What Does The Sequence Say?
Powering Productivity with Polymorphism

Rich Edelman
Siemens Digital Industries Software
Siemens EDA
Fremont, CA 94538

***Abstract*-In a SystemVerilog UVM testbench a UVM sequence is much like a program or a function call or a test. Writing interesting sequences can help with productivity and coverage closure. On one hand a sequence is simply a list of instructions, but on the other hand how those instructions are built or how they are used with other instructions can improve the test. This paper will demonstrate easy ways to incorporate new transactions and sequences into a SystemVerilog UVM Testbench.**

I. INTRODUCTION

This paper assumes some familiarity with SystemVerilog [1] and the UVM [2]. The examples here will be simple, but some of the concepts will be easier with a deeper background. Certainly, new users to the UVM can understand these concepts and can apply them to their first testbench. This is not a primer for new object-oriented programmers.

Our UVM testbenches in this paper will resemble the architecture below. A collection of UVM components (sequencer and driver) will be constructed in an environment (or agent). A sequence will be constructed by a test or a virtual sequence. It will be "started" (run) on the sequencer. It will generate transactions which will "go into the sequencer" and then get picked up by the driver. The driver will cause those transactions to be interpreted and the appropriate bus signals to be driven. This is basic UVM. Nothing fancy. The basic testbench is very simple. Our productivity boost will come from not changing any existing code, but by simply adding a few lines of new code and "overriding" the factory constructions.

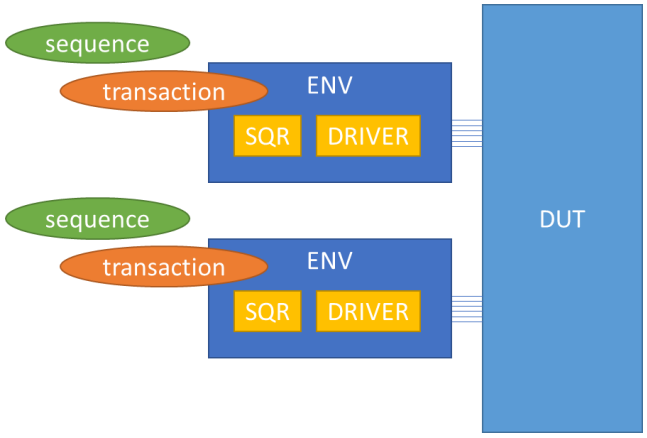

Figure 1 - Basic UVM Testbench with Two Interfaces

II. UNDERSTANDING INHERITANCE AND POLYMORPHISM

The Base Class

Polymorphism is about inheritance in an object-oriented programming world. Polymorphism is simple. But the details get hard sometimes. A polymorphic object is much like the base class, but "better". For example, if we had a base class describing a BALL object, we could derive a BASKETBALL and a RACQUETBALL from the BALL object. We could further derive a kids BASKETBALL and a professional BASKETBALL from a BASKETBALL

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

```
  endfunction

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

The class ‘ball’ is our base class. It defines two functions, get_name() and print(). They are virtual functions. Virtual functions are important later when the polymorphic object handles are used. This is important. When a handle is used to call a function, the function called could be from the DECLARED class of the object (ball) or the actual handle type (in the case of polymorphic assignments) – the function in the ACTUAL, DERIVED class. If the function is declared as a virtual function, then the derived version is used in the case of a polymorphic assignment. That’s what we want. Generally, just always declare functions and tasks as virtual. It will usually be what is wanted.

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

The class ‘basketball’ is an extended class, or a refinement, or specialization of a ball. It is by some measure a “better ball”. Notice it redefines get_name() and calls super.get_name(). Calling super.get_name() is the way we can call the base class function get_name(). Notice also that basketball does not define a new print() function. It will inherit the implementation from ball.

```
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
```

In a manner like basketball, we define kids_basketball and pro_basketball as extended classes from basketball, and a racquetball class extended from ball.

The Extended Class (derived class)

For the extended class, say ‘pro_basketball’, calling the print function will call the implementation in the base class, which calls get_name(). This is where the ‘virtual’ definitions help us. Which get_name() should get called? Ball? Basketball? Pro_basketball? The answer is pro_basketball. The pro_basketball.get_name() will call super.get_name() for the basketball, which will call super.get_name() for the ball.

Definitions of class handles

Simple declarations of class handles. Nothing special yet.

```
ball BALL;
basketball BASKETBALL;
kids_basketball KIDS_BASKETBALL;
pro_basketball PRO_BASKETBALL;
racquetball RACQUETBALL;
ball ANY_BALL[5];
```

Class Object creation and handle assignment

The objects get constructed with new(). Nothing special yet.

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

Calling print()

Using the object handles to call a function (print). Nothing special yet.

```
BALL.print();
BASKETBALL.print();
KIDS_BASKETBALL.print();
PRO_BASKETBALL.print();
RACQUETBALL.print();
```

Assigning handles polymorphically

The array of handles (ALL_BALLS) is an array of "balls". The ball, basketball, kids_basketball, pro_basketball and racquetballs are all "balls". They are "assignment compatible" with ball. They can be assigned to a ball.

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

Special. ANY_BALL is declared as an array of 'balls' types. Yet, we're assigning those array elements object handles that are BASKETBALL, KIDS_BASKETBALL, PRO_BASKETBALL and RAQUETBALL. This is allowed, since those listed object handles are all "better balls".

Calling print() polymorphically

For each of the "balls", call the print() function

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

Very special. This is the power of polymorphism at work. Print() gets calls in each of the objects – ANY_BALL[4].print() calls the RACQUETBALL print() routine.

Directly assigning an extended class handle to a base class handle

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

The print() routine that gets called in this case is the RACQUETBALL.print() routine. The print function is a virtual function, so the particular print function that will be called will be the print() function defined in the object type of the object handle – not the object type of the declared variable (BALL is a declared ball object).

The ball example is a simple example. The remainder of this paper will use the virtual functions, inheritance and polymorphism to describe how to easily extend a UVM testbench to improve productivity.

III. THE BASIC UVM TRANSACTION

The design of a UVM testbench often starts with a transaction class. The transaction class below defines a few random variables (rw, addr, data), and some non-random variables (low_addr, high_addr, id). It also defines two random constraints to control the value of addr and the contents of the data array. It has a constructor and defines convert2string and do_record. Convert2string is a simple way to return a string value of a transaction. Do_record describes the transaction data that will be recorded.

Transaction Base Class

```
typedef enum bit {READ=1, WRITE=0} rw_t;

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;
  }

  function new(string name = "transaction");
    super.new(name);
    low_addr = 0;    // Defaults
    high_addr = 256;
  endfunction

  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
endclass
```

Extended Transaction Types

Three extended transaction types are defined below. They extend the basic transaction – making it in some ways better. These specialized transactions will be used to augment our generated transactions from the sequence. But the sequence doesn't know anything about these new types. And it doesn't need to.

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

  constraint odd_address {
    addr[0] == 1;
  }
  ...
endclass
```

The odd transaction adds a constraint so that every address is odd.

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

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

The big_transaction changes (overrides) the base class constraint 'max_length'.

```
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
  ...
endclass
```

The double_transaction adds a new class member variable (addr2) and two constraints for its value. Additionally, it implements a new convert2string().

IV. THE BASIC SEQUENCE

Basic Sequence

The basic sequence is simple. It creates, randomizes and issues 'maximum_transactions' and ends. It only knows about the 'transaction' type. It is a basic sequence.

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

  transaction tr;
  string name;
  int maximum_transactions;
  ...

  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
```

Extended Sequence – The Checker

The checker_sequence below is an extended sequence. It extends the basic sequence, adds a data field and changes the behavior. This sequence only knows about the transaction type 'transaction'.

```
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);
```

```
      // 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;

      // Read
      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
```

The checker_sequence is a self-checking sequence. It issues a randomized WRITE, and then it READS from the same location, checking the data read against the data written.

V. THE UVM FACTORY

In the sequences above – and the remaining classes in this testbench, the UVM Factory is used to create objects. The UVM Factory is a polymorphic system. The call below calls the factory interface to return a class of type 'transaction'.

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

Normally, when the call above returns, a 'transaction' object handle would be returned. But the UVM Factory has a way to override the type of object returned. The call below tells the factory to return a different type – a derived type, instead of a 'transaction'. It is quite a mouthful. But this syntax means "anytime the factory is asked for a transaction object instance, instead, please construct an odd_address_transaction object instance.

```
transaction::type_id::set_type_override(odd_address_transaction::get_type(),1);
```

There are multiple different ways to override the types in the factory. The above syntax is the recommended way to over a type. A specific instance can also be overridden using

```
transaction::type_id::set_inst_override(object_type,"inst_path", parent);
```

Productivity can be increased by overriding transaction and then the basic transaction can generate many different transaction streams. In addition to overriding the transaction, the basic_sequence could also be overridden with the checker_sequence. Overriding the sequence allows a different "program" to generate and check transactions.

The syntax below tells the factory that anytime a basic_sequence is requested to instead return an object of type checker_sequence.

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

With both overrides in place, the old testbench with a basic_sequence and a transaction are now a new test with a checker_sequence and an odd_address_transaction

VI. FIRST PRODUCTIVITY

This is the first step in productivity. By defining new transactions and new sequences, an existing testbench can be easily reused. But there are some limitations. The UVM Factory allows a type or instance to be overridden. That replaces one type for the other type. But what if we wanted to randomly pick a transaction type? For example, instead of getting just a 'transaction' or the overridden 'odd_address_transaction', what if we wanted to randomly choose between three different transactions at the same time? We need a way to choose from a set of object types – not just have one that replaces the other.

The Transaction Picker

We can create an object that will pick from a list and give us back one of the objects. This is a kind of factory. The factory is still being used, but the picker allows for many types to replace one type, not just one for one.

The transaction picker below is really nothing more than a global array of object types. From that array, when 'get()' is called, one of the object types will be constructed and returned. This transaction picker can return any polymorphically related object.

```
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
```

Putting objects into the picker

The transaction picker is a static object with static variables. Calling 'add' using a mouthful of syntax is easy.

```
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());
```

The picker is really just a global array that randomly returns a value from the array when get() is called. Exactly what is desired. When a transaction is needed, the picker will pick one from the array.

Getting objects from the picker

Any object which desires to get an object handle to a derived class of transaction simply has to call the 'get()' function in the picker. Instead of calling new() or TYPE::type_id::create() a sequence would call the get() call.

Replace the factory interface

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

with the picker interface

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

Anyone needing a homework assignment can make the picker parameterized so that it works for any type of class.

VII. SECOND PRODUCTIVITY

With the picker in place in the checker_sequence, the basic_sequence test has been extended to be a self-checking sequence with a few different types of transactions. Without changing any code, productivity is improved with random transaction types and a self-checking sequence.

The astute reader might be wondering "Why not use the UVM Sequence Library?" – at least for sequence picking. The answer is simplicity, transparency and debuggability. The UVM Sequence Library is 815 lines long. The transaction picker is 21 lines long and is readable.

VIII. INTERRUPT SEQUENCES

The job of an interrupt sequence is to wait for an interrupt to occur, then "service" that interrupt. A sequence can start a transaction which will cause the driver to begin a thread waiting for the interrupt. When the interrupt occurs, the thread will wake up and trigger an event in the transaction handle, which the original interrupt sequence is waiting on. Once that trigger has happened the interrupt sequence can do whatever is necessary to service the interrupt.

Interrupt transaction

```
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
```

Simply supply an event to wait on and trigger. Any additional data could also be here (like icount).

Interrupt sequences

```
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
      `uvm_info(get_type_name(),
        $sformatf("...servicing interrupt A, icount=%0d", itrA.icount), UVM_MEDIUM)
    end
  endtask
endclass
```

Create an interrupt transaction handle and send it to the driver. When the start/finish returns, wait for the even to trigger. When it triggers, service the interrupt. After servicing the interrupt, setup another interrupt service thread.

Handling interrupt transactions in the driver

```
  int interrupt_serviceA_count;
  interrupt_transactionA itrA;

  task interrupt_serviceA_routine(transaction t);
    wait (bus.interruptA);
    itrA.icount = interrupt_serviceA_count++;
    -> itrA.w;
  endtask

  task run_phase(uvm_phase phase);
    forever begin
      seq_item_port.get_next_item(t);
      if ($cast(itrA, t)) begin
        fork
          interrupt_serviceA_routine(itrA);
        join_none;
      end
      else begin // Normal processing
        process_sequence_item(t);
        `uvm_info(get_type_name(),
          $sformatf("sending '%s'", t.convert2string()),
```

```
          UVM_MEDIUM)
      end
      seq_item_port.item_done();
    end
  endtask
```

In the driver use $cast to create a handler for that interrupt_transactionA type. The handler will run in a fork/join_none thread, waiting for the interrupt bus signal. When the bus signal happens, the transaction handle event will be triggered and the driver handler thread will end. It sounds complicated but it's not. The sequence starts a transaction and then waits for any interrupt. The driver simply waits for the interrupt in a thread, and marks the transaction object when the interrupt occurs. (In the same transaction object that originated the interrupt watching).

IX. OTHER SEQUENCES

Fire and forget sequences

These fire and forget sequences are just "regular" sequences. The sequences does start_item(t) and finish_item(t) and moves to the next generation of transaction. The basic_sequence is a fire and forget sequence.

Virtual Sequences. Sequences that communicate with each other. Handle sharing, event triggers

Virtual sequences can be used to start other sequences. A virtual sequence is nothing more than a puppet master getting other sequences to do work for it. It has special knowledge and special access. In the example code below, the virtual sequence has a handle to a collection of environments. It reaches in and starts sequences. And then it coordinates their activities. In this example, each sequence is made to wait for the others before advancing. Each sequence is released to WRITE at the same time. And then each sequence is released to READ at the same time.

The UVM test below builds a "virtual sequence" named coordinated_sequence. It fills in the environments (e1, e2, e3, e4, e5 and e6) that it inherited from 'test'. Then the test starts the virtual sequence on a null sequencer. (coordinated_sequences.start(null)).

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

  coordinated_sequence coordinated_seq;
  ...
  function void connect_phase(uvm_phase phase);
    coordinated_seq.e1 = e1;
    coordinated_seq.e2 = e2;
    coordinated_seq.e3 = e3;
    coordinated_seq.e4 = e4;
    coordinated_seq.e5 = e5;
    coordinated_seq.e6 = e6;
  endfunction

  task run_phase(uvm_phase phase);
    phase.raise_objection(this);
    coordinated_seq.start(null);
    phase.drop_objection(this);
  endtask
endclass
```

The coordinated_sequence is simple. It constructs the lower level sequences (synced_sequences) and starts them on the appropriate sequencers. At the same time, it starts a synchronizer thread that signals to the sequences, and interacts with them.

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

  env e1, e2, e3, e4, e5, e6;
  synced_sequence seq1, seq2, seq3, seq4, seq5, seq6;

  task synchronizer();
    forever begin
      ...
      -> seq1.proceed_to_write;
      -> seq2.proceed_to_write;
      -> seq3.proceed_to_write;
      -> seq4.proceed_to_write;
      -> seq5.proceed_to_write;
      -> seq6.proceed_to_write;
      ...
      -> 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

  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
```

The synced_sequence is a lower level – worker – sequence. It's really the same as the checker sequence. It writes data to an address and then reads that same addresses, comparing the written data to the read data. The synced_sequence adds a couple of events that it waits on. These events allow the virtual sequence to synchronize each of the synced_sequences. There is also some logic to signal back to the virtual sequence that was too busy for this snippet. Please contact the author for the complete source code.

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

  event proceed_to_read;
  event proceed_to_write;

  reg [7:0] wdata [];

  task body();
    for (int i = 0; i < maximum_transactions; i++) begin
      name = $sformatf("transaction%0d", i);
      tr = transaction_picker::get();

      @(proceed_to_write);
```

```
      // Write
      state = WRITING;
      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;

      @(proceed_to_read);

      // Read
      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
```

X. CONCLUSION

Polymorphism is a useful concept for increasing productivity but can strike fear into the hearts of verification engineers. It shouldn't. A polymorphic object is a "better object", in the same way that a kids_basketball is a "better ball" and a pro_basketball is a "better kids_basketball". By using polymorphism, less code can be written and the code that is written is smaller and easier to understand and debug.

Polymorphic coding can become unhelpful when it is overused. For example, a given testbench could have every object type overridden. In this case, examining the source code can be deceiving, since the code has been completely replaced. In the UVM, polymorphism in the transactions and sequences improves productivity without too much confusion. It is a good way to have "more or better ammunition" to send into the design-under-test. Creating a simple extension and using an override produces new tests. Replacing UVM components can also improve productivity but should be used judiciously. When the structure is changed it can make the testbench harder to understand and debug.

Happy Coding!

The complete source code is available from the author. It is Apache licensed (the same as UVM).

XI. REFERENCES

[1] SystemVerilog LRM, "1800-2017 - IEEE Standard for SystemVerilog--Unified Hardware Design, Specification, and Verification Language", https://ieeexplore.ieee.org/document/8299595

[2] UVM 1.1d - https://www.accellera.org/downloads/standards/uvm

XII. APPENDIX I

Polymorphic Testbench

```
// ====================================================
//   vip_pkg.sv
//   vip_pkg.sv
//   vip_pkg.sv
// ====================================================
package vip_pkg;
  import uvm_pkg::*;
  `include "uvm_macros.svh"

  `include "transactions.svh"
  `include "drivers.svh"
  `include "sequences.svh"
  `include "env.svh"
  `include "tests.svh"
endpackage


// ====================================================
//   transactions.svh
//   transactions.svh
//   transactions.svh
// ====================================================
typedef enum bit {READ=1, WRITE=0} rw_t;

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;
  }

  function new(string name = "transaction");
    super.new(name);
    low_addr = 0;    // Defaults
    high_addr = 256;
  endfunction

  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
endclass

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

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

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

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
```

```
// ======================================================
//   sequences.svh
//   sequences.svh
//   sequences.svh
// ======================================================
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);
      tr = transaction_picker::get();
      start_item(tr);
      if (!tr.randomize()) begin
        `uvm_info(get_type_name(), "Randomize FAILED",
          UVM_MEDIUM)
      end
      finish_item(tr);
    end
  endtask
endclass

class checker_sequence extends basic_sequence;
  `uvm_object_utils(checker_sequence)

  reg [7:0] wdata [];

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

  task body();
    for (int i = 0; i < maximum_transactions; i++) begin
      name = $sformatf("transaction%0d", i);
      //tr = transaction::type_id::create(name);
      tr = transaction_picker::get();

      // 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;

      // Read
      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
```

```
// ======================================================
//   drivers.svh
//   drivers.svh
//   drivers.svh
// ======================================================
class driver extends uvm_driver#(transaction);
  `uvm_component_utils(driver)

  transaction t;
  int d;
  int id;

  virtual abus bus;

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

  task bad(transaction t); // Bad behavior
    int r;
    r = $urandom_range(100, 0);
    if (r < 5) begin
      t.data[0] = -1;
    end
  endtask

  task process_sequence_item(transaction t);
    reg [31:0]a;
    `uvm_info(get_type_name(),
      $sformatf("Got     '%s'", t.convert2string()),
      UVM_MEDIUM)
    d = $urandom_range(100, 50);
    #d;
    bad(t);
    bus.rw = t.rw;
    a = t.addr;
    if (t.rw == 1) begin : READ
      foreach (t.data[i]) begin
        bus.addr = a;
        bus.valid = 1;
        bus.id = id++;
        @(posedge bus.clk);
        while (bus.ready != 1)
          @(posedge bus.clk);
        t.data[i] = bus.rdata;
        t.id = bus.id;
        a = a + 1;

        bus.valid = 0;
        @(negedge bus.clk);
      end
    end
    else if (t.rw == 0) begin : WRITE
      foreach (t.data[i]) begin
        bus.addr = a;
        bus.wdata = t.data[i];
        bus.valid = 1;
        bus.id = id++;
        t.id = bus.id;
        @(posedge bus.clk);
        while (bus.ready != 1)
          @(posedge bus.clk);
        a = a + 1;

        bus.valid = 0;
        @(negedge bus.clk);
      end
    end
  endtask

  task run_phase(uvm_phase phase);
    forever begin
      seq_item_port.get_next_item(t);
      process_sequence_item(t);
      `uvm_info(get_type_name(),
        $sformatf("sending '%s'", t.convert2string()),
        UVM_MEDIUM)
      seq_item_port.item_done();
    end
  endtask
endclass
```

```
// ======================================================
//   env.svh
//   env.svh
//   env.svh
// ======================================================
class env extends uvm_env;
  `uvm_component_utils(env)

  driver d;
  uvm_sequencer#(transaction) sqr;

  virtual abus bus;

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

  function void build_phase(uvm_phase phase);
    d = driver::type_id::create("driver", this);
    sqr = new("sqr", this);
  endfunction

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


// ======================================================
//   tests.svh
//   tests.svh
//   tests.svh
// ======================================================
class test extends uvm_test;
  `uvm_component_utils(test)

  env e;

  basic_sequence seq;

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

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

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());

//transaction::type_id::set_type_override(
//                  odd_address_transaction::get_type(),1);
//transaction::type_id::set_type_override(
//                           big_transaction::get_type(),1);

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

    if (!uvm_config_db#(virtual abus)::get(
          this, "*", "bus", e.bus))
      `uvm_fatal(get_type_name(),
        "Cannot find VIF for b1")
  endfunction

  task run_phase(uvm_phase phase);

    phase.raise_objection(this);
    seq = basic_sequence::type_id::create("sequence");
    seq.start(e.sqr);
    phase.drop_objection(this);
    `uvm_info(get_type_name(), "...finished", UVM_MEDIUM)
  endtask
endclass


// ======================================================
//   top.sv
//   top.sv
//   top.sv
// ======================================================
import uvm_pkg::*;
`include "uvm_macros.svh"

import vip_pkg::*;

module top();
  reg clk;
  reg reset;

  abus bus(clk, reset);

  dut DUT(bus);

  initial begin
    run_test("test");
  end

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

  initial begin
    uvm_config_db#(virtual abus)::set(
      null, "*", "bus", bus);

    reset = 0;
    repeat (10) @(posedge clk);
    reset = 1;
    repeat (10) @(posedge clk);
    reset = 0;

    repeat (10000)
      @(posedge clk);
    $finish(2);
  end
```