Scalable Mixed Features Stimulus Generation for Cluster Network Using Sequence Decorator

Chi-Ming Li, Synopsys Inc., Hsinchu City, Taiwan (*Chi-Ming.Li@synopsys.com*)

***Abstract*—Verification engineers put in a great effort designing stimulus for bug hunting and coverage closure. In Constrained random verification (CRV) methodology like UVM, generating versatile stimulus for stressing DUT usually requires a lot of hand-crafted sequence classes with constraints for addressing specific verification concerns. As the project goes on, the number of such sequence classes surge which eventually leads to combinatorial explosion. In this paper we propose a scalable way to design and manage sequence classes using the sequence decorator, which is a structural design pattern commonly used in Object-oriented programming (OOP). We also demonstrate how the sequence decorator can be applied to stressing DUT with massive concurrent transitioning state space like an out-of-order processor cluster network.**

Keywords*—*UVM; Mixed Features Test; Design Pattern

I. INTRODUCTION

Finding corner-case bugs is a major challenge for design verification as many of them are mixed features related and deep-state dependent, which often require comprehensive constrained randomness to hit. Ideally we want such mixed features stimulus activating multiple hardware features concurrently, when they might also be required to drive DUT states into a specific condition that is not obvious when we profile direct cases. Unfortunately, determining and implementing all possible mixed features stimulus in an efficient and scalable manner is a big challenge because the number of feature combinations can grow exponentially as the DUT becomes more complicated. As a result, verification engineers either end up produces over-proliferated stimulus codes that often overloads available resources, or compromises the stimulus versatility and wish that no bugs left behind. In this paper, we briefly summarize a commonly used stimulus development flow, followed by illustration of stimulus scalability issue arises from the flow when verifying a complicated design like an out-of-order capable cluster network (CLN). And then at the end we proof sequence decorator could solve the scalability issue of mix-features stimulus generation in a reasonable and affordable manner

II. COMMON STIMULUS DEVELOPMENT FLOW

The prevailing technique for conducting stimulus generation in UVM is to apply scenario constrains at various levels of abstraction. One common hierarchy for modeling stimulus is depicted in Figure 1. A transaction is a primitive operation of a hardware interface. A sequence aggregates multiple transactions to implement a single scenario for an interface. A virtual sequence controls and coordinates sequences which are dispatched to various hardware interfaces for modeling a high-level scenario. The test level provides user control knobs for steering or switching among high-level scenario sequences without knowledge of implementation detail.

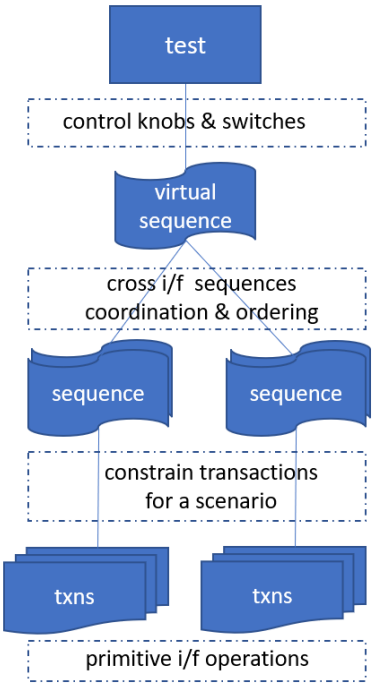


Figure 1. Common stimulus modeling hierarchy

For example, in CLN verification, a transaction is usually a memory-mapped read or write with many attributes like data size, cacheability, transaction ID and so on. We usually constrain legal transaction attributes space at the transaction-level itself and implement verification engineering's intents into scenarios by creating constraints at sequence level. This technique is crucial for verifying complex DUT with massive concurrent transitioning state space like CLN as many corner cases requires deep-state dependent mixed feature combination to be hit, which usually needs to be constructed with certain levels of abstraction by engineers for steering constraints at higher level.

The constraints inside a scenario sequence can be categorized into two kinds: declarative constraints and prerequisite constraints. The declarative constraints are described in conventional SV constraint constructs, which specify parameters and attributes of intent for a scenario sequence. The prerequisite constraints, however, are setups or preparations steps required before execution of a scenario sequence. An example of scenario-based sequence implementation can be found in Figure 2. The intent is to create scenarios having multiple concurrent cacheable transaction sequences, which result in back-to-back collisions on same cache structure index. In the declarative constraints part, we constrain the cacheability attribute (i.e., axcache) to be cacheable and transaction addresses to be mapped to same cache index regardless read or write. In the prerequisite constraints part, we enable the cache feature and set up the related memory maps for CLN whenever necessary. This sequence is highly reusable and can be randomized on-demand to construct more interesting scenarios by running in parallel or layering for abstraction with other sequences.

```
class cln_txn_item extends uvm_sequence_item;

  rand int data;
  rand int address;
  rand read_write_e read_write;
  rand bit cacheable;
  rand bit[7:0] qos_id;
  rand bit[7:0] cmd_id;

  rand bit[15:0] tag;
  rand bit[9:0] idx;

  constraint c_cache {
    address[15:6] == idx;
    address[31:16] == tag;
  }

endclass

class cln_sequence extends uvm_sequence #(cln_txn_item);

  int len = `SEQ_SIZE;
  rand cln_txn_item txns[];

  function new(string name="cln_sequence");
    super.new(name);
    txns = new[len];
    foreach(txns[i])
      txns[i] = new();
  endfunction

  virtual task prerequisite_setup();
    // to be extended, do nothing in base class
  endtask

  function void pre_randomize();
  endfunction

  task body();
    prerequisite_setup();
    foreach(txns[i]) begin
      start_item(txns[i]);
      // don't need randomize() here
      // as they were already randomized
      // at higher level of sequence hierarchy
      finish_item(txns[i]);
    end
  endtask

endclass
```

```
class cache_conflict_sequence extends cln_sequence;

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

  task prerequisite_setup();
    $display("enable cache and memory maps");
    // access configuration registers
    // to enable cache and memory maps
  endtask

  constraint c_same_idx {
    foreach(txns[i]) {
      txns[i].idx == txns[0].idx;
    }
  }

  constraint c_cacheable {
    foreach(txns[i]) {
      txns[i].cacheable == 1;
    }
  }

endclass
```

Figure 2. cache line conflict stress sequence

III. SEQUENCE SCALABILITY ISSUE

In the early phase of verification, a scenario sequence usually addresses single hardware feature. As the project goes on, the verification work usually evolves from single feature to mixed features, which demands much more sequence classes for taking care of mixed features concerns. Such constraints crafted for mixed features are very likely to duplicate from the ones already scattered around the existing sequences, which leads to tedious coding of fusing old constraints from various sequence classes into a new sequence class. Such duplication of sequence constraints is hard to maintain and error-prone. Most significantly, when the number of features grows to a larger number N, the corresponding mixed features combinations surges to 2^N, which is also known as combinational explosion. Such a scalability issue is shown in Figure 3.

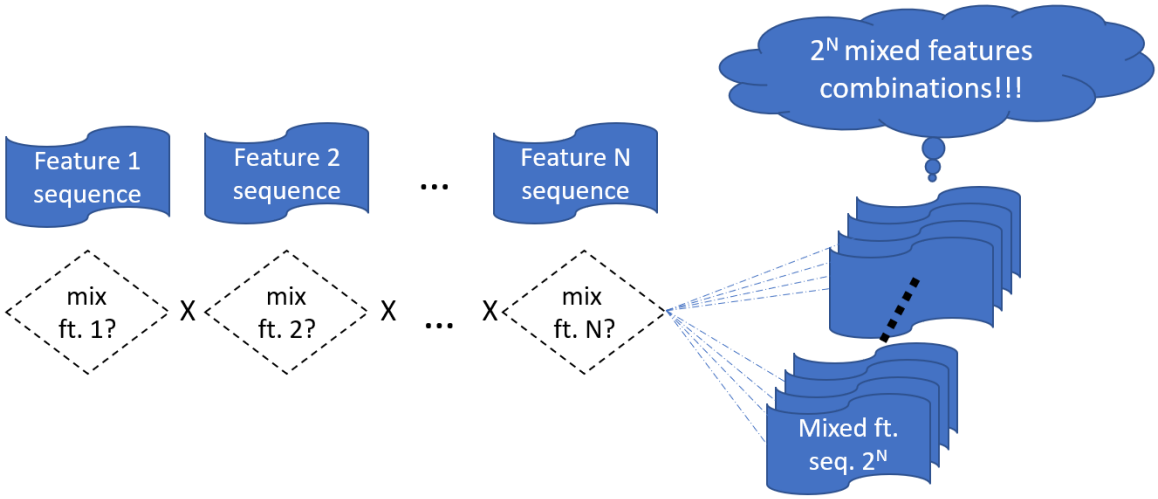


Figure 3. Combinational explosion

For example, a QoS feature used to be verified separately, which is configured by specifying priority of QoS ID associated with each transaction. Such priority setting affects cache replacement policy when cache conflicts occur, where the cache line with lower QoS ID priority is evicted. A sequence for verifying the QoS is shown in Figure 4. To stress DUT further with mixed scenarios, we need to fuse existing scenario-based sequences. Suppose we want to fuse the cache-conflict sequence with a QoS sequence to verify the mixed features scenario where cache line conflicts occur frequently when the QoS feature is also enabled. By an intuitive approach, we must create a new sequence class called "cache-conflict-QoS sequence" which inevitably duplicates both declarative and prerequisite constraints from the original. It is possible to reduce the duplication to half by extending the cache-conflict sequence with new constraints from cache-QoS (or the other way around), but this could result in over-complicated sequences inheritance hierarchy. Such tricky code proliferation due to mixing two features is illustrated in Figure 4.

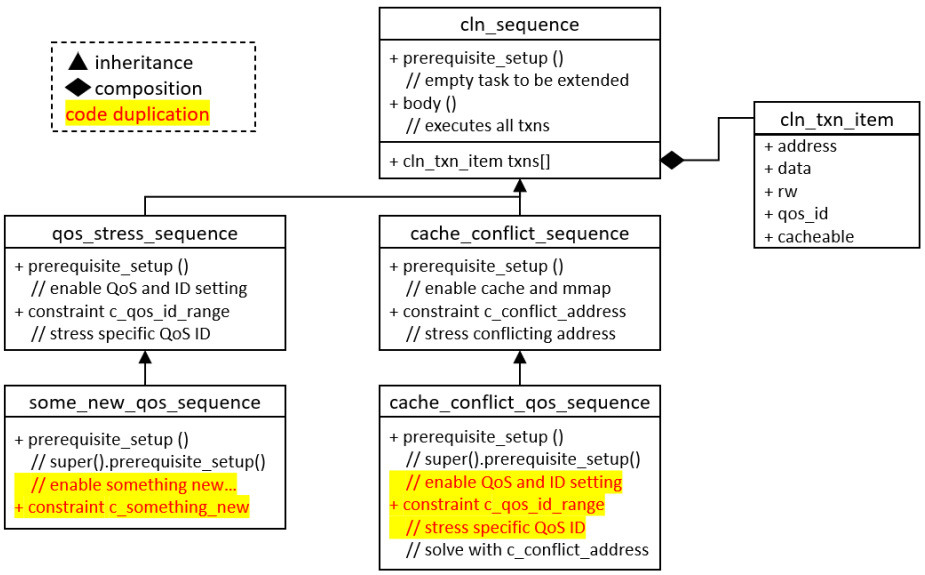


Figure 4. Code proliferation when mixing two features.

Bromley et al. proposed the pairwise-testing pattern [1] to stimulus generation, which reduce the mixed features combinations effectively using the pairwise heuristic. Nevertheless, the stimulus constraint duplication persists. To mitigate such code duplication, Dickol et al. proposed the constraint policy method [2], where the principle is to

encapsulate sequence constraints into a dedicated policy class, which is latter aggregated by the sequence to construct mixed constraints. However, this approach only applies to declarative constraints but prerequisite constraints. In addition, the policy class brings additional implementation overhead of refactoring the existing sequence code.

IV. SEQUENCE DECORATOR

To address the sequence scalability issue effectively with minimum modification to existing sequences code. We propose the sequence decorator method, which is inspired by the decorator design pattern introduced by Gang of Four (GoF) [3]. Figure 5. Illustrates the UML diagram of the sequence decorator class. Instead of re-implementing constraints explicitly in new sequence code, it aggregates various sequences dynamically with a rand sequence queue. Each individual sequence binds its declarative constraints to the sequence decorator using add_mixture() function, which assigns transaction handles of individual sequence to the corresponding transaction objects of the sequence decorator so that the declarative constraints can later be applied to the transaction objects. Once multiple sequences are bound to the sequence decorator, randomizing this sequence decorator enables the constraint solver to put all declarative constraints from the sequence queue together and produce an aggregated constrained randomization result for transaction objects. On the other hand, the prerequisite constraints are aggregated and applied by the sequence decorator inside its prerequisite_setup() task, which walks through all the individual prerequisites_setup() tasks from the sequence queue. This sequence decorator patterns provides several advantages for mixed features verification. In the example of cache-QoS stimulus, the sequence code can be greatly simplified, even into a one-liner for each individual feature or scenario with macros. The code snippet is shown in Figure 6.

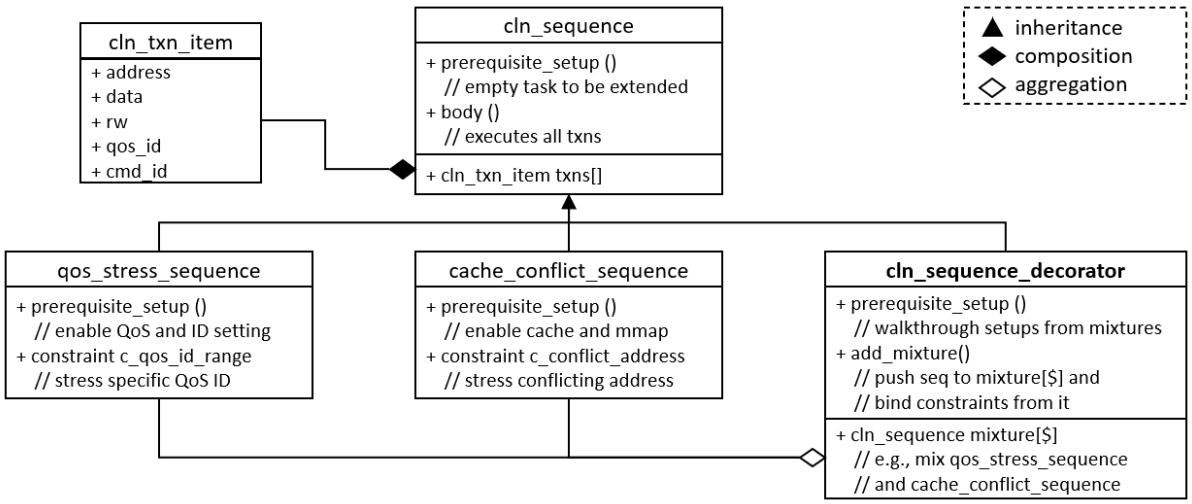
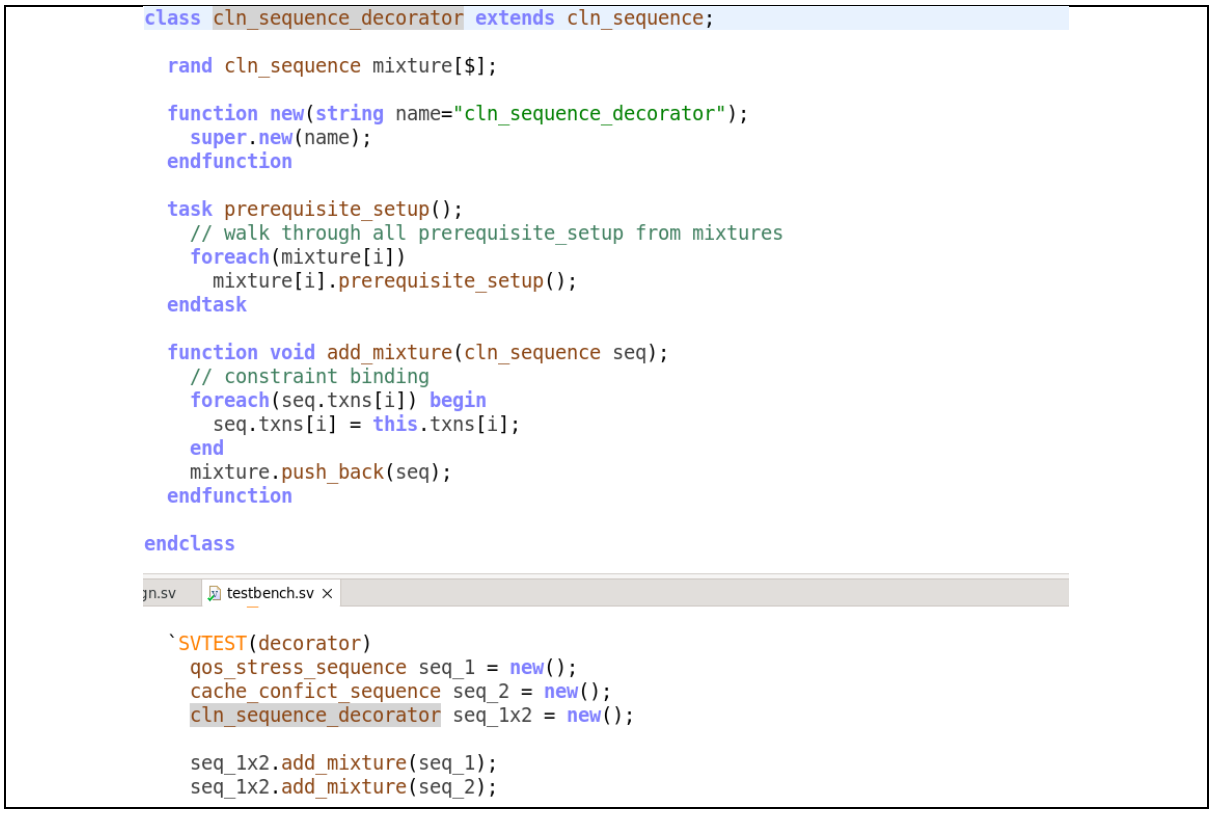


Figure 5. Sequence Decorator



```
class cln_sequence_decorator extends cln_sequence;

  rand cln_sequence mixture[$];

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

  task prerequisite_setup();
    // walk through all prerequisite_setup from mixtures
    foreach(mixture[i])
      mixture[i].prerequisite_setup();
  endtask

  function void add_mixture(cln_sequence seq);
    // constraint binding
    foreach(seq.txns[i]) begin
      seq.txns[i] = this.txns[i];
    end
    mixture.push_back(seq);
  endfunction

endclass
```

```
gn.sv   testbench.sv ×

  `SVTEST(decorator)
    qos_stress_sequence seq_1 = new();
    cache_confict_sequence seq_2 = new();
    cln_sequence_decorator seq_1x2 = new();

    seq_1x2.add_mixture(seq_1);
    seq_1x2.add_mixture(seq_2);
```

Figure 6. Simplified code with sequence decorator.

To generate all combinations for n mixed features efficiently, we can further extend the sequence decorator with n-wise consideration where n individual feature sequences have been added to its sequence queue. We implement the pre_randomize() hook which maps a serial number k inside [0 .. 2^n-1] to a combination of mixed features. We can consider the binary of k as a representation of features mixture, where m-th set denoting m-th feature is mixed. The mapping is a bijection so all the combinations can be traversed exhaustively by iterating the number from 0 to 2^n-1. The code snippet is shown in Figure 7.

```
class all_combination_sequence extends cln_sequence_decorator;

  int comb_id;
  cln_sequence mixture_backup[$];

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

  function void pre_randomize();
    cln_sequence new_mixture[$];
    // save full mixture so we can restore when iterating next comb_id
    mixture_backup = mixture;
    for(int i=0; i<mixture.size(); i++) begin
      // each digit of comb_id represents sequence mixture or not
      if(comb_id[i])
        new_mixture.push_back(mixture[i]);
    end
    mixture = new_mixture;
  endfunction

  function restore_mixture();
    mixture = mixture_backup;
  endfunction

endclass
```

gn.sv | testbench.sv ×

```
  `SVTEST(all_comb)
    all_combination_sequence seq_1x2 = new();
    qos_stress_sequence seq_1 = new();
    cache_confict_sequence seq_2 = new();
    seq_1x2.add_mixture(seq_1);
    seq_1x2.add_mixture(seq_2);

    for(int i=0; i<2**seq_1x2.mixture.size(); i++) begin
      seq_1x2.comb_id = i;
      if(!seq_1x2.randomize())
        `uvm_fatal("RAND FAIL", $psprintf(seq_1x2.get_type_name()))
      seq_1x2.start(sequencer);
      seq_1x2.restore_mixture();
    end
```

Figure 7. All combinations sequence example

V. CONCLUSION

In this paper, we introduce the sequence scalability issue for mixed features test which is crucial for corner-case bugs finding. We inspect the prevailing stimulus development flow and how the scalability issue arises with CLN verification example. We demonstrate the sequence decorator with code example to address this issue. With the sequence decorator, the verification engineers can develop a specific mixed features stimulus rapidly, as well as traverse all combinations of mixed features stimulus.

REFERENCES

[1] Jonathan Bromley and Kevin Johnston, “Is Your Testing N-wise or Unwise? Pairwise and N-wise Patterns in SystemVerilog for Efficient Test Configuration and Stimulus” Design and Verification Conference & Exhibition Europe, 2015.

[2] John Dickol,” I Didn’t Know Constraints Could Do That!”, DVClub Europe, 2018.

[3] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Design Patterns : Elements of Reusable Object-Oriented Software. Reading, Mass. :Addison-Wesley, 1995.