2022 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

What Does The Sequence Say? Powering Productivity with Polymorphism

Rich Edelman

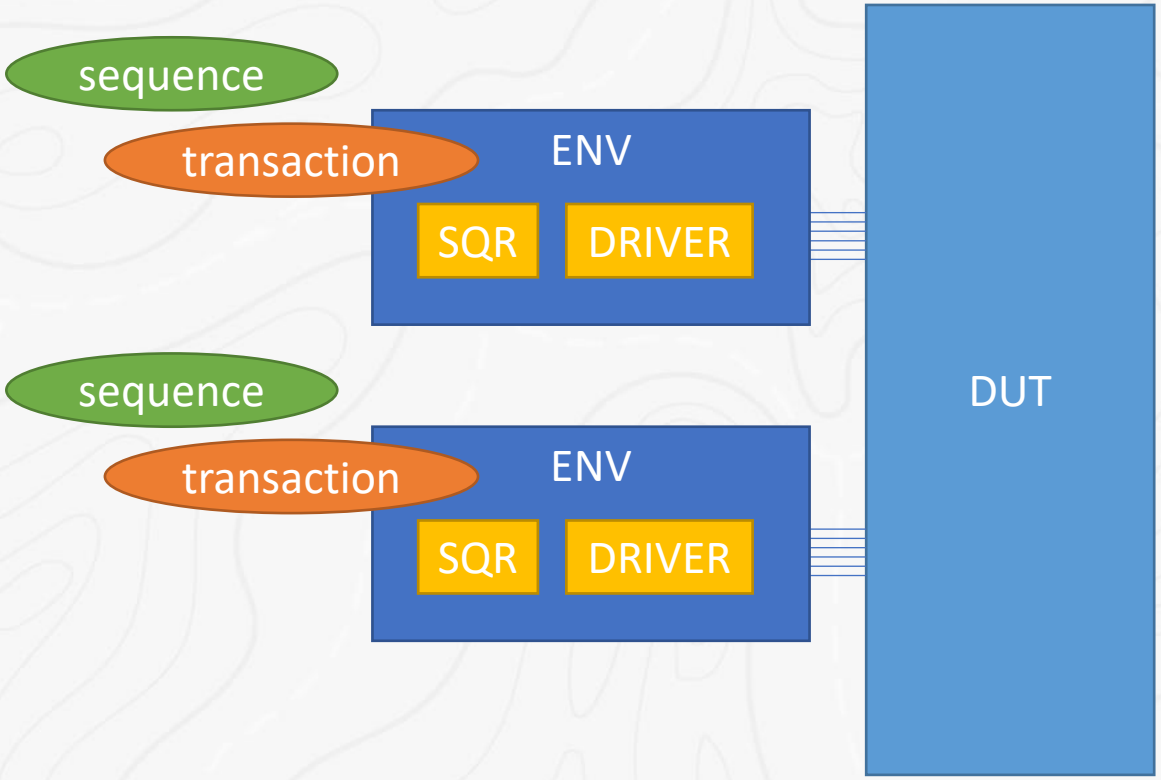
Siemens EDA

SIEMENS

accellera SYSTEMS INITIATIVE

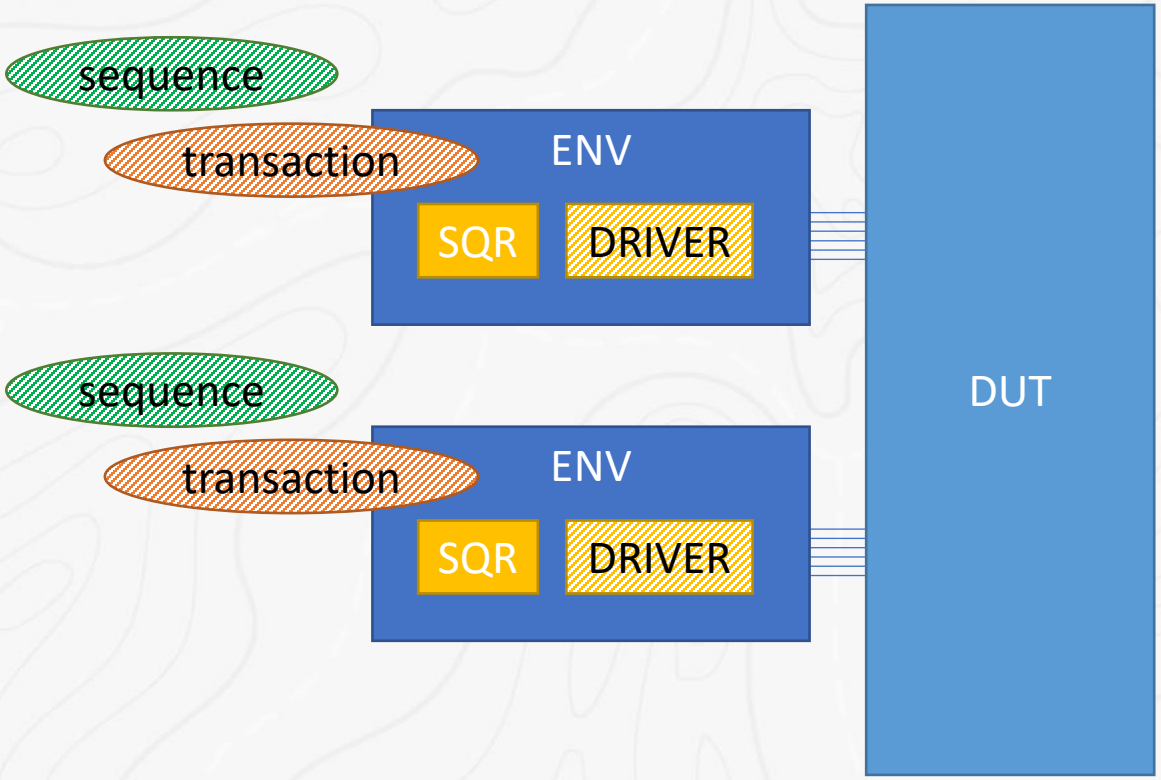
The UVM Environment

- Two interfaces
- One
  - Driver
  - Transaction
  - Sequence
- Used Twice



The Polymorphic UVM Environment

- Replace sequences
- Replace transactions
- With “BETTER” versions



Get More By Doing Less – "BETTER" objects

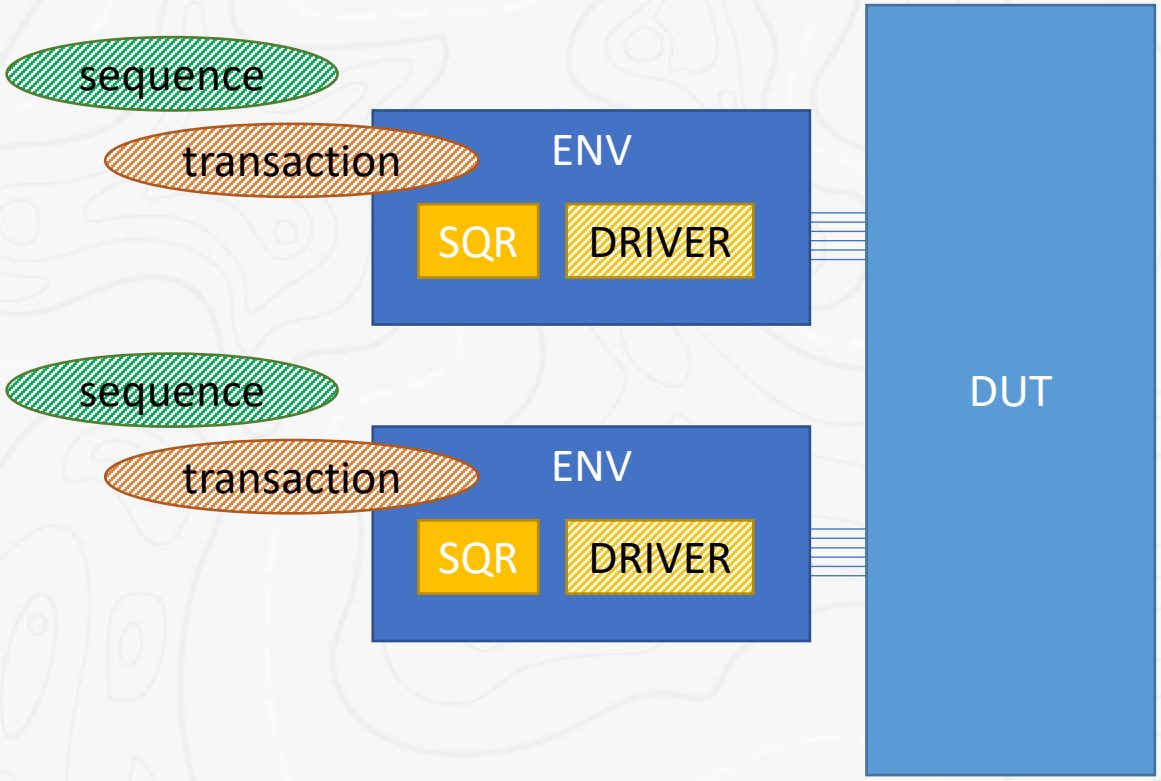
- The promise of object-oriented programming and polymorphism
- **What IS Polymorphism?**
  - In biology, polymorphism is the **occurrence of two or more clearly different morphs or forms**, also referred to as alternative phenotypes, in the population of a species. ... The term polyphenism can be used to clarify that the **different forms arise from the same genotype**.
  - In programming languages and type theory, polymorphism is the provision of a single interface to entities of different types or the use of a single symbol to represent multiple different types. The concept is borrowed from a principle in biology where **an organism or species can have many different forms** or stages.

What Makes A Better UVM Verification Object?

- A better transaction
  - Bigger
  - Smaller
  - Different constraints
- A better sequence
  - Different “program”
    - Self checking
    - Bandwidth requirements / Latency – video, uart, memory, etc
- A better driver
  - Handle Interrupts

Our Verification Strategy

- Build a simple UVM testbench
- Extend the test capabilities using polymorphism
  - transaction
  - sequence
  - driver
- Reuse this strategy on a larger testbench
  - virtual sequence
  - interrupt testing
  - synchronized testing



SystemVerilog UVM →Polymorphism

- Simple idea. Easy to say it out loud -- *"is-a" relationship*
  - A racquetball is-a ball
  - A basketball is-a ball
  - A racquetball is-a basketball

HOMEWORK: ONE of these bullets is incorrect.

- SystemVerilog
  - `class super_transaction extends transaction;`

- A super_transaction **IS-A** transaction

OOP – Ball – class definitions

```
class ball;
  virtual function string get_name();
    return "Ball";
  endfunction

  virtual function void print();
     $display("%s", get_name());
  endfunction
endclass

class basketball extends ball;
  virtual function string get_name();
    return {"Basketball", "<=", super.get_name()};
  endfunction
endclass

class kids_basketball extends basketball;
  virtual function string get_name();
    return {"Kids Basketball", "<=", super.get_name()};
  endfunction
endclass
```

```
class pro_basketball extends basketball;
  virtual function string get_name();
    return {"Pro Basketball", "<=",
      super.get_name()};
  endfunction
endclass

class racquetball extends ball;
  virtual function string get_name();
    return {"Racquetball", "<=",
      super.get_name()};
  endfunction
endclass
```

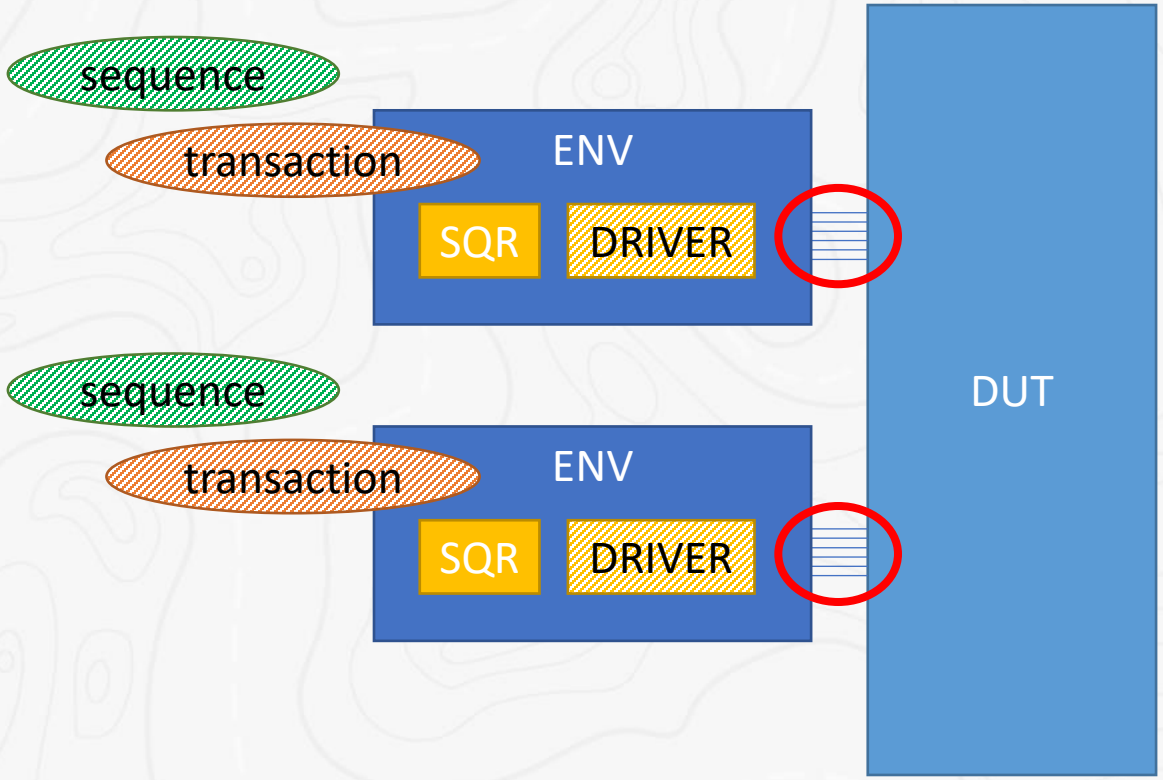
OOP – Ball – usage

```
module top();

  ball            BALL;
  basketball      BASKETBALL;

  kids_basketball KIDS_BASKETBALL;
  pro_basketball  PRO_BASKETBALL;

  racquetball     RACQUETBALL;

  ball            ANY_BALL[5];
```

```
  initial begin
    BALL = new();
    BASKETBALL = new();
    KIDS_BASKETBALL = new();
    PRO_BASKETBALL = new();
    RACQUETBALL = new();

    ANY_BALL[0] = BALL;
    ANY_BALL[1] = BASKETBALL;
    ANY_BALL[2] = KIDS_BASKETBALL;
    ANY_BALL[3] = PRO_BASKETBALL;
    ANY_BALL[4] = RACQUETBALL;

    foreach (ANY_BALL[i])
      ANY_BALL[i].print();

    BALL = RACQUETBALL;
    BALL.print();
  end
endmodule
```

Our Verification environment – The abus

```
interface abus (input clk, input reset);
  reg rw;
  reg [31:0] addr;
  reg  [7:0] wdata;
  reg  [7:0] rdata;
  reg  [5:0] id;
  reg ready;
  reg valid;
  reg interruptA;
  reg interruptB;
endinterface
```



The Transaction – Class Variables and Constraints

- Class member variables
  - Rw
  - Addr
  - Data
  - Low_addr
  - High_addr
  - Id
- Constraints
  - Legal_range
  - Max_length

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

  // Read and Write to Memory
  rand rw_t       rw;
  rand reg [31:0] addr;
  rand reg  [7:0] data [];

       reg [31:0] low_addr;
       reg [31:0] high_addr;

       reg  [7:0] id;

  constraint legal_range {
    addr >= low_addr;
    addr <= high_addr;
  }

  constraint max_length {
    data.size() < 16;
    data.size() > 0;
  }
```

The Transaction – Helpers

- Convert2string
  - Pretty print
- Do_record
  - Transaction recording
- Do_compare

```
  function string convert2string();
    return $sformatf("%s %0d %p (%s) id=%0d",
      (rw==READ)?"READ":"WRITE", addr, data, get_type_name(), id);
  endfunction

  function void do_record(uvm_recorder recorder);
    super.do_record(recorder);
    `uvm_record_field("name", get_name())
    `uvm_record_field("rw", rw)
    `uvm_record_field("addr", addr)
    `uvm_record_field("data", data)
    `uvm_record_field("id", id)
  endfunction
  function bit do_compare(uvm_object rhs, uvm_comparer comparer);
    transaction rhs_;
    $cast(rhs_, rhs);
    if (id != rhs_.id ) return 0;
    if (rw != rhs_.rw ) return 0;
    if (addr != rhs_.addr ) return 0;
    if (data.size() != rhs_.data.size()) return 0;
    foreach (data[i])
      if (data[i] != rhs_.data[i] ) return 0;
    return 1;
  endfunction
endclass
```

The Basic Sequence

- Class member variables
  - Transaction Handle
  - Name
  - Maximum Transactions
- Loop, sending random transactions

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

  transaction tr;
  string name;
  int maximum_transactions;

  function new(string name = "basic_sequence");
    super.new(name);
    if (maximum_transactions == 0)
      maximum_transactions = 100;
  endfunction

  task body();
    for (int i = 0; i < maximum_transactions; i++) begin
      name = $sformatf("transaction%0d", i);
      tr = transaction::type_id::create(name);
      start_item(tr);
      if (!tr.randomize()) begin
        `uvm_info(get_type_name(), "Randomize FAILED",
         UVM_MEDIUM)
      end
      finish_item(tr);
    end
  endtask
endclass
```

Odd Addresses Transaction – Extended

- Add a new constraint

```
class odd_address_transaction extends transaction;
  `uvm_object_utils(odd_address_transaction)

  constraint odd_address {
    addr[0] == 1;
  }

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

Big Transaction – Extended

- Override an existing constraint

```
class big_transaction extends transaction;
  `uvm_object_utils(big_transaction)

  constraint max_length {
    data.size() < 255;
    data.size() > 63;
  }

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

Double Transaction – Extended

- Add class member
- Add two constraints
- Enhance convert2string
- Update compare and record

```
class double_transaction extends transaction;
  `uvm_object_utils(double_transaction)

  rand reg [31:0] addr2;

  constraint legal_range_addr2 {
    addr2 >= low_addr;
    addr2 <= high_addr;
  }

  constraint addr2_value {
    addr2 != addr;
  }

  function string convert2string();
    return $sformatf("%s (addr2=%0d)",
      super.convert2string(), addr2);
  endfunction

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

Checker Sequence – Extended

- A “Better” basic sequence
- Add Class member
  - Wdata – array of data written
- Loop
  - Send WRITE
  - Send READ from same address
  - Do a compare

```
class checker_sequence extends basic_sequence;
  `uvm_object_utils(checker_sequence)

  reg [7:0] wdata [];
  task body();
    for (int i = 0; i < maximum_transactions; i++) begin
      name = $sformatf("transaction%0d", i);
      tr = transaction::type_id::create(“name”);
      start_item(tr);
      if (!tr.randomize()) begin ... end
      tr.rw = WRITE;   // Force WRITE.
      finish_item(tr);
      wdata = tr.data; // Save the data written
      tr.rw = READ;    // Force READ.
      start_item(tr);
      finish_item(tr);

      // Check
      if (wdata != tr.data)
        `uvm_info(get_type_name(),
         $sformatf("Mismatch Wrote %p, Read %p",
           wdata, tr.data), UVM_MEDIUM)
    end
  endtask
endclass
```

How to get “better” objects? The Factory!

- The UVM Factory replaces one thing with one thing

```
basic_sequence::type_id::set_type_override(
    checker_sequence::get_type(),1);
```

- For example
  - Every basic_sequence with a checker_sequence
  - Every transaction with an odd_address_transaction
  - Every transaction with a big_transaction

Oops!
We don’t want just ONE kind of **transaction** – we want a mix

Instead of a “factory”, we use a “picker”?

- Register by using ‘add()’
- Get a type using ‘get()’
- Simple queue of objects

- Many possible implementations
- This one is complete and small (vs. uvm_factory.svh)
  - 1.1d - 1632 lines
  - IEEE - 1967 lines

```
class transaction_picker;
  static uvm_object_wrapper types[$];
  static int                number[$];

  static function void add(uvm_object_wrapper w);
    types.push_front(w);
  endfunction

  static function transaction get();
    transaction tr;
    string name;
    uvm_object o;
    int d;
    d = $urandom_range(types.size()-1, 0);
    o = types[d].create_object("transaction_picker");
    name = $sformatf("%s%0d", types[d].get_type_name(),
      number[d]++);
    o.set_name(name);
    $cast(tr, o);
    return tr;
  endfunction
endclass
```

Using the Picker

- Add the types to the picker

```
transaction_picker::add(            transaction::get_type());
transaction_picker::add(odd_address_transaction::get_type());
transaction_picker::add(        big_transaction::get_type());
transaction_picker::add(     double_transaction::get_type());
```

- Replace the call to the factory interface – **create()**

```
tr = transaction::type_id::create(name);
```

- With a call to the Picker interface – **get()**

```
tr = transaction_picker::get();
```

The Interrupt Transaction

- Has a ‘count’
- Has an ‘event’
- Convert2string()

```
class interrupt_transactionA extends transaction;
  `uvm_object_utils(interrupt_transactionA)

  int icount;
  event w;

  function string convert2string();
    return $sformatf("%s (icount=%0d)",
      super.convert2string(), icount);
  endfunction
endclass
```

The Interrupt Sequence

```
class interrupt_sequenceA extends basic_sequence;
  `uvm_object_utils(interrupt_sequenceA)

  interrupt_transactionA itrA;

  task body();
    forever begin
      itrA = interrupt_transactionA::type_id::create("interrupt_transactionA");
      start_item(itrA);
      finish_item(itrA);
      wait(itrA.w); // Will hang here (in the driver fork/join_none thread)
                    //   until the interrupt occurs
    end
  endtask
endclass
```

The Driver Interrupt Handler

```
task run_phase(uvm_phase phase);
  forever begin
    seq_item_port.get_next_item(t);
    if ($cast(itrA, t)) begin // ISR processing
      fork
        isr(itrA);
      join_none;
    end
    else if ($cast(oooA, t)) begin // Other processing
      ...
    end
    else begin // Normal processing
      process_sequence_item(t);
    end
    seq_item_port.item_done();
  end
endtask
```

```
int d_icount;
interrupt_transactionA itrA;

task isr(transaction t);
  wait (bus.interruptA);
  itrA.icount = d_icount++;
  -> itrA.w;
endtask
```

Fire & Forget Sequence

- Define a sequence
  - Background traffic
  - Checker
  - Monitor
  - Long running process

- Start it
- Forget it

```
// Starter
task body();

  fork

    other_sequence.start();

  join_none

endtask
```

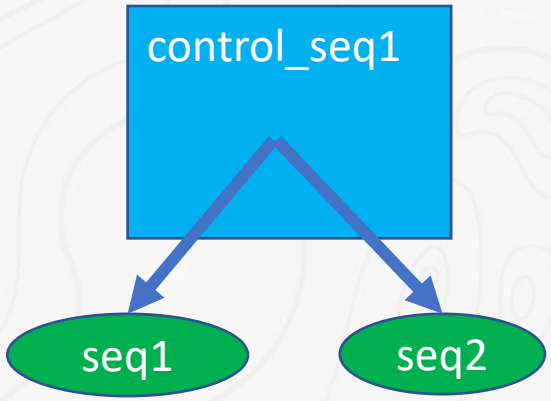
Video Sequence

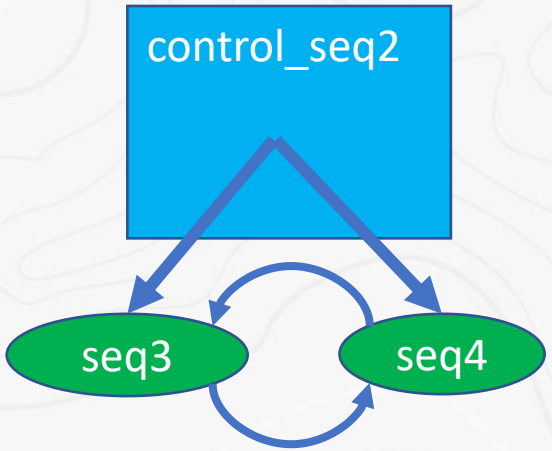
- Refresh rate

```
class video_sequence extends basic_sequence;
  `uvm_object_utils(video_sequence)
  ...
  transaction video_tr;
  task body();
    for (int i = 0; i < 100; i++) begin
      video_tr = transaction::type_id::create("video_tr");
      video_tr.max_length.constraint_mode(0);
      start_item(video_tr);
      if (!video_tr.randomize() with
          {video_tr.data.size() == (...);}) begin
        `uvm_info(get_type_name(), "Randomize FAILED",
         UVM_MEDIUM)
      end
      video_tr.rw = WRITE;
      finish_item(video_tr);
    end
  endtask
endclass
```

Virtual sequences

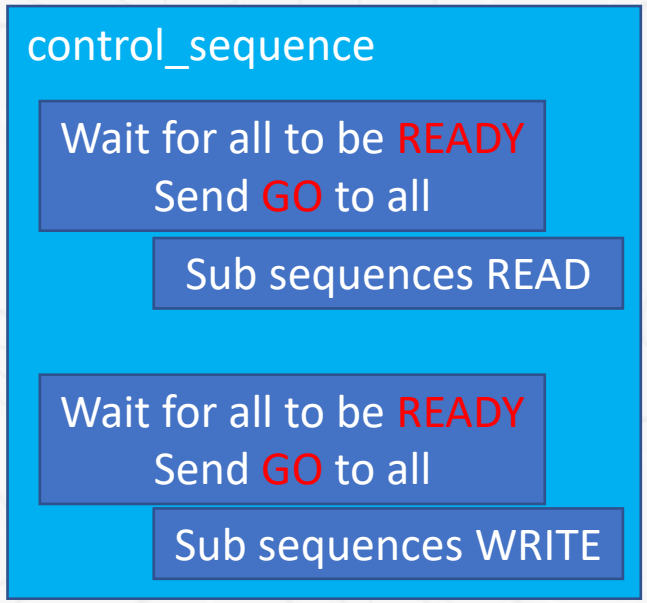
- Nothing special – but useful
- Just sequences that start other sequences
- Communicating Sequences
  - Handle sharing
  - Event triggers





Larger System

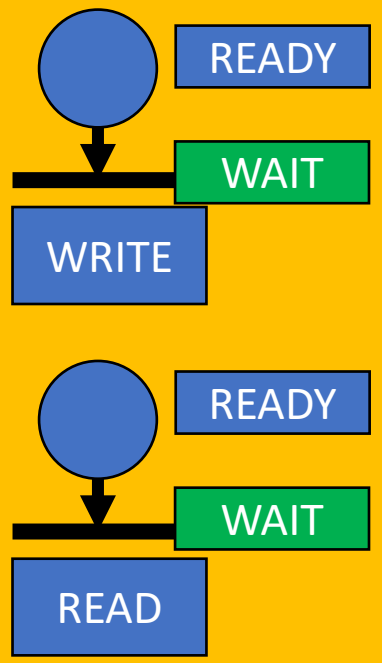
- 6 Interfaces
- 6 sequences
- Synchronized read and write



Coordinated Sequences (concept)

```
class CONTROLLER_sequence extends basic_sequence;
  `
  forever begin
    Wait for EACH child to be READY
    When each child is READY, then send an event to GO
  end
```

```
class CONTROLLED_sequence extends basic_sequence;

 forever begin
   Set READY to 1
   Wait for the GO event
   Set READY to 0

   ... Do the Write

   Set READY to 1
   Wait for the GO event
   Set READY to 0

   ... Do the Read

   ... Check
 end
```



Lower Level Sequence

```
class synced_sequence extends basic_sequence;
  `uvm_object_utils(synced_sequence)

  bit waiting_before_read;
  event proceed_to_read;
  bit waiting_before_write;
  event proceed_to_write;

  reg [7:0] wdata [];
```

Lower Level Sequence – body()

```
task body();
  for (int i = 0; i < maximum_transactions; i++) begin
    name = $sformatf("transaction%0d", i);
    tr = transaction_picker::get();
    waiting_before_write = 1;
    @(proceed_to_write);
    waiting_before_write = 0;
    // Write
    start_item(tr);
    if (!tr.randomize()) begin
      `uvm_info(get_type_name(),
       "Randomize FAILED", UVM_MEDIUM)
    end
    tr.rw = WRITE; // Force WRITE.
    finish_item(tr);
    wdata = tr.data;
```

```
    waiting_before_read = 1;
    @(proceed_to_read);
    waiting_before_read = 0;

    // Read
    tr.rw = READ; // Force READ.
    start_item(tr);
    finish_item(tr);

    // Check
    if (wdata != tr.data)
      //Error message…
  end
 endtask
endclass
```

Upper Level Sequence – The Synchronizer code

```
class coordinated_sequence extends basic_sequence;
  `uvm_object_utils(coordinated_sequence)

  synced_sequence seq1;
  synced_sequence seq2;
  synced_sequence seq3;
  synced_sequence seq4;
  synced_sequence seq5;
  synced_sequence seq6;

  task synchronizer();
    forever begin
      wait (seq1.waiting_before_write == 1);
      wait (seq2.waiting_before_write == 1);
      wait (seq3.waiting_before_write == 1);
      wait (seq4.waiting_before_write == 1);
      wait (seq5.waiting_before_write == 1);
      wait (seq6.waiting_before_write == 1);
      -> seq1.proceed_to_write;
      -> seq2.proceed_to_write;
      -> seq3.proceed_to_write;
      -> seq4.proceed_to_write;
      -> seq5.proceed_to_write;
      -> seq6.proceed_to_write;
```

```
      wait (seq1.waiting_before_read == 1);
      wait (seq2.waiting_before_read == 1);
      wait (seq3.waiting_before_read == 1);
      wait (seq4.waiting_before_read == 1);
      wait (seq5.waiting_before_read == 1);
      wait (seq6.waiting_before_read == 1);
      -> seq1.proceed_to_read;
      -> seq2.proceed_to_read;
      -> seq3.proceed_to_read;
      -> seq4.proceed_to_read;
      -> seq5.proceed_to_read;
      -> seq6.proceed_to_read;
    end
  endtask
```

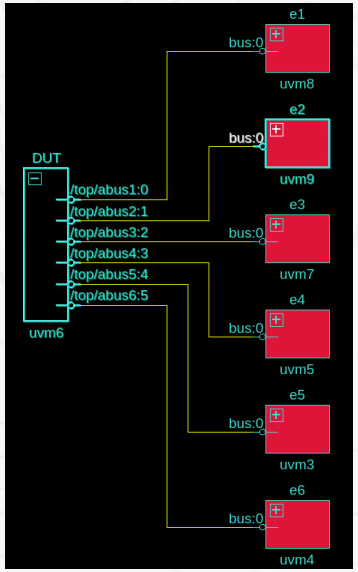
Upper Level Sequence – The body()

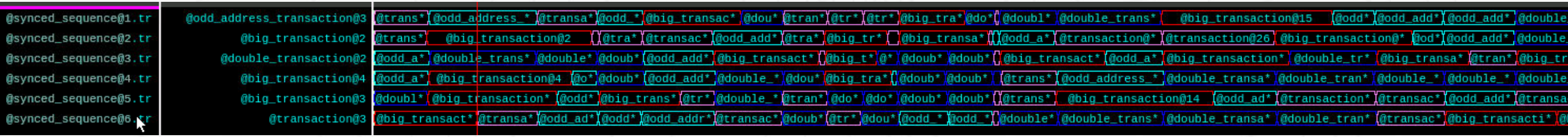
- body()
- Construct
- Start synchronizer
- Start lower level sequences

```
  task body();
    seq1 = synced_sequence::type_id::create("sequence1");
    seq2 = synced_sequence::type_id::create("sequence2");
    seq3 = synced_sequence::type_id::create("sequence3");
    seq4 = synced_sequence::type_id::create("sequence4");
    seq5 = synced_sequence::type_id::create("sequence5");
    seq6 = synced_sequence::type_id::create("sequence6");

    fork
      synchronizer();
    join_none

    fork
      seq1.start(e1.sqr);
      seq2.start(e2.sqr);
      seq3.start(e3.sqr);
      seq4.start(e4.sqr);
      seq5.start(e5.sqr);
      seq6.start(e6.sqr);
    join
  endtask
endclass
```

Conclusion

- Simple UVM Testbench
- With just a small amount of code
  - Expanded to N different transaction types
  - Expanded to N different sequence types
- Easy interrupt handler
- Easy synchronized sequences
- Easy to get lots of activity





Thank You!

Questions?

rich.edelman@siemens.com