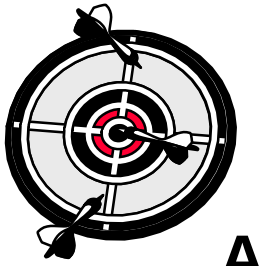
Agenda: Day 1

DAY 1

1. OOP Inheritance Review
2. UVM Structural Overview
3. UVM Transaction
4. UVM Sequence

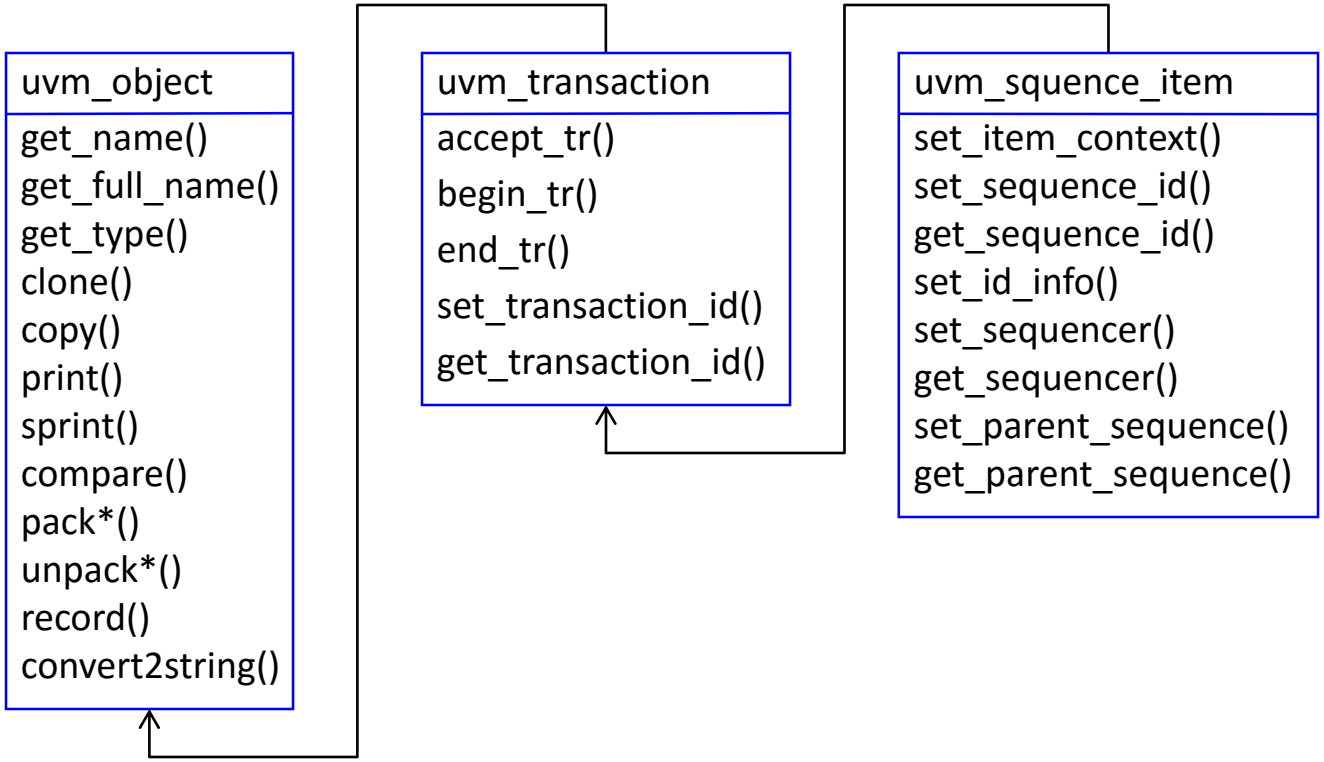
Unit Objectives



After completing this unit, you should be able to:

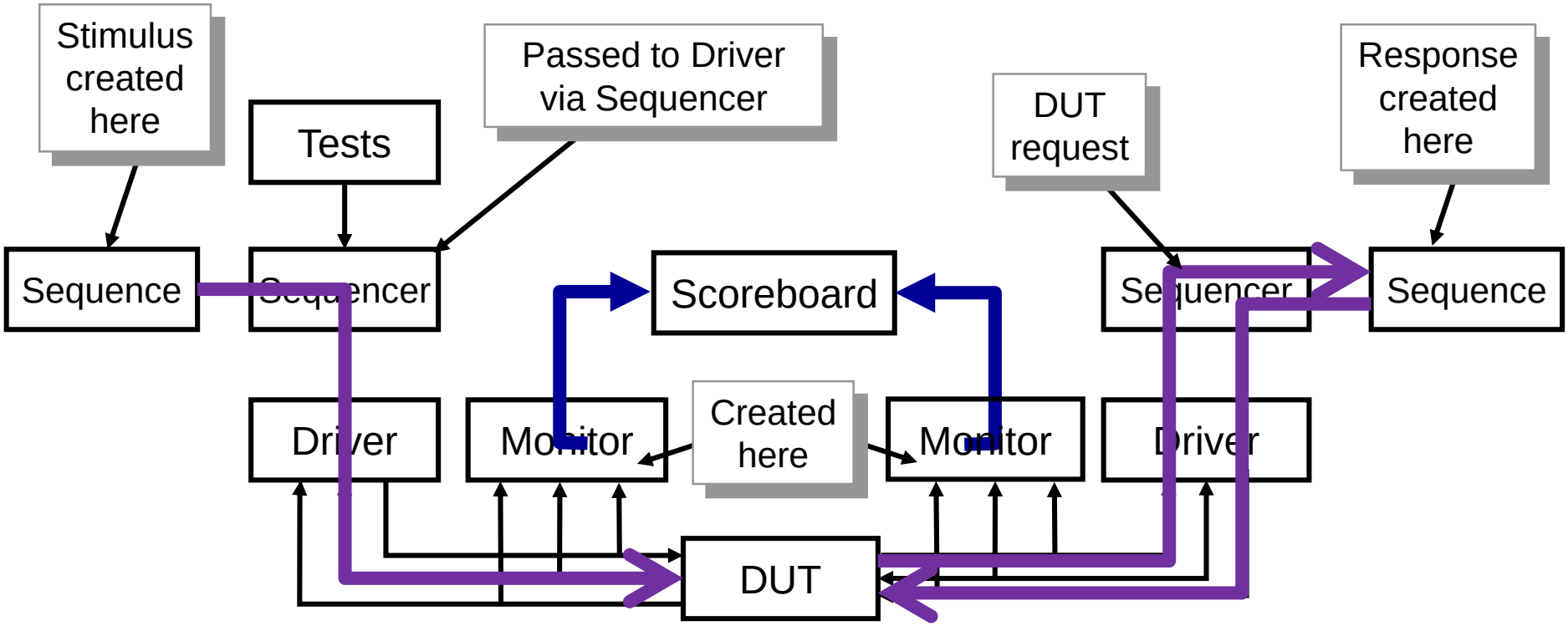
- **Build data models by inheriting from `uvm_sequence_item`**
- **Use macros or implement method to enable processing of `uvm_sequence_item` fields**
- **Modify constraint with inheritance and override**
- **Implement parameterized classes**
- **Simplify report messages**

UVM Transaction Base Classes



UVM Transaction Flow

- **Transactions typically do not have a fixed component parent**



Modeling Transactions

- **Derive from `uvm_sequence_item` base class**
  - Built-in support for stimulus creation, printing, comparing, etc.
- **Properties should be <u>public</u> by default**
  - Must be visible to constraints in other classes
- **Properties should be <u>rand</u> by default**
  - Can be turned off with `rand_mode`

```
class packet extends uvm_sequence_item;
  `uvm_object_utils(packet)
  rand bit [47:0] sa, da;
  rand bit [15:0] len;
  rand bit [ 7:0] payload[$];
  rand bit [31:0] crc;

  function new(string name = "packet");
    super.new(name);
    this.crc.rand_mode(0);
  endfunction
endclass
```

No **`uvm_component`** parent

Default required

Other Properties to be Considered (1/2)

- **Embed transaction descriptor**
  - Component interprets transaction to execute

```
class cpu_data extends uvm_sequence_item;
  typedef enum {READ, WRITE} kind_t;
  rand int delay = 0;
  rand kind_t kind;
  rand bit [31:0] addr, data;
  function new(string name="cpu_data");
    super.new(name);
    this.delay.rand_mode(0);
  endfunction
endclass
```

```
class cpu_driver extends uvm_driver #(cpu_data);
  virtual task execute(cpu_data tr);
    repeat(tr.delay) @(vif.drvClk);
    case (tr.kind) begin
      cpu_data::READ:
        tr.data = this.read(tr.addr);
      cpu_data::WRITE:
        this.write(tr.addr, tr.data);
    endcase
  endtask
endclass
```

Other Properties to be Considered (2/2)

- **Embed transaction status flags**
  - Set by component for execution status

```
class cpu_data extends uvm_sequence_item;
  typedef enum {IS_OK, ERROR, HAS_X} status_e;
  rand status_e status = IS_OK;  ...
  function new(string name="cpu_data");
    super.new(name);  ...
    this.status.rand_mode(0);
  endfunction
endclass
```

```
class cpu_driver extend uvm_driver #(cpu_data);
  virtual task execute(cpu_data tr);
    repeat(tr.delay) @(vif.drvClk);
    case (tr.kind) begin
      ...
    endcase
    if (error_condition_encountered)
      tr.status = cpu_data::ERROR;
      `uvm_info("DEBUG", tr.sprint(), UVM_HIGH)
  endtask
endclass
```

Transactions: Must-Obey Constraints

- **Define constraint block for the must-obey constraints**
  - Never turned off
  - Never overridden
  - Name "*class_name*_valid"
- **Example:**
  - Non-negative values for **`int`** properties

```
class packet extends uvm_sequence_item;
   rand int len;
   ...
   constraint packet_valid {
      len > 0;
   }
endclass
```

Transactions: Should-Obey Constraints

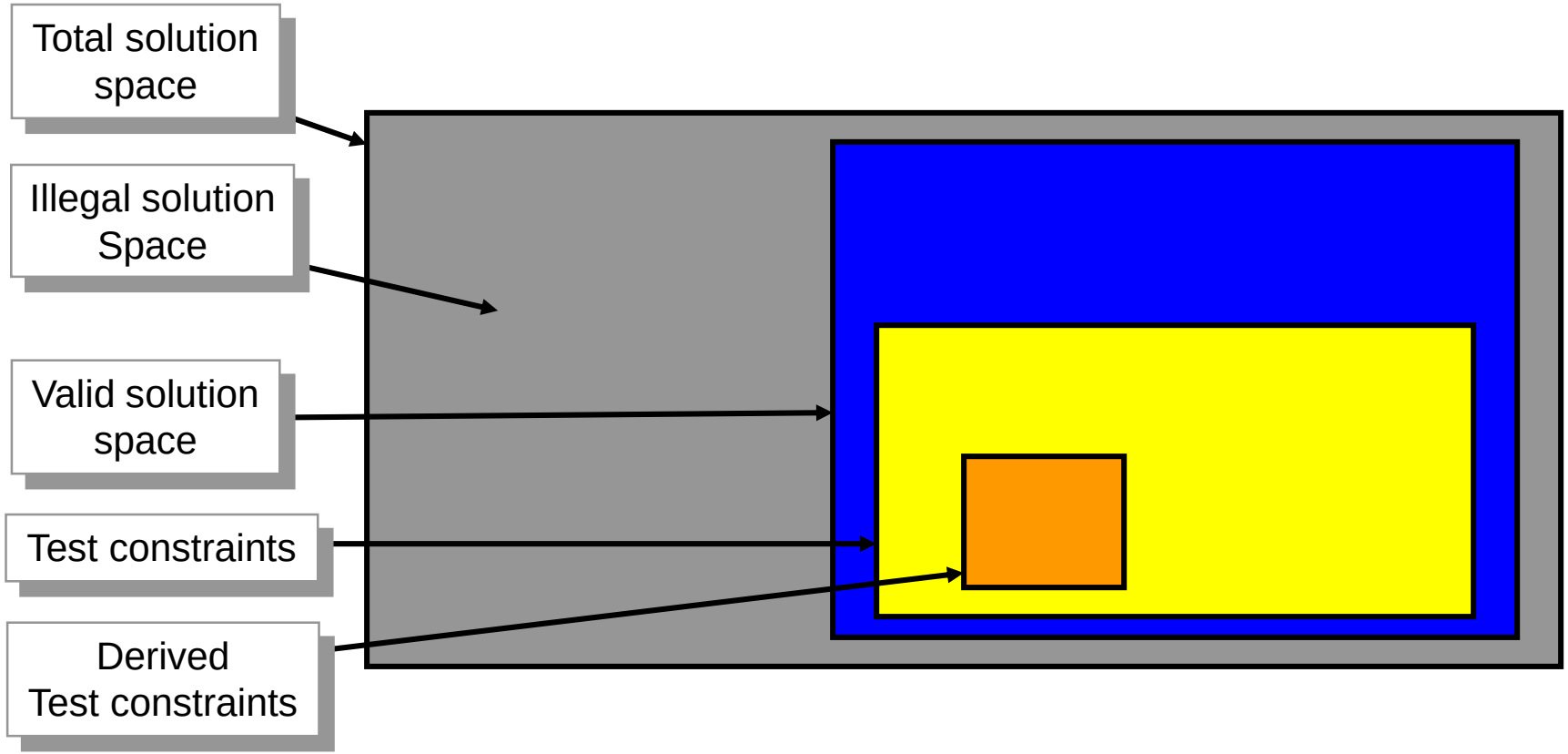
- **Define constraint block for should-obey constraints**
  - Can be turned off to inject errors
  - One block per relationship set
    - Can be individually turned off or overloaded
  - Name "*class_name_rule*"

```
class packet extends uvm_sequence_item;
   ...
   constraint packet_sa_local  {
      sa[41:40]   == 2'b0;
   }
   constraint packet_ieee {
      len      inside {[46:1500]};
      data.size() == len;
   }
...
```

```
...
   constraint packet_fcs {
      crc == 32'h0000_0000;
   }
endclass
```

Transactions: Constraint Considerations

- **Can't accidentally violate valid constraints**
  - Constraint solver will fail if the user constraints conflict with *valid* constraints



Transaction Class Methods

- **How to process transaction's fields in `uvm_object` methods?**
  - Print, copy, compare, record, pack
- **Use macros to support processing of fields**

```
`uvm_object_utils_begin(cname)
  `uvm_field_*(ARG, FLAG)
`uvm_object_utils_end
```

```
class packet extends uvm_sequence_item;
   rand bit [47:0] sa, da;
   rand bit [ 7:0] payload[$];
        packet     next;

   `uvm_object_utils_begin(packet)
     `uvm_field_int(sa, UVM_ALL_ON | UVM_NOCOMPARE)
     `uvm_field_int(da, UVM_ALL_ON)
     `uvm_field_queue_int(payload, UVM_ALL_ON)
     `uvm_field_object(next, UVM_ALL_ON)
    `uvm_object_utils_end
endclass
```

Transaction Methods are Not Virtual

- **Transaction access methods not virtual!**

For debugging

```
~~virtual~~ function void print(uvm_printer printer = null)
~~virtual~~ function string sprint(uvm_printer printer = null)
```

For making copies

```
virtual function uvm_object clone();  // exception!
~~virtual~~ function void copy(uvm_object rhs);
```

Used by checker

```
~~virtual~~ function bit  compare(uvm_object rhs,
                                  uvm_comparer comparer = null);
```

Used by transactors

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

Use to record transactions for debugger

```
~~virtual~~ function void record(uvm_recorder recorder = null);
```

- **User must <u>NOT</u> override these methods**
  - Exception is clone() which has a default implementation but can be overridden if required

Customization of Field Processing

- User can customize processing of fields via the following do_* methods

```
virtual function void do_print(uvm_printer printer = null)

virtual function void do_copy(uvm_object rhs);

virtual function bit  do_compare(uvm_object rhs,
                                 uvm_comparer comparer = null);

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

virtual function int  do_unpack(ref bit bitstream[],
                                input uvm_packer packer = null);

virtual function void do_record(uvm_recorder recorder = null);
```

Using Transaction Methods

Name string

```
packet pkt0, pkt1, pkt2;
bit bit_stream[];
pkt0 = packet::type_id::create("pkt0");
pkt1 = packet::type_id::create("pkt1");
pkt0.sa = 10;

pkt0.print();                          // display content of object on stdio

pkt0.copy(pkt1);                       // copy content of pkt1 into memory of pkt0
                                       // name string is not copied

$cast(pkt2, pkt1.clone());             // make pkt2 an exact duplication of pkt1
                                       // name string is copied

if(!pkt0.compare(pkt2)) begin          // compare the contents of pkt0 against pkt2
  `uvm_fatal("MISMATCH", {"\n", pkt0.sprint(), pkt2.sprint()});
end                                    // sprint() returns string for logging

pkt0.pack(bit_stream);                 // pack content of pkt0 into bit_stream array

pkt2.unpack(bit_stream);               // unpack bit_stream array into pkt2 object
```

Modify Constraint in Transactions by Type

```
class packet extends uvm_sequence_item;
  rand bit[3:0] sa, da;
  rand bit[7:0] payload[];
  constraint valid {payload.size() inside {[2:10]};}
  .
end
```

```
class packet_da_3 extends packet;
  constraint da_3 {da == 3;}
  `uvm_object_utils(packet_da_3)
  function new(string name = "packet_da_3");
    super.new(name);
  endfunction
```

The most common transaction modification is adding constraint

```
class test_da_3_type extends test_base;
 `uvm_component_utils(test_da_3_type)
 function new(string name, uvm_component parent);
  super.new(name, parent);
 endfunction
 virtual function void build_phase(uvm_phase phase);
  super.build_phase(phase);
//set_type_override("packet", "packet_da_3");
  set_type_override_by_type(packet::get_type(), packet_da_3::get_type());
 endfunction
endclass
```

Preferred override (compile-time check)

All **packet** instances are now **packet_da_3**

Modify Constraint in Transaction by Instance

```
class test_da_3_inst extends test_base;
  // utils and constructor not shown
  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
//  set_inst_override("env.i_agt.sqr.*", "packet", "packet_da_3");
    set_inst_override_by_type("env.i_agt.sqr.*", packet::get_type(),
                                                 packet_da_3::get_type());
  endfunction
endclass
```

Only packet instances in matching sequencers are now packet_da_3

Simulate with:
`simv +UVM_TESTNAME=`*`test_da_3_inst`*

```
UVM_INFO @ 0: uvm_test_top.env.i_agt.drv [Normal] Item in Driver
req: (packet_da_3@95)  {
  sa: 'h7
  da: 'h3
}
Instance Overrides:
  Requested Type  Override Path                      Override Type
  --------------  --------------------------------   --------------
  packet          uvm_test_top.env.i_agt.sqr.*       packet_da_3
```

Command-line Override

- **User can override factory objects at command line**
  - Objects must be constructed with the factory ***class_name*::type_id::create(…)** method

```
+uvm_set_inst_override=<req_type>,<override_type>,<inst_path>
+uvm_set_type_override=<req_type>,<override_type>
```

- **Works like the overrides in the factory**
  - **set_inst_override()**
  - **set_type_override()**
- **Example:**

```
+uvm_set_inst_override=packet,my_packet,*.agt.*
+uvm_set_type_override=packet,my_packet
```

No space character!

Parameterized Transaction Class (1/3)

- **Parameterized transaction requires a different macro**

  `` `uvm_object_param_utils(cname#(param)) ``

  Or

  ```
  `uvm_object_param_utils_begin(cname#(param))
    `uvm_field_*(ARG, FLAG)
  `uvm_object_utils_end
  ```

- **Use typedef to make it more manageable**

```
class my_data #(width=48) extends uvm_sequence_item;
  typedef my_data#(width) this_type;
  `uvm_object_param_utils(this_type)
  ...
endclass
```

Parameterized Transaction Class (2/3)

- **Must supplement macro with definition of `type_name`**
  - For non-parameterized classes, the macro creates these
  - For parameterized classes, user must define these

```
class my_data #(width=48) extends uvm_sequence_item;
  typedef my_data#(width) this_type;
  `uvm_object_param_utils(this_type)
  const static string type_name = $sformatf("my_data#(%0d)", width);
  virtual function string get_type_name();
    return type_name;
  endfunction
  ...
endclass
```

- **If not done, print will show `uvm_sequence_item` as type**

```
Name          Type                  Size   Value
-----------------------------------------------
req           uvm_sequence_item     -      @1548
  address     integral              4      'hc
  data        integral              16     'hc472
```

Parameterized Transaction Class (3/3)

- **Creation of parameterized transaction object requires parameter (even if none is needed)**

```
class my_component extends uvm_component;
  my_data d;
  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    d = my_data#()::type_id::create("d", this);
  endfunction
endclass
```

- **Use typedef to eliminate potential problem**

```
class my_component extends uvm_component;
  typedef my_data#() my_data_t;
  my_data_t d;
  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    d = my_data_t::type_id::create("d", this);
  endfunction
endclass
```

Simplifying Report Messages

- **The `print/sprint()` method may be too verbose**
  - Good for debug, bad for quick analysis

```
task driver::run_phase(uvm_phase phase);
  forever begin
    seq_item_port.get_next_item(req);
    `uvm_info("RUN", {"\n", req.sprint()}, UVM_MEDIUM);
    ...
  end
endtask
```

```
UVM_INFO driver.sv(34) @ 12.0ns: uvm_test_top.env.agt.drv [RUN]
----------------------------------------------------------------
Name                Type      Size   Value
----------------------------------------------------------------
req                  packet     -    @1548
  sa                 integral   4    'h3
  da                 integral   4    'h4
  ...
----------------------------------------------------------------
```

Applying `convert2string()`

- **Implement and use `convert2string()` method**

```
function string packet::convert2string();
  return $sformatf("sa = %2d, da = %2d", sa, da);
endfunction
```

```
task driver::run_phase(uvm_phase phase);
  forever begin
    seq_item_port.get_next_item(req);
    `uvm_info("RUN", req.convert2string(), UVM_LOW);
    ...
  end
endtask
```

- **Output now simplifies to**

```
UVM_INFO driver.sv(34) @ 12.0ns:  uvm_test_top.env.agt.drv [RUN] sa =  3, da =  4
UVM_INFO driver.sv(34) @ 24.0ns:  uvm_test_top.env.agt.drv [RUN] sa = 14, da = 15
UVM_INFO driver.sv(34) @ 36.0ns:  uvm_test_top.env.agt.drv [RUN] sa =  5, da =  1
UVM_INFO driver.sv(34) @ 48.0ns:  uvm_test_top.env.agt.drv [RUN] sa = 12, da =  6
UVM_INFO driver.sv(34) @ 60.0ns:  uvm_test_top.env.agt.drv [RUN] sa =  5, da =  1
```

Unit Objectives Review

Having completed this unit, you should be able to:

- **Build data models by inheriting from `uvm_sequence_item`**
- **Use macros or implement method to enable processing of `uvm_sequence_item` fields**
- **Modify constraint with inheritance and override**
- **Implement parameterized classes**
- **Simplify report messages**