("execute_requests",
          "Null transaction")
      if (t.id != tag)
        `uvm_fatal("execute_requests", "Tag mismatch")

      // Execute.
      // The transaction finally has a chance to
      //   run on the hardware.
      t.item_really_started();
      vif.execute(t.id);
    end
  endtask

  task service_interrupts();
    bit [7:0] tag;
    bit [7:0] output_value;
    trans t;

    forever begin
      // Wait for an interrupt / response.
      vif.handle_interrupt(tag, output_value);

      // Figure out which trans this tag represents
      t = t_outstanding[tag];

      // Get the data from the interrupt / response.
      // This is the response we'll send back
      t.output_value = output_value;

      tags_that_have_completed_mb.put(tag);
    end
  endtask

  task dispatch_responses();
    trans t;
    bit [7:0] tag;

    forever begin
      // Get the "response tags"
      tags_that_have_completed_mb.get(tag);

      // Find out WHICH transaction this tag is.
      t = t_outstanding[tag];

      // Remove it from the table.
      t_outstanding[tag] = null;

      // Signal 'item_really_done()'
      // The transaction is done.
      t.item_really_done();
    end
  endtask
endclass

class env extends uvm_env;
  `uvm_component_utils(env)

  driver d;
  uvm_sequencer#(trans) sqr;

  virtual dut_if vif;

  function new(string name = "env",
      uvm_component parent = null);
    super.new(name, parent);
  endfunction

  function void build_phase(uvm_phase phase);
    // This env is connected to what VIF?
    if (!uvm_config_db#(virtual dut_if)::get(
        null, "", "vif", vif))
      `uvm_fatal("DRVR", "Cannot find VIF!")

    d = driver::type_id::create("driver", this);
    sqr = uvm_sequencer#(trans)::type_id::create(
      "sequencer", this);
  endfunction

  function void connect_phase(uvm_phase phase);
    // Share the vif handle with the driver.
    d.vif = vif;
    d.seq_item_port.connect(sqr.seq_item_export);
  endfunction
endclass

class test extends uvm_test;
  `uvm_component_utils(test)

  env e;

  function new(string name = "test",
      uvm_component parent = null);
    super.new(name, parent);
  endfunction

  function void build_phase(uvm_phase phase);
```

```
    e = env::type_id::create("e", this);
  endfunction

  task run_phase(uvm_phase phase);
    sequenceN seq;
    phase.raise_objection(this);
    seq = sequenceN::type_id::create("sequenceN");
    if (!seq.randomize())
      `uvm_fatal("TEST",
        "Randomization failed for sequenceN")
    seq.start(e.sqr);
    phase.drop_objection(this);
  endtask
endclass

module top();
  reg clk;

  dut_if dut_if_i(
    .clk(clk));

  dut    dut_i(
    .clk(clk),
    .go(dut_if_i.go),
    .itag(dut_if_i.itag),
    .interrupt(dut_if_i.interrupt),
    .otag(dut_if_i.otag),
    .output_value(dut_if_i.output_value)
  );

  initial begin
    // Streams in use:
    //   /uvm_root/uvm_test_top/e/sequencer/sequenceN
    //   /top/dut_i/DUT
    uvm_config_db#(int)           ::set(
      null, "", "recording_detail", 1);
    uvm_config_db#(virtual dut_if)::set(
      null, "", "vif",              dut_if_i);

    run_test("test");
  end

  always begin
    #10 clk = 1;
    #10 clk = 0;
  end
endmodule
```

D. *Packet Example*

```
import uvm_pkg::*;
`include "uvm_macros.svh"

// -----------------------------------------

class packet extends uvm_sequence_item;
  `uvm_object_utils(packet)

  rand bit[7:0] som;
  rand bit[7:0] addr;
  rand bit[7:0] payload [8];
       bit[7:0] checksum;
  rand bit[7:0] eom;

  constraint val {
    som == '0;
    eom == '1;
    foreach (payload[i]) payload[i] inside {[0:100]};
  }

  function new(string name = "packet");
    super.new(name);
  endfunction

  function void post_randomize();
    calc_checksum();
  endfunction

  function string convert2string();
    return $sformatf(
     "(%s) packet=[%3x] %2x %2x %2x %2x %2x %2x %2x
[%2x]",
      get_type_name(),
      addr,
      payload[0], payload[1], payload[2], payload[3],
      payload[4], payload[5], payload[6], payload[7],
      checksum
      );
  endfunction

  virtual function void calc_checksum();
    checksum = 0;
    foreach(payload[i])
      checksum = checksum + payload[i];
  endfunction

  function void do_record(uvm_recorder recorder);
    $add_attribute(recorder.tr_handle, som, "som");
    $add_attribute(recorder.tr_handle, addr, "addr");
    $add_attribute(recorder.tr_handle, payload,
      "payload");
    $add_attribute(recorder.tr_handle, checksum,
      "checksum");
    $add_attribute(recorder.tr_handle, eom, "eom");
  endfunction
endclass

class randc_addr_c;
  randc bit[7:0] addr;
endclass

class packet_with_randc_addr extends packet;
  `uvm_object_utils(packet_with_randc_addr)

  static randc_addr_c randc_addr;

  function new(string name = "packet_with_randc_addr");
    super.new(name);
  endfunction

  function void post_randomize();
    if (randc_addr == null)
      randc_addr = new();
    if (!randc_addr.randomize())
      `uvm_fatal("RANDC", "Randomize failed for
randc_addr")
    addr = randc_addr.addr;
  endfunction
endclass

class big_packet extends packet;
  `uvm_object_utils(big_packet)

  rand bit[7:0] payload [32]; // Size goes from 8 to 32.

       bit[7:0] io_type;      // New attribute

  function new(string name = "big_packet");
    super.new(name);
    super.payload.rand_mode(0); // Going to be replaced.
  endfunction

  virtual function void calc_checksum();
    checksum = 0;
    foreach(payload[i])
      checksum = checksum + payload[i];
  endfunction

  function string convert2string();
    string msg;

    msg = $sformatf("(%s) packet=", get_type_name());
    for(int i = 0; i < 32; i++) begin
      if ((i % 16) == 0)
        msg = {msg, "\n", $sformatf("  [%3x]", addr+i)};
      msg = {msg, $sformatf(" %2x", payload[i])};
    end
    msg = {msg, $sformatf(" [%2x]", checksum)};
    return msg;
  endfunction

endclass

class n_packets extends uvm_sequence#(packet);
  `uvm_object_utils(n_packets)

  int packet_priority = 100; // 100 = UVM default.
                             // Higher priority is a
                             //   higher number.

  rand int how_many;
```

```
  constraint val {
    how_many >= 10; how_many <= 256; }

  function new(string name = "n_packets");
    super.new(name);
  endfunction

  task create_called_N_times();
    packet t;

    for(int i = 0; i < how_many; i+=8) begin
      for (int j = 0; j < 8; j++) begin
        fork
          begin
            packet my_t;
            my_t = packet::type_id::create(
              "packet",,get_full_name());

            if (!my_t.randomize() )
              `uvm_fatal("SEQ", "Randomize failed")

            start_item(my_t, packet_priority);
            finish_item(my_t);
          end
        join_none
      end
      wait fork;

      #5;
    end
  endtask

  task create_called_once();
    packet t;

    t = packet::type_id::create(
      "packet",,get_full_name());
    for(int i = 0; i < how_many; i++) begin
      if (!t.randomize() )
        `uvm_fatal("SEQ", "Randomize failed")

      start_item(t, packet_priority);
      finish_item(t);
      #5;
    end
  endtask

  task body();
    create_called_N_times();
    // create_called_once();
  endtask

  function void do_record(uvm_recorder recorder);
    super.do_record(recorder);
    $add_attribute(recorder.tr_handle,
      how_many, "how_many");
  endfunction
endclass

// ----------------------------------------

class driver extends uvm_driver#(packet);
  `uvm_component_utils(driver)

  function new(string name = "driver",
      uvm_component parent = null);
    super.new(name, parent);
  endfunction

  task run_phase(uvm_phase phase);
    forever begin
      packet t;
      seq_item_port.get_next_item(t);
      `uvm_info("DRVR",
        {" Got ", t.convert2string()}, UVM_MEDIUM)
      foreach (t.payload[i]) // Consume time.
        #10;
      seq_item_port.item_done();
    end
  endtask
endclass
```

```
class env extends uvm_env;
  `uvm_component_utils(env)

  driver d;
  uvm_sequencer#(packet) sqr;

  function new(string name = "env",
      uvm_component parent = null);
    super.new(name, parent);
  endfunction

  function void build_phase(uvm_phase phase);
    d = driver::type_id::create("driver", this);
    sqr = uvm_sequencer#(packet)::type_id::create(
      "sequencer", this);

    sqr.set_arbitration(SEQ_ARB_STRICT_FIFO);
  endfunction

  function void connect_phase(uvm_phase phase);
    d.seq_item_port.connect(sqr.seq_item_export);
  endfunction
endclass

class test extends uvm_test;
  `uvm_component_utils(test)

  env e1;

  function new(string name = "test",
      uvm_component parent = null);
    super.new(name, parent);
  endfunction

  function void build_phase(uvm_phase phase);
    e1 = env::type_id::create("e1", this);
  endfunction

  task run_phase(uvm_phase phase);
    n_packets seq1, seq2;

    phase.raise_objection(this);

    packet::type_id::set_type_override(
      packet_with_randc_addr::get_type);

    packet::type_id::set_inst_override(
      big_packet::get_type(),
      "uvm_test_top.e1.sequencer.seq1.packet");

    seq1=n_packets::type_id::create(
      "seq1",,get_full_name());
    seq2=n_packets::type_id::create(
      "seq2",,get_full_name());

    seq1.packet_priority = 1000;

    factory.print();

    if (!seq1.randomize() with {how_many == 256;})
      `uvm_fatal("TEST",
        "Randomization failed for seq1")
    if (!seq2.randomize() with {how_many == 256;})
      `uvm_fatal("TEST",
        "Randomization failed for seq2")

    fork
      seq1.start(e1.sqr);
      seq2.start(e1.sqr);
    join

    phase.drop_objection(this);
  endtask
endclass

module top();
  initial begin
    uvm_config_db#(int)::set(null, "",
"recording_detail", 1);
    run_test("test");
  end
endmodule
```