Keeping Your Sequences Relevant

Nicholas Zicha, Eric Combes
Accedian Networks
2351 Blvd. Alfred-Nobel, Suite N-410
Saint-Laurent (Montreal), Quebec, H4S 2A9, Canada

***Abstract*- In UVM testbenches, the job of generating transactions as stimulus generally falls on sequences which have many underlying mechanisms and use models that allow for complex scenarios to be created, from user override-able callbacks, to sequencer manipulation using locking and grabbing. In this paper we hope to add another tool to a verification engineer's toolbox by demonstrating the use of a lesser known part of the sequence-sequencer interaction already built into UVM, the relevance API. Leveraging other object-oriented programming techniques such as mixins and the visitor pattern, and prioritizing composition rather than strict inheritance, we will show some reuseable ways of enhancing sequences with little modification necessary. An example controlling the individual and aggregate rate of a packet-based design will be demonstrated.**

I. Introduction

Generating stimulus to apply to a design under test (DUT) is an important part of any verification environment. UVM testbenches prescribe transaction-based stimulus using the concept of sequences which can generate sequence items (transactions) or can start other sequences. They are ephemeral objects that are created, run, and destroyed over the course of a test, in contrast with classic BFM-style generators that are static in a test environment. Often referred to as functors or function objects with a primary role of simply executing their `body()` function, they in fact execute many layers of callbacks, handshakes, and sequencing mechanisms.

As sequences represent a significant piece in achieving coverage closure, having both a good understanding and a large toolbox of techniques when using them is important. Only a cursory explanation is presented here to set a context for what follows, as other texts provide a more thorough exposition [1].

A. Interaction Between a Sequence, Sequencer, and Driver

Sequences "run" or are "started" on sequencers acting as brokers with a component that consumes sequence items, typically a driver. Figure 1 shows the standard high-level flow of events when a sequence is ran [2][3] using some of the actual function and task calls involved.

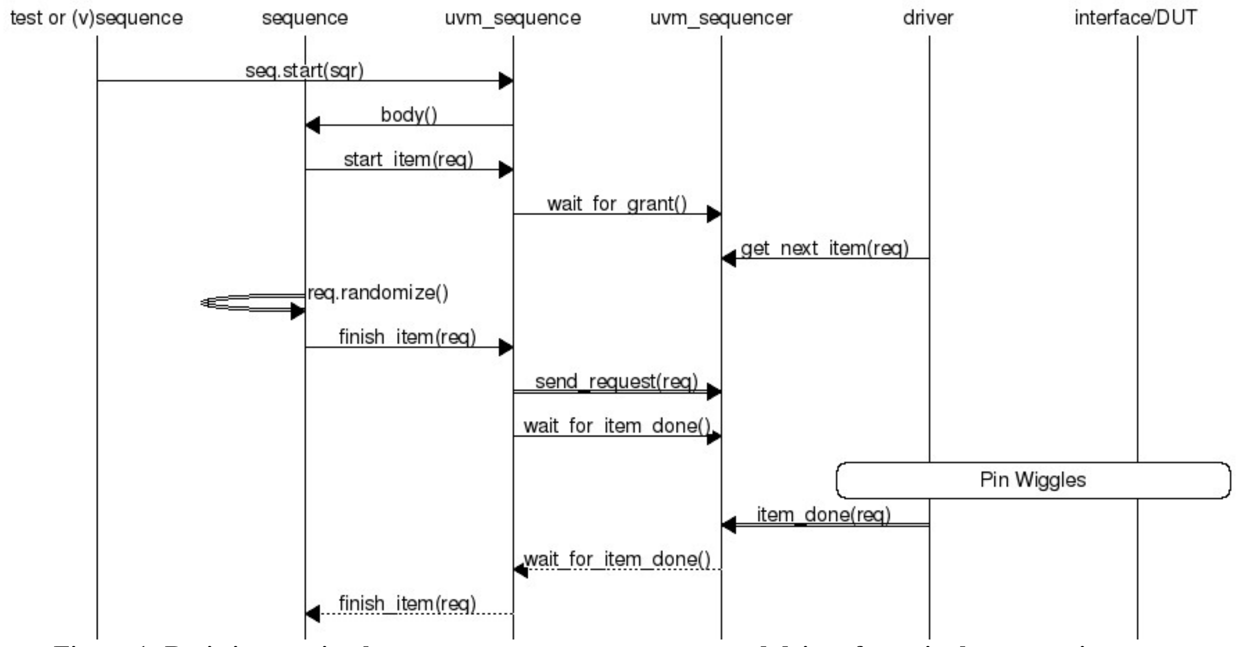

Figure 1: Basic interaction between a sequence, sequencer, and driver for a single transaction.

Initially the driver fetching an item (using `peek/get_next_item`), the sequencer waiting for sequences, and the component starting the sequence are independent. Once one or more sequences have been started on a sequencer and have started an item (using `start_item`) within their `body()` task, the sequencer selects the next transaction to be executed. The selected sequence can then randomize or manipulate the transaction at the last instant before finally sending it, often referred to as "late randomization" since simulation cycles might have passed between the time the request was initiated and was finally granted access to the driver. The item is delivered to the driver by calling the blocking `finish_item` task, which will wait for a response. The driver will map the transaction to the bus interface through "pin wiggles" and declare that it is done by calling `item_done` with or without an item in response, thus unblocking the sequence and allowing it to continue.

B. Multiple Sequences

More advanced scenarios often require running multiple sequences in parallel. These sequences can be started on individual sequencers, or a common sequencer. Parallelizing sequences in SystemVerilog can be done by forking several "threads", and starting a sequence on each one.

```
fork
   seq1.start(sequencerA);
   seq2.start(sequencerB);
   seq3.start(sequencerB);
join(_none/any)
```

Code Stub 1: Starting sequences in parallel.

Notice that sequences 2 and 3 are both started on a common sequencer (sequencerB), whereas sequence 1 is started on its own (sequencerA). How sequences running on a common sequencer are serviced is based on built in selection and arbitration mechanisms.

II. Selecting From Multiple Sequences

When multiple sequences overlap in their execution on the same sequencer, the sequencer needs to determine which sequence will have access to the driver at any given time. The sequences are running in parallel, but the items are processed one at a time, therefore some sort of selection and arbitration is required.[1]

A. Sequencer Arbitration Modes

Controlling the interaction between multiple sequences running in parallel on a common sequencer is most often presented using the arbitration modes of `uvm_sequencer`. A list of available sequences is constructed by the sequencer to grant a request from, using one of the following built-in modes:

Table 1: Sequencer arbitration modes

Mode	Grant
UVM_SEQ_ARB_FIFO	in FIFO order
UVM_SEQ_ARB_RANDOM	randomly
UVM_SEQ_ARB_STRICT_FIFO	by priority in FIFO order
UVM_SEQ_ARB_STRICT_RANDOM	by priority in random order
UVM_SEQ_ARB_WEIGHTED	randomly by weight
UVM_SEQ_ARB_USER	based on user-definable `user_priority_arbitration` function

The user can change the arbitration mode of a sequencer using the `set_arbitration` function (actually defined in `uvm_sequencer_base`). By default, sequencers start in FIFO mode, though the initial ordering may vary if several sequences are started concurrently due to SystemVerilog's non-determinism. Several of the modes have some configurability using a priority that can be specified when calling the `start` task of a sequence. Depending on the mode, this value will be used either as a strict priority value or as a weighing priority. The modes provide a good range of control for generating interesting stimulus [4].

For anything more precise or exotic, extending `uvm_sequencer` to override the default `user_priority_arbitration` function is necessary.

[1] For brevity, sequencer locking and grabbing will not be discussed, but also partake in the process.

```
virtual function integer user_priority_arbitration(integer avail_sequences[$]);
```

Code Stub 2: Function signature.

Unfortunately, identifying and applying specific arbitration to the available sequences is somewhat cumbersome since the given list is of integers, providing little useful information to identify them individually. Many of the properties and methods in the base sequencer classes are qualified as local, removing visibility from inheriting classes. Applying user arbitration also requires overriding the sequencer type when built, which is straightforward in UVM using the factory, but limits certain dynamism of the environment during tests.

B. Sequence Relevance

While rarely mentioned (but documented in plain sight), there are other steps when selecting a sequence in addition to the arbitration modes. The following sequence diagram resembles the earlier one, but adds more detail to the steps taken between the `start_item` and `finish_item` calls.

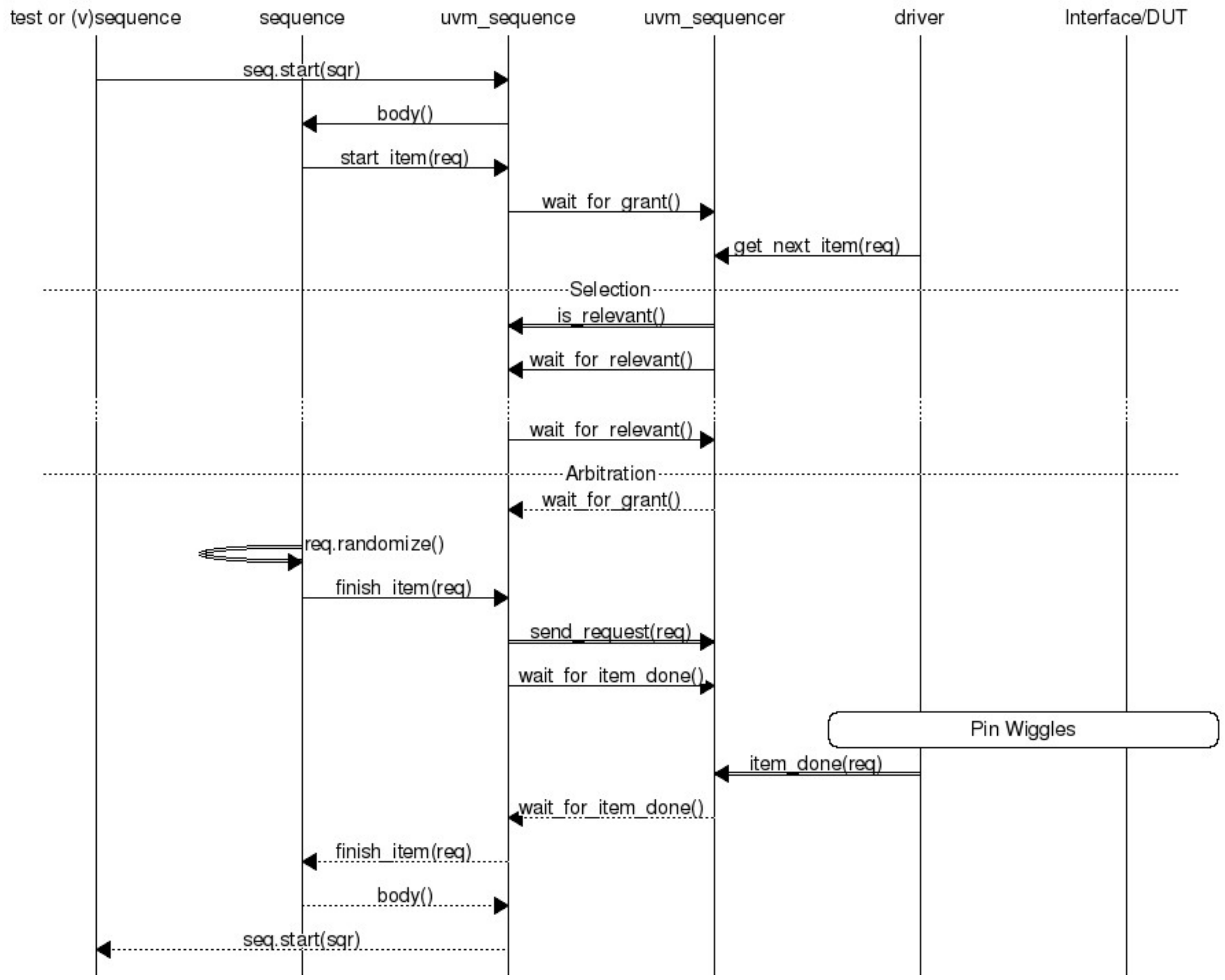

Figure 2: Sequence diagram from when an uvm_sequence is started until completion. Note the addition of the selection based on relevance and arbitration steps.

Of particular interest are the calls involved in the "Selection". A sequence's relevance is validated during the start/finish item cycle, and is determined by the `is_relevant` function defined in `uvm_sequence_base`, having a relatively nondescript signature:

```
virtual function bit is_relevant();
```

Code Stub 3: Function signature.

By default, sequences assert that they are always relevant and are therefore available for arbitration. Otherwise, if no sequence asserts itself as relevant, the sequencer calls the `wait_for_relevant` task.

```
virtual task wait_for_relevant();
```

Code Stub 4: Task signature.

Re-arbitration of a sequence's relevance is performed after the wait task returns. This implies that time should be consumed by the `wait_for_relevant` task in order to avoid an endless `is_relevant/wait_for_relevant` loop. It is not necessary for a sequence to be relevant when the task returns, providing some additional flexibility in determining how long to wait.

By default, this duo of relevance methods are short-circuited by always asserting relevant, which may be why they are not commonly used. They do, however, provide yet another way to manipulate the flow of transactions in the packaging most commonly used to generate stimulus: sequences. They also provide a clarification of the aforementioned sequencer arbitration modes. Sequencer arbitration is only applied to sequences that are relevant.

To provide more insight, the following is the output of a simple testbench where several overridable sequence, sequencer, and driver methods are instrumented by extending their respective base classes and adding `` `uvm_info `` calls to print the order in which they occurred. Two sequences are started in parallel, with transactions that do nothing more than generate a random delay. The sequencer arbitration mode is configured to UVM_SEQ_ARB_USER, providing another point of reference.

```
@ 0:    uvm_test_top.drv       [driver]                   get_next_item
@ 0:    uvm_test_top.sqr       [uvm_sequencer_base]       wait_for_sequences
@ 0:    uvm_test_top.sqr@@seq1 [uvm_sequence]             start_item
@ 0:    uvm_test_top.sqr       [uvm_sequencer_base]       wait_for_grant
@ 0:    uvm_test_top.sqr@@seq2 [uvm_sequence]             start_item
@ 0:    uvm_test_top.sqr       [uvm_sequencer_base]       wait_for_grant
@ 0:    uvm_test_top.sqr       [uvm_sequencer_base]       wait_for_sequences
@ 0:    uvm_test_top.sqr@@seq1 [uvm_sequence]             is_relevant
@ 0:    uvm_test_top.sqr@@seq2 [uvm_sequence]             is_relevant
@ 0:    uvm_test_top.sqr       [uvm_sequencer_base]       user_priority_arbitration
@ 0:    uvm_test_top.sqr@@seq1 [uvm_sequence]             finish_item
@ 0:    uvm_test_top.sqr       [uvm_sequencer_param_base] send_request
@ 0:    uvm_test_top.sqr       [uvm_sequencer_base]       wait_for_item_done
@ 0:    uvm_test_top.drv       [driver]                   got next_item
@ 311:  uvm_test_top.sqr       [uvm_sequencer]            item_done
@ 311:  uvm_test_top.sqr       [uvm_sequencer_base]       wait_for_sequences
@ 311:  uvm_test_top.sqr@@seq2 [uvm_sequence]             is_relevant
@ 311:  uvm_test_top.sqr@@seq2 [uvm_sequence]             finish_item
@ 311:  uvm_test_top.sqr       [uvm_sequencer_param_base] send_request
@ 311:  uvm_test_top.sqr       [uvm_sequencer_base]       wait_for_item_done
@ 1527: uvm_test_top.sqr       [uvm_sequencer]            item_done
@ 1527: uvm_test_top.sqr       [uvm_sequencer_base]       wait_for_sequences
```

Figure 3: Two sequences running in parallel with instrumented messages to reveal the order of events.

The relevance check is performed after the `start_item` call, and since both sequences are relevant by default, they must be arbitrated as indicated by the `user_priority_arbitration` event. One sequence is granted first, and its item completed through the driver and `finish_item` steps. Note that the other sequence is in fact checked once again for relevance before it is granted and completes its own execution.

II. Application: Shaping Packet Traffic

The need to control stimulus rates, in either packets per second (pps) or bits per second (bps), is a common requirement when verifying packet-based designs. Some DUTs may have limited buffering capacity or processing capabilities, while others may themselves regulate or measure traffic rates.

Common IEEE 802.3 Ethernet frames [5][6] will be used as transactions for the application examples that follow. Ignoring Jumbo frames, Ethernet frames range in size from 64 bytes to 1518 bytes. When transmitted, a 7 byte preamble and single SFD byte are pre-appended. Furthermore, a minimum of 12 bytes is required between frames, referred to as the minimum inter-packet or inter-frame gap (IPG or IFG). Using these values, the minimum and maximum packet rates for a 1Gbps link can be calculated as follows:

Equation 1: Minimum and Maximum packets per second (pps)

$$pps_{max} = (1{,}000{,}000{,}000\frac{bits}{sec})/((64+8+12)\frac{bytes}{packet} * 8\frac{bits}{byte}) = 1{,}488{,}095\frac{pkts}{sec}$$

$$pps_{min} = (1{,}000{,}000{,}000\frac{bits}{sec})/((1518+8+12)\frac{bytes}{packet} * 8\frac{bits}{byte}) = 81{,}274\frac{pkts}{sec}$$

Small frames tend to stress processing capability as new headers arrive more often, while large frames tend to stress bandwidth and buffering since fewer bits are "wasted" on overhead. Irrespective of how a frame transaction type is implemented, a means of determining the number of bits it contains is necessary for rate calculations. UVM already has a function built into `uvm_object` that can serve this purpose, `pack()`.

```
function int pack (ref bit bitstream[], input uvm_packer packer = null);
virtual function void do_pack (uvm_packer packer);
```

Code Stub 5: Packing functions.

The packing function is intended to serialize the fields of an object into the referenced bit array, returning the total number of bits packed. Although the `pack()` function itself cannot be overridden, classes extending `uvm_object` can implement the `do_pack()` hook and use the built-in packing macros[2]. The packer object in the parameters is a policy class used by the packing macros, defaulting to `uvm_default_packer` if none is provided[3].

```
bit [47:0] da;      // destination address
bit [47:0] sa;      // source address
bit [15:0] etype;   // length or type field
bit [7:0]  data[];  // payload
bit [31:0] fcs;     // frame check sequence (for error detection)

virtual function void do_pack (uvm_packer packer);
    super.do_pack(packer);
    `uvm_pack_intN(da, $bits(da));
    `uvm_pack_intN(sa, $bits(sa));
    `uvm_pack_intN(etype, $bits(etype));
    `uvm_pack_arrayN(data, 8);
    `uvm_pack_intN(fcs, $bits(fcs));
endfunction
```

Code Stub 6: An example implementation of the `do_pack()` function for a simple Ethernet packet.

Many algorithms exist for regulating the rate at which packets are generated. A token bucket is a relatively simple algorithm, embodying the analogy of a fixed-sized bucket that is filled at a constant rate [7]. Filling and emptying the bucket are referred to as crediting and debiting, respectively, and the unit quantity measured in tokens. For packets, the tokens are simply bits and the bucket is credited at the desired bit-rate. The bucket can be modeled as a signed integer, and when the bucket has reached a certain threshold, say a positive number, it is said to be compliant. Compliance signifies that a packet can be generated, with the bucket debited by the size of a packet. The bucket typically also has a maximum fill level, used to limit the maximum number of bits that can be burst consecutively.

The algorithm can be described in pseudo-code as follows:

```
int signed Bucket
@(Update = 1/Rate)
   Bucket += Token
```

[2] The `` `uvm_field_* `` macros can also be used instead of explicitly defining the packing function(s).

[3] The default packer implements its internal bitstream as a fixed-sized array, with a maximum size of 4096 bytes. Depending on the application, this may be limiting, and unfortunately requires recompiling the UVM source and (re)defining `` `UVM_MAX_STREAMBITS `` to a larger value.

```
   If (Bucket > Max) : Limit
       Bucket = Max
   If (Bucket >= 0)
       Compliant = true
   Else
       Compliant = false

If Compliant
   Send
   Debit Packet Size
Else
   Wait until next Update
```

Figure 4: Pseudo-code for token bucket.

B. Using the Relevance API to Create a Base Sequence

Since the relevance of a sequence is checked prior to the execution of each item, it presents an opportunity to restrict how often a transaction will be selected based on the result of `is_relevant`. It also presents a clean API for deferring relevance with the `wait_for_relevant` task, analogous to a delay before a token bucket is (re)filled to the threshold and another packet can be sent. A partial sequence implementation is shown in Code Stub 7.

```
class rate_relevant_sequence extends uvm_sequence#(item);
   // Configuration properties
   rand int      iter = 10;      // number of transaction sent in body
   int unsigned  bitrate;        // desired rate
   time          update_period;  // how often to update the bucket
   int unsigned  max_burst;      // maximum burst size (bucket size)
   protected int bucket;         // account balance in number of bits

   // Send transactions
   virtual task body();
       req = item::type_id::create("req");
       repeat(iter) begin
           start_item(req);
           if (!req.randomize() with {
               data.size() == 84; // Minimum packet + overhead
           }) `uvm_fatal("RANDERR", "item")
           finish_item(req);
       end
   endtask

   // Determine relevance based on bucket level
   virtual function bit is_relevant();
       update_bucket();
       return (bucket >= 0);
   endfunction

   // Wait until relevant using the time elapsed since the last update
   virtual task wait_for_relevant();
       time wait_time = update_period - ($time() - last_update_time);
       #(wait_time);
   endtask

   // Debit bucket
   virtual task finish_item (uvm_sequence_item item, int set_priority = -1);
       bit bitstream[];
       int nb_of_bits;

       nb_of_bits = item.pack(bitstream);
```

```
        bucket -= nb_of_bits;
        super.finish_item(item, set_priority);
    endtask

    // Fill bucket
    virtual function void update_bucket();
        time            t_since_last_update;
        int unsigned    quotient_time;
        int             new_bucket;

        // calculate time since last update
        t_since_last_update = $time() - last_update_time;

        if(t_since_last_update > 0) begin : NeedUpdate
            // Calculate credit amount
            quotient_time = t_since_last_update / update_period;
            new_bucket = bucket + (quotient_time * bits_per_period);
            // Limit bucket level
            if(new_bucket > int'(max_burst))
                new_bucket = max_burst;
            // Update state
            last_update_time = $time;
            bucket = new_bucket;
        end
    endfunction
endclass
```

Code Stub 7: Sequence definition, including standard `body()` task, relevance, and token bucket implementation. Note that portions of the code have been left out in order to concentrate on the essential.

Other than the presence of the `is_relevant()` function and `wait_for_relevant()` task, the sequence is fairly standard UVM. Several properties have been added to control the rate: `bitrate` to define the desired rate, `bucket` implemented as a signed integer to represent the token bucket, `max_burst` to limit the bucket size, and `update_period` to specify how often credits should be dispensed. At each relevance check the bucket is updated and the compliance is validated based on the level being positive. The wait for relevance determines the time before the bucket should be updated next. Once the sequence has been served, the size of the last packet sent can be extracted in the `finish_item` task to debit the bucket. Figure 5 shows a sample transaction recording when sequences are ran using different rates (100 Mbps in Code Stub 8) with a constant frame size of 64 bytes (minimum Ethernet packet plus 20 bytes to account for minimum IPG, SFD, and preamble). [4]

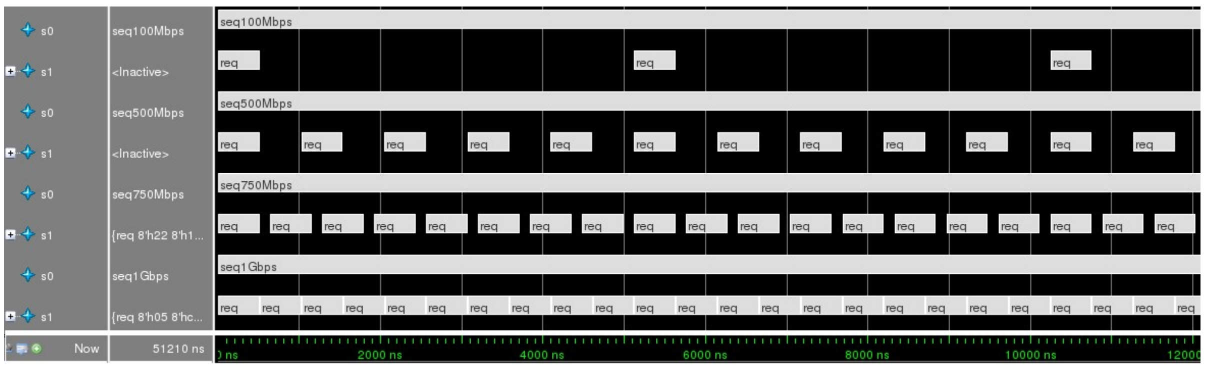

Figure 5: Transaction recording of transaction rate limited by relevance sequence on a 1 Gbps link. From top to bottom: 100 Mbps, 500 Mbps, 750 Mbps, 1Gbps.

[4] Unit testing rates is simplified using fixed packet sizes since the time between packets over short periods is constant.

At line rate (1Gpbs in this case), there are no gaps between transactions. At 500 Mbps and 100 Mbps, it is clear that only half and a tenth of the bandwidth is used, respectively. Configuring and starting the sequence is shown in Code Stub 8.

```
class rate_relevant extends test;
   ...
   virtual task main_phase(uvm_phase phase);
       rate_relevant_sequence seq1 = rate_relevant_sequence::type_id::create("seq1");
   ...
       seq1.iter = 10;
       seq1.set_bitrate(100_000_000);
       seq1.set_update_period(10ns);
       seq1.start(sqr);
```

Code Stub 8: Sample usage of the relevant rate sequence.

III. Composable Relevance Sequence

While the sequence in Code Stub 7 can be used as a base class that other sequences can extend, it imposes a class hierarchy and sequence item type. An approach to wrap arbitrary sequences that already exist using different transaction types would provide a far more interesting and reusable solution. SystemVerilog's type parameterization can be employed to accomplish this.

A. Mixin

A mixin is a class whose methods are added to another class without the need of an inheritance relationship, enabling aspect oriented programming [8]. Many programming languages enable this type of "mixing in" methods out of the box, either by leveraging multiple inheritance (C++, Python), traits or interface classes with partial function implementations (Scala [9]), or other language features.

SystemVerilog does not support multiple inheritance, but interface classes were added in the 2012 LRM enabling protocol-style multiple inheritance. Unfortunately, complete or partial function implementations and properties are not allowed, so the techniques used in other languages are not applicable. Despite this limitation, there is a way to quickly combine classes in the *spirit* of mixins [10]. The following is a class definition with a particular parameterization and inheritance signature.

```
class mixin #(type T) extends T;
```

Code Stub 9: Mixin class signature.

The class extends a class of the type it is parameterized by. A more concrete implementation gives more insight into why this can be interesting.

```
// Note: the default type is not strictly necessary, but hints intent.
class bar#(type T=uvm_sequence) extends T;
...
   constraint bar_constraints {
       ...
   }

   task finish_item (uvm_sequence_item item, int set_priority = -1);
       `uvm_info("BAR", "Mixing into finish_item!", UVM_DEBUG)
       super.finish_item(item, set_priority);
   endtask
...
endclass
```

Code Stub 10: Defining a class in the mixin style to add a constraint and function override.

The `bar` class adds a constraint block and a `finish_item` task to a class of an as-yet undefined type. The default type of `T`, while not strictly necessary, gives an indication of what `T` may be, in this case tailored specifically to add features to an `uvm_sequence`. Though unusual, the class above allows for quickly mixing in modified behaviour to a sequence `baz` as follows:

```
bar#(baz) barbaz;
```

Code Stub 11: Mixing `bar` into `baz`.

If another mixin `foo` exists, perhaps with different constraints and method overrides, it is just as easy to combine with other mixins:

```
foo#(bar#(baz)) foobarbaz;
```

Code Stub 12: Mixing `foo` and `bar` into `baz`.

The classes `barbaz` and `foobarbaz` have not actually inherited from multiple classes of course, as that is not allowed in SystemVerilog. The apparent composability is an illusion. What the examples above have done is create a chain of inheritance applied at compile time very succinctly. The resulting hierarchy is as follows, with the parent class starting on the left:

```
uvm_object ← … ← uvm_sequence#(item) ← baz ← bar ← foo
```

Figure 6: Class inheritance of foobarbaz.

What is also practical about this syntax, in addition to its terseness, is the ability to mix in different classes with different implementations in any order:

```
bar#(foo#(baz))  barfoobaz;
fiz#(buzz#(baz)) fizbuzzbaz;
```

Code Stub 13: Different mixin ordering.

A word of caution: in reality the mixin classes cannot be applied to any arbitrary class as in other languages. The portability of the mixins is often still limited to a certain type of class, especially for classes whose constructors have parameters without default values. In spite of this, they can still be useful and reusable way to inject functionality into objects.

B. Refactoring the Relevance Sequence as a Mixin

The sequence originally defined in Code Stub 7 can be modified by changing its class signature and removing its `body()` task to create a mixin that adds rate control to other sequences by implementing `is_relevant`, `wait_for_relevant`, and `finish_item` methods.

```
// Same as previous rate relevance sequence, but without the body() task.
class rate_mixin#(type T) extends T;

   int unsigned  bitrate;       // desired rate
   time          update_period; // how often to update the bucket
   int unsigned  max_burst;     // maximum burst size (bucket size)
   ...
   `uvm_object_param_utils(rate_mixin#(T))
   function new(string name="rate_mixin");
       super.new(name);
   endfunction

   virtual function bit is_relevant(); ...
   virtual task wait_for_relevant();  ...
   virtual task finish_item (uvm_sequence_item item, int set_priority = -1);  ...
   virtual function void update_bucket(); ...
   virtual function time time_since_last_update();  ...
   virtual function time time_before_next_update();  ...
    ...
endclass

// A fairly simple sequence that randomizes packet-like items.
class standard_seq extends uvm_sequence#(item);
```

```
   rand int iter = 1; // number of transaction sent in body
   ...
   virtual task body();
       req = item::type_id::create("req");

       repeat(iter) begin
           start_item(req);
           if (!req.randomize() with {
               data.size() == 84;
           }) `uvm_fatal("RANDERR", "item")
           finish_item(req);
       end
   endtask
endclass

class test extends uvm_test;
   ...
   typedef rate_mixin#(standard_seq) rate_seq_t;

   virtual task main_phase(uvm_phase phase);
       rate_seq_t seq1 = rate_seq_t::type_id::create("seq1");
       ...
       seq1.set_bitrate(100_000_000);
       seq1.set_update_period(10ns);
       ...
       seq1.start(sqr);
       ...
```
Code Stub 14: A mixin based on the original relevance control sequence.

Since the sequence only refers to the item's packing function in order to determine its bit length for rate control, the actual type of item is irrelevant so long as it is a descendant of `uvm_object` (true for `uvm_sequence_items`) and that its `do_pack()` function is implemented. Code Stub 15 shows an example where two sequences of different traffic (transaction) types [6] have their rates limited using the mixin sequence, and started in parallel on the same sequencer. A third sequence is started afterwards at a much lower rate. A transaction recording is shown in Figure 7.

```
virtual task main_phase(uvm_phase phase);
   rate_mixin#(eth_sequence)  seq_eth  = rate_mixin#(eth_sequence)::type_id::create("seq_eth");
   rate_mixin#(ipv4_sequence) seq_ipv4 = rate_mixin#(ipv4_sequence)::type_id::create("seq_ipv4");
   rate_mixin#(ptp_sequence)  seq_ptp  = rate_mixin#(ptp_sequence)::type_id::create("seq_ptp");

   // Configure rate
   seq_eth.set_bitrate(500_000_000);  // 500 Mbps
   seq_eth.set_update_period(10ns);
   seq_ipv4.set_bitrate(100_000_000); // 100 Mbps
   seq_ipv4.set_update_period(10ns);
   seq_ptp.set_bitrate(50_000_000);   // 50 Mbps
   seq_ptp.set_update_period(10ns);
   ...
   // Start sequences
   fork
       fork
           seq_eth.start(sqr);
           seq_ipv4.start(sqr);
       join
       begin
           #5us;
           seq_ptp.start(sqr);
```

```
        end
    join
    ...
```

Code Stub 15: Using the sequence mixin to create a more complex scenario.

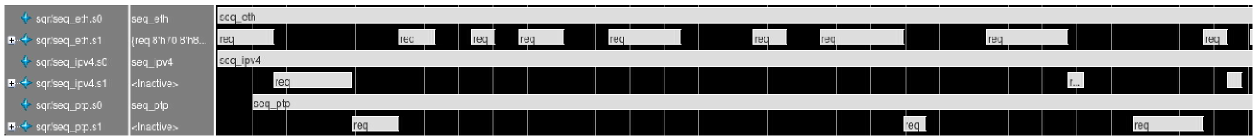

Figure 7: Transaction recording of a simulation using multiple sequences at different rates.

IV. MULTIPLE RELEVANCE CHECKS

Having a technique to control the rate of transaction generation in a sequence already provides a means for creating interesting scenarios. Adding mixins adds an ease of composability to existing sequences. The concept can be expanded upon. Suppose other types of relevance checks need to be applied? Creating new mixins and chaining them together could be used. But suppose that the resulting relevance should be either the conjunction (*AND*) or disjunction (*OR*) of each check? Or suppose multiple sequences need to share the same relevance check, for example to share the same bandwidth on an interface? Here the aforementioned approaches will not work as-is. What is needed is a way to combine arbitrary algorithms, as well as a way to handle multiple relevance checks.

A. Visitor Pattern

In object-oriented programming, known patterns are often used as solutions to common problems [11][12]. The visitor pattern is used to separate an algorithm from a structure, most often demonstrated as a chain of classes that are visited. The resulting structure can allow for the addition and combination of different algorithms, using a more compositional approach rather than strict inheritance. This pattern inspires a way to achieve the goal of an even more flexible relevance control API.

B. Relevance Delegation

The first step is to break out the algorithm determining relevance from the base mixin. The `is_relevant` and `wait_for_relevant` hooks will be delegated to a new `relevance` class.

```
class relevance extends uvm_object;
   // Sequences associated with this algorithm
   protected relevance_sequence_mixin assoc_q[$];

   // Function for connecting sequence & sanity checks.
   assoc()
   // Delegation
   is_relevant()
   wait_for_relevant()
   start_item(uvm_sequence_item item)
   finish_item(uvm_sequence_item item)
endclass
```

Code Stub 16: Structure of a relevance checking delegate class, used as a base class for different algorithms.

The actual base class of this object is inconsequential, but using `uvm_object` is common practice in a UVM testbench if only to register with the UVM factory. The start and finish item tasks have also been broken out to this class in order to have the ability to delegate even more control outside of the sequence. What distinguishes this class further is the queue of sequences to which it is associated, providing a link to multiple calling sequences as will become apparent. This class can be extended to create different relevance algorithms.

C. Relevance Visiting Mixin

Refactoring is applied once again to the relevance sequence in Code Stub 14, keeping the mixin signature, but applying the visitor pattern by adding a queue of relevance objects that it can be associated with.

```
class relevance_sequence_mixin#(type SEQ = uvm_sequence) extends SEQ;
   // AND/OR relevance results of assoc (AND=1, OR=0)
   bit all_relevant;

   // Queue of controls associated with this object
   protected relevance assoc_q[$];

   // Relevance functions and tasks
   virtual function void assoc(relevance rel);
   virtual function bit is_relevant();
   virtual task wait_for_relevant();
   virtual task finish_item (uvm_sequence_item item, int set_priority = -1);
endclass
```

Code Stub 17: Final relevance mixin sequence.

Now, instead of simply providing a response when `is_relevant` is called, the function visits the `is_relevant` function of each associated relevance class in turn, combining their results for the final response. A configuration bit `all_relevant` is used to determine whether all or at least one should be relevant for the final relevance to be asserted, equivalent to *AND* or *OR* operators. The `wait_for_relevant` task waits only as long as needed based on its connected relevance classes (using `fork/join_any`). Other helper functions can be added to perform the internal connections and bookkeeping, such as null checks and duplicate association refusal.

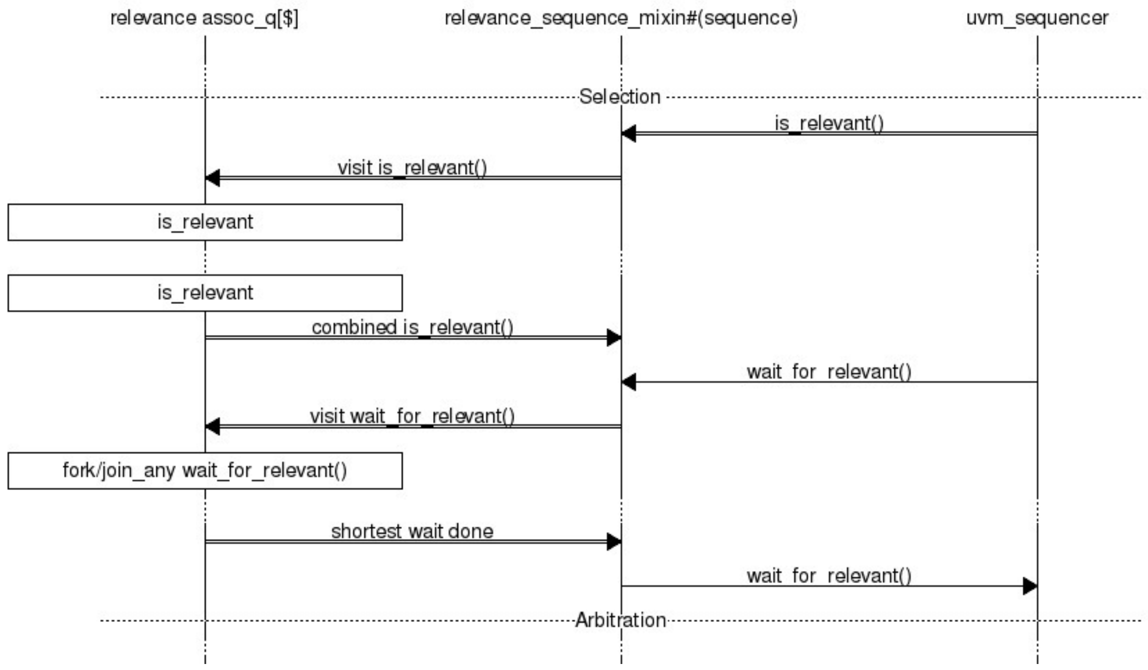

Figure 8: Interaction between the relevance sequence mixin and associated relevance controls to which `is_relevant()` and `wait_for_relevant()` calls are delegated.

V. Bringing It All Together

For a final example, two relevance algorithms will extend the `relevance` class to create algorithms that can be used in different combinations, providing an example of their potential. A complete implementation can be found in the Appendix, including a simple testing environment.

A. Rate Control

Code stubs from the original relevance sequence are taken and refactored to extend the base relevance class to (re)create a rate controlling algorithm.

```
// Limit based on rate
class rate extends relevance;
   // Configuration
   protected int unsigned    bitrate;          // desired rate
   protected time            update_period;    // how often to update the account balance ...
   protected int unsigned    max_burst;        // maximum burst size (bucket size)
...
   // Derived configuration
   protected int unsigned    bits_per_period;  // bpp: Bits per period

   // Work variables
   protected int             bucket;           // Account balance in number of bits
...
   // Check relevance based on bucket level
   virtual function bit is_relevant();
       ...
       update_bucket();
       ...
       return (bucket >= 0);
   endfunction

   // Wait until the next update to check again if relevant
   virtual task wait_for_relevant();
       time wait_time;
       wait_time = time_before_next_update();
       #(wait_time);
   endtask

   // Debit bucket by packet size
   virtual function void finish_item (uvm_sequence_item item);
       bit bitstream[];
       int nb_of_bits;

       nb_of_bits = item.pack(bitstream);
       bucket -= nb_of_bits;
   endfunction

   // Update bucked based on update period and time elapsed
   virtual function void update_bucket();
...
```

Code Stub 18: Relevance delegate extension controlling bit rate.

B. Buffering

Packet-based DUTs often have some limit in the quantity of bits that they can buffer, and packets being discarded or having backpressure applied may be undesirable for a certain types of coverage scenarios. If the test environment contains a scoreboard, the number of outstanding transactions yet to be matched can be used as a crude estimate for the quantity of bits buffered or “in-flight” in the design, and can potentially be used to limit the relevance of a given sequence.

For such a method to work effectively, the scoreboard can be instrumented by adding two features: a function that returns the number of unmatched bits, and an event generated each time a transaction is observed at the output. Most scoreboards contain some sort of internal queue or array to keep track of transactions to be matched with DUT responses.

```
// Queue of transactions scoreboarded to be matched
uvm_sequence_item expect_q[$];
```

Code Stub 19: Scoreboard design using a queue of transactions that are expected to be matched.

The number of bits can be calculated as follows (once again assuming that `pack()` has been implemented in the transaction):

```
// Calculates the number of bits in the list of unmatched transactions
virtual function int unsigned unmatched();
   bit b[];
   return (expect_q.sum() with (item.pack(b)));
endfunction
```

Code Stub 20: Calculate the number of unmatched bits.

While a simple queue implementation is presented here, the `unmatched` feature can be added to other scoreboard implementations such as those using `uvm_tlm_analysis_fifo`.

Most scoreboards are connected using TLM, and implement write functions (for example `write_*`) to receive transactions. An event can be added to the scoreboard and triggered in these functions.

With these two framework additions, a relevance class can be created such that instead of checking if there are enough credits in a bucket to be relevant, it checks the scoreboard to see if there are too many unmatched bits compared to a threshold. If so, rather than wait a period of time for the bucked to be credited with enough tokens, it waits for a transaction to have been observed by the scoreboard, potentially reducing the number of outstanding bits.

```
// Limit based on bitcount.
class buffering extends relevance;

   // Maximum number of unmatched bits to be relevant
   longint unsigned max = -1;
   scoreboard       sb;
...
   // Check if scoreboard is not too full
   virtual function bit is_relevant();
       longint unsigned size;
       size = sb.unmatched();
       return size > max ? '0 : '1;
   endfunction

   // Wait for a matched event
   virtual task wait_for_relevant();
       @sb.matched;
   endtask

endclass
```

Code Stub 21: Buffering relevance control.

C. *Complex Scenario*

The culmination of the above methods and patterns results in a simple API that can be used to control sequences. Figure 9 is a graphical representation of the relationships that will be presented.

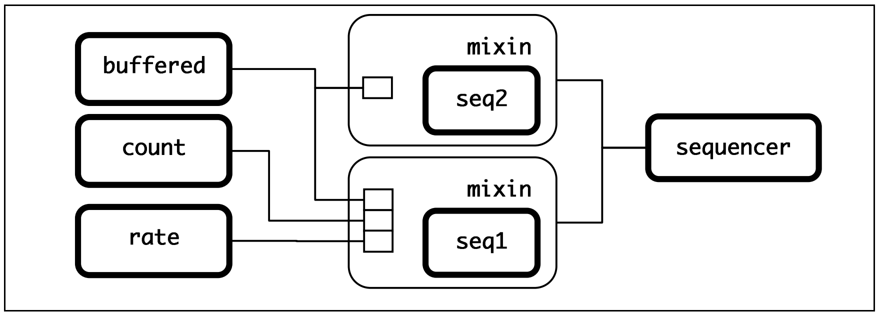

Figure 9: Sequence and relevance control relationship in complex scenario.

Two sequences have relevance controls mixed-in using `relevance_sequence_mixin`. Three relevance controls are created: rate, buffering, and count. The first sequence is associated with all the controls, whereas the second sequence "shares" the buffering control with the first. The first sequence represents a slow, but time-limited flow of traffic, whereas the second is high-bandwidth and long-running. Both sequences will respect a limited number of unmatched bits potentially buffered in the DUT.

```
// Convenience type definition of relevant traffic sequence
typedef relevance_sequence_mixin#(traffic_sequence) rt_seq_t;
rt_seq_t seq1;
rt_seq_t seq2;
...
// Control bitrate
rate = rate::type_id::create("rate");
rate.bitrate = 1_000_000; // 1Mbps
rate.burst = 2 * 1500 * 8; // two jumbo packets converted from bytes to bits

// Control the number of bits "in flight" using the number of unmatched bits in a scoreboard
buffered = buffering::type_id::create("buffered");
buffered.max = 16 * 1024 * 8; // 16 KiB -> bits

// Limit the number of packets
count = counter::type_id::create("count");
count.max = 100;

// First sequence is linked to all controls
seq1.assoc(rate);
seq1.assoc(buffered);
seq1.assoc(count);

// Second sequence shares in potential bits in flight
seq2.assoc(buffered);

// Apply AND/OR when combining the relevance of the individual controllers. Here, we are saying
the all must be relevant for the result to be asserted.
seq1.all_relevant = 1;

// Randomize sequences
if(!seq1.randomize()) `uvm_fatal("RNDERR", seq1.get_name())
if(!seq2.randomize()) `uvm_fatal("RNDERR", seq2.get_name())

// Start sequences
fork
    seq1.start(sequencer);
    seq2.start(sequencer);
join
```

Code Stub 22: Final example demonstrating multiple relevance controls, some of which are shared between sequences.

VI. CONCLUSION

The purpose of this paper was to provide deeper insight into a relatively unknown part of a sequence's selection and arbitration in order to add another tool to a verification engineer's proverbial toolbox. As an enhancement, other programming techniques such as mixins and the visitor pattern were introduced to improve reusability and composability. The outcome was used to control sequences in packet network applications. Other ways to use the relevance and arbitration mechanism exist, providing a rich canvas to create test scenarios and achieve coverage closure.

VII. References

[1] Rich Edelman, Raghu Ardeishar, "Sequence, Sequence on the Wall – Who's the Fairest of Them All? Using SystemVerilog UVM Sequences for Fun and Profit," in *DVCon*, San Jose, 2014.

[2] Accellera, Universal Verification Methodology (UVM) 1.1 User's Guide, 2011.

[3] Accellera, UVM User's Guide 1.2, 2014.

[4] Keisuke Shimizu, "ClueLogic UVM Tutorial for Candy Lovers – 26. Sequence Arbitration," 4 April 2015. [Online]. Available: http://cluelogic.com/2015/04/uvm-tutorial-for-candy-lovers-sequence-arbitration/.

[5] IEEE Std 802.3-2015.

[6] AMIQ Consulting, "amiq_eth," [Online]. Available: https://github.com/amiq-consulting/amiq_eth.

[7] Andrew S. Tanenbaum, Computer Networks, Fourth Edition, Prentice Hall PTR, 2003.

[8] James Strober, Corey Gross, "What Ever Happened to AOP?," in *DVCon*, San Jose, 2015.

[9] Martin Odersky, Lex Spoon, Bill Venners, Programming in Scala, Third Edition, Artima Press, 2016.

[10] Tudor Timi, "Verification Gentleman - Fake It 'til You Make It - Emulating Multiple Inheritance in SystemVerilog," 28 March 2015. [Online]. Available: http://blog.verificationgentleman.com/2014/09/emulating-multiple-inheritance-in-system-verilog.html.

[11] Eldon Nelson, "Design Patterns by Example for SystemVerilog Verification Environments Enabled by SystemVerilog 1800-2012," in *DVCon*, San Jose, 2016.

[12] Harry Foster, Michael Horn, Bob Oden, Pradeep Salla, Hans van der Schoot, "Verification Patterns – Taking Reuse to the Next Level," in *DVCon*, San Jose, 2016.

VIII. Appendix

The code for a complete example, including simple unit test, follows.

A. Relevance API Classes

```
// Forward declaration
typedef class relevance;

// Mixin that can be applied to a sequence
class relevance_sequence_mixin #(type SEQ = uvm_sequence) extends SEQ;
    localparam MSG_ID = "RELEVANT_SEQ";

    // AND/OR relevant result of associated controls (AND=1, OR=0)
    bit all_relevant = 1;

    // Suppressible warnings
    bit warn_no_assoc;                 // no controls were associated
    bit warn_already_assoc;            // control associated multiple times
    bit warn_repeat_wait_for_relevant; // zero time consumed when waiting

    //---------------------
    // Internal properties
    //---------------------

    // Queue of associated controls
    protected           relevance assoc_q[$];
    // Last call to wait_for_relevant
    protected time      last_wait_for_relevant;
    // Count consecutive wait calls
    protected int       wait_relevant_count;

    `uvm_object_param_utils(relevance_sequence_mixin#(SEQ))

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

    // Associates sequence and relevance control
    virtual function void assoc(relevance rel);
        register_assoc(rel);
        rel.register_sequence(this);
    endfunction

    // Determine relevance based on combining the relevance of all associated controls
    virtual function bit is_relevant();
        if(assoc_q.size() == 0) begin
            if(warn_no_assoc)
                `uvm_warning(MSG_ID, "(suppressible) No controls associated, deferring to superclass.")
            is_relevant = super.is_relevant();
        end else begin
            `uvm_info(MSG_ID, {"Check is_relevant(", all_relevant ? "any" : "all", ") with assoc queue: ",
                assoc_q_convert2string()}, UVM_DEBUG)

            is_relevant = 0;
            // Visit all controls
            foreach(assoc_q[i]) begin
                is_relevant = assoc_q[i].is_relevant();
                `uvm_info(MSG_ID, $sformatf("assoc '%s'.is_relevant() == %0d", assoc_q[i].get_name(), is_relevant), UVM_DEBUG)
                case(all_relevant)
                    0: if(is_relevant) break;  // Break if any is relevant (OR)
                    1: if(!is_relevant) break; // Break if one is not relevant (AND)
                endcase
            end
        end
    endfunction

    // Wait for shortest of associated controls' delay
    virtual task wait_for_relevant();
        time now;

        `uvm_info(MSG_ID, "wait_for_relevant()", UVM_DEBUG)
        now = $time;

        // Sanity check
        if(now == last_wait_for_relevant) begin
            wait_relevant_count++;
            if(warn_repeat_wait_for_relevant && (wait_relevant_count > 0))
                `uvm_fatal(MSG_ID, "Consecutive calls to wait_for_relevant during 0 simulation time.")
        end else begin
            wait_relevant_count    = 0;
            last_wait_for_relevant = now;
        end

        // Call associated controls' wait in parallel
        if(assoc_q.size() == 0)
            // Defer to superclass' wait
            super.wait_for_relevant();
        else begin
            bit wait_relevant_completed;
            fork
                begin
                    foreach(assoc_q[i]) begin
                        fork
                            automatic int k = i;
                            begin
                                assoc_q[k].wait_for_relevant();
                                wait_relevant_completed = 1;
                            end
                        join_none
                    end
                    wait(wait_relevant_completed == 1);
                end
            join_any
            disable fork;
        end
    endtask

    // Visit controls and finish item
    virtual task finish_item (uvm_sequence_item item, int set_priority = -1);
        `uvm_info(MSG_ID, "Enter finish_item()", UVM_DEBUG)
        super.finish_item(item, set_priority);
        foreach(assoc_q[i])
            assoc_q[i].finish_item(item);
    endtask

    // Bookkeeping
    virtual protected function void register_assoc(relevance rel);
        bit assoc_already_linked = 0;

        // Sanity check
        if(rel == null)
            `uvm_fatal(MSG_ID, $sformatf("'%s'.register_assoc() is called with 'null' handle", get_name()))
        `uvm_info(MSG_ID, $sformatf("Associating %s", rel.get_name()), UVM_DEBUG)
        // Scan all previously linked controls for duplicate association
        foreach(assoc_q[i])
            if(assoc_q[i] == rel)
                assoc_already_linked = 1;

        if(!assoc_already_linked)
            assoc_q.push_back(rel);
        else if(warn_already_assoc)
            `uvm_warning(MSG_ID, "(suppressible) Relevance already associated")
    endfunction

    // Stringify associated control queue
    virtual function string assoc_q_convert2string();
        string s = "{";
        if(assoc_q.size() > 0)
            s = {s, assoc_q[0].get_name()};
        for(int i = 1; i < assoc_q.size(); i++)
            s = {s, ",", assoc_q[i].get_name()};
        return {s, "}"};
    endfunction
endclass
```

File 1: relevance_sequence_mixin.sv

```
class relevance extends uvm_object;
    localparam MSG_ID = "RELEVANCE";

    // Suppressible warnings
    bit warn_seq_already_assoced;

    // Bit to detect if is_relevant() is implemented
    protected bit is_rel_default;

    // List of sequences associated with this control
    protected relevance_sequence_mixin assoc_q[$];
    `uvm_object_utils(relevance)

    function new (string name = "relevance");
        super.new(name);
        // Default Configuration
        warn_seq_already_assoced = 1;
    endfunction

    // Associate relevance control and sequence
    virtual function void register_sequence(relevance_sequence_mixin seq);
        bit seq_already_assoced = 0;

        // Sanity check
        if(seq == null)
            `uvm_fatal(MSG_ID, $sformatf("%s cannot be registered with null handle", get_name()))
        `uvm_info(MSG_ID, $sformatf("%s registering sequence %s", get_name(), seq.get_name()), UVM_DEBUG)
        // Scan all previously assoced sequences to figure out if this sequence is one of those.
        foreach(assoc_q[i])
            if(assoc_q[i] == seq)
                seq_already_assoced = 1;

        if(!seq_already_assoced)
            assoc_q.push_back(seq);
        else if(warn_seq_already_assoced)
            `uvm_warning(MSG_ID, $sformatf("(suppressible) Sequence %s already associated in sequence queue %s", seq.get_name(),
seq_q_convert2string()))
    endfunction

    // Default relevance, override for more interesting behaviour
    virtual function bit is_relevant();
        is_rel_default = 1;
        return 1;
    endfunction

    // Default wait_for_relevant, must be overridden to consume time
    virtual task wait_for_relevant();
        if(is_rel_default)
            `uvm_fatal(MSG_ID, "Sanity: should not execute when is_relevant() == 1")
        else
            `uvm_fatal(MSG_ID, "is_relevant() implemented without matching wait_for_relevant()")
    endtask

    // Offer pre actions for extending classes
    virtual function void start_item (uvm_sequence_item item);
    endfunction

    // Offer post actions for extending classes
    virtual function void finish_item (uvm_sequence_item item);
    endfunction

    // Stringify sequences associated with this control
    virtual function string seq_q_convert2string();
        string s = "{";
        if(assoc_q.size() > 0)
            s = {s, assoc_q[0].get_name()};
        for(int i = 1; i < assoc_q.size(); i++)
            s = {s, ",", assoc_q[i].get_name()};
        return {s, "}"};
    endfunction
endclass
```

File 2: relevance.sv

```
// Limit based on rate
class rate extends relevance;
    localparam MSG_ID = "RATE";

    //-----------------------------------------------------------------------
    // Configuration: protected to force use of setters/getters for validation
    //-----------------------------------------------------------------------
    protected int unsigned bitrate;          // desired rate
    protected time         update_period;    // how often to update bucket
    protected int unsigned max_burst;        // maximum burst size (bucket size)

    // Suppressible warnings for configuration validateion
    bit                    warn_long_update_period = 1;
    bit                    warn_slow_bitrate       = 1;

    // Work variables
    protected int unsigned bits_per_period;
    protected int unsigned bucket_limit;
    protected int unsigned bucket_limit_default = 10;
    protected bit          first_is_relevant_done;    // start bucket credits after first transaction
    protected int          bucket;                    // account balance in number of bits
    protected time         last_update_time;          // last time update was performed
    protected bit          var_has_changed = 1;       // flag indicating that the configuration changed
    // Validation flags
    protected bit          valid_bitrate;
    protected bit          valid_update_period;
    protected bit          valid_max_burst;

    `uvm_object_utils(rate)

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

    // Determines relevance based on bucket level and burst mode
    virtual function bit is_relevant();
        // If last_update_time is zero then assume we should start calculation from now
        if(first_is_relevant_done == 0) begin
            last_update_time       = $time();
            first_is_relevant_done = 1;
            update_derived_variable();
            bucket   = max_burst;
        end
        // Update bucket level
        update_bucket();
        `uvm_info(MSG_ID, $sformatf("is_relevant: bucket %0d bits", bucket), UVM_DEBUG)
        // relevant if bucket has tokens
        return (bucket >= 0) ? 1 : 0;
    endfunction

    // Waits until next update time where enough tokens may have accumulated.
    virtual task wait_for_relevant();
        time wait_time;

        wait_time = time_before_next_update();
        // Wait until the next Rate become relevant
        `uvm_info(MSG_ID, $sformatf("%s enter wait_for_relevant() with a delay of %0t", get_full_name(), wait_time), UVM_DEBUG)
        #(wait_time);
        `uvm_info(MSG_ID, $sformatf("%s exit wait_for_relevant()", get_full_name()), UVM_DEBUG)
    endtask

    // Debit item's size from bucket
    virtual function void finish_item (uvm_sequence_item item);
        bit bitstream[];
        int nb_of_bits;

        // use packing function to get number of bits
        nb_of_bits = item.pack(bitstream);
        bucket -= nb_of_bits;
    endfunction

    // Time delta passed
    virtual function time time_since_last_update();
        return $time() - last_update_time;
    endfunction

    // Time delta before update
    virtual function time time_before_next_update();
        return update_period - time_since_last_update();
    endfunction

    // Derive helper variables based on configuration
    virtual function void update_derived_variable();
        int unsigned    remainder_bpp;
        time            one_sec;

        if(!valid_update_period)
            `uvm_fatal(MSG_ID, "update_period not set, did you forget to call 'set_update_period(...)'?")
        if(!valid_bitrate)
            `uvm_fatal(MSG_ID, "bitrate not set, did you forget to call 'set_bitrate(...)'?")
        bits_per_period = (bitrate * update_period) / 1s;
        one_sec         = 1s;
        remainder_bpp   = (bitrate * update_period) % one_sec; // can be used for more precision

        // Limits
        if (max_burst == 0)
```

```
            // use default
            bucket_limit = bucket_limit_default*bits_per_period;
        else
            bucket_limit = max_burst;
    endfunction

    // Calculates the next bucket fill level
    virtual function void update_bucket();
        time            t_since_last_update;
        int unsigned    quotient_time;
        int unsigned    remainder_time;
        int             new_bucket;

        // Apply new config
        if(var_has_changed)
            update_derived_variable();
        // Determine delta in time
        t_since_last_update = time_since_last_update();

        if(t_since_last_update > 0) begin : NeedUpdate
            // Calculate time to derive number of bits to credit
            quotient_time  = t_since_last_update / update_period;
            remainder_time = t_since_last_update % update_period;

            // Sanity check
            if((quotient_time * update_period + remainder_time) != t_since_last_update)
                `uvm_fatal(MSG_ID, $sformatf("quotient_time(%0d) * get_period(%0d) + remainder_time(%0d) !=
t_since_last_update(%0t)",
                        quotient_time, update_period, remainder_time, t_since_last_update))

            // Determine new bucket level
            new_bucket = bucket + (quotient_time * bits_per_period);

            // Note: if(int > int unsigned) will make this true: if(-1000 > 10), need to cast
            if(new_bucket > int'(bucket_limit))
                new_bucket = bucket_limit;
            last_update_time = $time - remainder_time;

            // Debug info
            if (uvm_report_enabled(UVM_DEBUG)) begin
                string s = "";
                s = {s, $sformatf("\nt_since_last_update(%0t), update_period(%0t), quotient_time(%0d), remainder_time(%0d),
bits_per_period(%0d)",
                                   t_since_last_update, update_period, quotient_time, remainder_time, bits_per_period)};
                s = {s, $sformatf("\nlimit(%0d), old_bucket(%0d), new_bucket(%0d)",
                    bucket_limit, bucket, new_bucket)};
                `uvm_info(MSG_ID, {"Bucket update: ", s}, UVM_DEBUG)
            end

            // Update bucket
            bucket = new_bucket;
        end
    endfunction

    //---------------------------------------------------------------------
    // Setters & Getters for configuration parameters requiring validation
    //---------------------------------------------------------------------

    // Warn if bitrate is really slow, resulting in long simulation time
    virtual function void set_bitrate(int unsigned bitrate);
        valid_bitrate   = 1;
        var_has_changed = 1;
        if(warn_slow_bitrate && (bitrate < 1_000))
            `uvm_warning(MSG_ID, $sformatf("(suppressible) Bit rate of %0d bps seems so small!", bitrate))
        this.bitrate = bitrate;
    endfunction

    // Getter
    virtual function int unsigned get_bitrate();
        return bitrate;
    endfunction

    // Long update periods will result in poor precision
    virtual function void set_update_period(time update_period);
        valid_update_period = 1;
        var_has_changed     = 1;
        if(update_period == 0)
            `uvm_fatal(MSG_ID, $sformatf("update_period of %0t is invalid", update_period))
        if(warn_long_update_period &&(update_period > 1ms))
            `uvm_warning(MSG_ID, $sformatf("(suppressible) update_period of %0t seems long!", update_period))
        this.update_period  = update_period;
    endfunction

    // Getter
    virtual function time get_update_period();
        return update_period;
    endfunction

    // Max burst should be non-zero
    virtual function void set_max_burst(int unsigned max_burst);
        var_has_changed = 1;
        this.max_burst  = max_burst;
        if (this.max_burst == 0)
            `uvm_info(MSG_ID, "max_burst is zero, default bucket limit will be applied", UVM_FULL)
    endfunction

    // Getter
    virtual function int unsigned get_max_burst();
        return max_burst;
    endfunction
endclass
```

File 3: rate.sv

```
// Limit based on bitcount using a scoreboard's unmatched level.
class buffering extends relevance;
    localparam MSG_ID = "BUFFERING";

    // Maximum number of bits before not relevant
    longint unsigned max = -1;
    // Handle to scoreboard
    scoreboard       sb;

    `uvm_object_utils(buffering)

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

    // Relevant if the number of unmatched bits in scoreboard is less than configured
    virtual function bit is_relevant();
        longint unsigned size;
        size = sb.unmatched();
        `uvm_info(MSG_ID, $sformatf("is_relevant: %0d bits unmatched (max %0d)",
            size, max), UVM_DEBUG)
        return size > max ? '0 : '1;
    endfunction

    // Wait for a matched event, which should free some bits
    virtual task wait_for_relevant();
        @sb.matched;
    endtask

endclass
```

File 4: buffering.sv

B. *Unit Test Environment*

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

/***********************************************************************/
// Simple data packet modeled as stream of bytes
/***********************************************************************/
class item extends uvm_sequence_item;

    // Data stream
    rand byte unsigned data[];

    // Min to Jumbo Ethernet size
    constraint defaults {
        data.size() inside {[64:1536]};
    }

    `uvm_object_utils(item)
    function new(string name="item");
        super.new(name);
    endfunction
    // Pack item into stream of bits
    virtual function void do_pack(uvm_packer packer);
        super.do_pack(packer);
        `uvm_pack_arrayN(data, 8)
    endfunction

    virtual function bit do_compare(uvm_object rhs, uvm_comparer comparer);
        item rhs_;
        if (!$cast(rhs_, rhs))
            `uvm_fatal("CASTERR", "")
        do_compare = super.do_compare(rhs, comparer);
        do_compare &= (data == rhs_.data);
    endfunction

    virtual function void do_copy(uvm_object rhs);
        item rhs_;
        if(!$cast(rhs_, rhs))
            `uvm_fatal("CASTERR", "")
        super.do_copy(rhs);
        data    = rhs_.data;
    endfunction

    virtual function void do_record(uvm_recorder recorder);
        super.do_record(recorder);
        foreach(data[i])
            `uvm_record_field($sformatf("data%0d", i), data[i])
    endfunction

    virtual function string convert2string();
        string s = "";
        foreach (data[i]) begin
            if (i % 32 == 0) begin
                s = {s, "\n%4d", i};
            end
            s = {s, $sformatf(" %2h", data[i])};
        end
        return s;
    endfunction
endclass


/***********************************************************************/
// A fairly simple sequence to randomize "packets".
/***********************************************************************/
class fixed_length_seq extends uvm_sequence#(item);

    rand int iter   = 10;
    rand int length_min = 64;
    rand int length_max = 1500;

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

    virtual task body();
        req = item::type_id::create("req");

        repeat(iter) begin
            start_item(req);
            if (!req.randomize() with {
                data.size() inside {[local::length_min:local::length_max]};
            }) `uvm_fatal("RANDERR", "item")
            finish_item(req);
        end
    endtask
endclass


/***********************************************************************/
// Instrumented sequencer for better visibility of function/task calls
/***********************************************************************/
class sequencer extends uvm_sequencer#(item);
    `uvm_component_utils(sequencer)
    function new(string name="sequencer", uvm_component parent=null);
        super.new(name, parent);
    endfunction
    //-------------------------------------------------------
    // from UVM_SEQUENCER_BASE
    //-------------------------------------------------------

    virtual task wait_for_sequences();
        `uvm_info("uvm_sequencer_base", "wait_for_sequences", UVM_HIGH)
        super.wait_for_sequences();
    endtask

    virtual task wait_for_item_done(uvm_sequence_base sequence_ptr, int transaction_id);
        `uvm_info("uvm_sequencer_base", "wait_for_item_done", UVM_LOW)
        super.wait_for_item_done(sequence_ptr, transaction_id);
    endtask

    virtual task wait_for_grant(uvm_sequence_base sequence_ptr, int item_priority = -1, bit lock_request = 0);
        `uvm_info("uvm_sequencer_base", "wait_for_grant", UVM_HIGH)
        super.wait_for_grant(sequence_ptr, item_priority, lock_request);
    endtask

    virtual function integer user_priority_arbitration(integer avail_sequences[$]);
        `uvm_info("uvm_sequencer_base", "user_priority_arbitration", UVM_NONE)
        foreach(avail_sequences[i])
            `uvm_info("uvm_sequencer_base", $sformatf("    [%2d] %0d", i, avail_sequences[i]), UVM_LOW)
        foreach(reg_sequences[i])
            `uvm_info("uvm_sequencer_base", $sformatf("    [%2d] %0s", i, reg_sequences[i].get_name()), UVM_LOW)
        return super.user_priority_arbitration(avail_sequences);
    endfunction

    virtual task execute_item(uvm_sequence_item item);
        `uvm_info("uvm_sequencer_base", "execute_item", UVM_HIGH)
        super.execute_item(item);
    endtask

    //-------------------------------------------------------
    // from UVM_SEQUENCER_PARAM_BASE
    //-------------------------------------------------------

    virtual function void send_request(uvm_sequence_base sequence_ptr, uvm_sequence_item t, bit rerandomize = 0);
        `uvm_info("uvm_sequencer_param_base", "send_request", UVM_LOW)
        super.send_request(sequence_ptr, t, rerandomize);
    endfunction

    //-------------------------------------------------------
    // from UVM_SEQUENCER_PARAM_BASE
    //-------------------------------------------------------

    virtual function void item_done(RSP item = null);
        `uvm_info("uvm_sequencer", "item_done", UVM_LOW)
        super.item_done(item);
    endfunction
endclass


/***********************************************************************/
// Driver that consumes time based on stream length
/***********************************************************************/
class driver extends uvm_driver#(item);

    // testing environment without a monitor, send transactions for analysis from driver
    uvm_analysis_port#(item) p_notify;

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

    task run_phase(uvm_phase phase);
        fork
            begin
                item tr;
                forever begin
                    `uvm_info("driver", "get_next_item", UVM_LOW)
                    seq_item_port.get_next_item(tr);
                    `uvm_info("driver", "next_item", UVM_LOW)
                    void'(tr.begin_tr());

                    // generate delay based on size of data stream
                    #(1ns * tr.data.size() * 8);
                    tr.end_tr();
                    `uvm_info("driver", "item_done", UVM_LOW)
                    p_notify.write(tr);
                    seq_item_port.item_done(/*tr*/);
                end
            end
        join_none
    endtask
endclass


/*********************************************************************/
// Simple in-order (FIFO) scoreboard
/*********************************************************************/
class scoreboard extends uvm_scoreboard;
    // Declare analysis ports
    `uvm_analysis_imp_decl(_obs)
    `uvm_analysis_imp_decl(_exp)

    // Analysis ports - expected/input transactions
    uvm_analysis_imp_exp#(item, scoreboard) trans_exp_ap;
    // Analysis ports - observed/output transactions
    uvm_analysis_imp_obs#(item, scoreboard) trans_obs_ap;

    // Queues storing expected transactions by flow
    item expect_q[$];

    // Event when a match occurs
    event matched;

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

    function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        trans_obs_ap = new("trans_obs_ap", this);
        trans_exp_ap = new("trans_exp_ap", this);
    endfunction

    // Calculates the number of bits in the list of unmatched transactions
    virtual function int unsigned unmatched();
        bit b[];
        return (expect_q.sum() with (item.pack(b)));
    endfunction


    virtual function void write_exp(item trans);
        expect_q.push_back(trans);
    endfunction

    virtual function void write_obs(item trans);
        foreach(expect_q[i]) begin
            if (trans.data == expect_q[i].data) begin
                `uvm_info("MATCH","", UVM_LOW)
                expect_q.delete(i);
                // generate event that unmatched level has decreased
                -> matched;
                return;
            end
        end
    endfunction
endclass


/*****************************************************************************************************/
// Model a delay so that scoreboard may fill up with unmatched transactions (to test buffering control).
/*****************************************************************************************************/
class delay extends uvm_subscriber#(item);

    time delta = 20ns;

    uvm_analysis_port#(item) p_notify;

    `uvm_component_utils(delay)
    function new (string name="delay", uvm_component parent=null);
        super.new(name, parent);
        p_notify = new("p_notify", this);
    endfunction
    virtual function void write(T t);
        T cpy = new("cpy");
        cpy.copy(t);
        fork
            delta_delay(cpy);
        join_none
    endfunction

    virtual task delta_delay(T t);
        #delta;
        p_notify.write(t);
    endtask
endclass


/************************************************************************/
// Base unit testing environment
/************************************************************************/
class test extends uvm_test;

    sequencer  sqr;
    driver     drv;
    delay      buff;
    scoreboard sb;

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

    virtual function void build_phase(uvm_phase phase);
        super.build_phase(phase);

        uvm_config_db#(int)::set(null, "", "recording_detail", 1);

        sqr  = sequencer::type_id::create("sqr", this);
        drv  = driver::type_id::create("drv", this);
        buff = delay::type_id::create("buff", this);
        sb   = scoreboard::type_id::create("sb", this);
    endfunction

    virtual function void connect_phase(uvm_phase phase);
        super.connect_phase(phase);
        drv.seq_item_port.connect(sqr.seq_item_export);
        drv.p_notify.connect(sb.trans_exp_ap);
        drv.p_notify.connect(buff.analysis_export);
        buff.p_notify.connect(sb.trans_obs_ap);
    endfunction

    virtual task run_phase(uvm_phase phase);
        super.run_phase(phase);

        phase.raise_objection(this);
        phase.drop_objection(this);
    endtask

endclass
```

File 5: base.sv

```
`include "base.sv"
`include "relevance_sequence_mixin.sv"
`include "relevance.sv"
`include "rate.sv"
`include "buffering.sv"

/************************************************************************/
// Test relevance controls
/************************************************************************/
class relevance_api extends test;

    typedef relevance_sequence_mixin#(fixed_length_seq) seq_t;

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

    virtual task main_phase(uvm_phase phase);
```

```
        seq_t seq1          = seq_t::type_id::create("seq1");
        seq_t seq2          = seq_t::type_id::create("seq2");
        rate limit_250Mbps;
        rate limit_750Mbps;
        buffering limit_16KiB;

        phase.raise_objection(this);

        // Configure environment
        sqr.set_arbitration(SEQ_ARB_WEIGHTED);
        buff.delta = 10us;

        // Control bitrate
        limit_250Mbps = rate::type_id::create("limit_250Mbps");
        limit_250Mbps.set_bitrate(250_000_000); // 250Mbps
        limit_250Mbps.set_update_period(10ns);

        limit_750Mbps = rate::type_id::create("limit_750Mbps");
        limit_750Mbps.set_bitrate(750_000_000); // 750Mbps
        limit_750Mbps.set_update_period(10ns);

        // Controls the number of bits "in flight",
        // Implemented using the number of unmatched bits in a scoreboard
        limit_16KiB     = buffering::type_id::create("limit_16KiB");
        limit_16KiB.sb  = this.sb;
        limit_16KiB.max = 16 * 1024 * 8; // 16 KiB -> bits

        // Link all controls to be visited
        seq1.assoc(limit_250Mbps);
        seq1.assoc(limit_16KiB);
        // Apply AND/OR when combining the relevance of the individual controllers. Here, we are saying the all must be relevant
for the result to be asserted.
        seq1.all_relevant = 1;

        // Second sequence shares in potential bits in flight
        seq2.assoc(limit_16KiB);
        seq2.assoc(limit_750Mbps);
        seq2.all_relevant = 1;

        // Start sequences
        fork
            seq1.start(sqr);
            seq2.start(sqr);
        join

        phase.drop_objection(this);
    endtask

endclass

/***********************************************************************/
// Run test
/***********************************************************************/
module top();
    initial begin
        run_test();
    end
endmodule
```

File 6: unit.sv