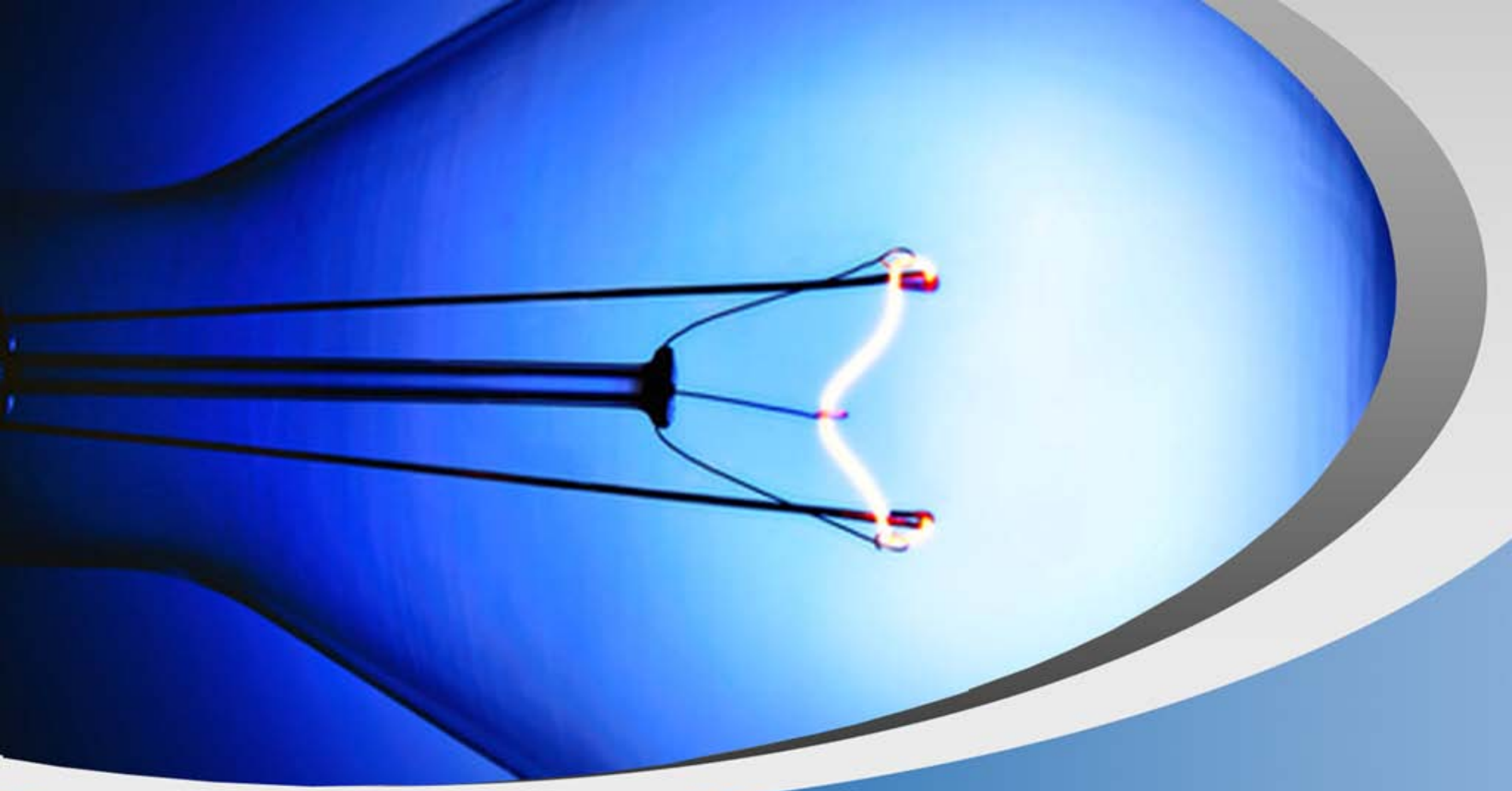


UVM: Ready, Set, Deploy!

UVM

DVCon 2012
Design & Verification Conference & Exhibition

accellera
SYSTEMS INITIATIVE

Base Classes in UVM

Tom Fitzpatrick

Verification Methodologist

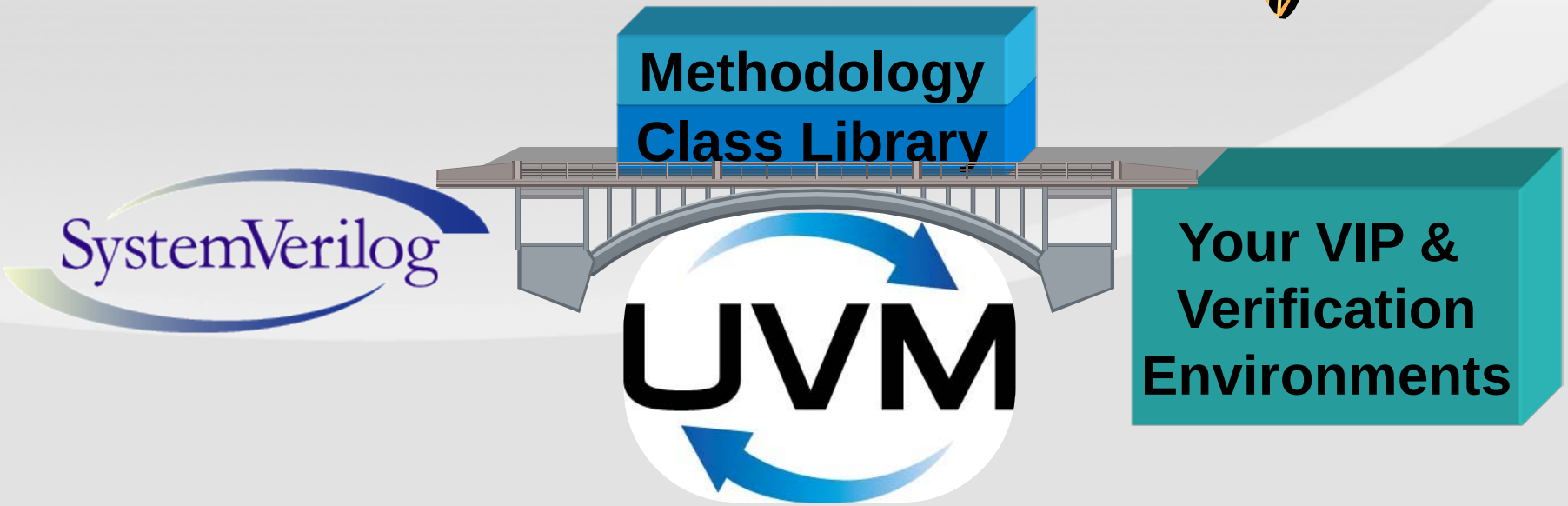
The Idea Behind The Methodology

- **UVM underpins best practices**
  - It's all about people...
  - Team Development
- **Peopleware is most important**
  - Develop Skill Set
  - Common language
  - Strategy and cohesion
  - Clarity and transparency
- **A Guiding Methodology**
  - Provides Freedom From Choice
  - Avoids Chaos and Repetition
  - Ease of Use APIs

  - Not just for Super-heroes!

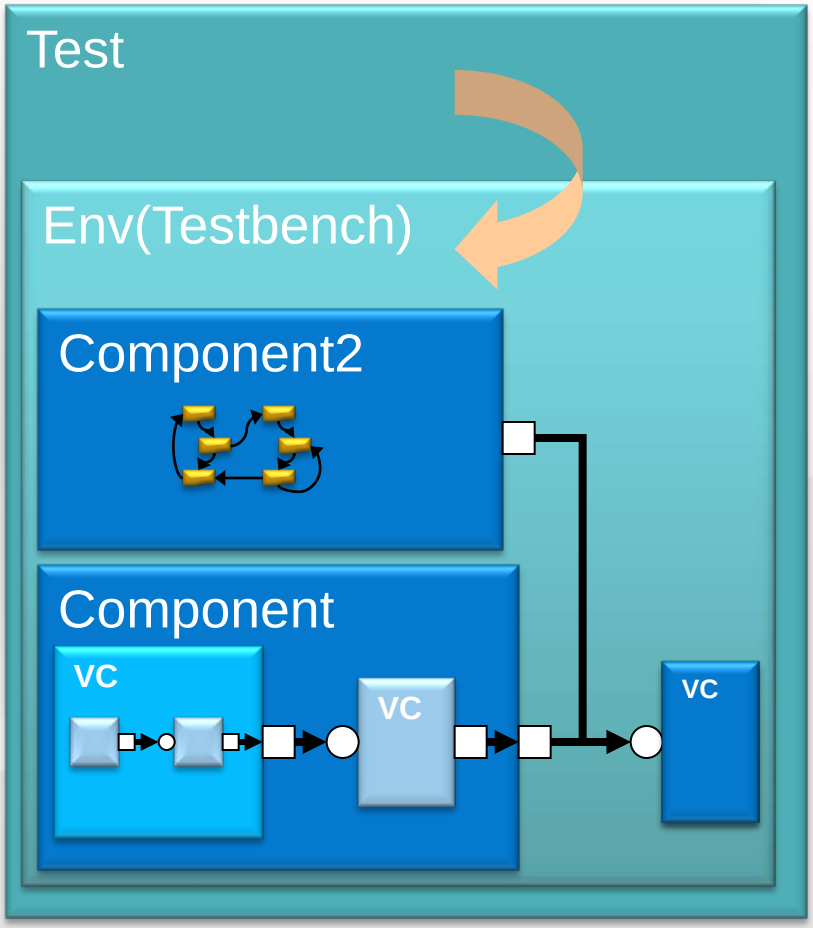
The Keys to Verification Productivity

- **Don’t reinvent the wheel**
  - Build on what’s there
- **Reuse, Reuse, Reuse**
  - Modularity, Flexibility and Portability



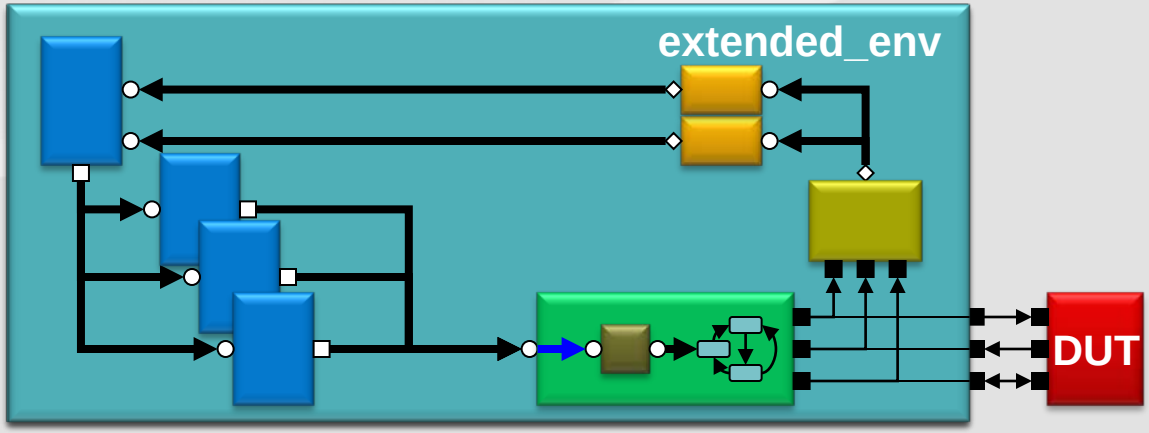
UVM Key Concepts

- **Test/Testbench Separation**
  - Improves reusability
  - Test customizes testbench
- **Phased build process**
- **Configurability**
  - Controlled by test
  - Structural, run-time parameters
  - Allows greater topological flexibility
- **TLM Communication**
  - Improves component modularity
  - Enables plug-n-play reuse
- **Hierarchical Sequential Stimulus**
  - Simplified Test Writer interface
  - Decouple stimulus from component hierarchy



UVM: Building an Environment

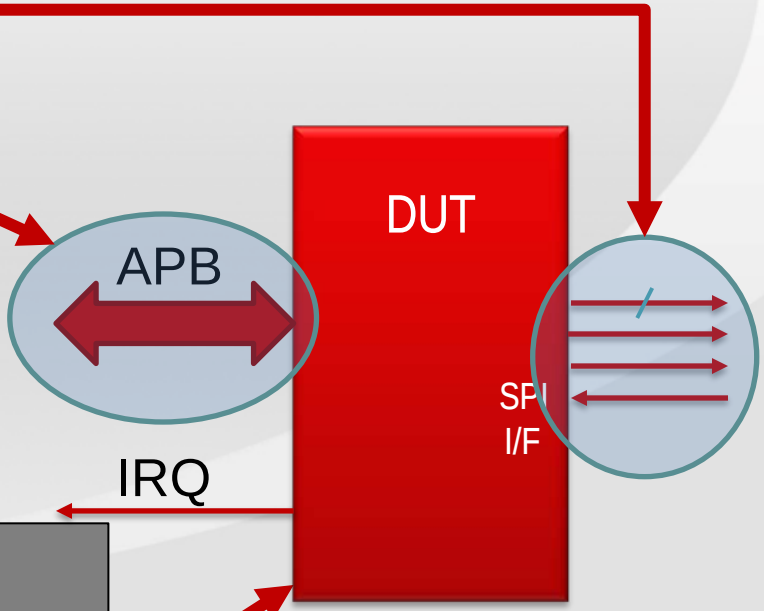
- **All verification components instantiated in `build_phase()`**
  - Factory class generates component types
    - `build_phase()` called hierarchically top-down
    - `parent.build_phase()` can set parameters for `child.build_phase()`
    - Components can be overridden by type or instance (with wildcarding too)
- **Connections defined in `connect_phase()`**
  - Called automatically after `build_phase()`
- **UVM environment is completely built after `end_of_elaboration_phase()`**
- **Env code doesn't change**
  - Test modifies env via the factory & config_db
  - Env itself can be overridden by test via factory



UVM Testbench - Architectural Design

For Each Interface:

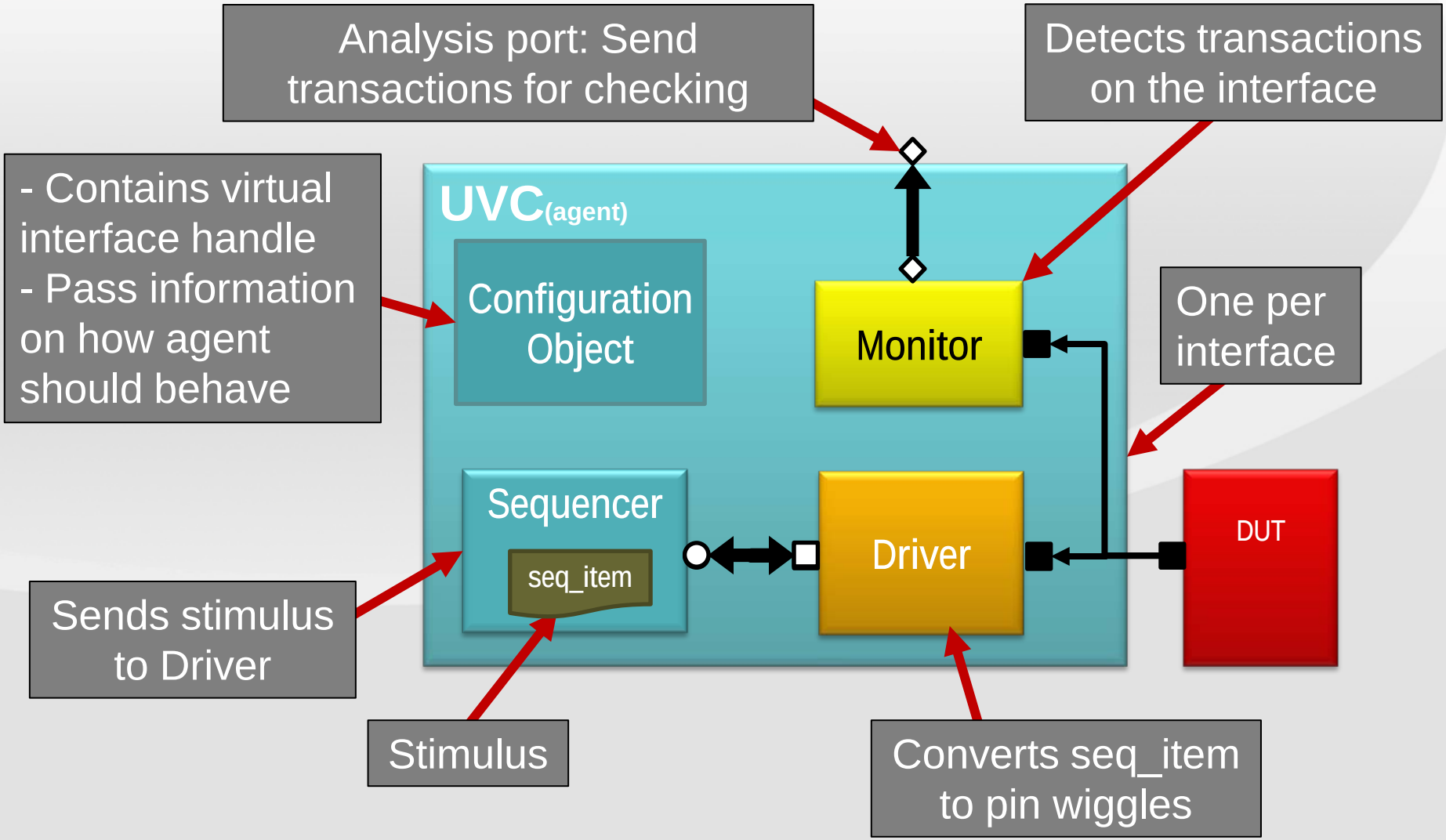
- How does the interface work?
- What information is transferred?
- Transaction variants?
- Uni/bidirectional? Pipelined?



For the Design:

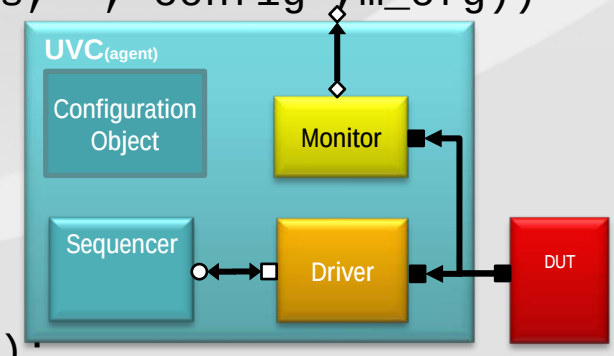
- What does it do?
- What are the use cases?
- Which test cases are required?
- What type of stimulus scenarios are required?
- What represents correct behavior?
- What kind of functional coverage do I need?

UVC Structural Building Block



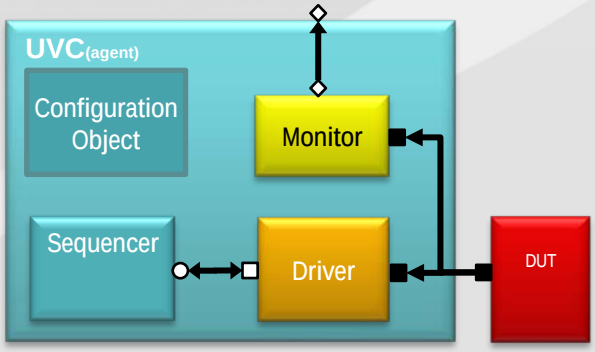
The Agent

```
class dut_agent extends uvm_component;
  `uvm_component_utils(dut_agent)
  dut_agent_cfg m_cfg;
  uvm_analysis_port #(dut_txn) ap;
  dut_monitor m_monitor;
  dut_driver m_driver;
  uvm_sequencer #(dut_txn) m_seqr;
  …
  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    if(!uvm_config_db #(dut_agent_cfg)::get(this,"","config",m_cfg))
      `uvm_fatal("Config fatal",
                 "Can't get config");
    if(m_cfg.active == UVM_ACTIVE) begin
      m_seqr = uvm_sequencer#(dut_txn)::
                   type_id::create("seqr",this);
      m_driver = dut_driver::
                   type_id::create("driver",this);
    end
    m_monitor = dut_monitor::type_id::create("monitor",this);
    …
  endfunction

endclass
```



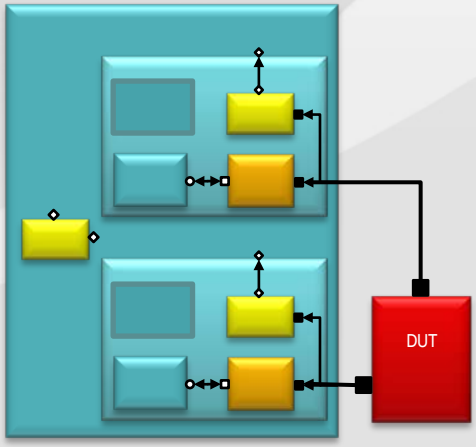
The Agent

```
class dut_agent extends uvm_component;
  `uvm_component_utils(dut_agent)
  dut_agent_cfg m_cfg;
  uvm_analysis_port #(dut_txn) ap;
  dut_monitor m_monitor;
  dut_driver m_driver;
  uvm_sequencer #(dut_txn) m_seqr;
  …
  function void build_phase(uvm_phase phase);
    …
  endfunction

  function void connect_phase(uvm_phase phase);
    m_monitor.dut_if = m_cfg.bus_if;
    ap = m_monitor.ap;
    if(m_cfg.active == UVM_ACTIVE) begin
      m_driver.seq_item_port.connect(
                        m_seqr.seq_item_export);
      m_driver.dut_if = m_cfg.bus_if;
    end
    …
  endfunction

endclass
```



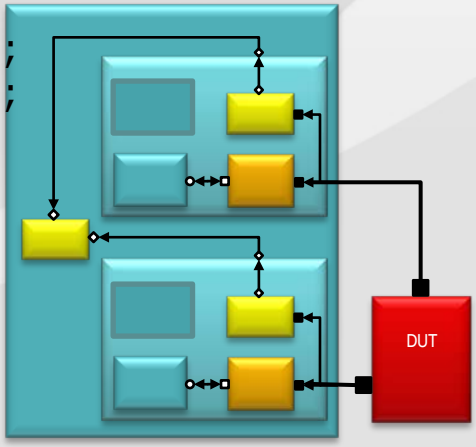
The Environment

```
class my_env extends uvm_env;
  `uvm_component_utils(my_env)
  agent1 m_agent1;
  agent2 m_agent2;
  my_scoreboard m_scoreboard;
  my_env_config m_cfg;
  function new(string name = “my_env”, uvm_component parent = null);
    super.new(name, parent);
  endfunction
  function void build_phase(uvm_phase phase);
    if(!uvm_config_db #( my_env_config )::get( this , "",
                          “my_env_config" , m_cfg ) begin
    `uvm_error(“Env: build", “can’t get env_config") end
    if(m_cfg.has_agent1) begin
      uvm_config_db #(agent1_config)::set( this ,
           "m_agent1*","agent1_config",m_cfg.m_agent1_cfg);
      m_agent1 = agent1::type_id::create("m_agent1", this);
    end
    if(m_cfg.has_agent2) begin
      uvm_config_db #(agent2_config)::set( this ,
           "m_agent2*","agent2_config",m_cfg.m_agent2_cfg);
      m_agent2 = agent2::type_id::create("m_agent2", this);
    end
    if(m_cfg.has_my_scoreboard) begin
      m_scoreboard = my_scoreboard::type_id::create("m_scoreboard", this);
    end
  endfunction:build_phase
```



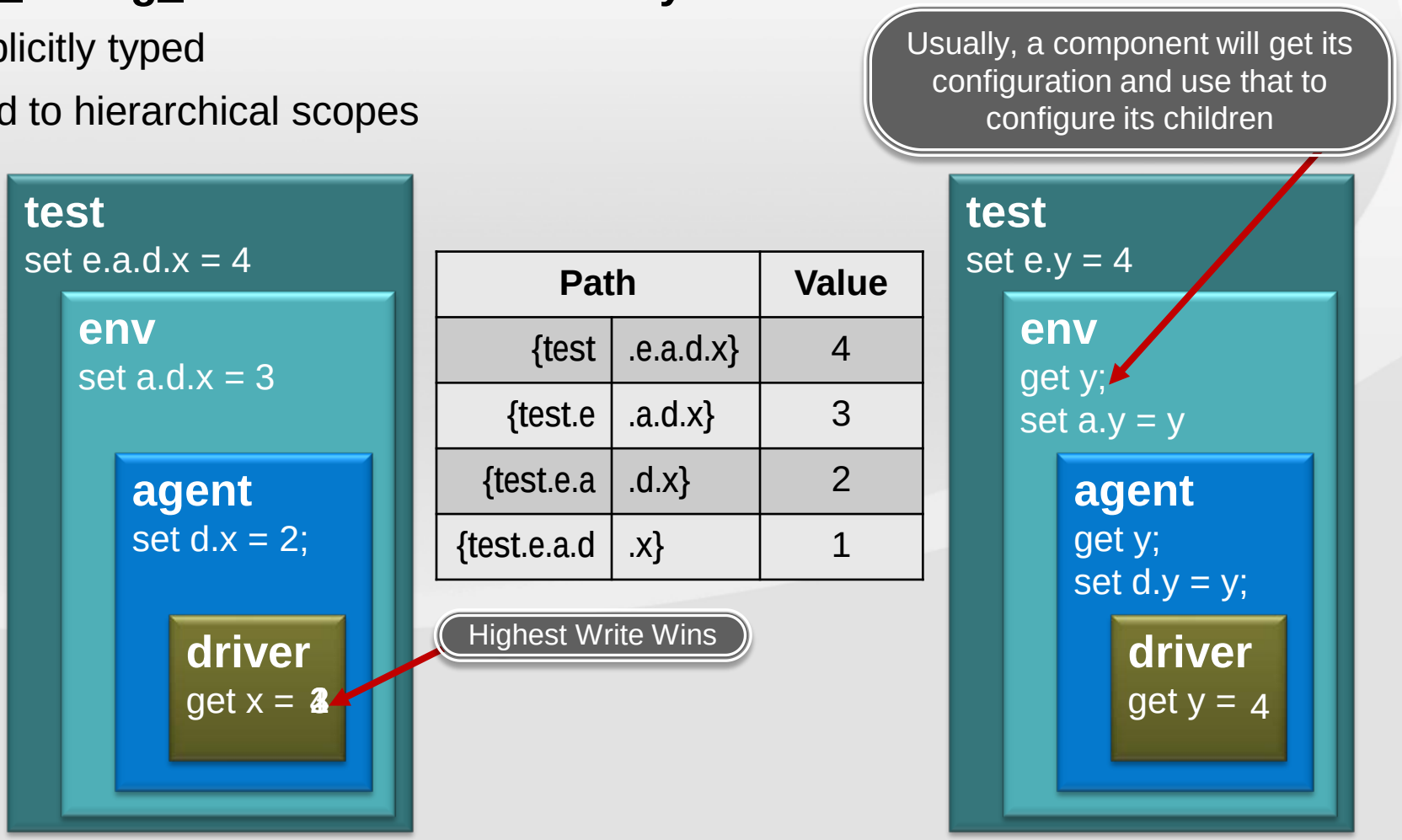
The Environment

```
class my_env extends uvm_env;
  `uvm_component_utils(my_env)
  agent1 m_agent1;
  agent2 m_agent2;
  my_scoreboard m_scoreboard;
  my_env_config m_cfg;
  function new(string name = “my_env”, uvm_component parent = null);
    super.new(name, parent);
  endfunction
  function void connect_phase( uvm_phase phase );
    if(m_cfg.has_spi_scoreboard) begin
      m_agent1.ap.connect(m_scoreboard.apb.analysis_export);
      m_agent2.ap.connect(m_scoreboard.spi.analysis_export);
     end
  endfunction: connect_phase
endclass
```



UVM Configuration Database

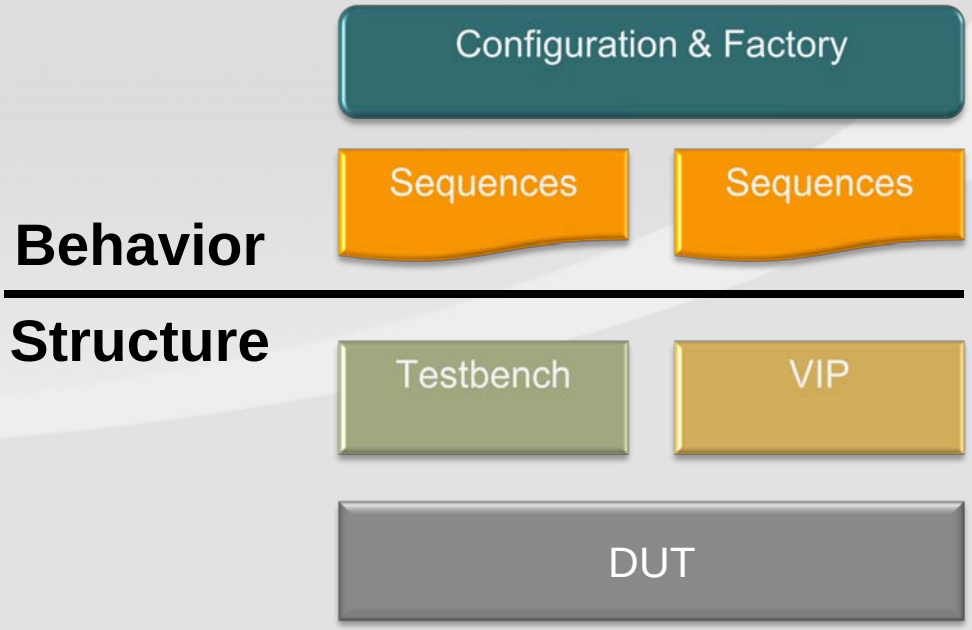
- **uvm_config_db is a convenience layer**
  - Explicitly typed
  - Tied to hierarchical scopes



Path		Value
{test	.e.a.d.x}	4
{test.e	.a.d.x}	3
{test.e.a	.d.x}	2
{test.e.a.d	.x}	1

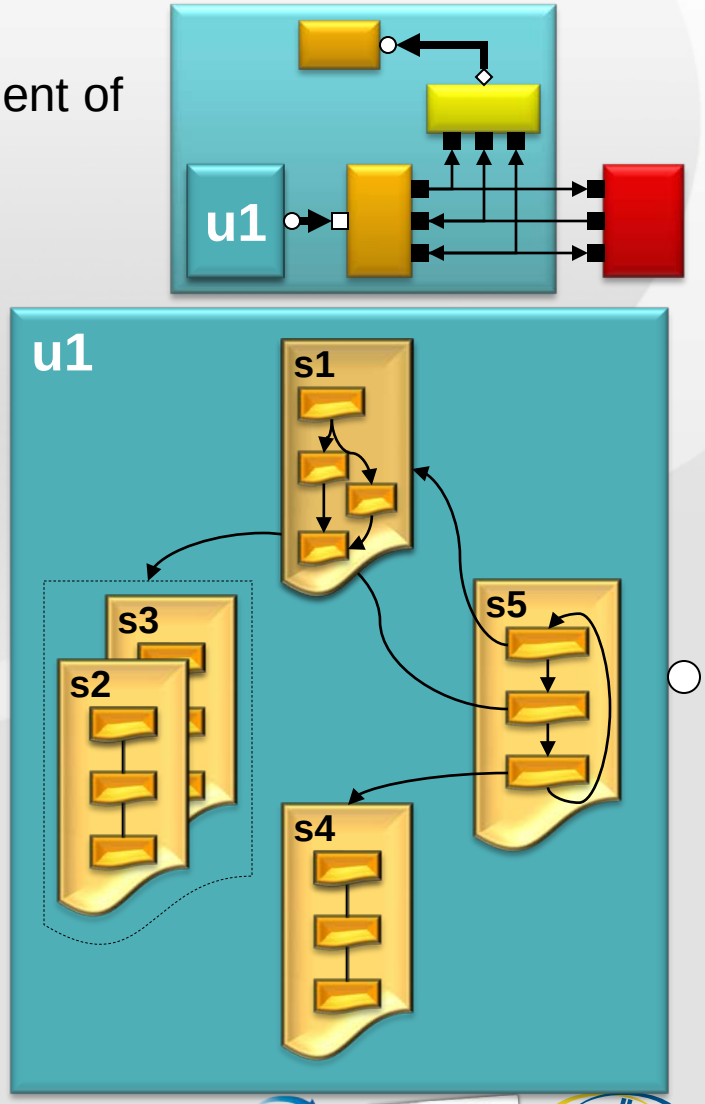
Separating Stimulus from the Testbench

- A key to reusability is to separate *Behavior* from *Structure*



Sequences

- **Decouple stimulus specification from structural hierarchy**
  - Add/remove/modify stimulus scenarios independent of testbench
  - Simplify test writer API
- **Sequences define transaction streams**
  - May start on any sequencer
- **Sequences can call children**
- **Sequences & transactions customizable via the factory**



Designing a Sequence Item

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

  rand int delay;
  rand logic[31:0] addr;
  rand op_code_enum op_code;
  rand logic[31:0] data[];
  string slave_name;
  bit response;

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

This is the "transaction"

Make all "input" properties **rand**

- do_copy()
- do_compare()
- convert2string()
- do_print()
- do_record()
- do_pack()
- do_unpack()

Methods for standard operation

Users call copy(), compare()…

```
endclass: bus_item
```

Designing a Sequence Item

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

  function void do_copy(uvm_object rhs);
    bus_item rhs_;
```

do_copy()

Virtual method

```
    do_copy()
    do_compare()
    convert2string()
    do_print()
    do_record()
    do_pack()
    do_unpack()
```

```
  endfunction: do_copy

endclass: bus_item
```

Designing a Sequence Item

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

  function void do_copy(uvm_object rhs);
    bus_item rhs_;

    if(!$cast(rhs_, rhs)) begin
      uvm_report_error("do_copy:", "Cast failed");
      return;
    end









  endfunction: do_copy

endclass: bus_item
```

Make sure argument is of correct type

do_copy()

Designing a Sequence Item

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

  function void do_copy(uvm_object rhs);
    bus_item rhs_;

    if(!$cast(rhs_, rhs)) begin
      uvm_report_error("do_copy:", "Cast failed");
      return;
    end
    super.do_copy(rhs);




  endfunction: do_copy

endclass: bus_item
```

Chain the copy with parent classes

Designing a Sequence Item

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

  function void do_copy(uvm_object rhs);
    bus_item rhs_;

    if(!$cast(rhs_, rhs)) begin
      uvm_report_error("do_copy:", "Cast failed");
      return;
    end
    super.do_copy(rhs);
    delay = rhs_.delay;
    addr = rhs_.addr;
    op_code = rhs_.op_code;
    slave_name = rhs_.slave_name;
    data = rhs_.data;
    response = rhs_.response;
  endfunction: do_copy

endclass: bus_item
```

do_copy()

Copy members of rhs to *this*

Designing a Sequence Item

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

  function void do_copy(uvm_object rhs);
    bus_item rhs_;

    if(!$cast(rhs_, rhs)) begin
      uvm_report_error("do_copy:", "Cast failed");
      return;
    end
    super.do_copy(rhs);
    delay = rhs_.delay;
    addr = rhs_.addr;
    op_code = rhs_.op_code;
    slave_name = rhs_.slave_name;
    data = rhs_.data;
    response = rhs_.response;
  endfunction: do_copy

endclass: bus_item
```

```
do_copy()
do_compare()
convert2string()
do_print()
do_record()
do_pack()
do_unpack()
```

USAGE:

```
bus_item A, B;
A.copy(B);
```

Deep copy B into A

OR

```
$cast(A, B.clone());
```

Clone returns a uvm_object

A Word About Phasing

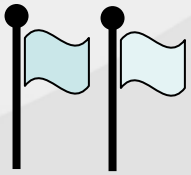
- **UVM adds 12 new phases in parallel with run_phase**
- **Consensus is to use the new phases to control stimulus**

```
class my_phase_test extends uvm_test_base;
  `uvm_component_utils(my_phase_test)

  task XXX_phase(uvm_phase phase);
    phase.raise_objection(this, “Starting Phase”);
    // Start sequence(s)
    // begin-end / fork-join
    phase.drop_objection(this, “Finished Phase”);
  endtask

endclass
```

- **Drivers and monitors should just use run_phase**
- **Don’t use phase domains or jumping**

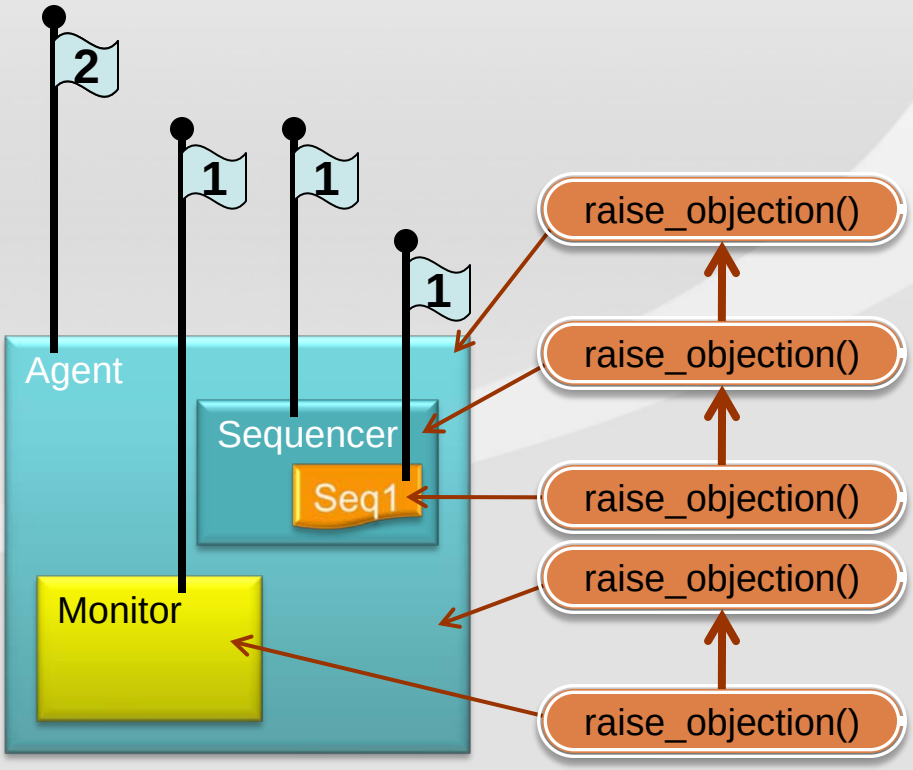
Phase Objections

- Components or Sequences can raise or drop objections
- Phase continues until all raised objections are dropped
- Objection must be raised at beginning of the phase



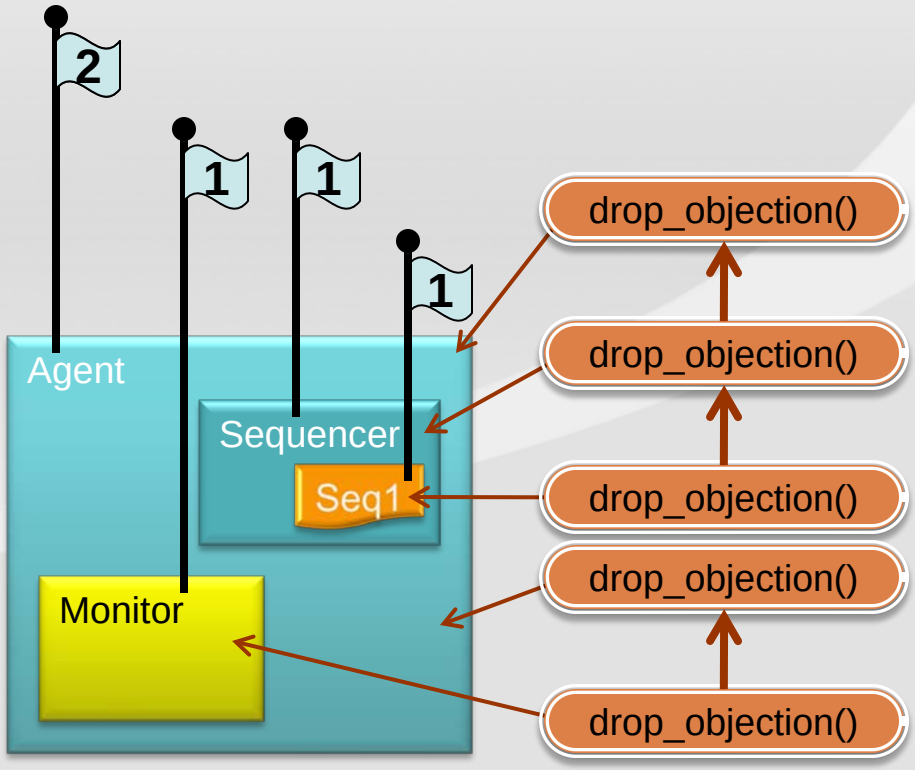
Objections are Hierarchical

- **Objections are raised up the hierarchy**



Objections are Hierarchical

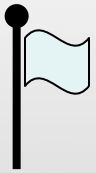
- Objections are raised up the hierarchy
- Objections are dropped hierarchically too



Using Objections

```
class my_mon extends uvm_component;
  ...
  task run_phase(uvm_phase phase);
    forever begin
      wait(tx_start);
      phase.raise_objection(this);
      collect(tr);
      ap.write(tr);
      phase.drop_objection(this);
      end
  endtask
endclass
```

When transaction starts



What if my_mon is the only objector?

Test ends when all objections dropped

Recommended

```
class my_test extends uvm_test;
  ...
  task run_phase(uvm_phase phase);
    phase.raise_objection(this);
    vseq_h.start(null);
    phase.drop_objection(this);
  endtask
endclass
```

Manage phase objections from the test

Requires that vseq_h eventually returns

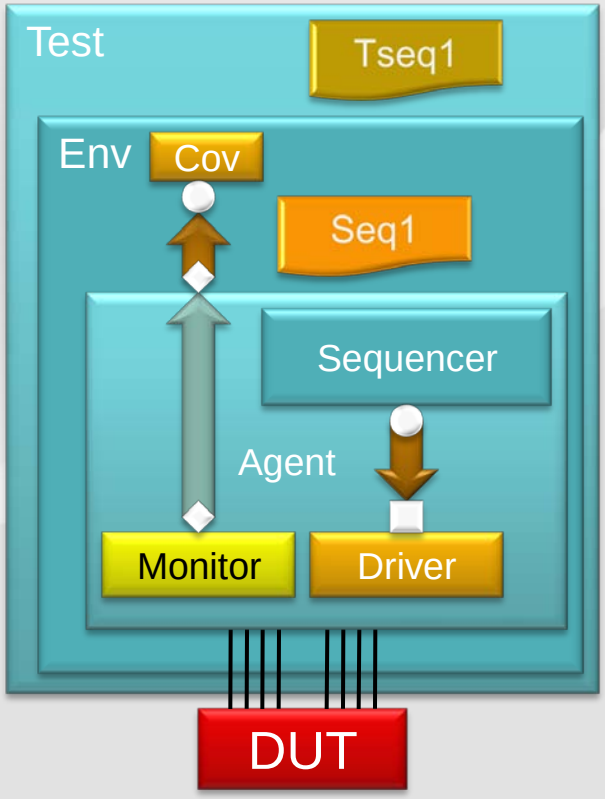
Recommendation: Objecting in a Monitor

```
class my_mon extends uvm_component;
  ...
  function void phase_ready_to_end( uvm_phase phase );
    if( !is_ok_to_end() ) begin
      phase.raise_objection( this , "not done yet" );
      fork begin
        wait_for_ok_end();
        phase.drop_objection( this , "ok to end phase" );
      end
      join_none
    end
  endfunction : phase_ready_to_end
endclass
```

Object at end of phase

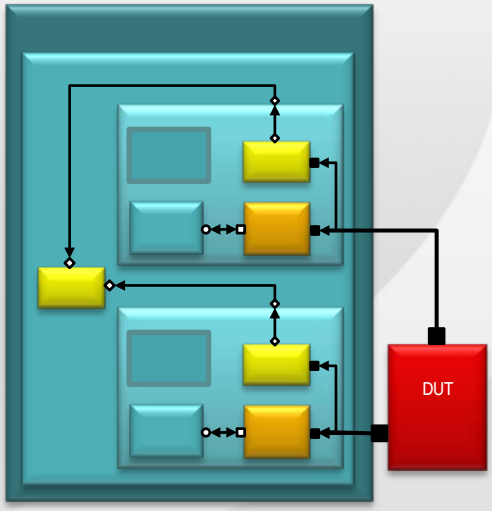
What is a Test?

- **The environment is the "Testbench"**
  - Defines what components are needed to verify the DUT
  - Specifies defaults
- **The test's job is to "tweak" the testbench**
  - Configuration
  - Factory overrides
  - Additional sequences
- **The test's other job is to ensure the simulation ends**



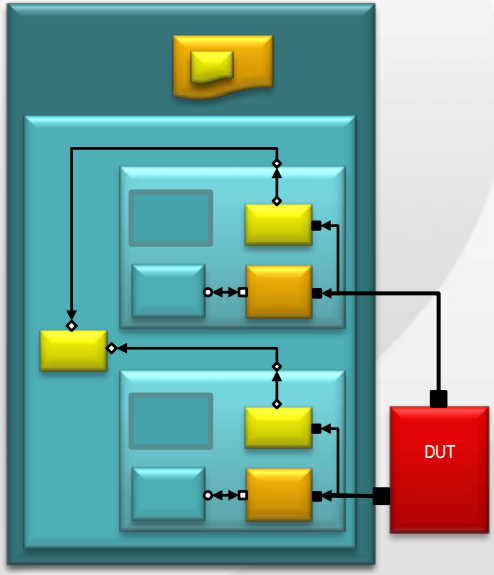
The Base Test

```
class my_test_base extends uvm_test;
  `uvm_component_utils(my_test_base)

  my_env m_env;
  my_env_config m_cfg;
  my_agent1_config m_a1_cfg;
  my_agent2_config m_a2_cfg;

  function new(string name = “my_test_base”,
                       uvm_component parent = null);
    super.new(name, parent);
  endfunction

  function void build_phase( uvm_phase phase );
    m_cfg = my_env_config::type_id::create(“m_env_cfg”);
    // setup configuration for env and agents
    uvm_config_db#(my_env_config)::set(this,"*", “my_env_config",
                                       m_cfg);
    m_env = my_env::type_id::create("m_env", this);
  endfunction

endclass
```



The Actual Test

```
class my_test extends uvm_test_base;
  `uvm_component_utils(my_test)

  my_virt_seq m_vseq;

  function new(string name = “my_test”,
               uvm_component parent = null);
    super.new(name, parent);
  endfunction

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

  task run_phase(uvm_phase phase);
    m_vseq = my_virt_seq::type_id::create(“my virtual sequence”);
    phase.raise_objection(this, “Starting virtual sequence”);
    m_vseq.start();
    phase.drop_objection(this, “Finished virtual sequence”);
  endtask

endclass
```



Setup and Invoke Test

```
module top;
  ...
  dut_if dut_if1();

  initial begin: blk
    uvm_config_db#(dut_if)::set("*", "dut_if", dut_if1, 0);


    run_test();
  end

endmodule: top
```

+UVM_TESTNAME="my_test1"

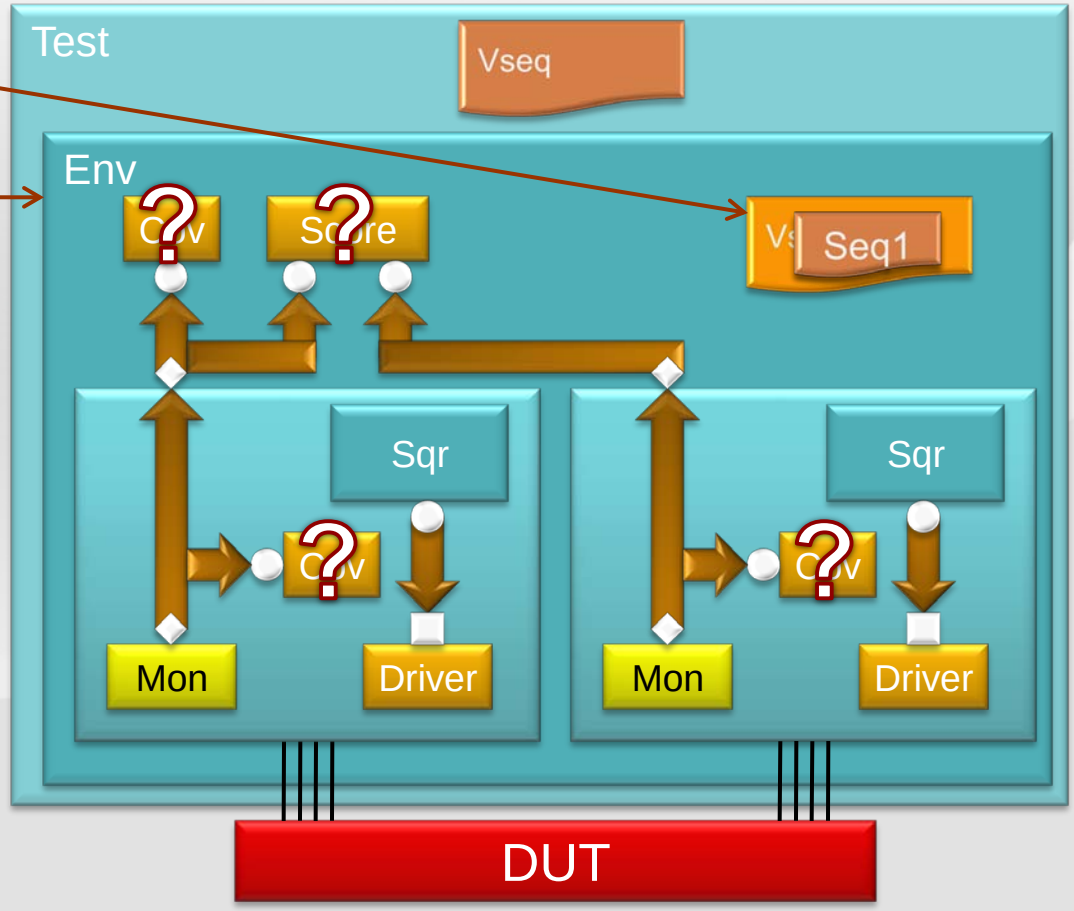
Complex Environment, Simple Test

Virtual Sequence embodies “the test”

Environment sets up agents, etc

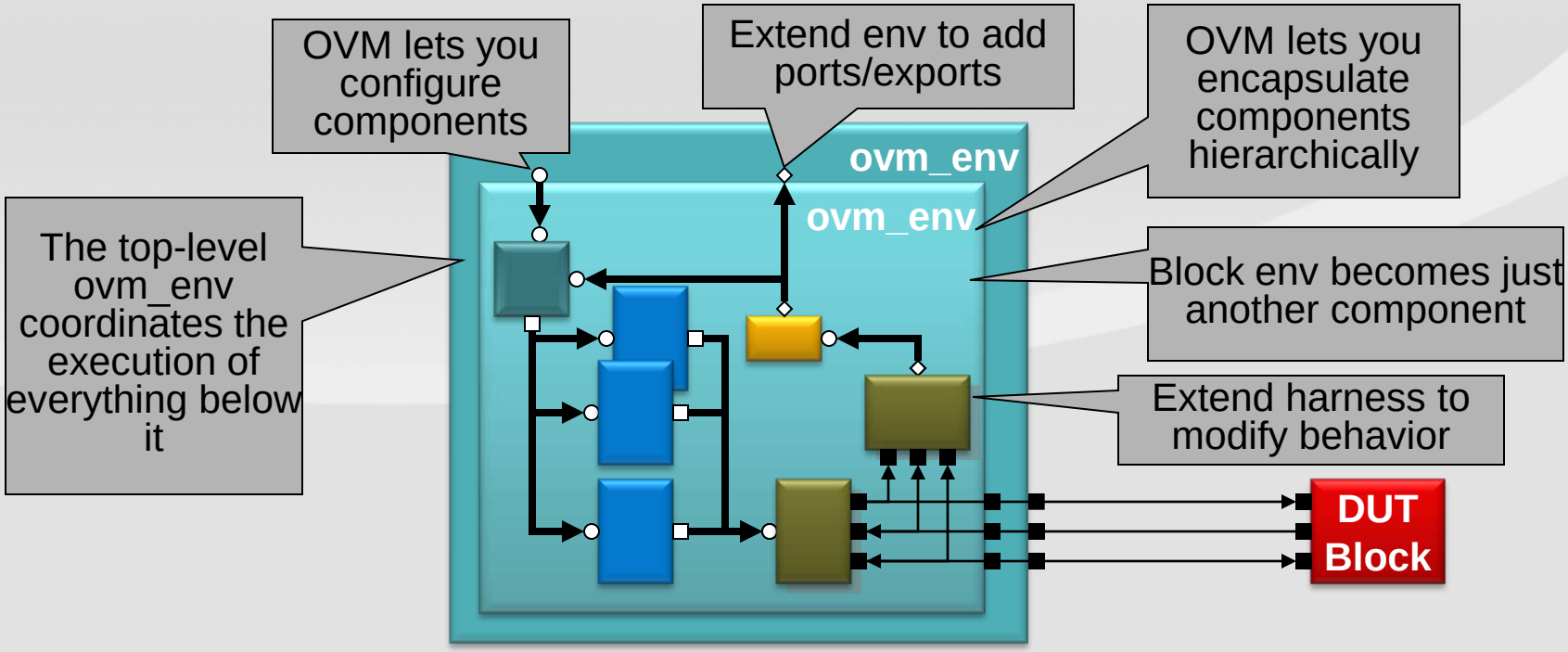
The Test simply chooses the virtual sequence to run

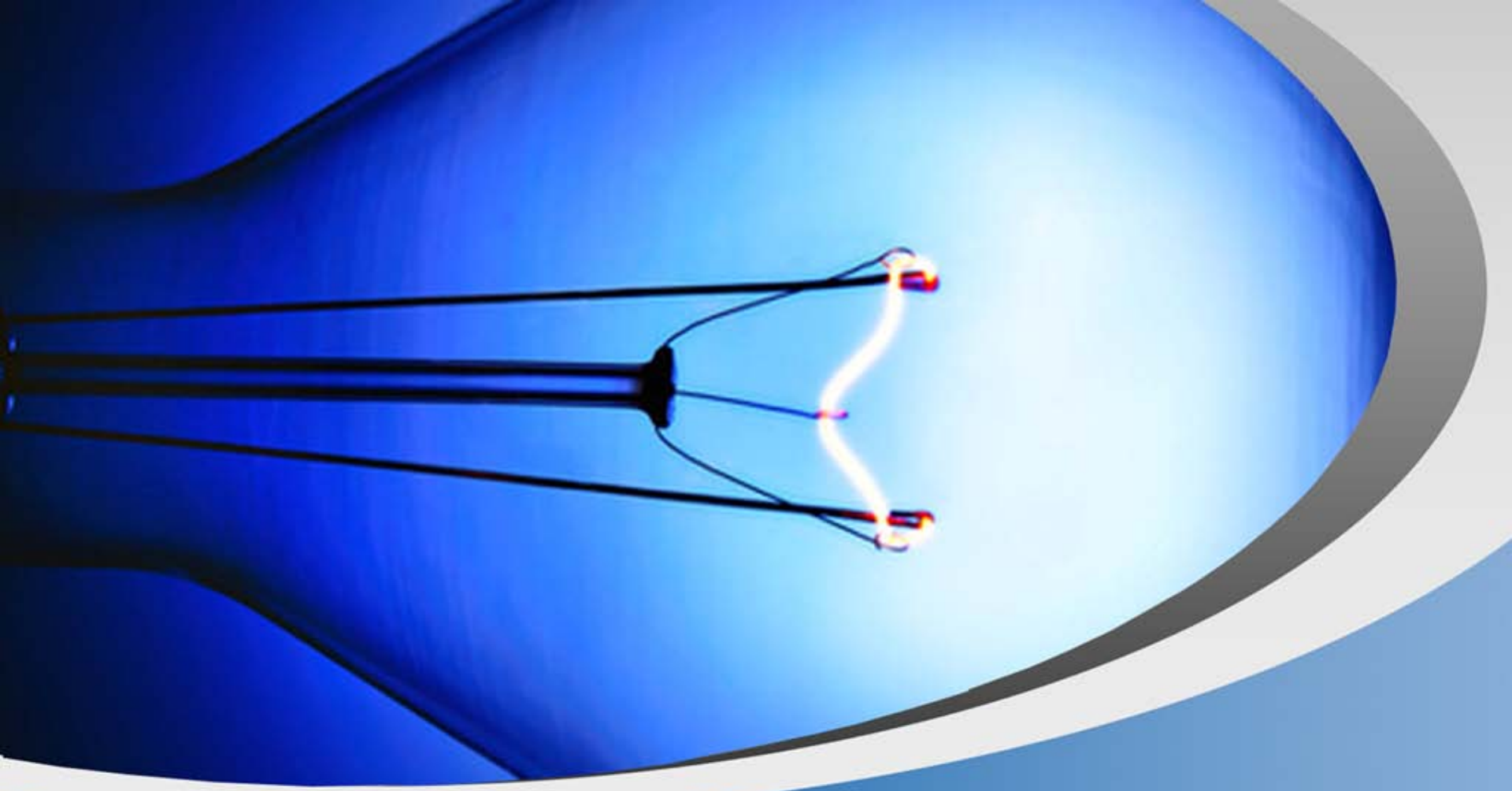
…and corresponding coverage collectors and scoreboard(s)



Environment Encapsulation

- **An OVM environment is itself an OVM component that can be instantiated hierarchically**
  - Test phases managed from top-level environment
  - Environment can be extended to add ports/exports





UVM: Ready, Set, Deploy!

UVM

DVCon 2012
Design & Verification Conference & Exhibition

accellera
SYSTEMS INITIATIVE

Communication and Sequences

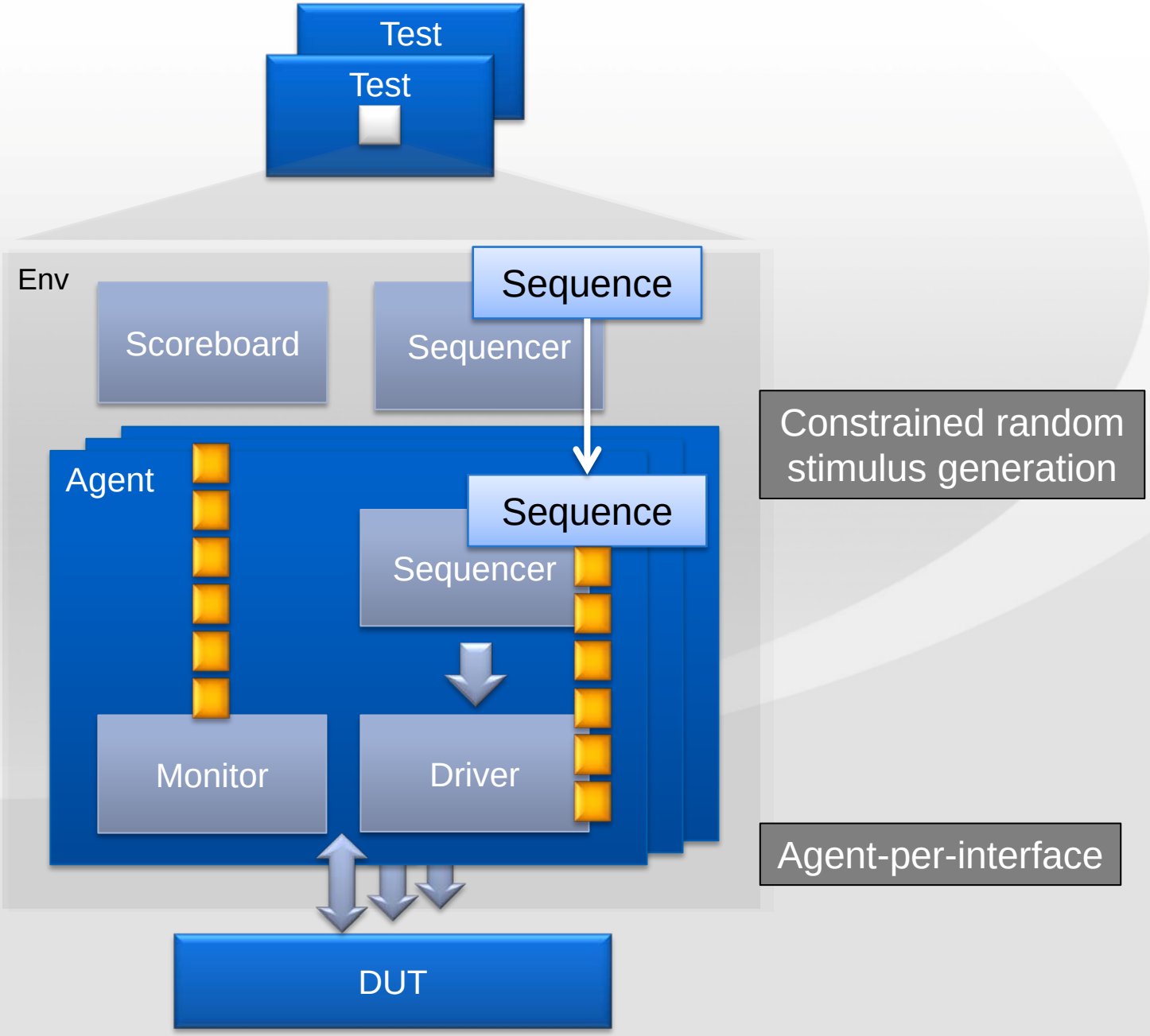
John Aynsley

CTO

Communication and Sequences

- TLM Communication in UVM
- Sequencer-Driver Communication
- Structuring Sequences

Review

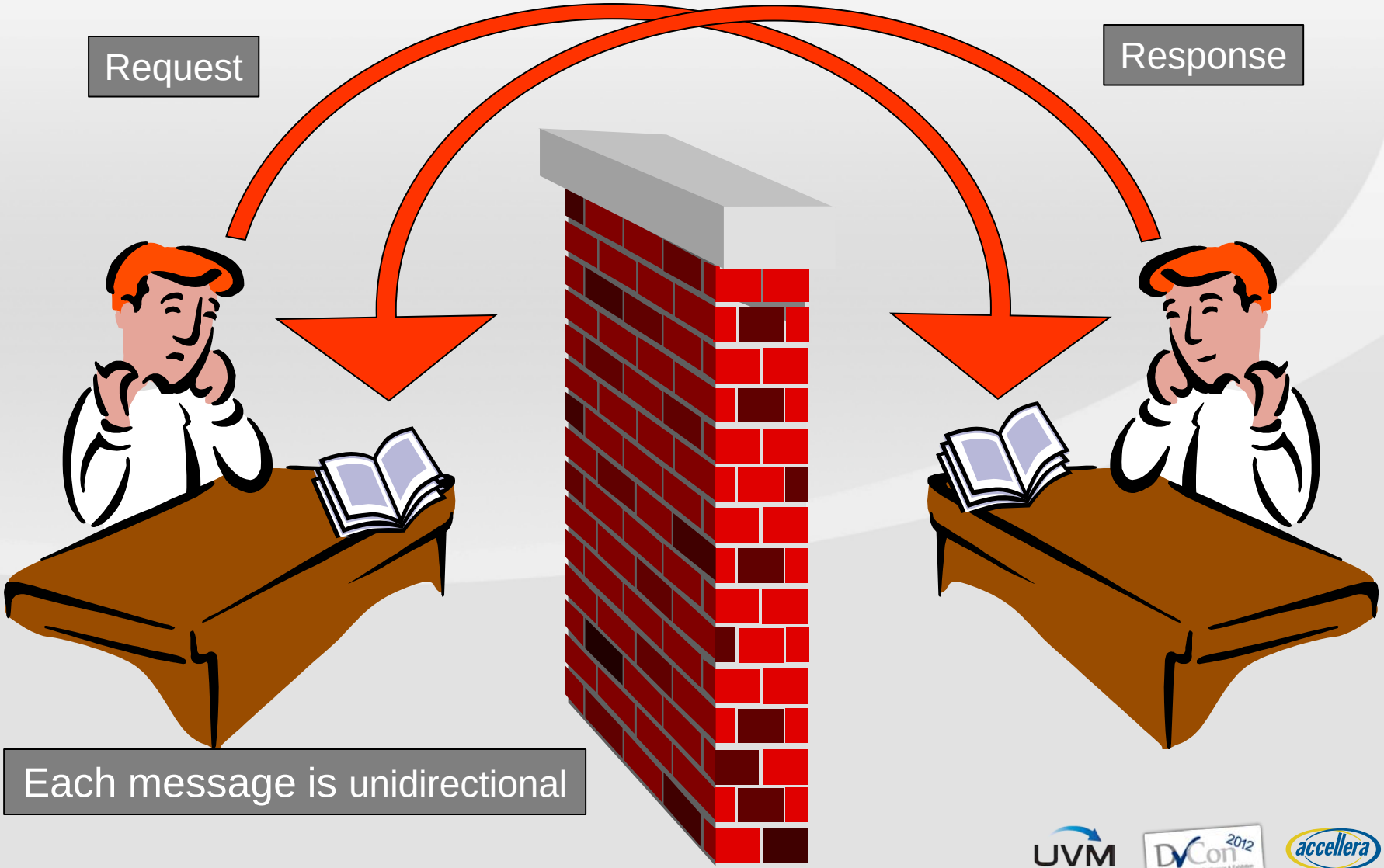


Checking and Coverage

Constrained random stimulus generation

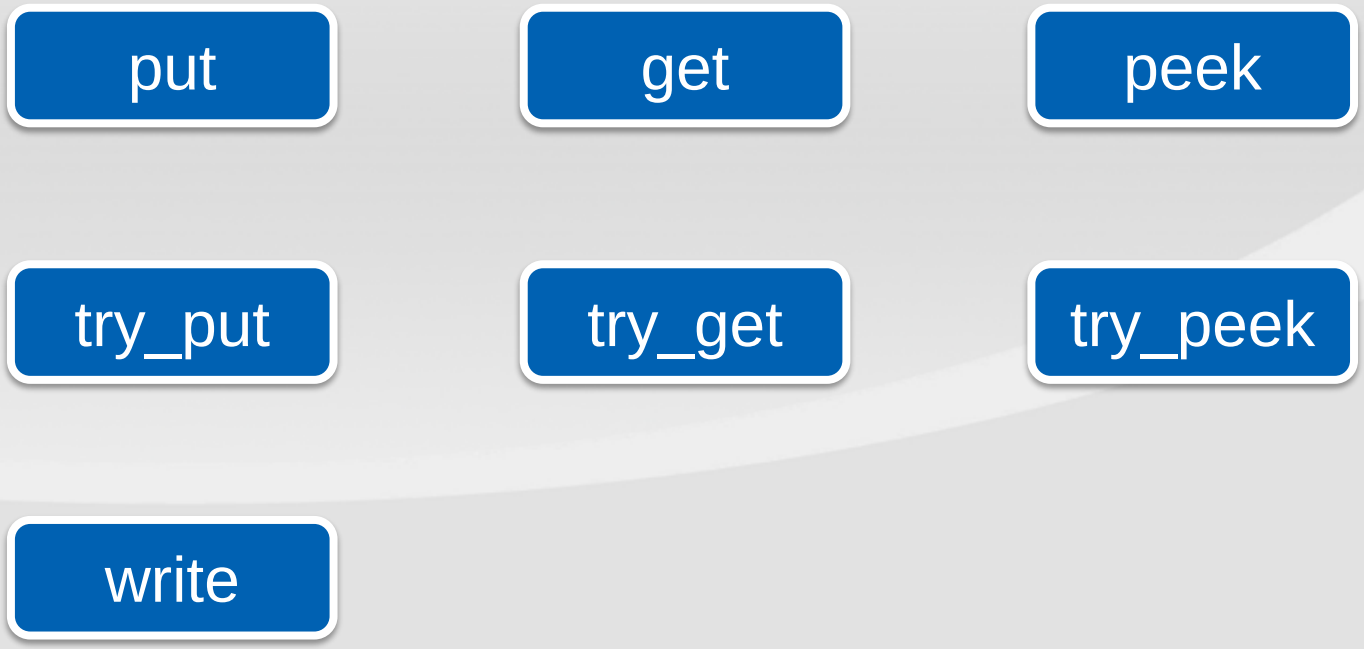
Agent-per-interface

TLM-1 = Message Passing

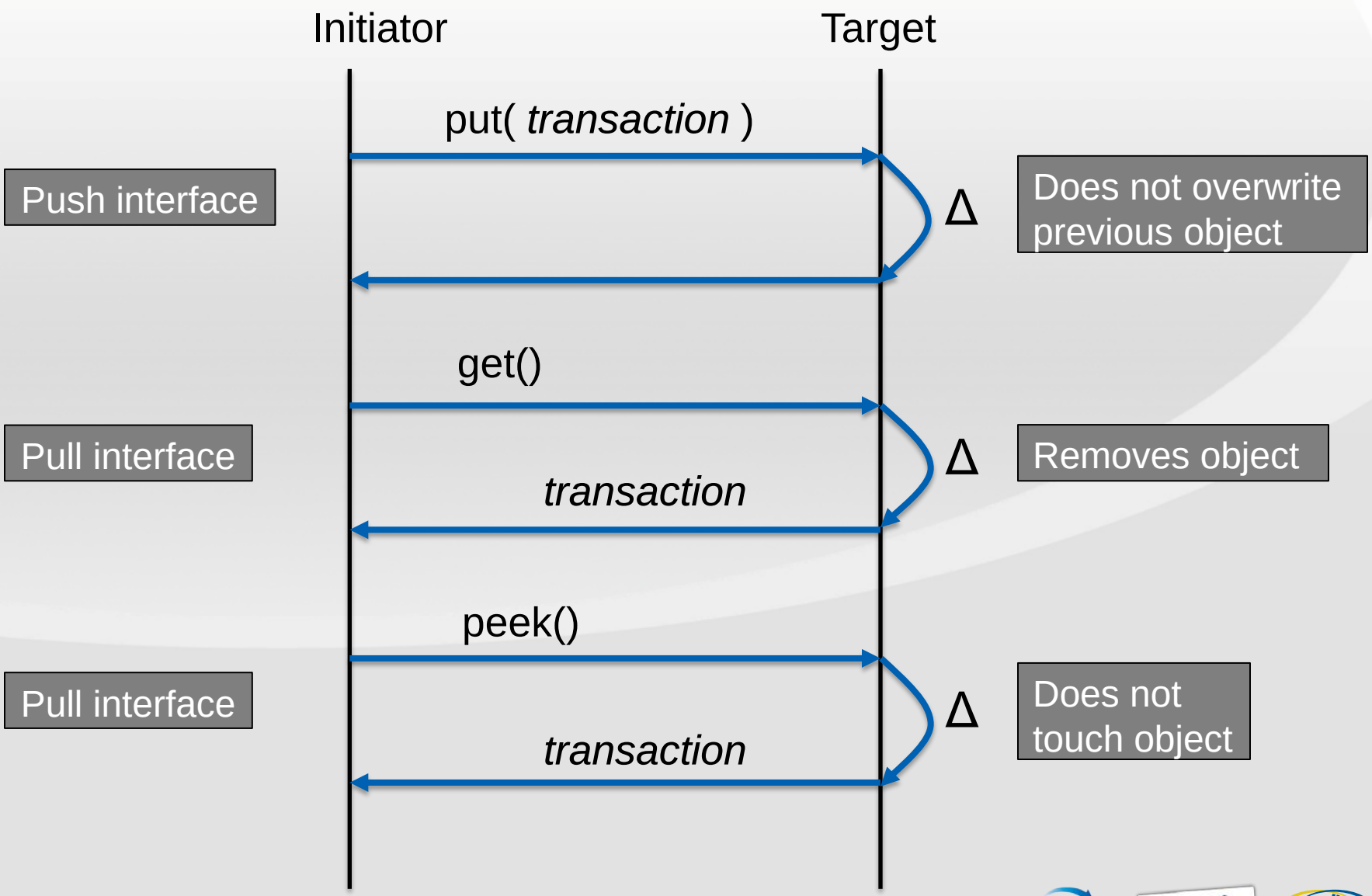


TLM-1 in UVM

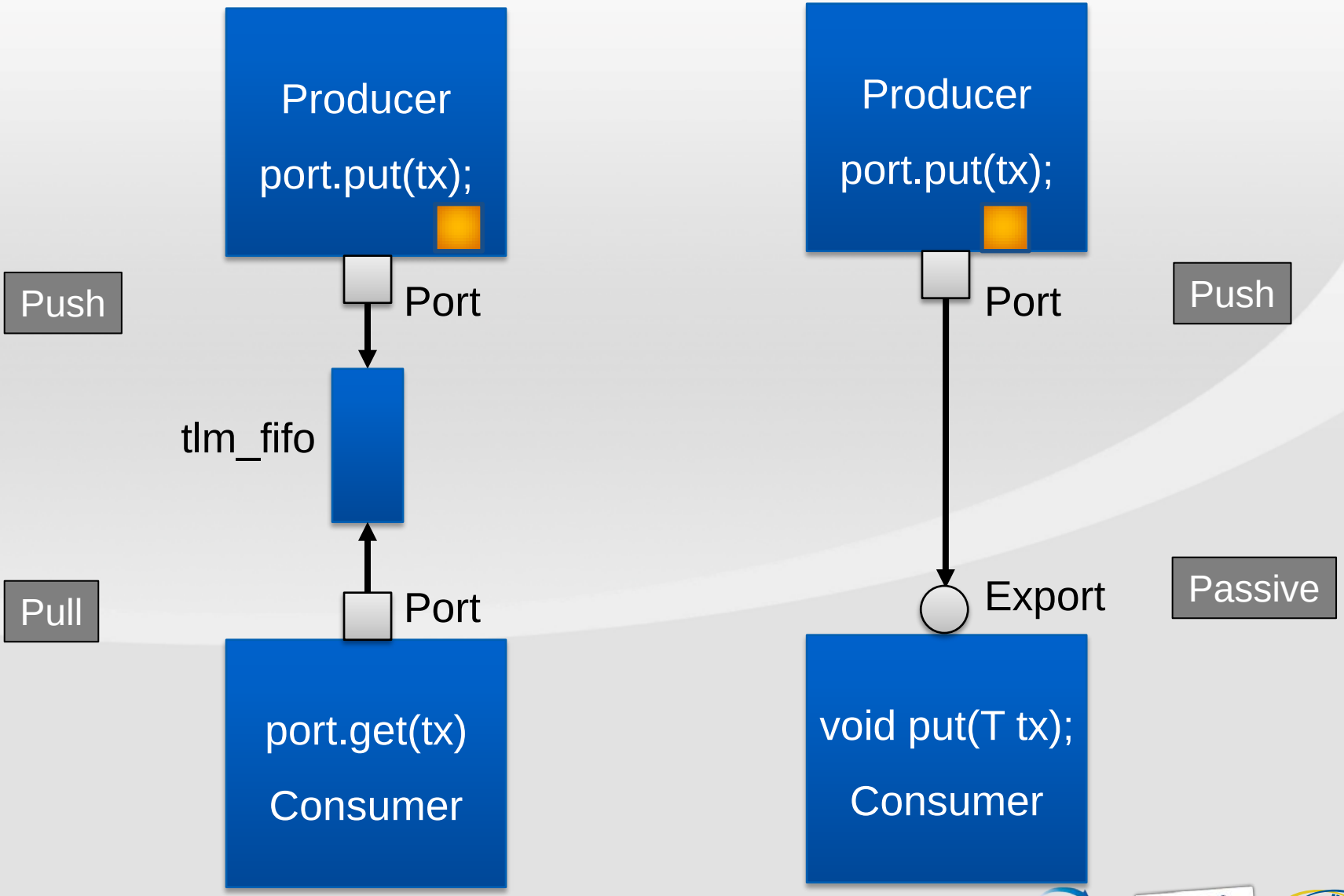
SystemC TLM-1 standard method calls



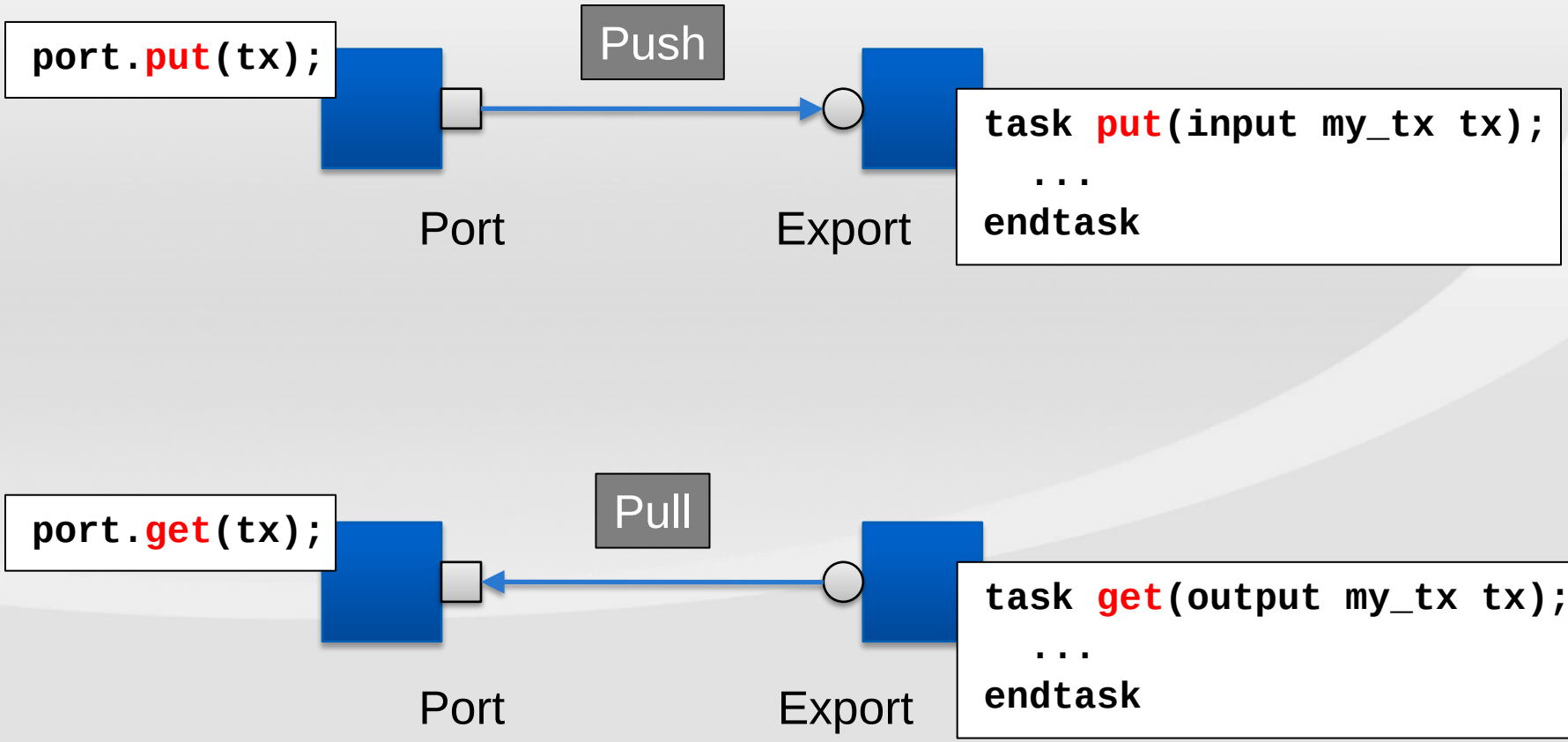
put/get/peek Semantics



Mediated vs Direct Communication



Push versus Pull



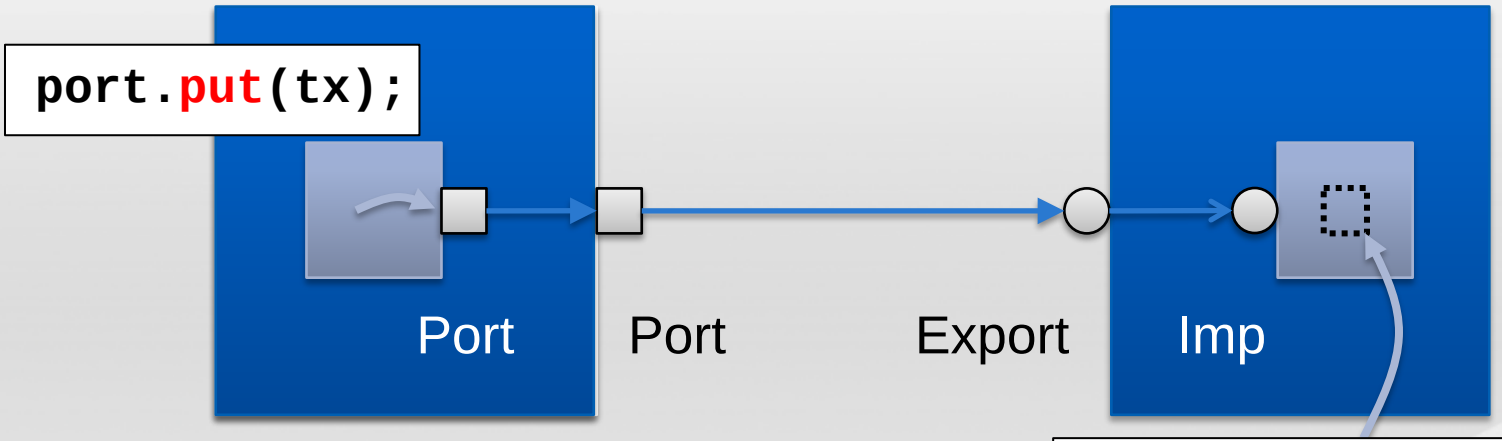
Required / Provided Interfaces


```
put_port.put(tx);
```

```
task put(T tx);
```


```
producer.put_port.connect( consumer.put_export );
```

Export versus Imp




```
task put(input my_tx tx);
  ...
endtask
```

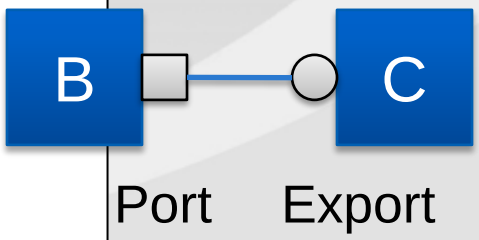
Export = a waypoint
Imp = the end of the line

Build Phase

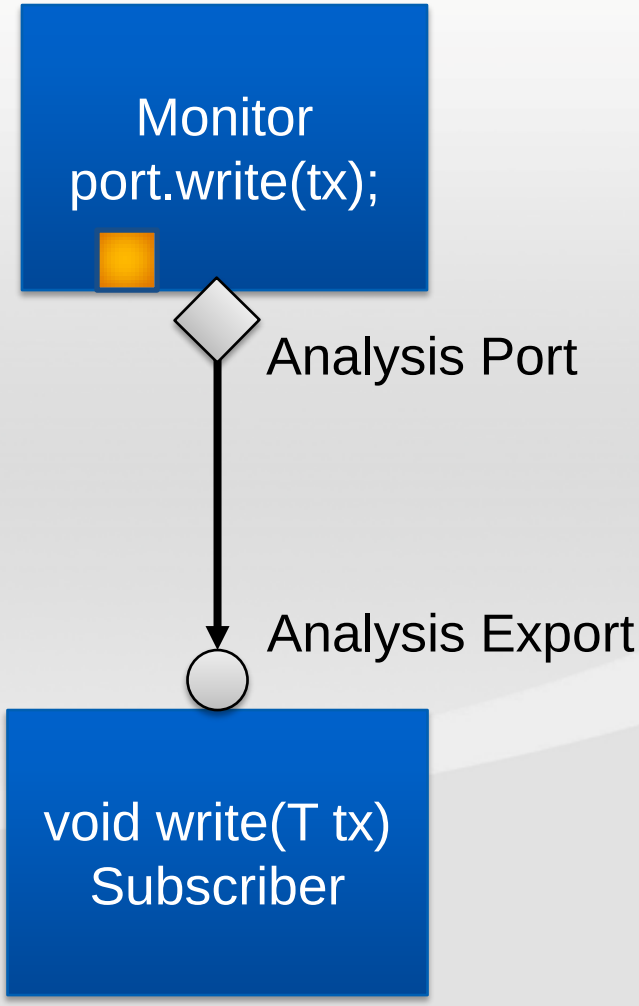
```
class A extends uvm_component;
  `uvm_component_utils(A)

  B  b;
  C  c;
  ...
  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    b = B::type_id::create("b", this);
    c = C::type_id::create("c", this);
  endfunction
```

Connect Phase

```
class A extends uvm_component;
  `uvm_component_utils(A)

  B  b;
  C  c;
  ...
  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    b = B::type_id::create("b", this);
    c = C::type_id::create("c", this);
  endfunction

  function void connect_phase(uvm_phase phase);
    b.port.connect( c.export );
  endfunction
```



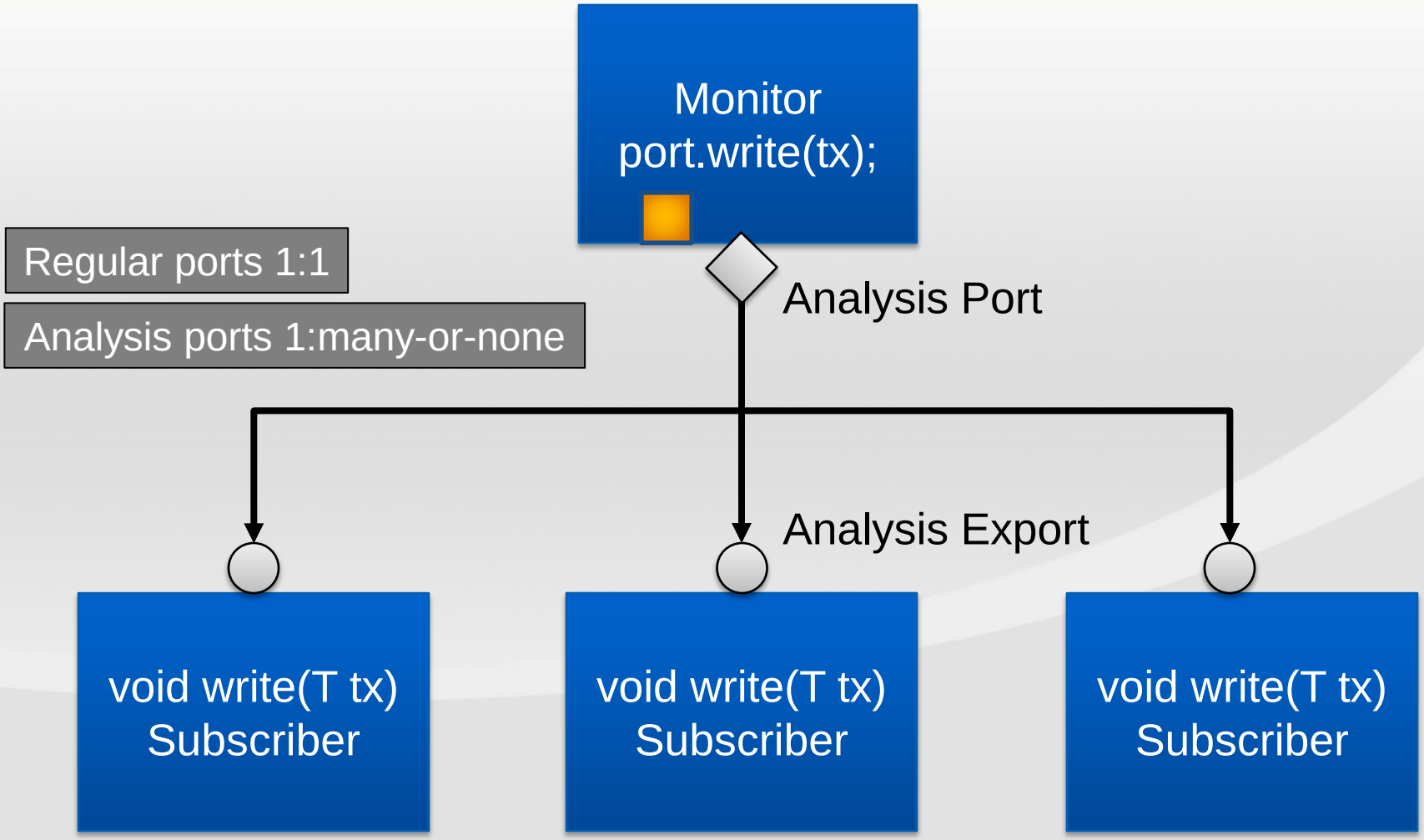
Analysis Ports



Write is non-blocking

subscribers are passive

Analysis Ports



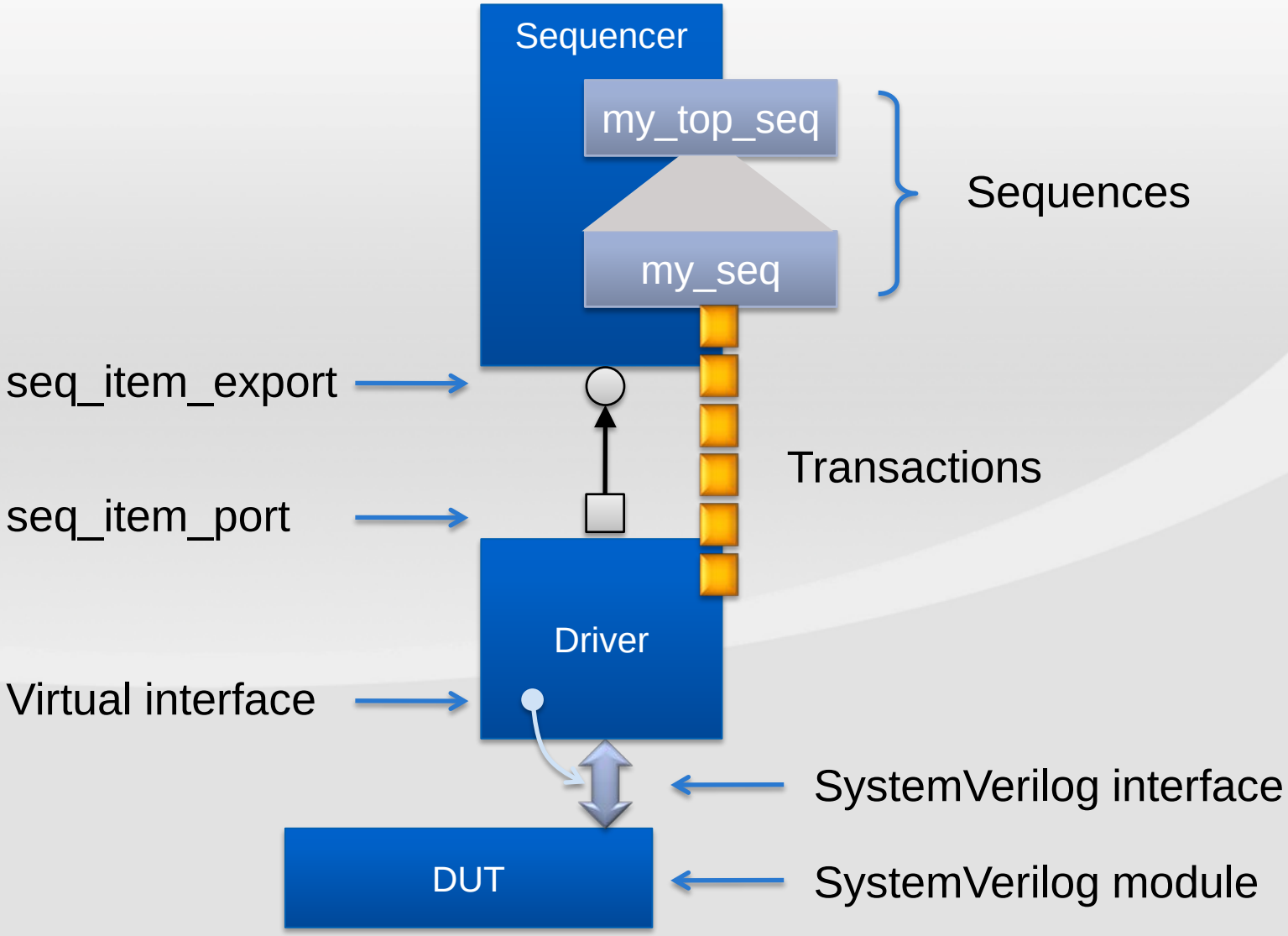
Write is non-blocking

subscribers are passive

Communication and Sequences

- TLM Communication in UVM
- Sequencer-Driver Communication
- Structuring Sequences

Sequencer - Driver - DUT



Body of a Sequence

```
class my_seq extends uvm_sequence #(my_tx);
  `uvm_object_utils(my_seq)

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

  task body;
    repeat (6)
    begin
      req = my_tx::type_id::create("req");

      start_item(req);

      assert(req.randomize() with {data > 127;});

      finish_item(req);
    end
  endtask
endclass
```

Handshake with driver

Late randomization

Or

```
`uvm_do_with(req, {data > 127;})
```

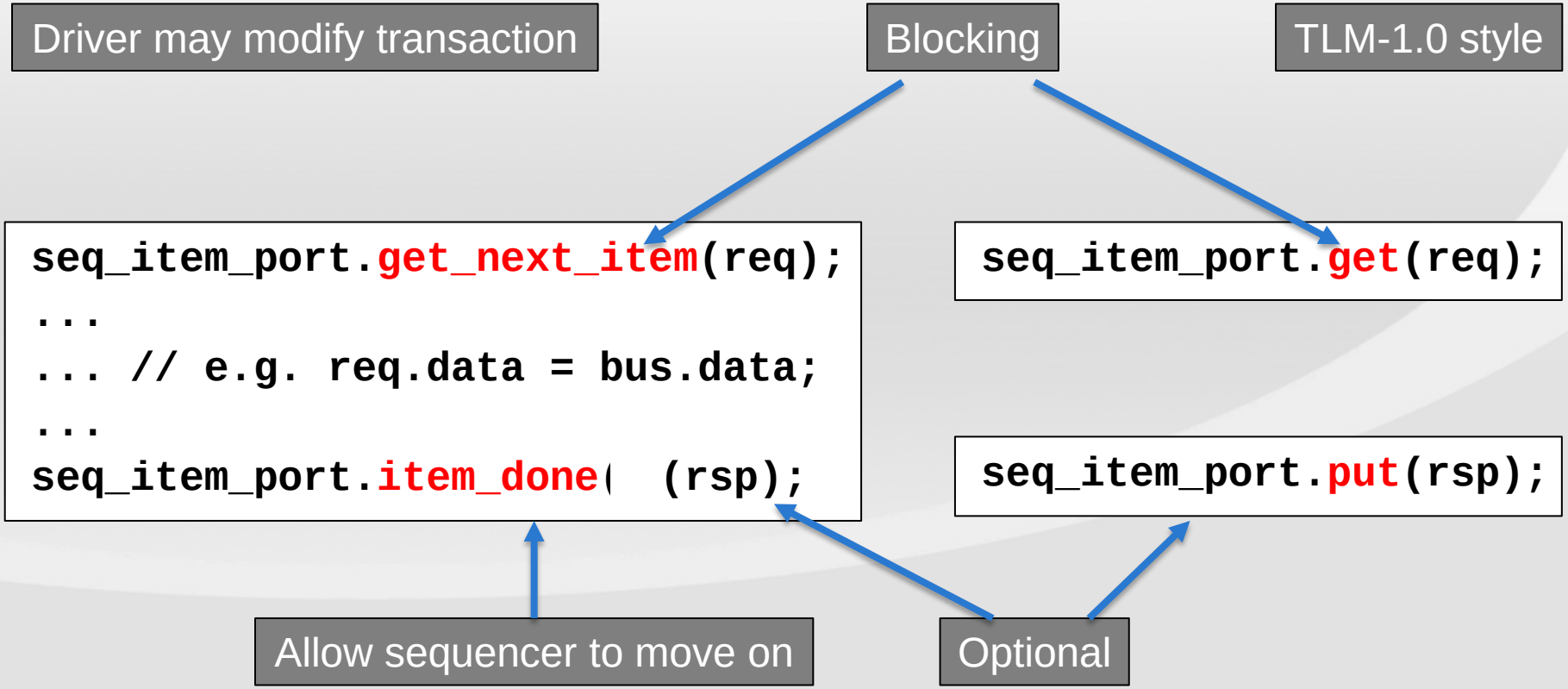
Driver Run Phase

Inherited port

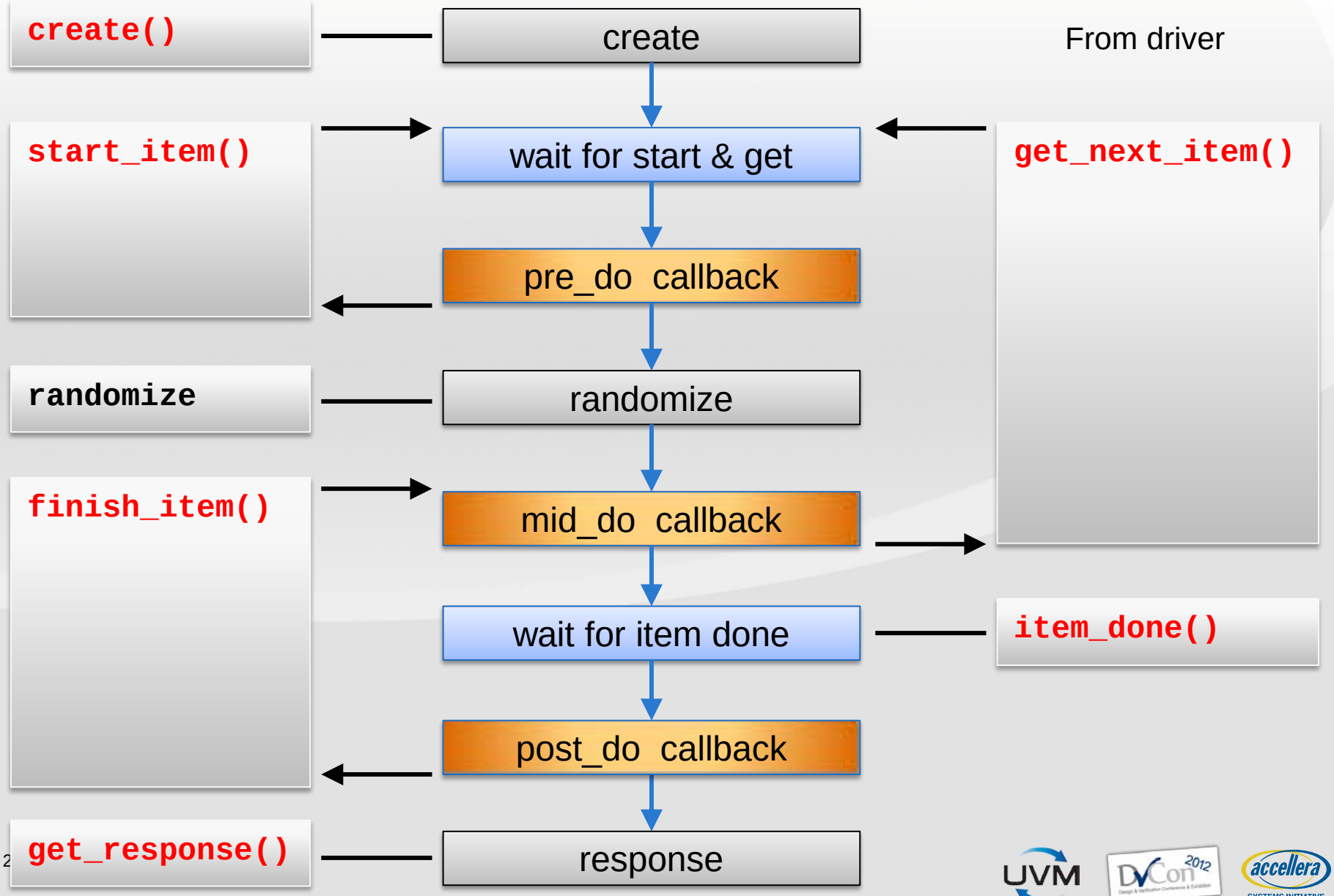
```
task run_phase( uvm_phase phase );
  forever
  begin
    seq_item_port.get_next_item(req);

    phase.raise_objection(this);
    @(posedge dut_vi.clock);
    dut_vi.cmd  = req.cmd;
    dut_vi.addr = req.addr;
    dut_vi.data = req.data;
    phase.drop_objection(this);

    seq_item_port.item_done();
  end
endtask
```

Wiggle DUT pins through interface

Sequencer-Driver Interface



Sequencer-Driver Interface



Overriding Sequence Behavior

```
class my_top_seq extends uvm_sequence #(my_top_tx);
  ...
  task body;
    req = my_top_tx::type_id::create("req");
    start_item(req);
    assert( req.randomize() );
    finish_item(req);
    ...
```

Built into VIP

```
class alt_top_seq extends my_top_seq;
  ...
  function void mid_do(uvm_sequence_item this_item);
    $cast(tx, this_item);
    tx.addr = prev_addr + $urandom_range(1, 7);
  endfunction

  function void post_do(uvm_sequence_item this_item);
    $cast(tx, this_item);
    prev_addr = tx.addr;
  endfunction
  ...
```

Called before releasing req

Called after releasing req

For a specific test

```
my_top_seq::type_id::set_type_override( alt_top_seq::get_type() );
```

Communication and Sequences

- TLM Communication in UVM
- Sequencer-Driver Communication
- Structuring Sequences

Starting a Sequence from a Test

```
task run_phase(uvm_phase phase);
  uvm_component seqr;
  my_sequence seq;

  seq = my_sequence::type_id::create();
  seq.starting_phase = phase;

  seqr = uvm_top.find("*.*_sequencer1");
  seq.start( seqr );
endtask
```

Must create sequence object before calling start

Starting from Another Sequence

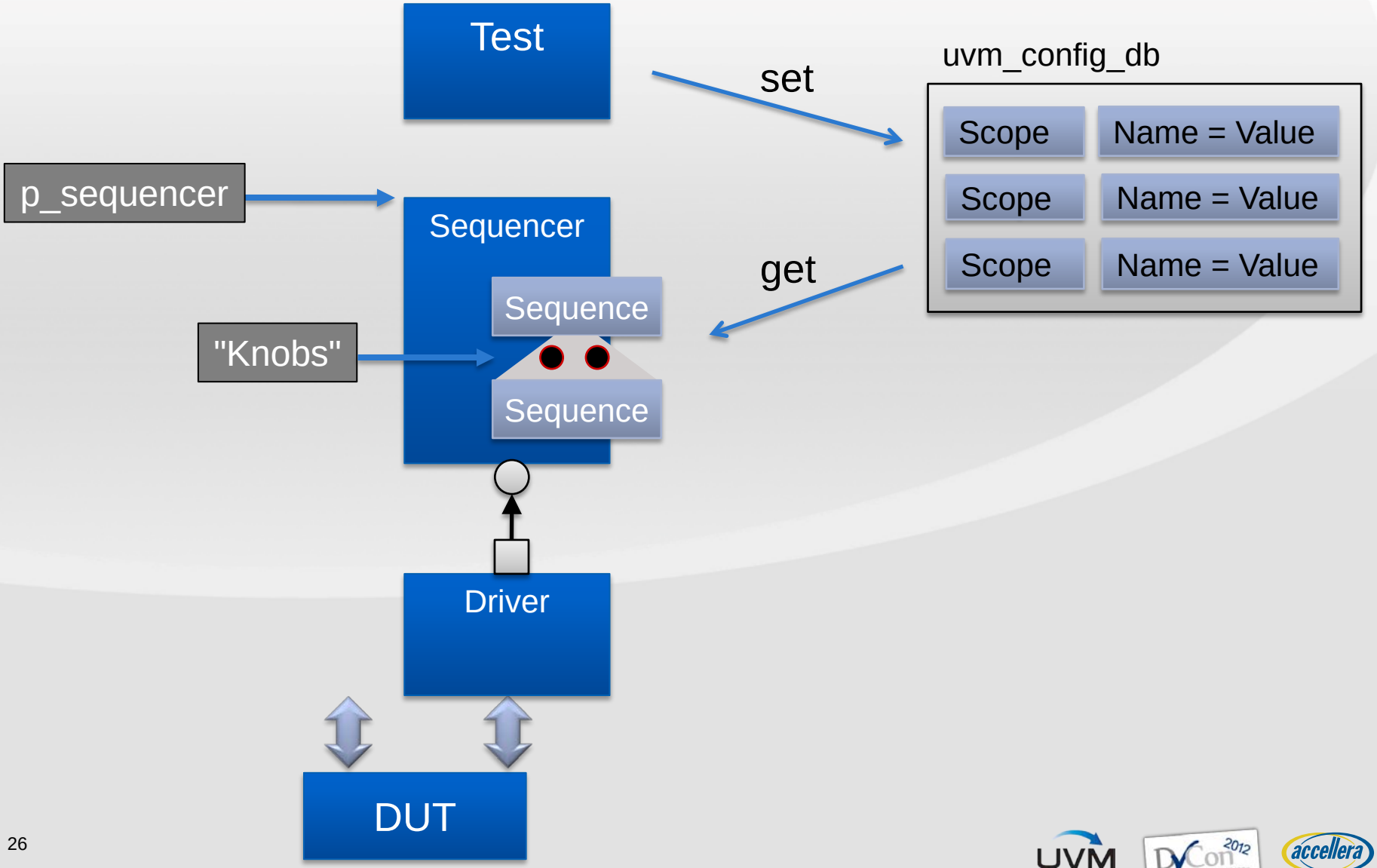
```
task body;
  seq = my_sequence::type_id::create();
  seq.starting_phase = phase;

  seq.start( p_sequencer, this, -1, 0 );
endtask
```

Sequencer, parent_sequence, priority, call_pre_post_body

Or

```
task body;
  my_sequence seq;
  `uvm_do(seq)
endtask
```

Constrained Random Sequences



Sequence Control Knobs

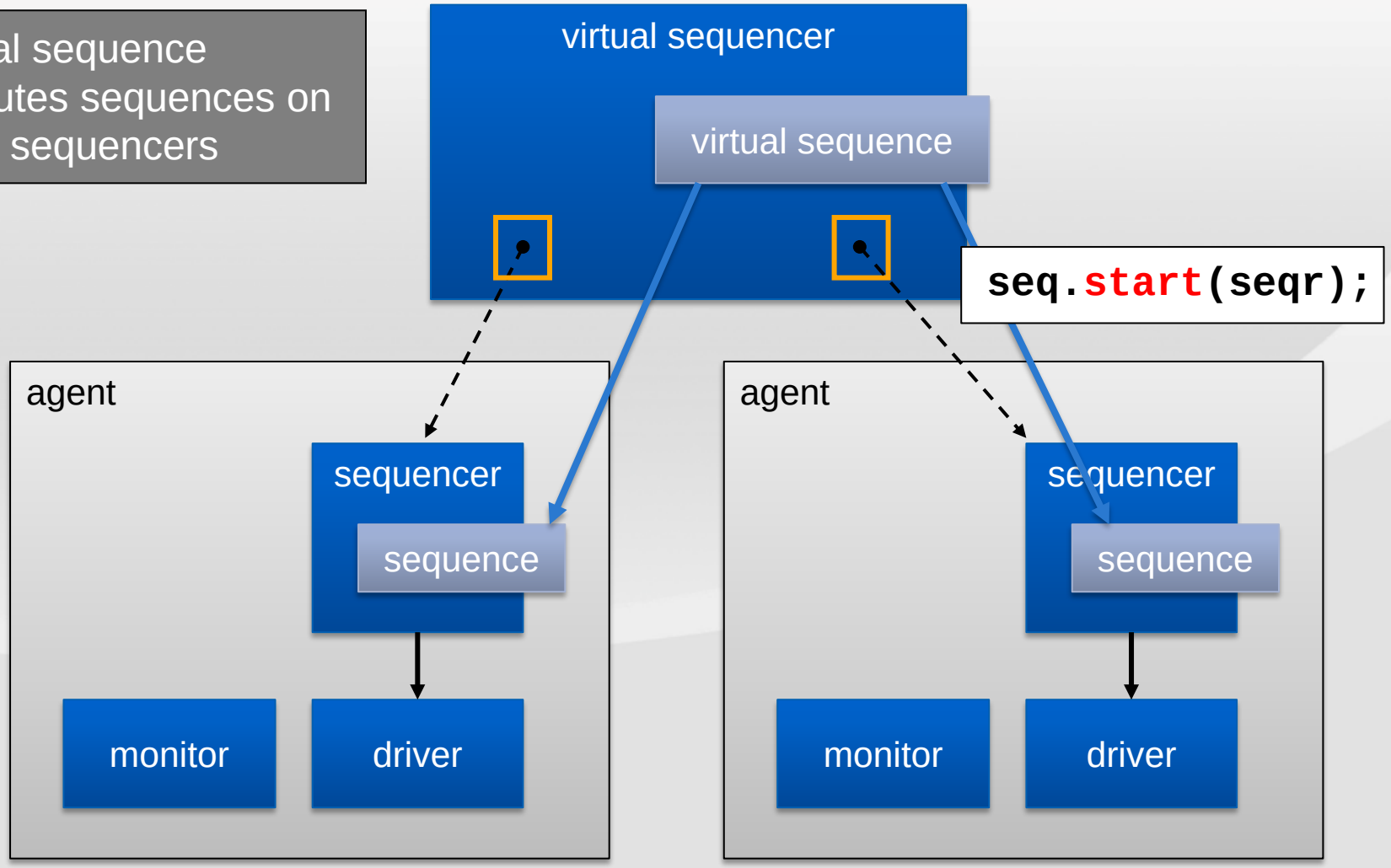
```
class mem_burst_read extends uvm_sequence #(instruction);
  rand shortint unsigned start, length;
  ...
  task body;
   `uvm_do_with(  req,
                 {op == movi_op;  {src,src2} == start[7:0]; } )
   `uvm_do_with(  req,
                 {op == movhi_op; {src,src2} == start[15:8]; } )
   ...
   for (int i = 0; i < length; i++) begin
     `uvm_do_with(req,
                 {op == load_op;  src == base; } )
     `uvm_do_with(req,
                 {op == addi_op;  src == base; src2 == 1; } )
     ...
```

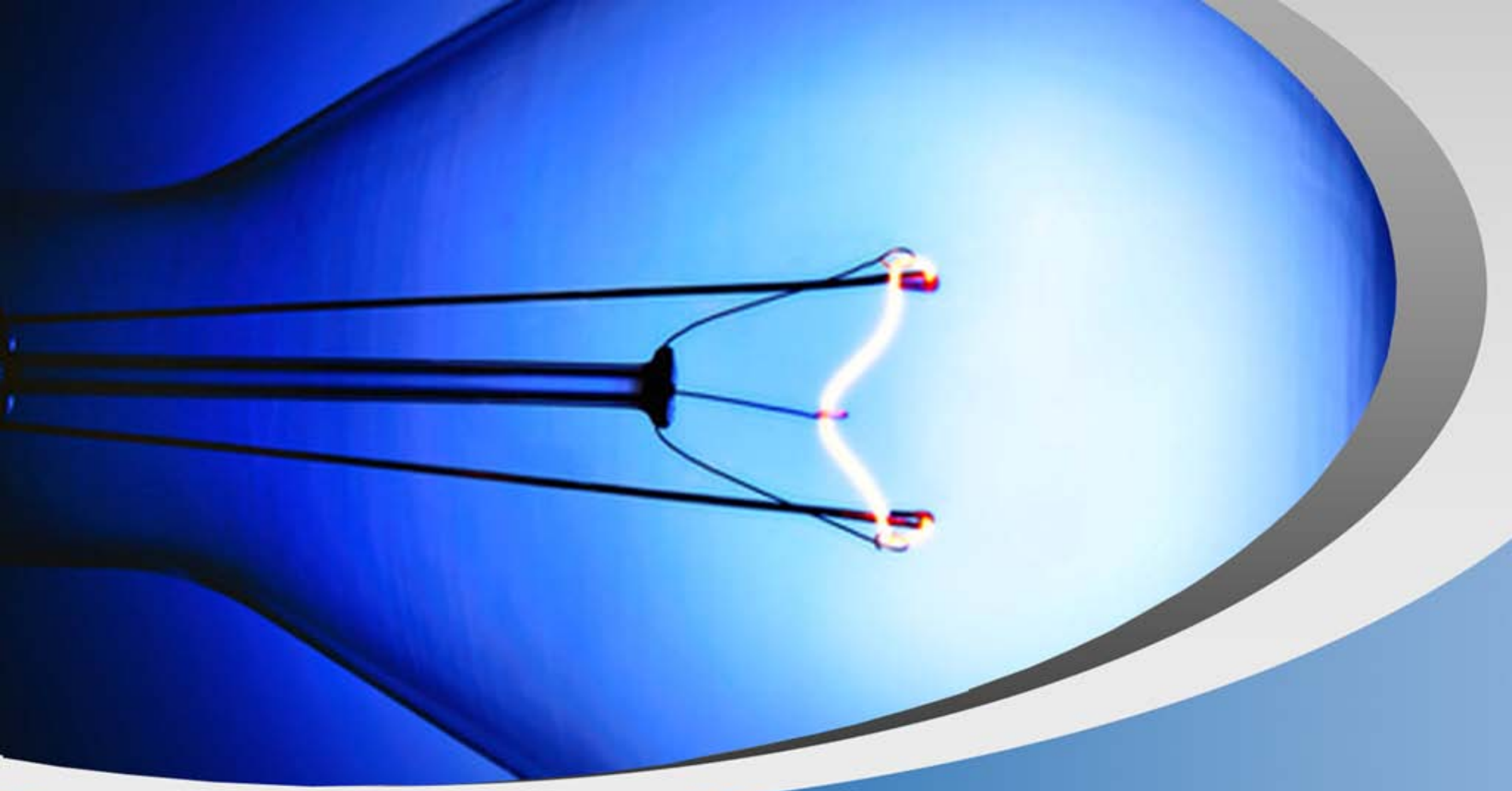
Nested Sequences

```
class mem_copy extends uvm_sequence #(instruction);
  ...
  rand int   addr1, addr2, size;

  constraint word_bndry  { addr1[1:0] == 2'b00; }
  constraint whole_words { size[1:0]  == 2'b00; }
  constraint no_overlap  { addr2 > addr1 + size; }

  mem_burst_read  burst1;
  mem_burst_write burst2;

  task body;
   `uvm_do_with(burst1, { start == addr1; length == size; } )
   `uvm_do_with(burst2, { start == addr2; length == size; } )
  endtask
  ...
```

Virtual Sequences

Virtual sequence executes sequences on other sequencers





UVM: Ready, Set, Deploy!

UVM

DVCon 2012
Design & Verification Conference & Exhibition

accellera
SYSTEMS INITIATIVE

Customizing Your UVM Environment

Kathleen Meade

Verification Solutions Architect

cadence®

UVM DVCon 2012 accellera SYSTEMS INITIATIVE

Customizing Your UVM Environment

- **UVM provides guidelines for developing reusable verification IP**
  - Base classes and standard API are provided for consistency and ease-of-use
- **UVM components can be easily configured**
  - Designed with default behavior that can be modified “from above”
    - Examples: active/passive agents, numbers of masters/slaves, enabling/disabling checking and coverage and specifying the virtual interface connection
- **UVM sequences generate random stimulus to exercise interesting scenarios**
  - Stimulus can be further constrained with constraint layering and other UVM techniques

Aspects of Your Environment

- **Setting topology during environment creation**
  - Number of master, slaves
  - Active/passive components
  - Specifying the virtual interface connection
- **Setting default parameters for testbench**
  - Data bus is 64 bits, 32 bits, etc.
  - Legal address ranges for slave devices

Configuration that must be set up before run time

- **Specifying additional constraints for stimulus**
  - Data items : Transaction delays < 10 clocks
  - DUT config: Set **max_len_reg** value range of 30 to 64
- **Turning off certain checks in the scoreboard**
- **Changing verbosity of print messages**

Can be changed during run time

UVM Facilities for Customization

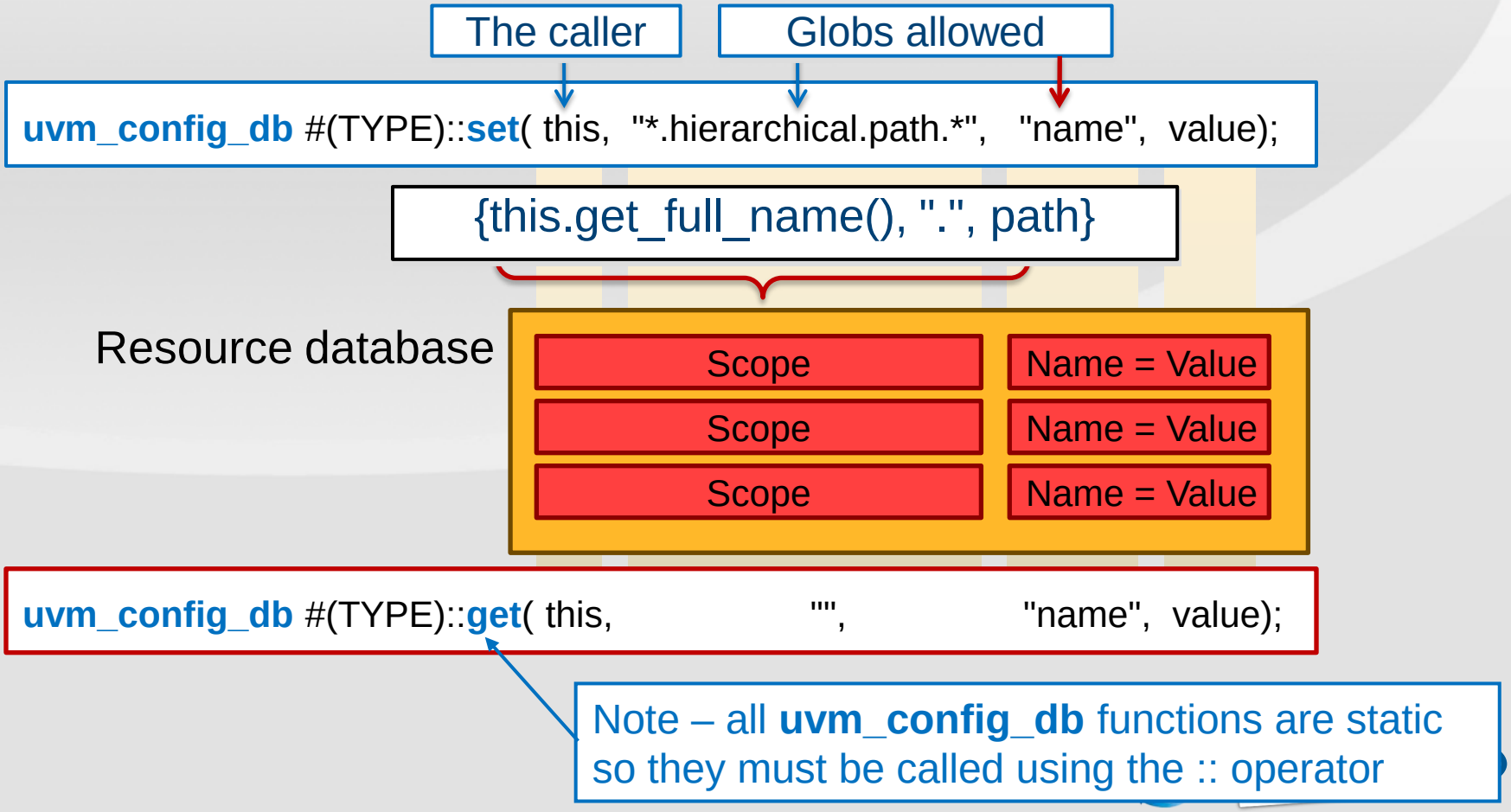
- **Configuration**
- **Factory**
- **Messaging**
- **Callbacks**
- **Command-line processor**

Configuring a Testbench Environment

- **Configuration can affect the structure of the testbench**
  - For example, **`is_active`** property of an agent determines whether sequencer/driver sub-components are created
- **Can also define or affect the stimulus for the test**
- **Desire to control configuration from a higher level**
  - Typically in a high-level testbench or test
- **Config properties must be set before components are created**
  - For example, set number of master agents before agent is constructed
- **We do *not* want to pass property values from the top via arguments to `new()` or `build_phase()`**

The Configuration Database

- **UVM provides a global resources database to implement its configuration mechanism**
  - Any **`uvm_object`** can access the database by a scope string and a type



Configuration Database Syntax

```
class uvm_config_db#(type T=int) extends uvm_resource_db#(T);
  static function void set ( uvm_component cntxt,
         string inst_name, string field_name, T value);
  static function bit get ( uvm_component cntxt,
         string inst_name, string field_name, inout T value);
  static function bit exists(...);
  static function void dump();
  static task wait_modified(...);
endclass
```

- **Note: UVM1.0EA included a configuration mechanism for components**
  - Used to configure integral types, strings and objects

```
virtual function void set_config_int(string inst_name,
                        string field, bitstream_t value);
Also: set_config_string(…), set_config_object(…);
```

- **These functions still exist as a convenience on top of the resources database**

Setting a Config Property in a Test

```
class my_test extends uvm_test;
  apb_env env;
  ...
  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);     // important!
    uvm_config_db#(uvm_bitstream_t)::set(this, "env.agent",
                                         "is_active", UVM_PASSIVE);
    env = apb_env::type_id::create("env", this);
  endfunction

endclass
```

Always call **super.build_phase()** to apply test config settings

The env build_phase will be called after the test build_phase

```
class apb_env extends uvm_env;
  apb_agent agent;
  ...
  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);     // important!
    agent = apb_agent::type_id::create("agent", this);
  endfunction
endclass
```

The agent build_phase is called after the env build_phase

Using Config Property in Agent (1)

apply_config_settings() in component build phase

Note: monitor not shown

```
class apb_agent extends uvm_agent;
  //uvm_active_passive_enum is_active = UVM_ACTIVE; //built-in field

  `uvm_component_utils_begin(apb_agent)
    `uvm_field_enum(uvm_active_passive_enum, is_active, UVM_ALL_ON)
  `uvm_component_utils_end

  apb_sequencer  sequencer;
  apb_driver     driver;
...
 virtual function void build_phase(uvm_phase phase);
   super.build_phase(phase);
   if (is_active == UVM_ACTIVE)
    begin
     sequencer = apb_sequencer::type_id::create("sequencer", this);
     driver    = apb_driver::type_id::create("driver", this);
    end
  endfunction
. . .
endclass
```

config property is automated

is_active set during super.build_phase()

build creates agent sub-components (which in turn are implicitly built)

Using Config Property in Agent (2)

Calling get() explicitly

```
class apb_agent extends uvm_agent;
  //uvm_active_passive_enum is_active = UVM_ACTIVE; //built-in field

  apb_sequencer  sequencer;
  apb_driver     driver;
...
 virtual function void build_phase(uvm_phase phase);
  super.build_phase(phase);
  uvm_config_db#(uvm_bitstream_t)::get(this, "",
                                       "is_active", is_active);

  if (is_active == UVM_ACTIVE)
   begin
     sequencer = apb_sequencer::type_id::create("sequencer", this);
     driver    = apb_driver::type_id::create("driver", this);
    end
  endfunction
. . .
endclass
```

get() looks to see if field has been set – or uses default value

agent sub-components are created based on config property

Setting the Virtual Interface

```
//Use the get function to retrieve a virtual interface
function void apb_monitor::connect_phase(uvm_phase phase);
  if (!uvm_config_db#(virtual ubus_if)::get (this,
                                        "", "vif", vif ) )
    `uvm_error("NOVIF", "virtual interface not set")
endfunction
```

Note: use **`uvm_error** (not fatal) to get all the connectivity errors in one simulation run

```
// Setting the virtual interface in the top module
module ubus_top;
  ubus_if ubus_if0(); // instance of the interface

  initial begin
    uvm_config_db#(virtual ubus_if)::set( null,
                        "*.ubus0*", "vif", ubus_if0);
    run_test();
  end
endmodule
```

Setting in the top removes hierarchy dependencies in the testbench

UVM Facilities for Customization

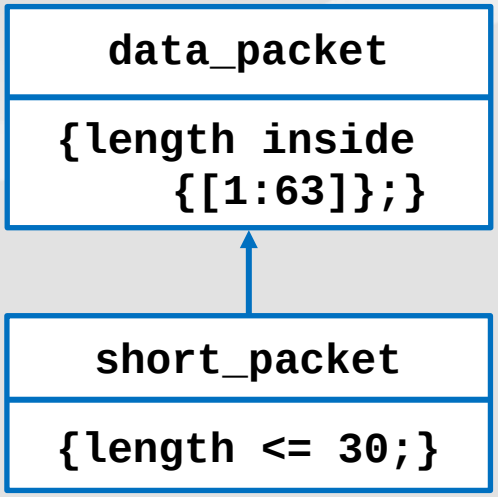
- Configuration
- Factory
- Messaging
- Callbacks
- Command-line processor

Overriding Components and Data

- **Verification IP should be designed such that its behavior can be modified without making major changes throughout the code**
  - For a specific test, you may want to further constrain the data items generated
  - You may want to replace the default driver implementation with a modified version.
- **Need this for exercising many different scenarios during testing**
  - Adding more attributes to a class and constraint layering
  - Changing procedural behavior
    - Different method for specific tests
    - Modify checking algorithm for error testing/recovery
- **The UVM Factory**
  - Is an implementation of the classic software factory design pattern
  - You design your code and defer to run-time the exact sub-type of the object to be allocated

Constraints Layering

- Advanced verification environments use randomness to explore unanticipated areas
- In a specific test, you might constrain your generated data to hit a corner case
- Object oriented programming allows for deriving a new sub-class from the parent class and adding the new constraints
- But how do we use these sub-classes?
- Let's look at an example …



Constraints Layering Problem

```
class data_packet extends uvm_sequence_item;
  rand bit [5:0] length;
  rand bit [7:0] data [];
  // default constraint
  constraint length_const { length >= 0; length < 64; }
  ...
endclass
```

```
// For short packet test
class short_packet extends data_packet;
  constraint short_length_const {length < 20;}
endclass
```

Extends from **data_packet** and adds another constraint

```
// Used in multiple locations throughout the design
data_packet   coll_packet = new("packet"); // MONITOR
...
data_packet   packet  = new("packet");    // SEQUENCE
...
data_packet   scbd_packet = new("packet"); // SCOREBOARD
```

Will need to change all `data_packet` to `short_packet`

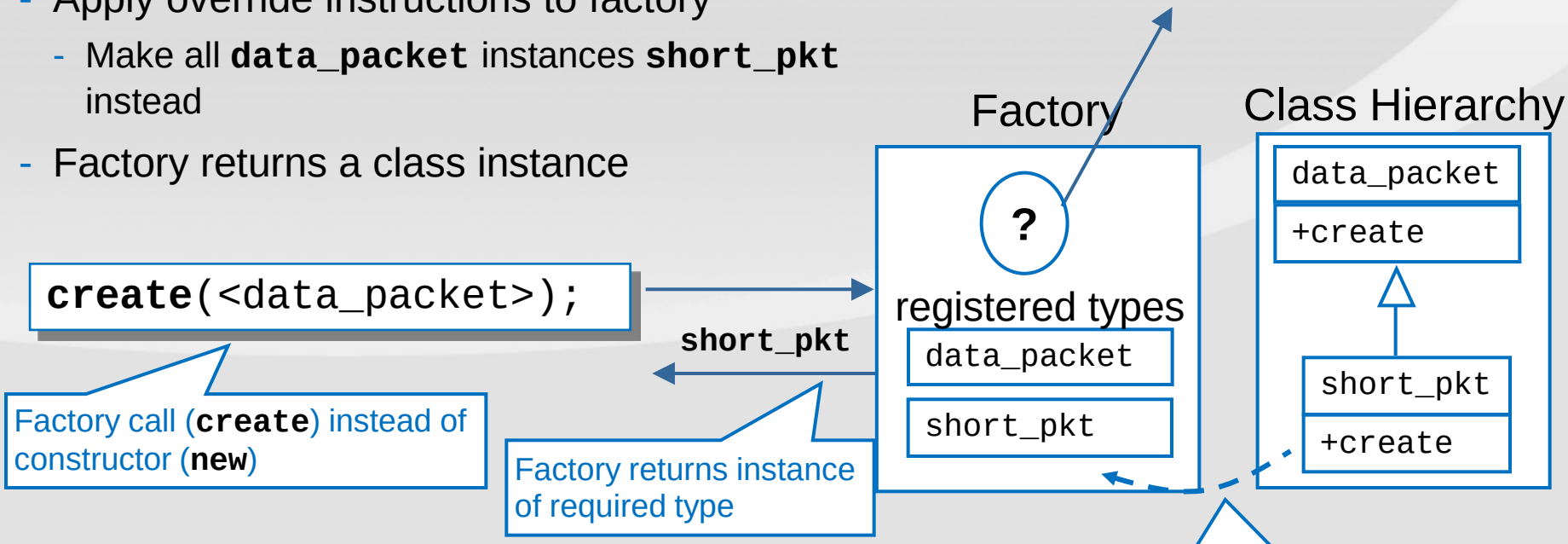
```
short_packet packet = new(...);
```

Solution: Factory

- **Central location to create class instances**
  - Each type is registered with factory
  - Instances created via factory **`create()`** method
    - Not via class constructor (**`new`**)
  - Apply override instructions to factory
    - Make all **`data_packet`** instances **`short_pkt`** instead
  - Factory returns a class instance

type and instance override list

from	to
data_packet	short_pkt
...	...



UVM Factory

- **Provides a mechanism for registering all uvm_object-based types**
  - Utility macros register the object to the factory and define create() method
    - **`uvm_object_utils(class_name)**
    - **`uvm_component_utils(class_name)**
- **Creates objects of registered types on demand**
  - **create()**
    - Creates uvm_sequence_item and its derivatives (data)
    - Creates uvm_component and its derivatives (components)
- **Provides powerful user-defined type and instance based overrides**
  - Auto instantiated at time 0
- **Overrides can be called at anytime.**
  - Subsequent calls to the factory for object creation reflect the override.

UVM Object Example: Factory create

```
class data_packet extends uvm_sequence_item;
  rand bit [5:0] length;
  rand bit [7:0] data[];
  `uvm_object_utils(data_packet)
  // default constraint
   constraint length_const { length >= 0; length < 100; }
endclass
```

adds data_packet to factory

```
class apb_monitor extends uvm_monitor;
  data_packet packet;
  task get_next_item();
    packet = new("packet");
    packet = data_packet::type_id::create("packet", this);
    ...
   endtask
  . . .
endclass
```

instead of **new**...

use **create**

- *All* object references should be created by calling factory **`create`** instead of constructor **`new()`**

Use create for All Components

```
class apb_agent extends uvm_agent;
  //uvm_active_passive_enum is_active = UVM_ACTIVE; //built-in agent field

  `uvm_component_utils_begin(data_agent)
    `uvm_field_enum(uvm_active_passive_enum, is_active, UVM_ALL_ON)
  `uvm_component_utils_end

  apb_sequencer  sequencer;
  apb_driver     driver;
  apb_monitor    monitor;

 virtual function void build_phase(uvm_phase phase);
   super.build_phase (phase);
    monitor = apb_monitor::type_id::create("monitor", this");
   if (is_active == UVM_ACTIVE) begin
     sequencer = apb_sequencer::type_id::create("sequencer", this");
     driver = apb_driver::type_id::create("driver", this");
    end
  endfunction

  . . .

endclass
```

Overriding Components and Objects

- **UVM provides an API for overriding the default data items and components in a testbench**

Replace ALL instances:

```
object::type_id::set_type_override(
                    derived_obj::get_type())
```

Example:

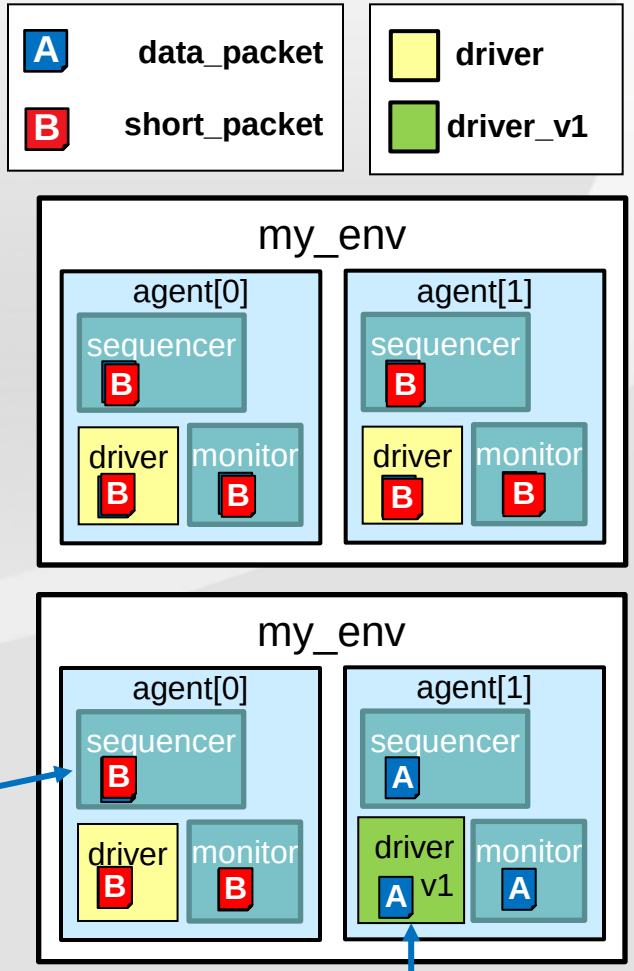
```
data_packet::type_id::set_type_override
                     (short_packet::get_type());
```

Replace specific instances:

```
object::type_id::set_inst_override
   (derived_obj::get_type(), "hierarchical_path");
```

Example:

```
data_packet::type_id::set_inst_override
   (short_packet::get_type(),
    "my_env.agent[0].*");
my_driver::type_id::set_inst_override
   (driver_v1::get_type(), "my_env.agent[1]");
```



UVM Facilities for Customization

- Configuration
- Factory
- Messaging
- Callbacks
- Command-line processor

UVM Messages for Debug and Errors

- **Debug of a large environment with many UVCs is critical**
  - Need to report: errors, progress, informational messages
- **Messages should be categorized via severity**
  - Fatal, Error, Warning, Info
- **Actions can be associated with messages**
  - Stop simulation on fatal or after 4 errors
  - Summarize number of messages reported
- **Requirements**
  - Uniformity of output format, turning messages on and off
  - User control over output format, message actions and filtering by severity and hierarchy
  - Minimal performance overhead

23

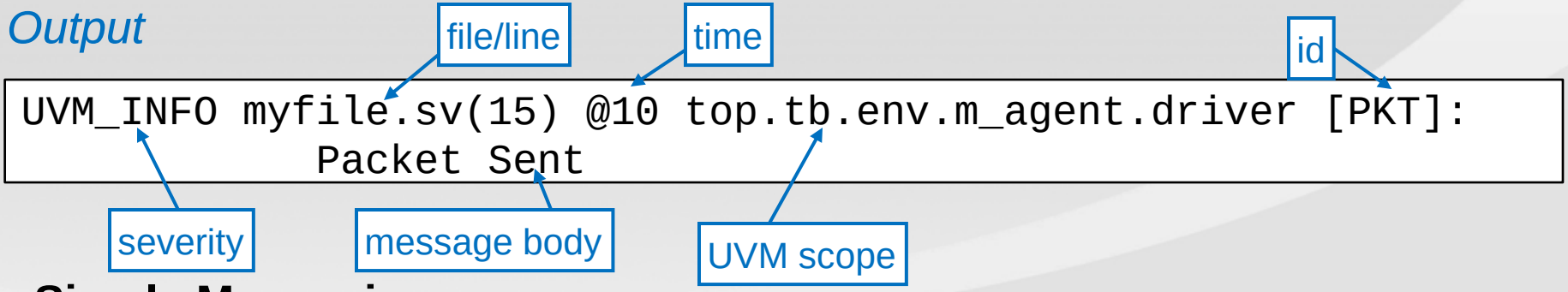
UVM Messaging Facility

Messages print trace information with advantages over $display:

- Aware of its hierarchy/scope in testbench
- Allows filtering based on **hierarchy**, **verbosity**, and **time**

```
`uvm_info("PKT", "Packet Sent“, UVM_LOW)
```

Output



- **Simple Messaging:**
  - `uvm_*(string id, string message, <verbosity>)
    - Where * (severity) is one of fatal, error, warning, info
    - <verbosity> is only valid for `uvm_info

UVM Facilities for Customization

- Configuration
- Factory
- Messaging
- Callbacks
- Command-line processor

Extensions Using Callbacks

- **Like the factory, callbacks are a way to affect an existing component from outside**
- **SystemVerilog and UVM include some built-in callbacks**
  - e.g. post_randomize(), sequences: pre_body(), post_body.
- **Callbacks require the developer to predict the extension location(s) and create the proper hook(s)**
- **Callback advantages:**
  - They do not require inheritance
  - Multiple callbacks can be combined

Callbacks: Developer Code

```
virtual class inj_err_cbs extends uvm_callback;
  function new(string name=inj_err_cbs");
    super.new(name);
  endfunction
  pure virtual function void inject_err(data_packet pkt);
endclass
class driver extends uvm_component;
  `uvm_register_cb(driver, inj_err_cbs)
  ...
  task drive_transfer(data_packet pkt);
    `uvm_do_callbacks(driver, inj_err_cbs, inject_err(pkt))
    ...
  endtask : drive_transfer
endclass

// For convenience, also create a callback typedef.
  typedef uvm_callbacks#(driver, inj_err_cbs) inj_err_cb;
```

Callback function is an extension of uvm_callback

Register the callback

Call the callback at the appropriate time

typedef simplifies usage

Callbacks: User Code

```
class my_driver_cb extends inj_err_cbs;
  function new(string name="my_driver_cb");
    super.new(name);
  endfunction
  virtual function void inject_err(data_packet pkt);
    `uvm_info("DRIVER_CB", "inject_err() called", UVM_LOW)
  endfunction
endclass

my_driver_cb cb = new;
. . .
inj_err_cb::add(top.tb.env.agent.driver, cb)
inj_err_cb::add(null, cb);
```

Define desired logic for the callback implementation

Attaches the callback to a specific instance or to all instances of the type. A null handle specifies a type-wide callback.

Callback implementations are usually created by the test writer and applied to specific components in the testbench or tests

UVM Report Catcher Callback

Goal: Message Manipulation

UVM built-in callback

```
class my_catcher extends uvm_report_catcher;
  virtual function action_e catch();
    if(get_severity()==UVM_ERROR && get_id()=="MYID") begin
      set_severity(UVM_INFO);
      set_action(get_action() = UVM_COUNT);
    end
    return THROW; // throw the message for more manipulation
                  // or catch it to avoid further processing
  endfunction
endclass
```

This example demotes MYID to be an info message

```
// In testbench run_phase
my_catcher catcher = new;
uvm_report_cb::add(null,catcher);
`uvm_error("MYID", "This one should be demoted")
#100;
catcher.callback_mode(0);  //disable the catcher
`uvm_error("MYID", "This one should not be demoted")
```

Disable callback using the built-in callback_mode() method

UVM Facilities for Customization

- Configuration
- Factory
- Messaging
- Callbacks
- Command-line processor

Command-line Processor Class

- **Provides a vendor independent general interface to the command line arguments**
  - Use command-line options to further customize your test or testbench
  - This is above the capability of $value$plusargs()
- **Supported categories:**
  - Basic Arguments and values
    - get_args(), get_plusargs(), get_arg_matches(), get_uvmargs()
  - Tool information
    - get_tool_name(), get_tool_version()
  - Built-in UVM aware Command Line arguments
    - Set UVM variables: +UVM_VERBOSITY, +UVM_TESTNAME
    - Override settings: +uvm_set_config_int, +uvm_set_config_string, +uvm_set_severity, +uvm_set_type_override, +uvm_set_severity

Built-In Command-Line Examples

- **+uvm_set_config_int=<comp>,<field>,<value>**

```
+uvm_set_config_int="*,coverage_enable,1"
+uvm_set_config_int=uvm_test_top.my_tb.ubus0.bus_monitor,coverage_enable,1
```

Use “” if wildcards are used

- **+uvm_set_severity=<comp>,<id>,<curr_severity>,<new_severity>**

```
+uvm_set_severity=“*, NOVIF,UVM_ERROR,UVM_WARNING”
```

- **+uvm_set_action=<comp>,<id>,<severity>,<action>**

```
+uvm_set_action=“*slave[0].monitor,_ALL_,UVM_NO_ACTION"
+uvm_set_action=“*.bus_monitor,NOVIF,UVM_FATAL,UVM_STOP”
```

- **+uvm_set_inst_override=<reg_type>,<override_type>,<full_path>**
  +uvm_set_type_override=<reg_type>,<override_type>[,<replace>]

```
+uvm_set_type_override=data_packet,short_packet
```

Note: This is the string-based factory. Parameterized types cannot be overridden

Command-line Processor Example

```
class test extends uvm_test;
  ...
  function void start_of_simulation();
    uvm_cmdline_processor clp;
    string arg_values[$];
    string tool, version;
    clp = uvm_cmdline_processor::get_inst();
    tool = clp.get_tool_name();
    version = clp.get_tool_version();
    `uvm_info("MYINFO1", ("Tool: %s, Version : %s", tool, version,
                UVM_LOW)

    void'(clp.get_arg_values("+foo=", arg_values));
    `uvm_info("MYINFO1","arg_values size : %0d", arg_values.size(),
                UVM_LOW))
    for(int i = 0; i < arg_values.size(); i++)
       uvm_info("MYINFO1", "arg_values[%0d]: %0s", i, arg_values[i],
                UVM_LOW))
  endfunction
endclass
```

Fetching the command line processor singleton class

Use the class methods

Get argument values

Summary – Customizing with UVM

- **This section introduced various techniques for customizing your verification environment with UVM**

  Configuration Mechanism – Configure components and data objects from higher-levels of hierarchy

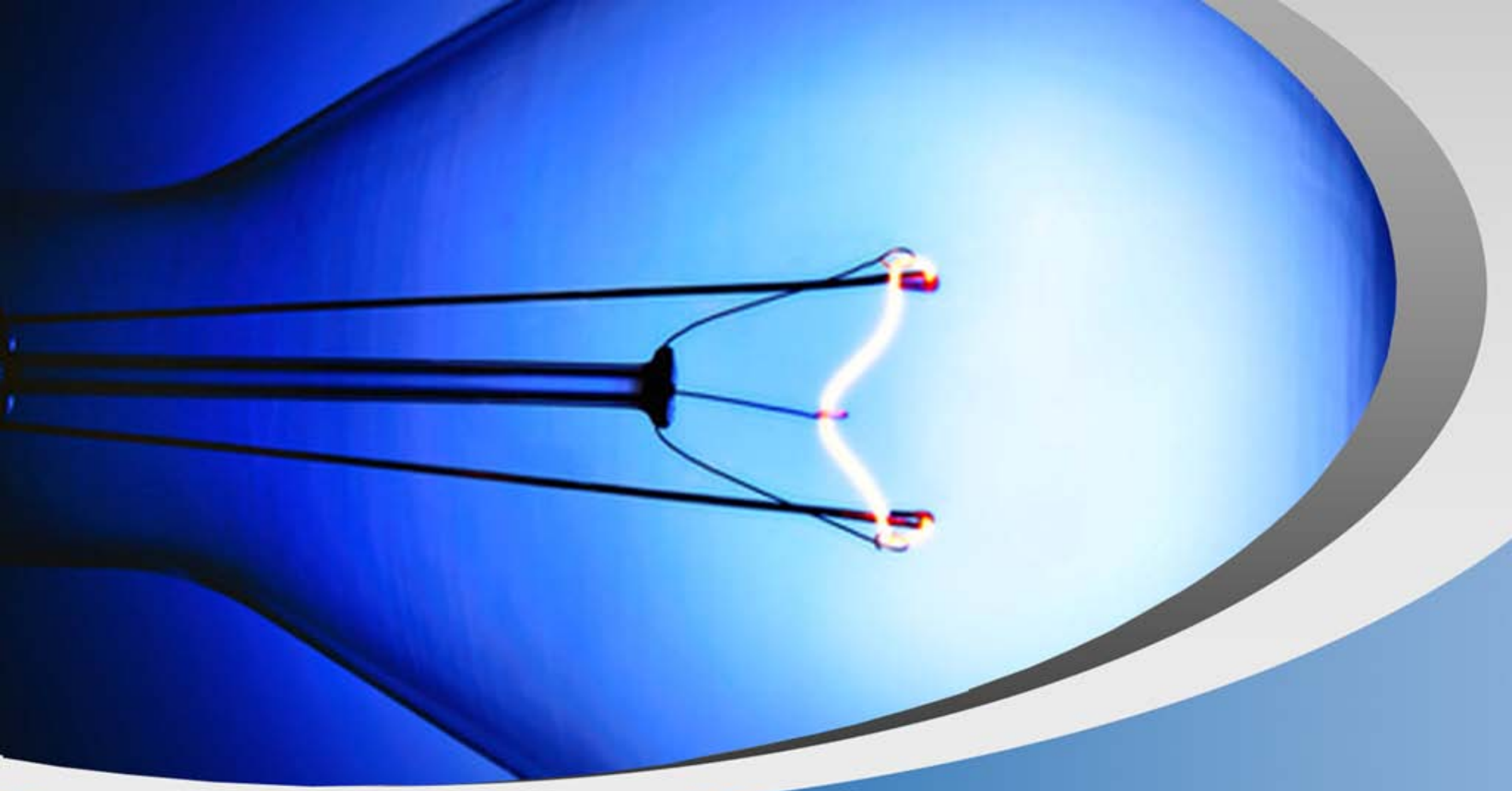
  Factory – Override components and data objects for a specific test

  Messaging – Display and control debug information

  Callbacks – Add functionality to existing components at pre-determined points

  Command-line processor – Provide flexible, simulator-independent, run-time control

- **UVM provides these built-in capabilities to make reuse possible**



UVM: Ready, Set, Deploy!

UVM

DVCon 2012
Design & Verification Conference & Exhibition

accellera
SYSTEMS INITIATIVE

Register Modeling in UVM

Adiel Khan

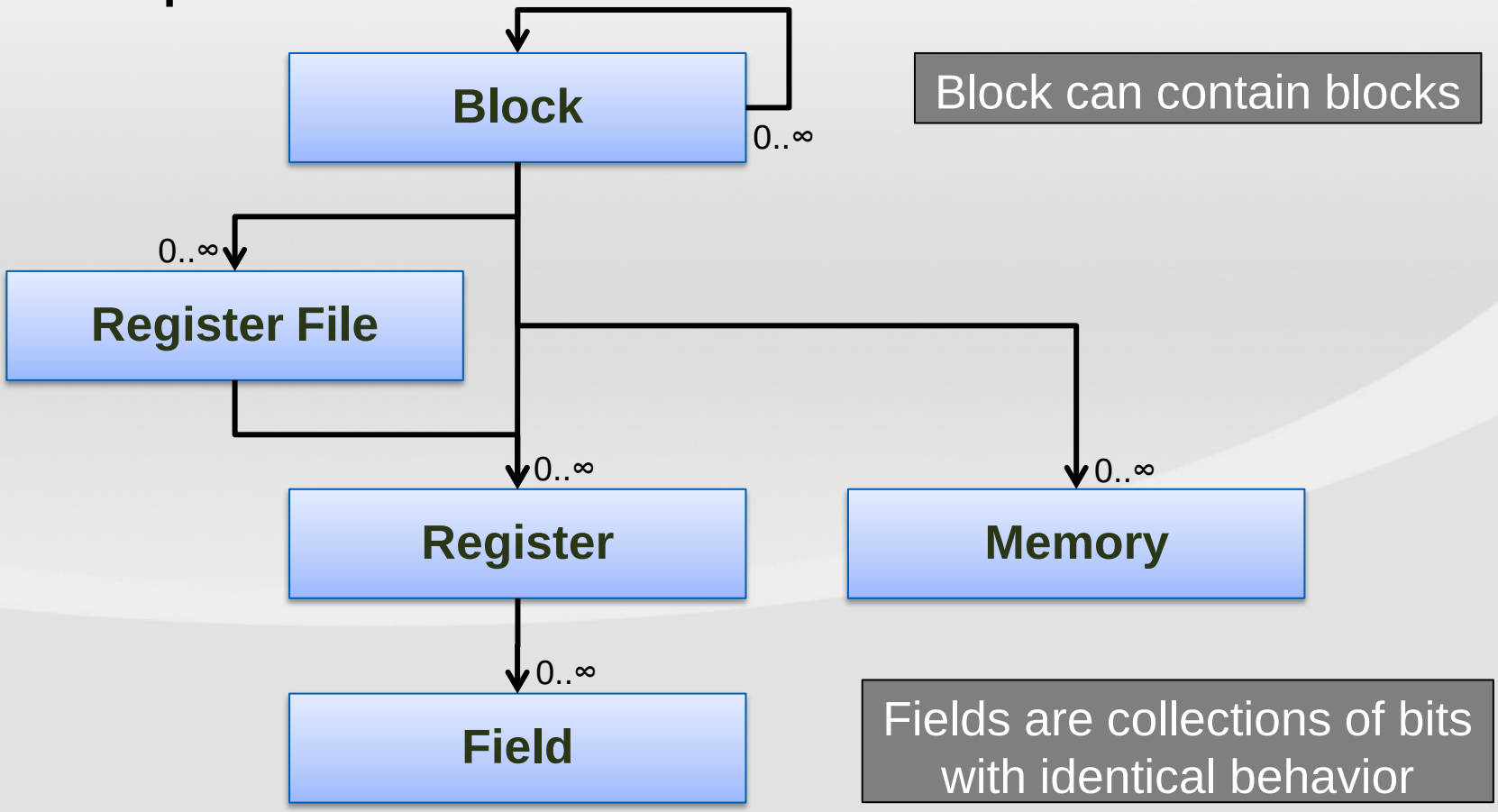
Verification Expert

Outline

- Register Model Elements
- Address Maps vs Blocks
- Access Granularity
- Coverage Models

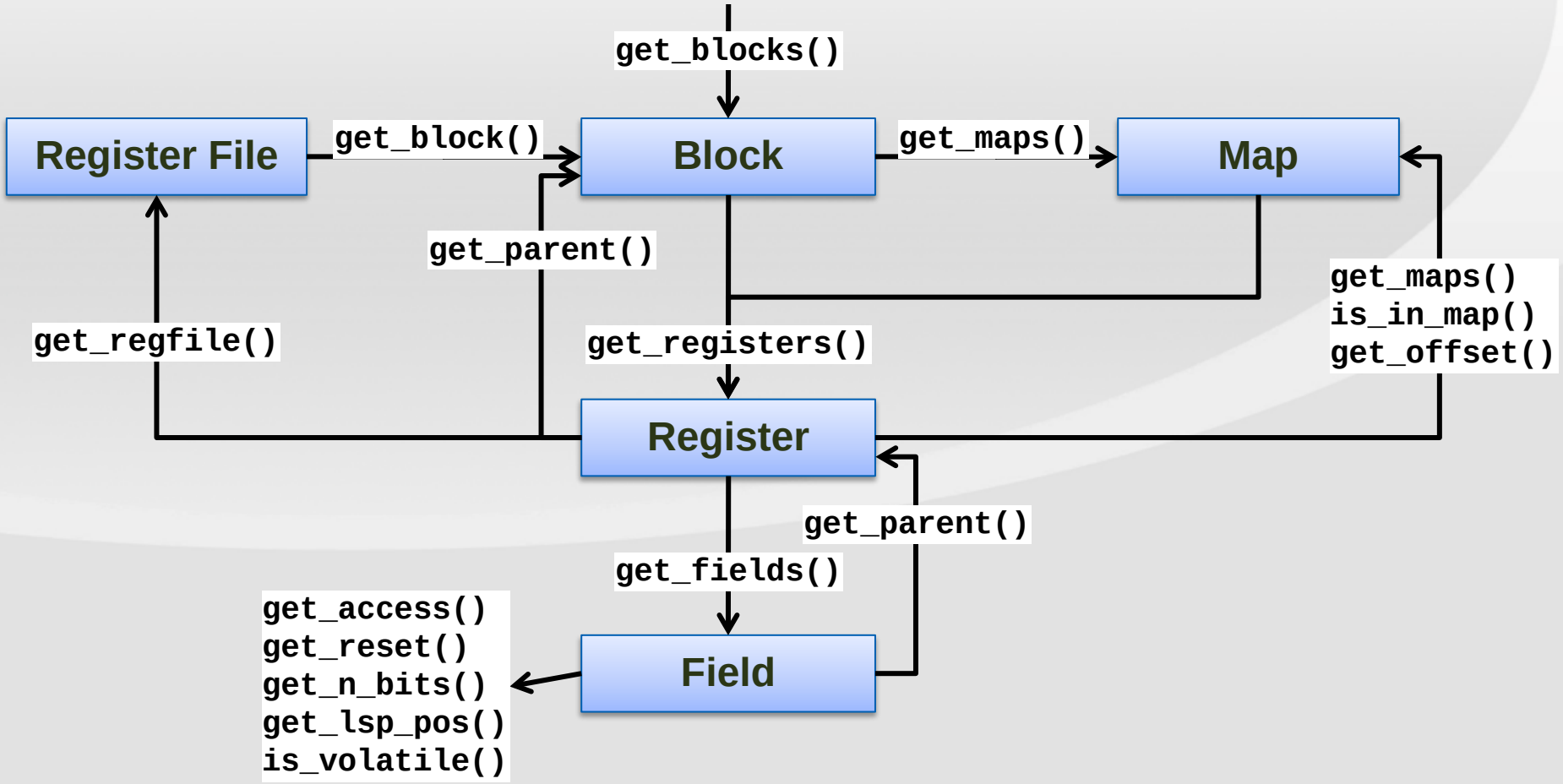
Register Model Hierarchy

- **Composed of classes**

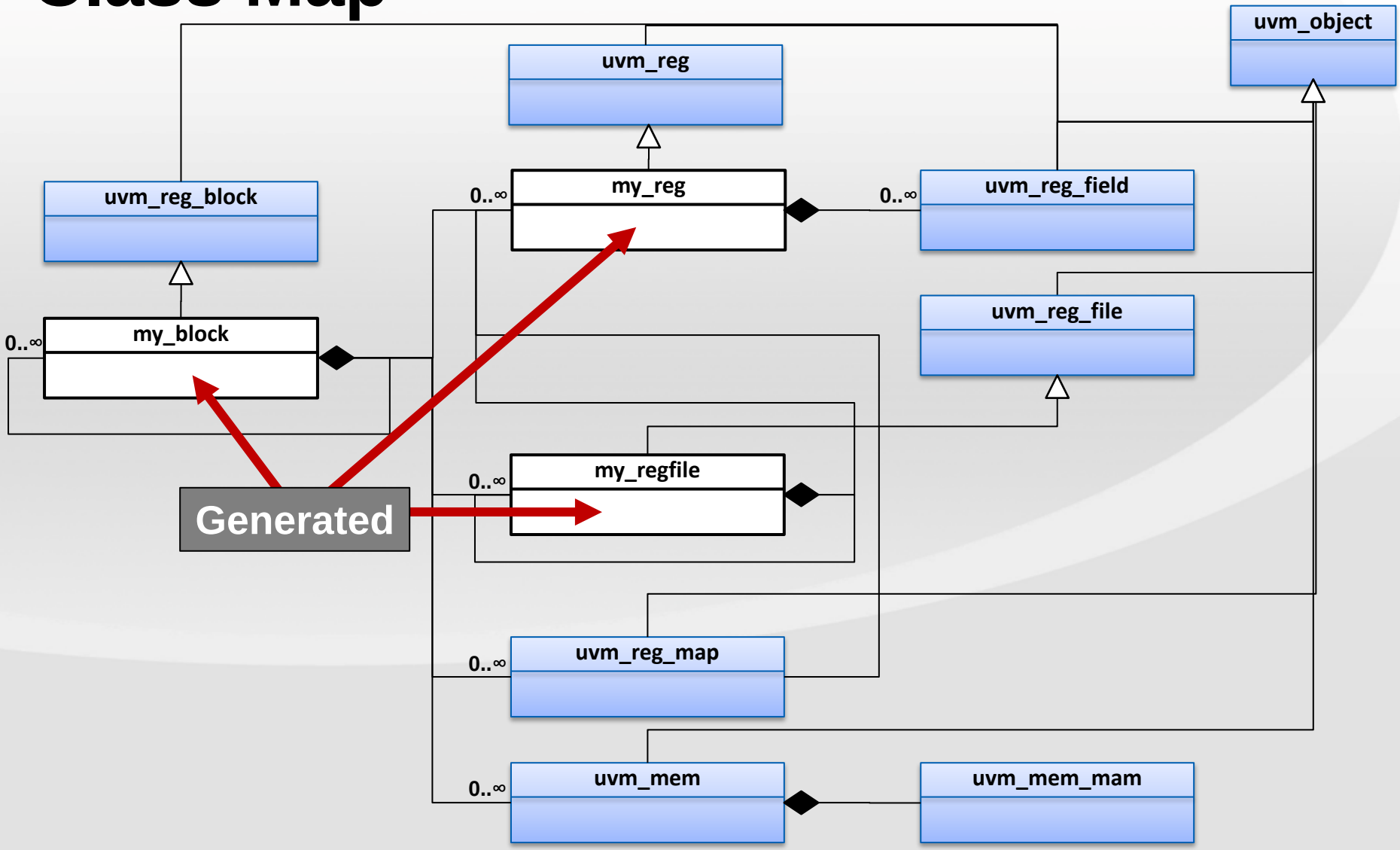


Introspection

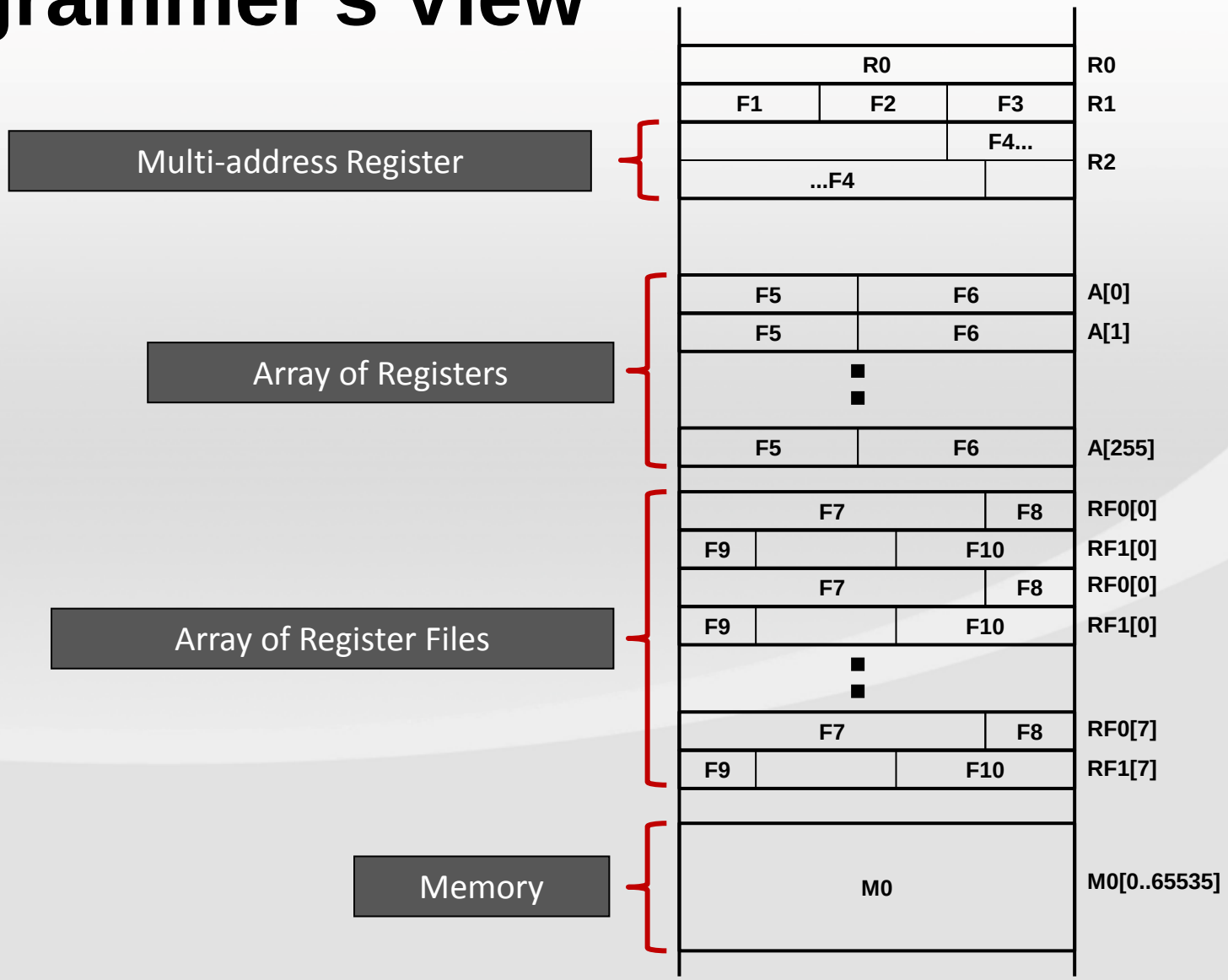
- Rich introspection API



Class Map


uvm_object
uvm_reg
uvm_reg_block
0..∞
my_reg
0..∞
uvm_reg_field
uvm_reg_file
0..∞
my_block
0..∞
my_regfile
Generated
0..∞
uvm_reg_map
0..∞
uvm_mem
uvm_mem_mam

Programmer's View

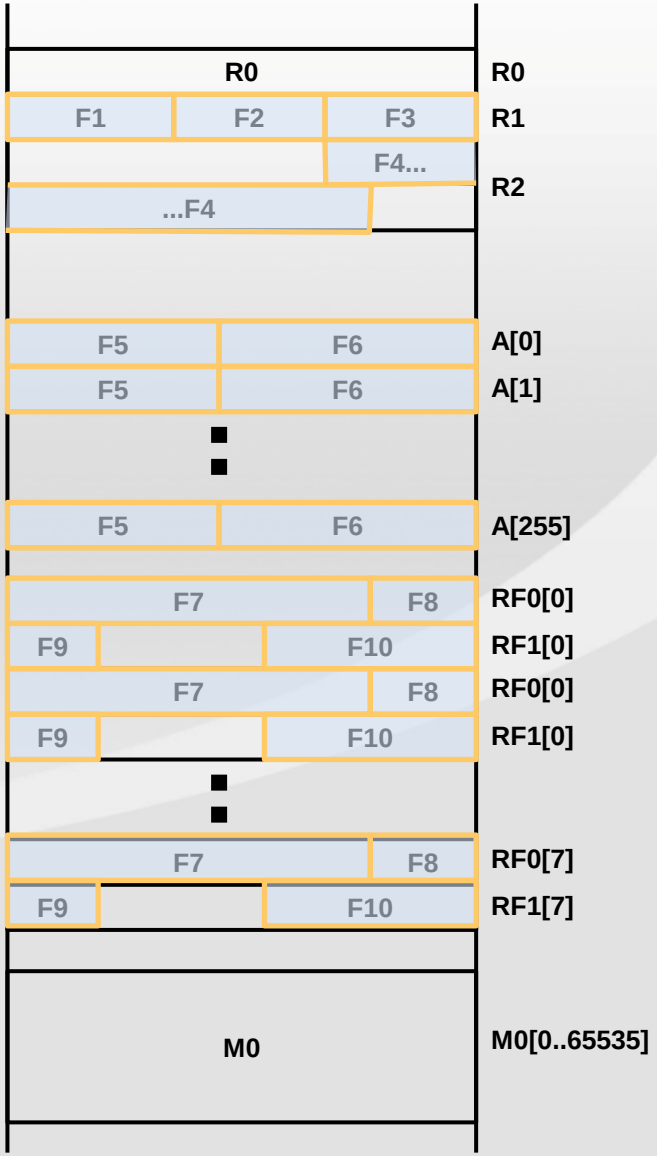


Class View

- **One class per field**

```
class uvm_reg_field extends uvm_object;
```

  - Name
  - Position
  - Access Policy
  - Read/write methods



Class View

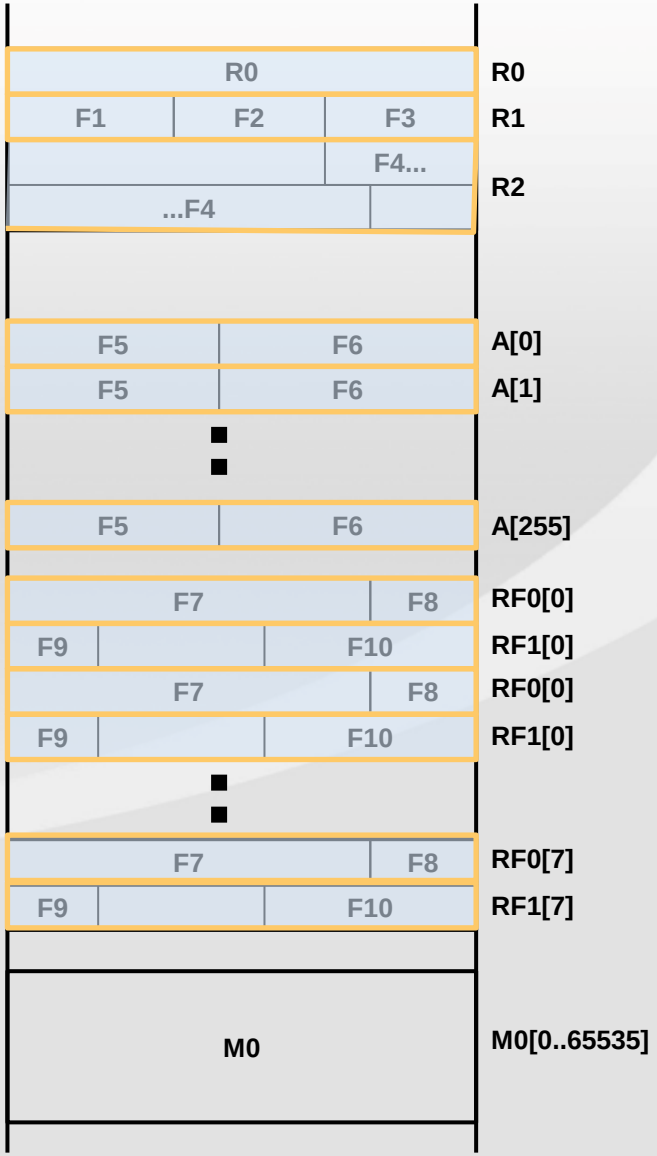
- **One class per field**
- **One class per register**

```
class R1_reg extends uvm_reg;
   uvm_reg_field F1;
   uvm_reg_field F2;
   uvm_reg_field F3;
endclass
```

Generated

Make sure names are different from base class methods

- Name
- Address
- Contained fields
- Read/Write methods

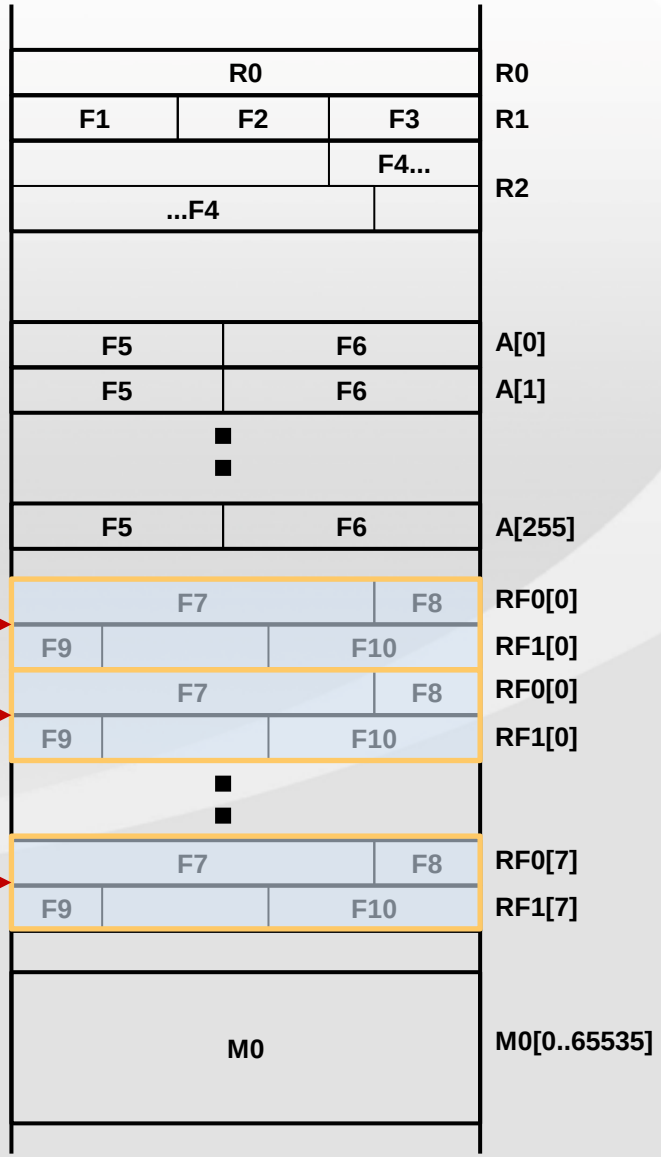


Class View

- **One class per field**
- **One class per register**
- **One class per register file**

```
class RF_rfile extends uvm_object;
   RF0_reg R0;
   RF1_reg R1;
endclass
```

Generated



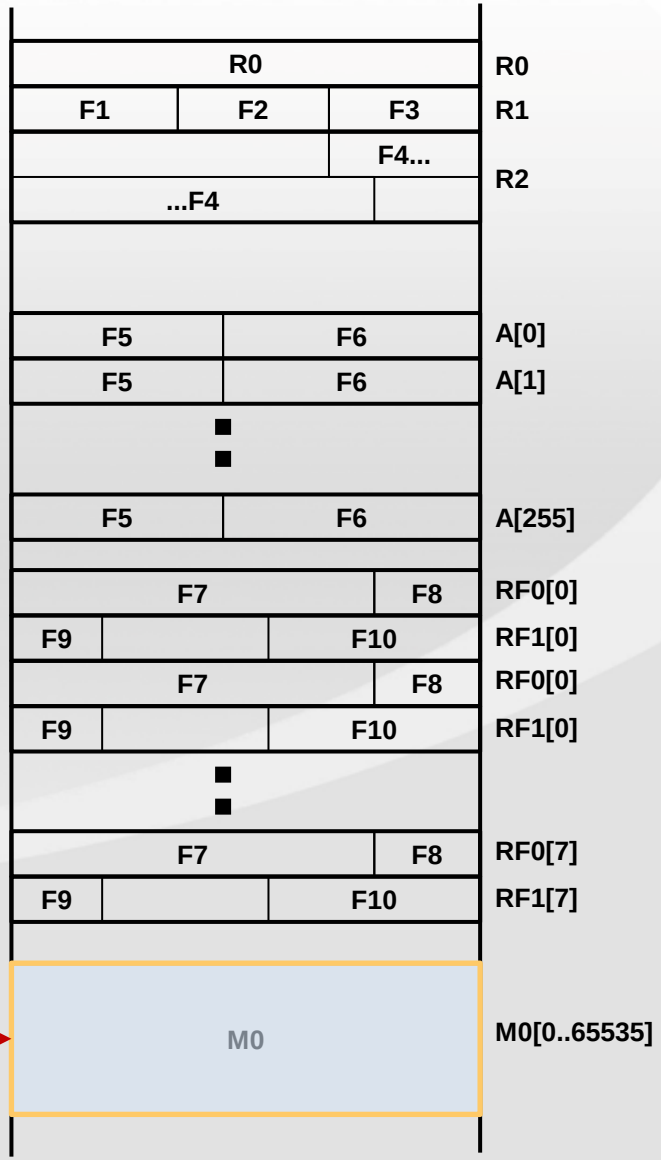
- Contained registers and register files
  - May be arrays

Class View

- One class per field
- One class per register
- One class per register file
- One class per memory

```
class uvm_mem extends uvm_object;
```

  - Name, size, address
  - Read/Write methods



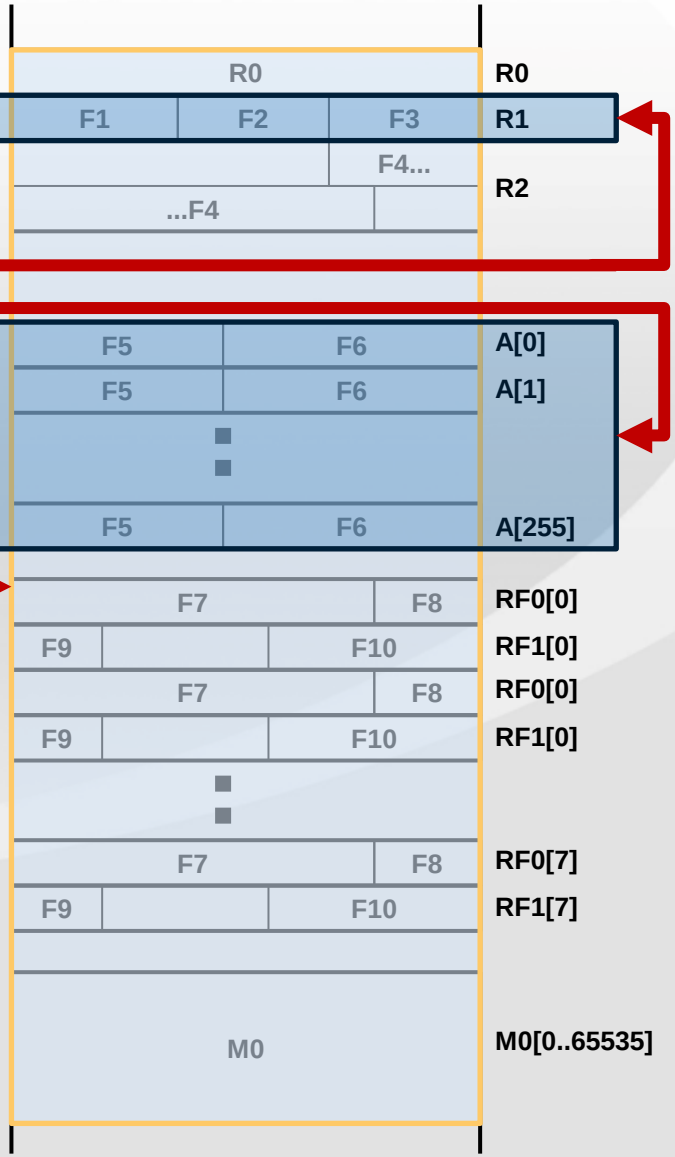
Class View

Make sure names are different from base class methods

- **One class per block**

```
class B_blk extends uvm_reg_block;
   R0_reg          R0;
   R1_reg          R1;
   R2_reg          R2;
   A_reg           A[256];
   RF_rfile        RF[8];
   uvm_reg_mem     M0;
endclass
```

Generated



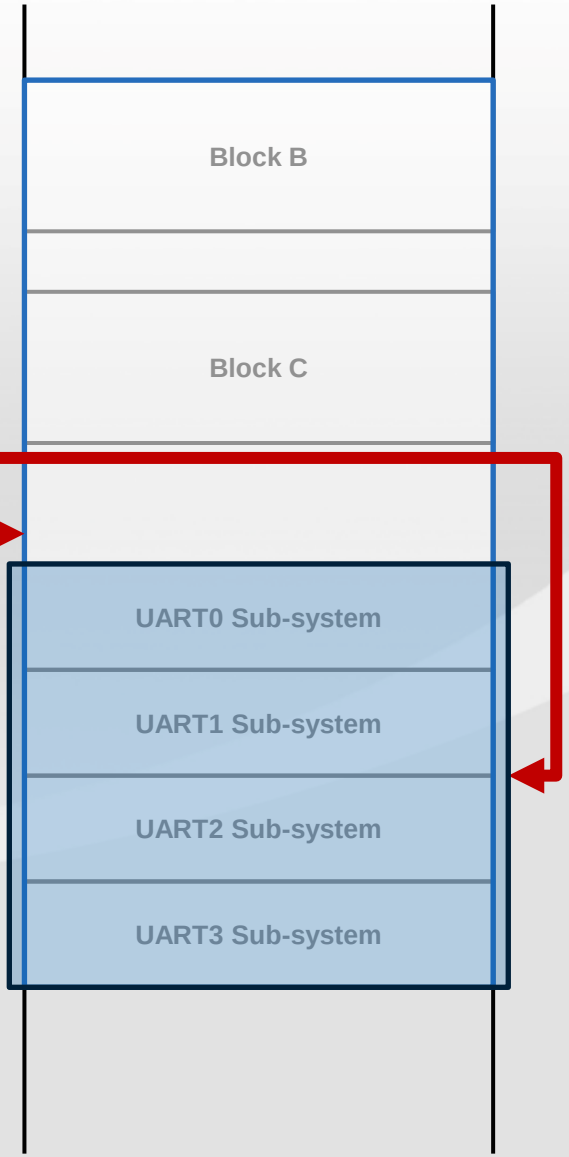
  - Name, Base address
  - Contained registers, register files, memories
    - May be arrays
    - Optional: contained fields

Class View

- **One class per system**

```
class S_blk extends uvm_reg_block;
   B_blk    B;
   C_blk    C;
   UART_blk IO[4];
endclass
```

Generated



  - Contained registers, register files, memories and sub-blocks
    - May be arrays

API View

- Methods in relevant class instance

```
blk.R0.get_full_name()
```

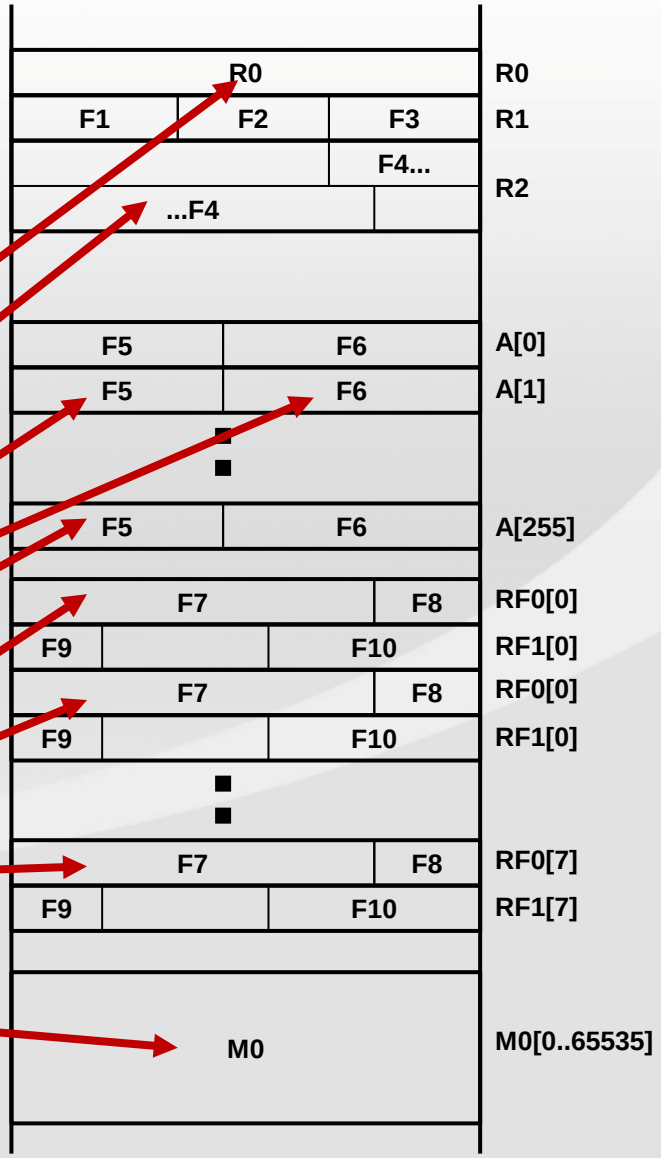
```
blk.F4.read(...)
```

```
blk.A[1].write(...)
```

```
blk.A[255].F5.write(...)
```

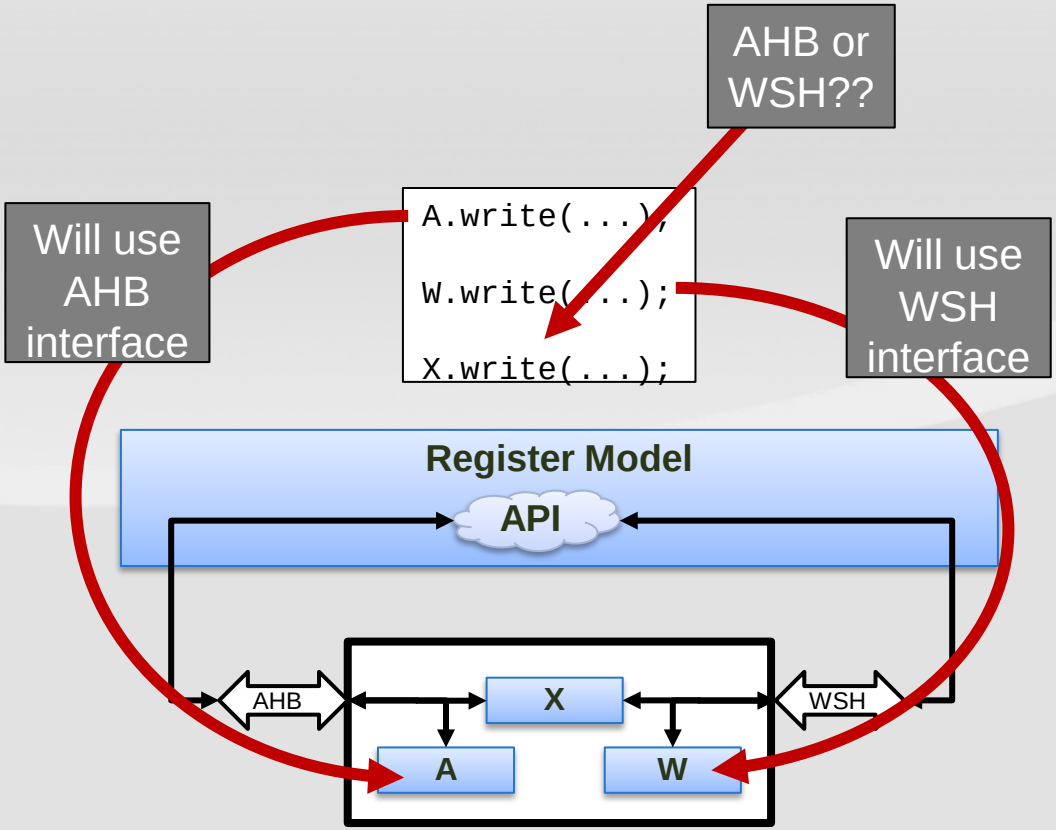
```
foreach (blk.RF[i]) begin
   blk.RF[i].R0.F7.read(..._);
end
```

```
blk.M0.read(...)
```



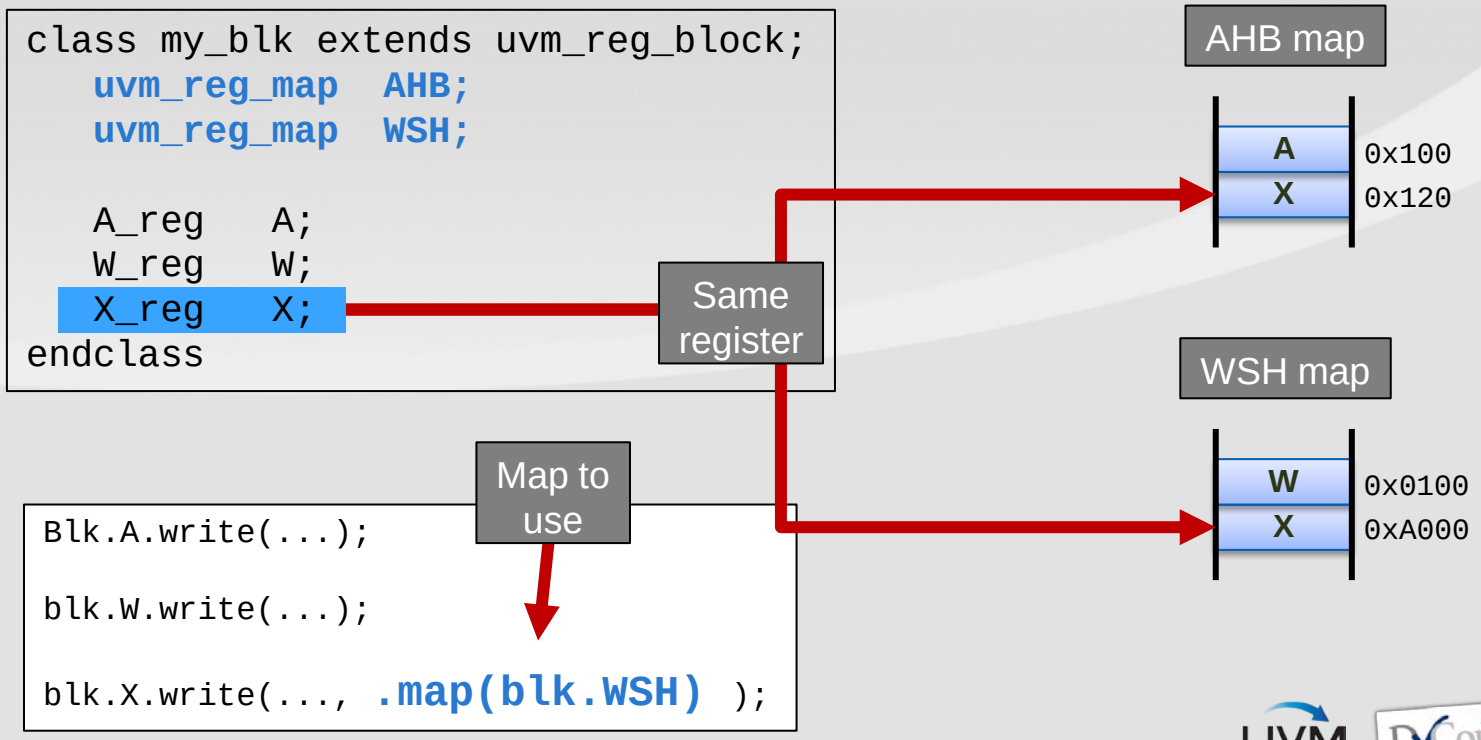
Address Maps

- DUT may have multiple physical interfaces
- Registers may be shared across multiple physical interfaces



Address Maps

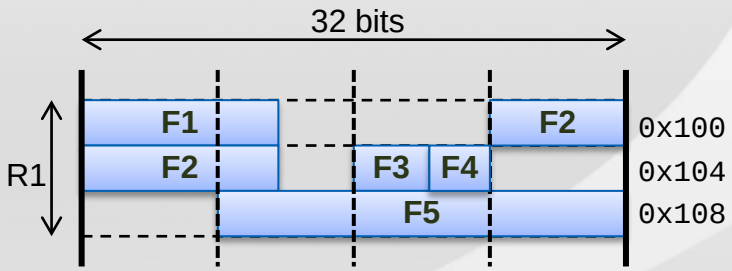
- **Registers collected in *address maps***
- **One *map* per physical interface**
  - "default_map" is only one

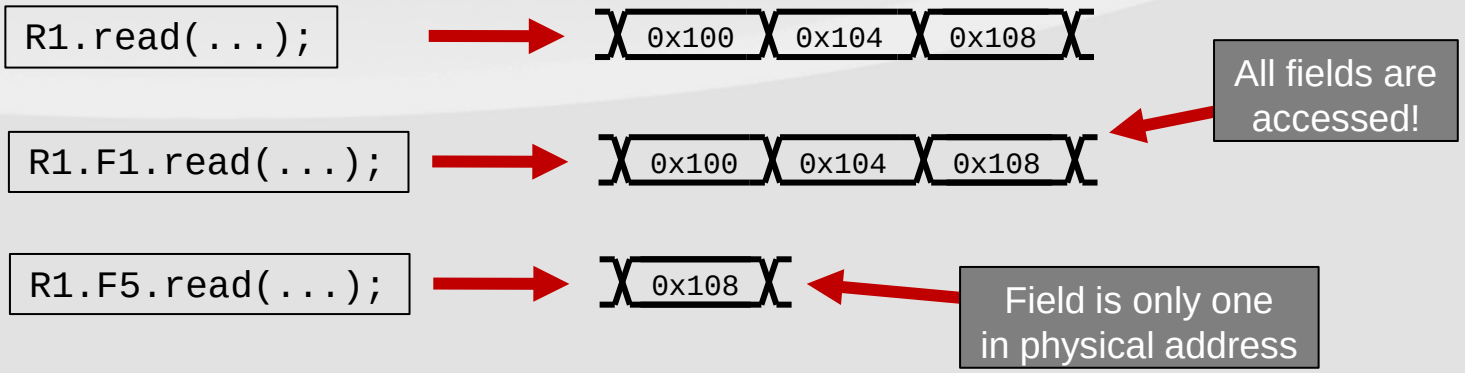


Access Granularity

- **By default, always access entire register**

```
class R1_reg extends uvm_reg;
   uvm_reg_field F1;
   uvm_reg_field F2;
   uvm_reg_field F3;
   uvm_reg_field F4;
   uvm_reg_field F5;
endclass
```





Access Granularity

- **IF protocol supports byte-enabling…**
  - Can access individual fields if sole occupant

```
R1.F1.read(...);
```

ADDR 0x100
BE 0b1100

```
R1.F2.read(...);
```

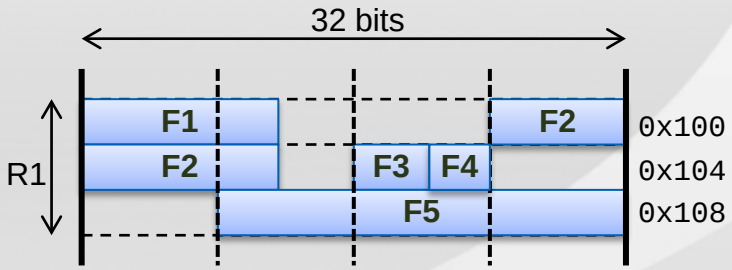
ADDR 0x100 0x104
BE 0b0001 0b1100

```
R1.F4.read(...);
```

ADDR 0x100 0x104 0x108
BE 0b1111 0b1111 0b1111

```
R1.F5.read(...);
```

ADDR 0x108
BE 0b0111



Reads entire register because F3 & F4 share byte lane

Coverage Models

- **Register models *may* contain coverage models**
  - Up to the generator
- **Not instantiated by default**
  - Can be large. Instantiate only when needed.
  - To enable:

```
uvm_reg::include_coverage("*", UVM_CVR_ALL);
```

All coverage models

In all blocks and registers

- **Not collected by default**
  - To recursively enable

```
blk.set_coverage(UVM_CVR_ALL);
```

All coverage models in block

Coverage Models

- **Pre-defined coverage models**
  - Register bits
  - Address map
  - Field values
- **Details of coverage points, bins left to generator**
- **Generator may define more models**

- **Stay Tuned for user level coverage experience**
  - User Experience with UVM: Getting Started as a Beginner

UVM Register Library

INTEGRATION WITH DUT

Outline

- Integration Steps
- Instantiating a Register Model
- Bus Adapter
- Implicit & Explicit Monitoring
- Predefined functionality Methods & Sequences
- Basic Read & Write API
- User defined Sequences

Necessary Steps

- **Add register model to environment**
- **Create register-to-bus adapter class**
- **Smoke test using a predefined sequence**

Think ahead:

- Allow for vertical reuse
  - Register model may be part of a larger model

Instantiating a Register Model

- **In the DUT's verification environment:**

```
class my_env extends uvm_env;

   my_model  regmodel;
   bus_agent agt;

   function void build_phase(uvm_phase phase);
      agt = bus_agent::type_id::create(…);
      if (regmodel == null) begin
         regmodel = my_model::type_id::create(…);
         regmodel.build();
         regmodel.lock_model();
      end
   endfunction

   function void connect_phase(uvm_phase phase);
      if (regmodel.get_parent() == null) begin
         reg_to_bus_adapter adapter = new();
         regmodel.default_map.set_sequencer(agt.m_sequencer, adapter);
         regmodel.set_auto_predict(1);
      end
```

Build component as usual

Use upper-level regmodel if present

Create adapter and give it to address-map

Register-to-Bus Adapter

- **Register layer uses abstract transactions**
- **Mandatory functions convert between register and physical transactions**

```
class reg_to_i2c_adapter extends uvm_reg_adapter;
   `uvm_object_utils(reg_to_i2c_adapter)
   function uvm_sequence_item reg2bus(uvm_reg_bus_op rw);
      i2c_trans i2c = i2c_trans::type_id::create("i2c_item");
      i2c.addr = rw.addr[7:0];
      ...
      return i2c;
  endfunction
  function void bus2reg(uvm_sequence_item bus_item, uvm_reg_bus_op rw);
      ...
  endfunction
endclass
```

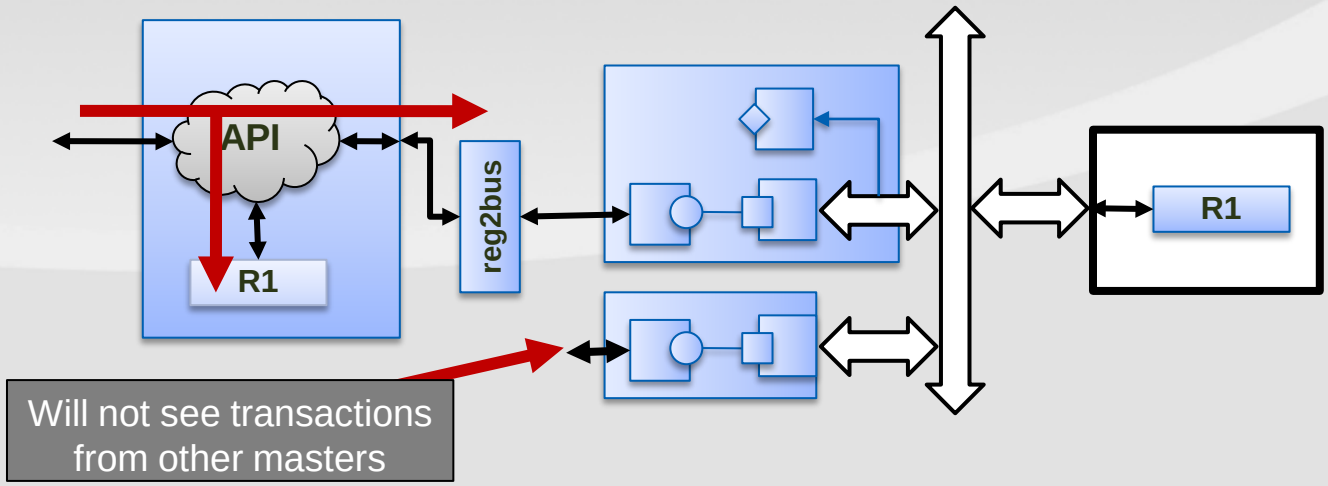
Populate bus transaction object from *rw*

Check *bus_item* is correct type, populate *rw* from its contents

Implicit Monitoring

- **Register model predicts value internally**
- **OFF by default**

```
reg_to_bus_adapter adapter = new();
regmodel.default_map.set_sequencer(agt.m_sequencer, adapter);
regmodel.set_auto_predict(1);
```

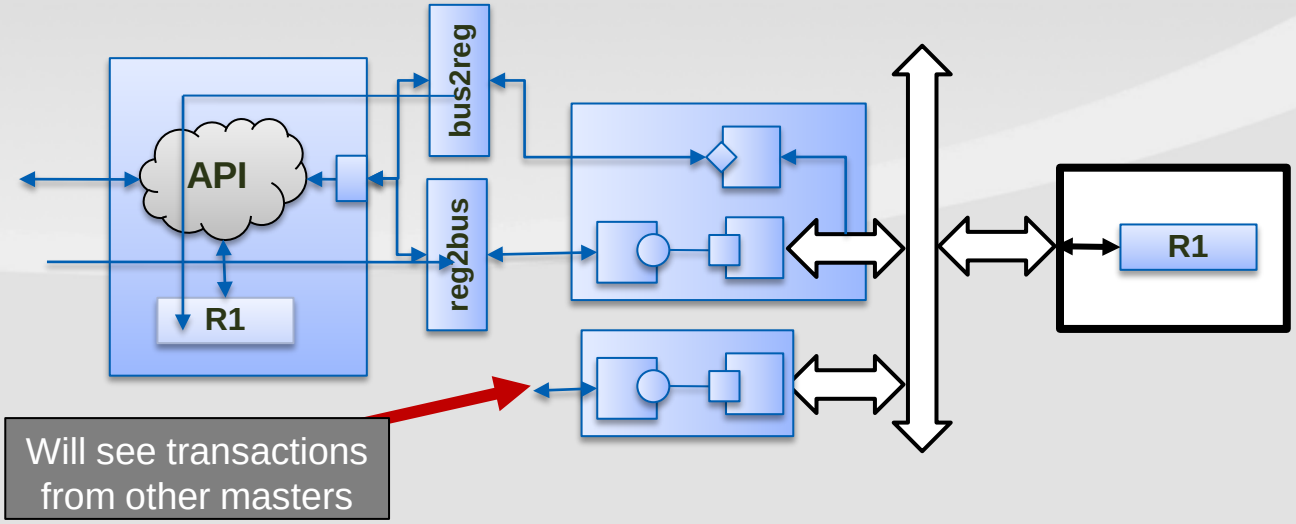


Explicit Monitoring

Stay tuned later today
User Experience with UVM:
"Getting Started as a Beginner"

- **Register model predicts values based on observed transactions**

```
uvm_reg_predictor#(i2c_trans) i2c_predictor;
reg_to_bus_adapter adapter = new();
regmodel.default_map.set_sequencer(agt.m_sequencer, adapter);
i2c_predictor.map = regmodel.default_map;
12c_predictor.adapter = adapter;
agt.mon.ap.connect(i2c_predictor.bus_in);
```



Resetting the Register Model

- **Reset the register model during reset phase**

```
class my_test extends uvm_env;
   my_model  regmodel;
   ...
   task reset_phase(uvm_phase phase);
      phase.raise_objection(this, “Resetting DUT”);

      vif.rstn = 1’b0;
      repeat (10) @ (posedge vif.clk);
      vif.rstn = 1’b1;

      regmodel.reset();

      phase.drop_objection(this);
   endtask
endclass
```

Smoke Test

- **Use a pre-defined register layer sequence**
  - Read all reset values

```
class smoke_test extends uvm_test;
   my_env env;
   ...
   task main_phase(uvm_phase phase);
      uvm_reg_hw_reset_test test_seq = new();

      phase.raise_objection(this, “Running hw_reset test”);

      test_seq.model = env.regmodel;
      test_seq.start(null);

      phase.drop_objection(this);
   endtask
endclass
```

Basic Reading and Writing API

- **Specify target register by hierarchical reference in register model**
  - Compile-time checking

```
blk.blk[2].regfile[4].reg.fld
```

By-name API also available

- **Use *read()* and *write()* method on**
  - Register
  - Field
  - Memory

```
blk.blk[2].regfile[4].reg.fld.read(...);
blk.mem.write(...);
```

User defined Register Sequence

```
class my_blk_config_sequence extends uvm_sequence;
   `uvm_object_utils(my_blk_config_sequence)

   my_blk regmodel;

   task body();
      regmodel.R1.write(...);
      foreach (my_blk.A[i]) begin
         regmodel.A[i].write(...);
      end
      for (int i = 0; i < regmodel.mem.get_size(), i++) begin
         regmodel.mem.write(..., i, ...);
      end
   endtask
endclass
```

Perform read/write in *body()* task

Pass root register model as property

Register Sequence

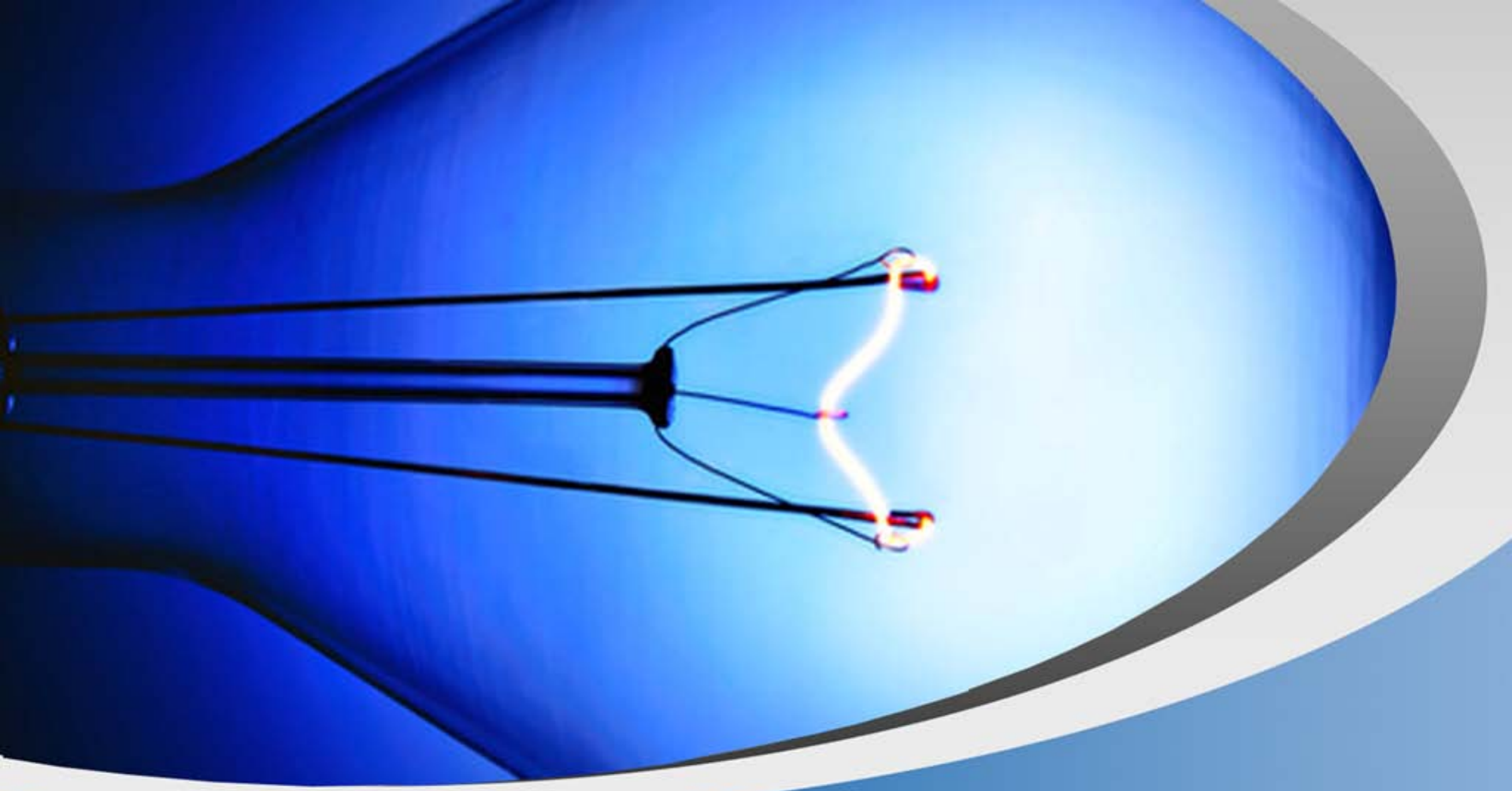
- Create the sequence
- Assign root register model to property
- Start sequence

```
my_blk_config_sequence seq;

seq = my_blk_config_sequence::type_id::create(“seq”,,
                                              get_full_name());
seq.regmodel = regmodel.blk1;
seq.start(null);
```

Register API Functionality Overview

- **Randomization of registers**
- **Update & mirror**
- **Read/Write return status**
- **peek/poke DUT**
- **get/set Mirror**
- **Customization API**
- **Generator hooks**
- **X handling**
- **Factory Replacements**

- **Callbacks**
- **Concurrent Accesses**
- **Frontdoor (over the bus access)**
  - User-defined frontdoor
    - Non-linear addressing: Different endianness:Additional transactions
- **Backdoor (snoop DUT access)**
  - User-defined backdoor
    - Instead of default VPI routine:Vendor-provided backdoor API:Add/check ECC:Hardware encryption



UVM: Ready, Set, Deploy!

UVM

DVCon 2012
Design & Verification Conference & Exhibition

accellera
SYSTEMS INITIATIVE

Getting Started with UVM

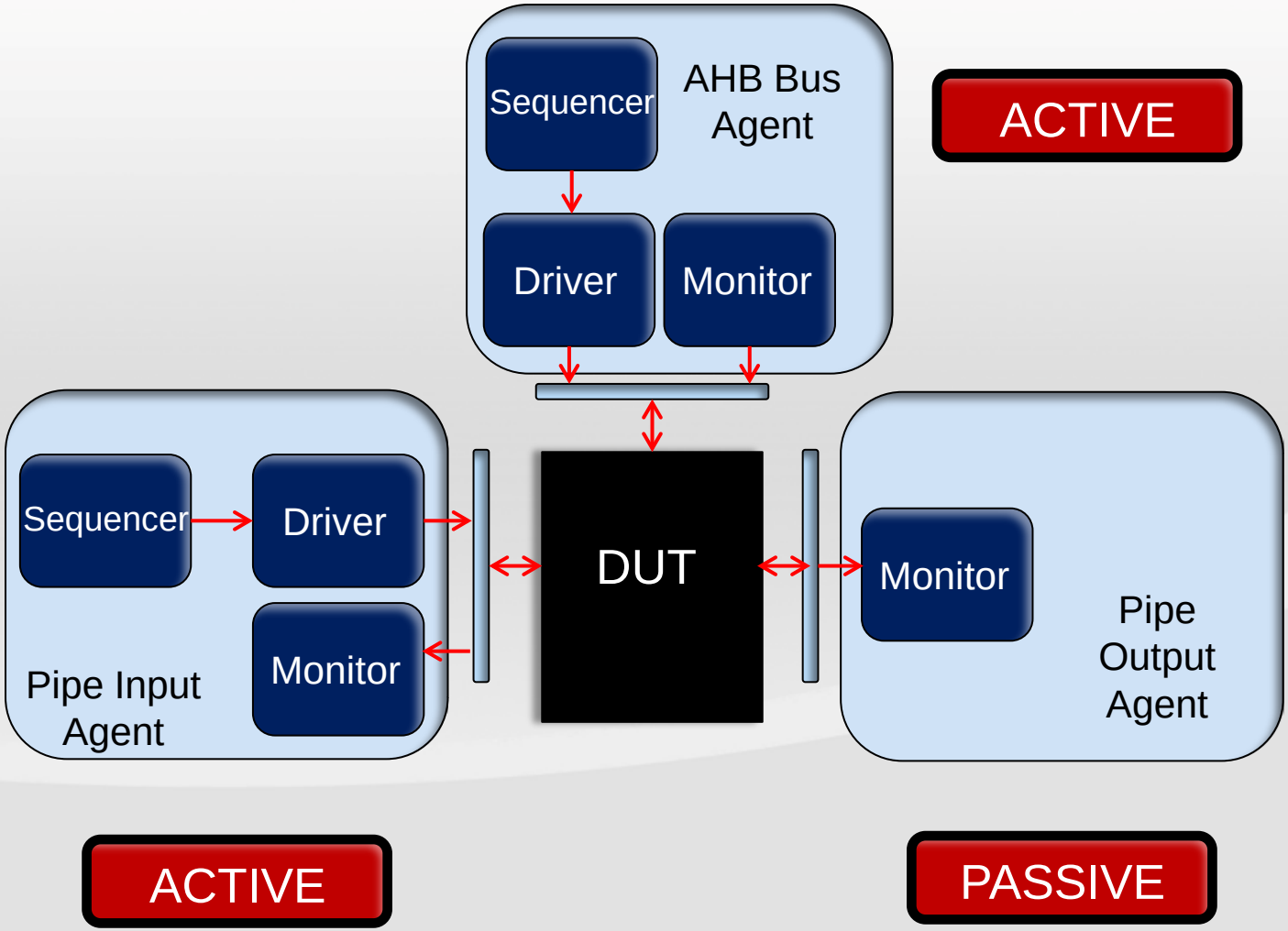
Vanessa Cooper

Verification Consultant

Agenda

- Testbench Architecture
- Using the Configuration Database
- Connecting the Scoreboard
- Register Model: UVM Reg Predictor
- Register Model: Coverage

Testbench Architecture



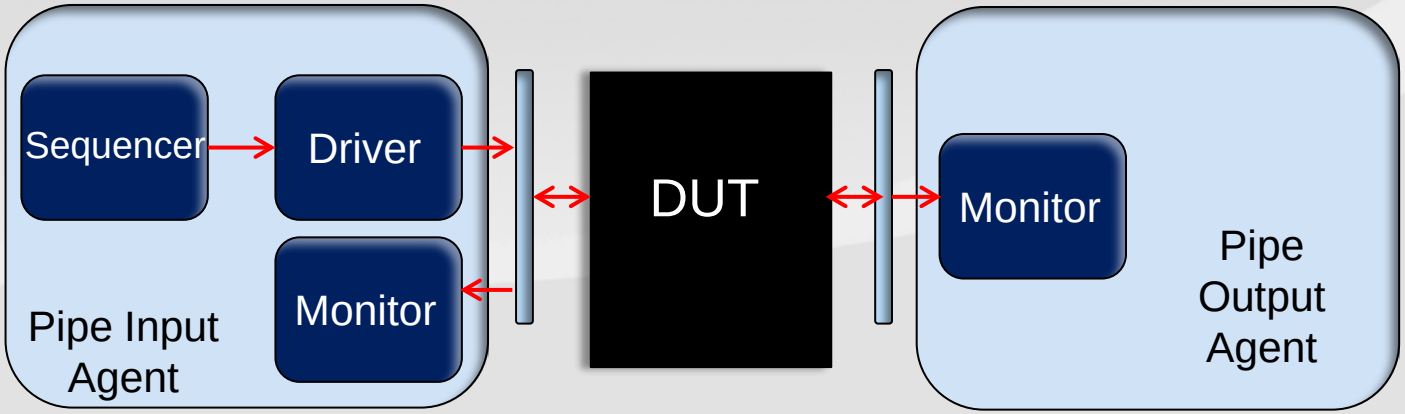
Agenda

- Testbench Architecture
- **Using the Configuration Database**
- Connecting the Scoreboard
- Register Model: UVM Reg Predictor
- Register Model: Coverage

Using the Configuration Database

PROBLEM

- **Reuse Monitor and Interface for Input and Output**
- **Ensure Monitor selects correct Interface**



Using the Configuration Database

```
dut_if vif(.clk(clk),.rst_n(rst_n));

uvm_config_db#(virtual dut_if)::set(uvm_root::get( ), "*",
                                    "dut_intf", vif);
```

Top

```
static function void set(uvm_component cntxt,
                         string        inst_name,
                         string        field_name,
                         T             value)
```

Using the Configuration Database

```
uvm_config_db#(virtual dut_if)::get(this, "", "dut_intf",
                                    vif);
```

Monitor

```
static function bit get(     uvm_component cntxt,
                             string        inst_name,
                             string        field_name,
                         ref T             value)
```

Using the Configuration Database

```
uvm_config_db#(virtual dut_if)::get(this, "", "dut_intf",
                                    vif);
```

Monitor

```
if(!uvm_config_db#(virtual dut_if)::get(this, "", "dut_intf",
                                        vif))
   `uvm_fatal("NOVIF", {"virtual interface must be set for:
                        ", get_full_name( ), ".vif"});
```

Using the Configuration Database

```
dut_if dut_ivif(.clk(clk), .rst_n(rst_n));
dut_if dut_ovif(.clk(clk), .rst_n(rst_n));

uvm_config_db#(virtual dut_if)::set(uvm_root::get( ), "*",
                                    "input_dut_intf",dut_ivif);

uvm_config_db#(virtual dut_if)::set(uvm_root::get( ), "*",
                                   "output_dut_intf", dut_ovif);
```

Top

Instantiate 2 Interfaces

Put both in uvm_config_db

Using the Configuration Database

```
class dut_monitor extends uvm_monitor;
   virtual dut_if vif;
   string monitor_intf;
   ...
endclass: dut_monitor
```

Monitor

```
uvm_config_db#(string)::set(this,"input_env.agent.monitor",
                            "monitor_intf", "input_dut_intf");
```

ENV

```
uvm_config_db#(string)::set(this, "output_env.agent.monitor",
                           "monitor_intf",  "output_dut_intf");
```

Using the Configuration Database

Monitor

```
class dut_monitor extends uvm_monitor;
   virtual dut_if vif;
   string monitor_intf;

   uvm_config_db#(string)::get(this, "", "monitor_intf",
                                          monitor_intf);

   uvm_config_db#(virtual dut_if)::get(this, "",
                                       monitor_intf, vif);
   ...
endclass: dut_monitor
```

Using the Configuration Database

```
uvm_config_db#(virtual dut_if)::set(uvm_root::get( ), "*",
                                    "input_dut_intf",dut_ivif);
```

Top

```
uvm_config_db#(virtual dut_if)::set(uvm_root::get( ),
          "*.dut_agent.monitor", "input_dut_intf",dut_ivif);
```

Using the Resource Database

- Let the interface name itself

```
interface dut_if(input clk, rst_n);
   string if_name = $sformatf("%m");
endinterface
```

- Put the interfaces in the Interface Registry

```
uvm_resource_db#(virtual dut_if)::set("Interface Registry",
                                      dut_ivif.if_name, dut_ivif);

uvm_resource_db#(virtual dut_if)::set("Interface Registry",
                                      dut_ovif.if_name, dut_ovif);
```

Using the Resource Database

```
uvm_resource_db#(virtual dut_if)::set("Interface Registry",
                                      dut_ivif.if_name, dut_ivif);
```

```
static function void set(input string scope,
                         input string name,
                               T val,
                         input uvm_object accessor = null)
```

Using the Resource Database

```
class dut_monitor extends uvm_monitor;
   virtual dut_if vif;
   string monitor_intf;

   uvm_resource_db#(virtual dut_if)::read_by_name("Interface
                                    Registry", monitor_intf, vif);
   ...
endclass: dut_monitor
```

Monitor

```
static function bit read_by_name(input string scope,
                                 input string name,
                                   ref T val,
                                 input uvm_object accessor = null)
```

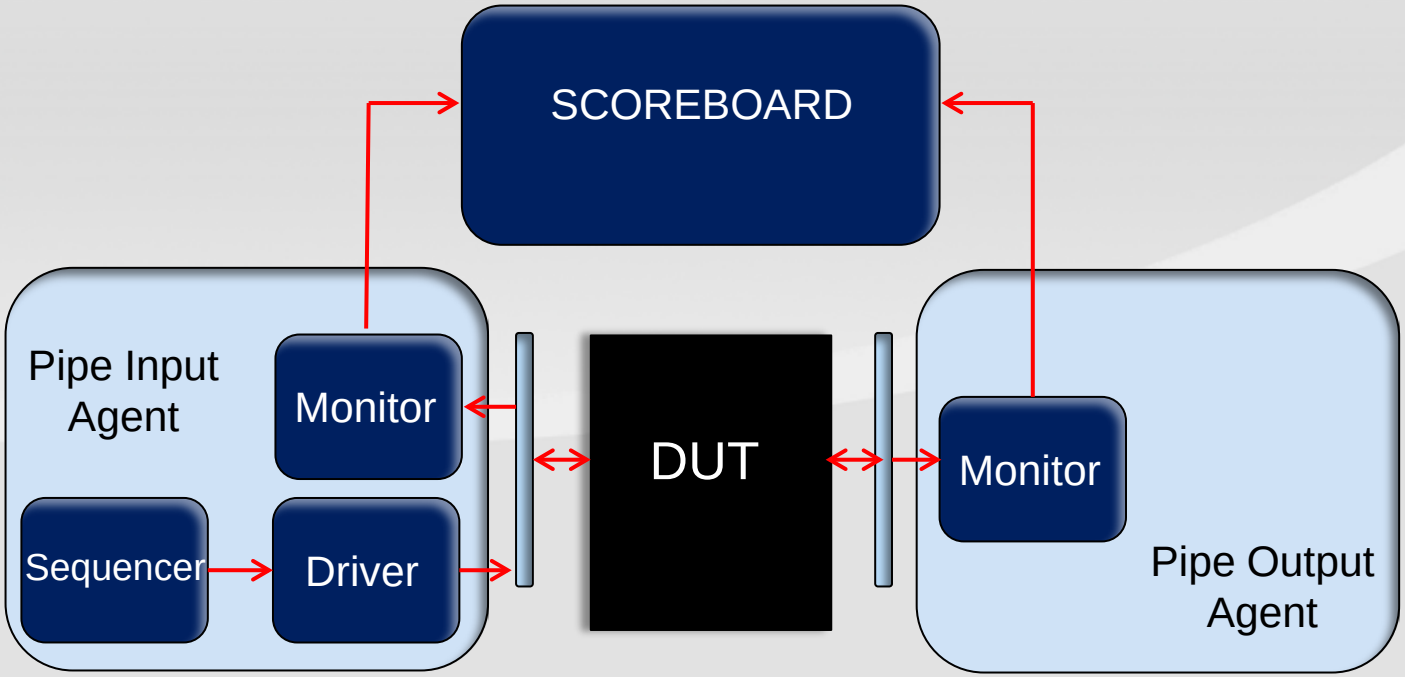
Agenda

- Testbench Architecture
- Using the Configuration Database
- **Connecting the Scoreboard**
- Register Model: UVM Reg Predictor
- Register Model: Coverage

Connecting the Scoreboard

PROBLEM

- **What is a simple way to connect the monitors to the scoreboard**

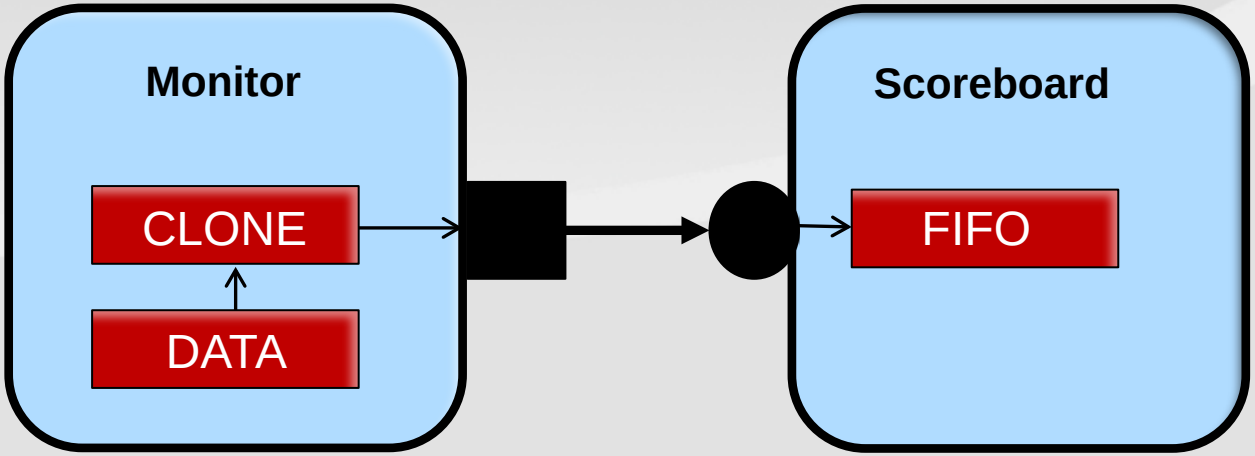


Connecting the Scoreboard

```
class dut_monitor extends uvm_monitor;
   ...
   uvm_analysis_port #(data_packet) items_collected_port;
   data_packet data_collected;
   data_packet data_clone;
   ...

endclass: dut_monitor
```

Analysis Port

Data Packets

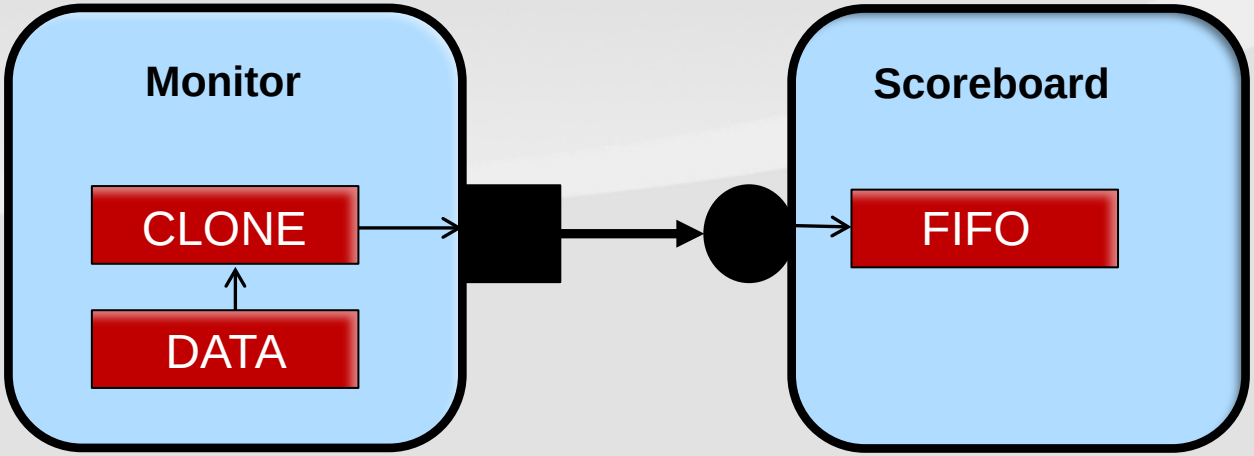


Connecting the Scoreboard

```
class dut_monitor extends uvm_monitor;
   ...
   virtual task collect_packets;
      ...
      $cast(data_clone, data_collected.clone( ));
      items_collected_port.write(data_clone);
   endtask: collect_packets
   ...
endclass: dut_monitor
```

Clone data packets

Write to the port



Connecting the Scoreboard

```
class dut_scoreboard extends uvm_scoreboard;
   ...
   uvm_tlm_analysis_fifo #(data_packet) input_packets_collected;
   uvm_tlm_analysis_fifo #(data_packet) output_packets_collected;
   ...

   virtual task watcher( );
      forever begin
         @(posedge top.clk);
         if(input_packets_collected.used( ) != 0) begin
            ...
         end
      end
   endtask: watcher
endclass: dut_scoreboard
```

TLM Analysis Ports

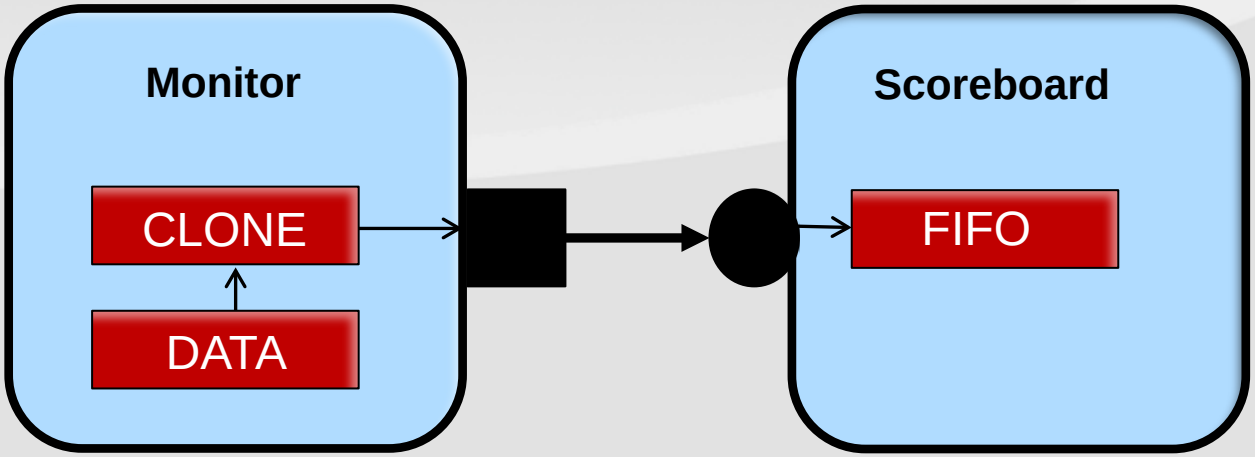
.used() not .size()

Connecting the Scoreboard

```
input_env.agent.monitor.items_collected_port.connect
      (scoreboard.input_packets_collected.analysis_export);
```

ENV

```
output_env.agent.monitor.items_collected_port.connect
      (scoreboard.output_packets_collected.analysis_export);
```



Connecting the Scoreboard

```
virtual task watcher( );
   forever begin
      @(posedge top.clk);
      if(input_packets_collected.used( ) != 0) begin
         ...
      end
   end
endtask: watcher
```

```
virtual task watcher( );
   forever begin
      input_packets_collected.get(input_packets);
         ...
      end
   end
endtask: watcher
```

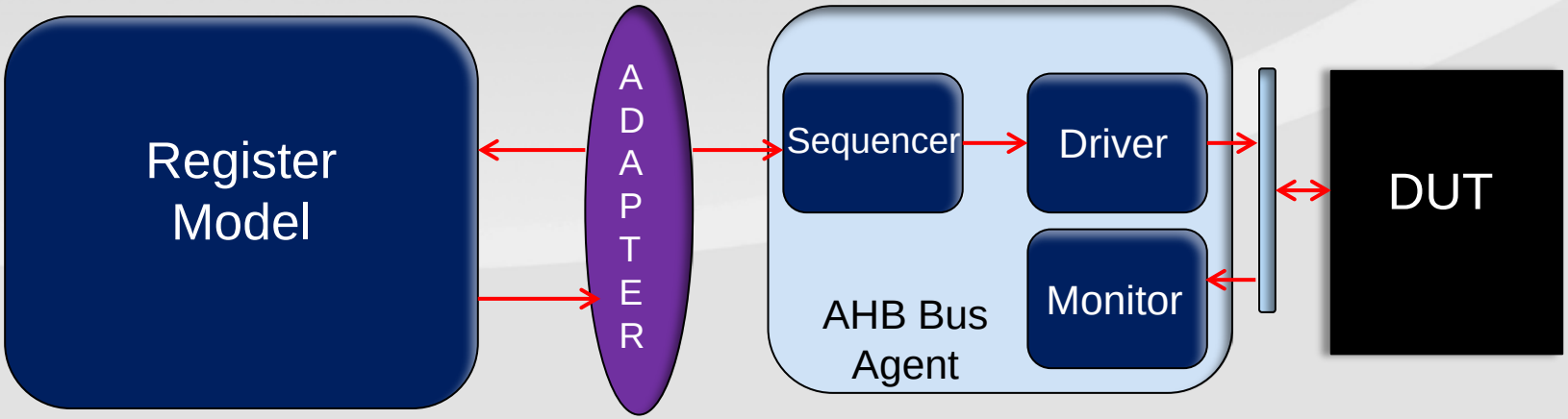
Agenda

- Testbench Architecture
- Using the Configuration Database
- Connecting the Scoreboard
- **Register Model: UVM Reg Predictor**
- Register Model: Coverage

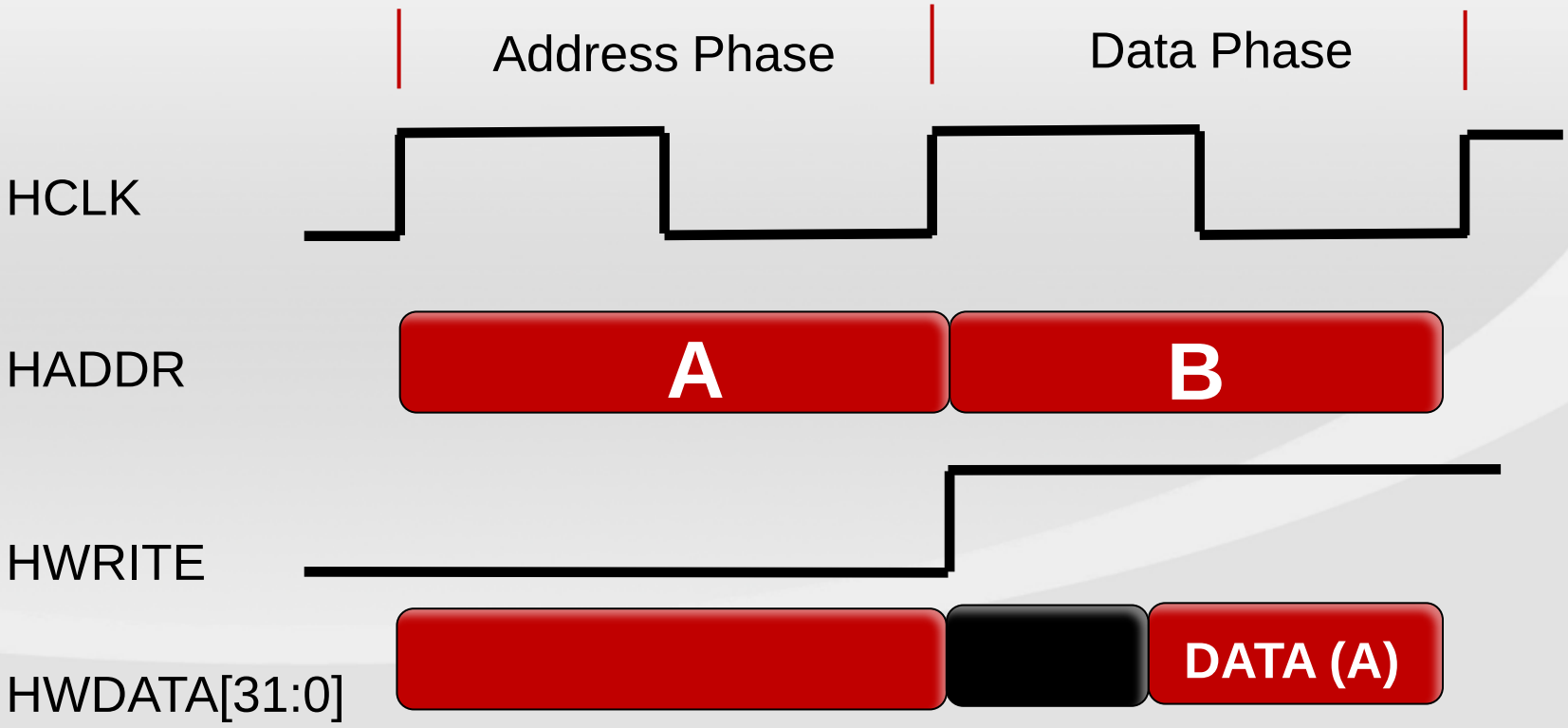
Register Model: UVM Reg Predictor

PROBLEM

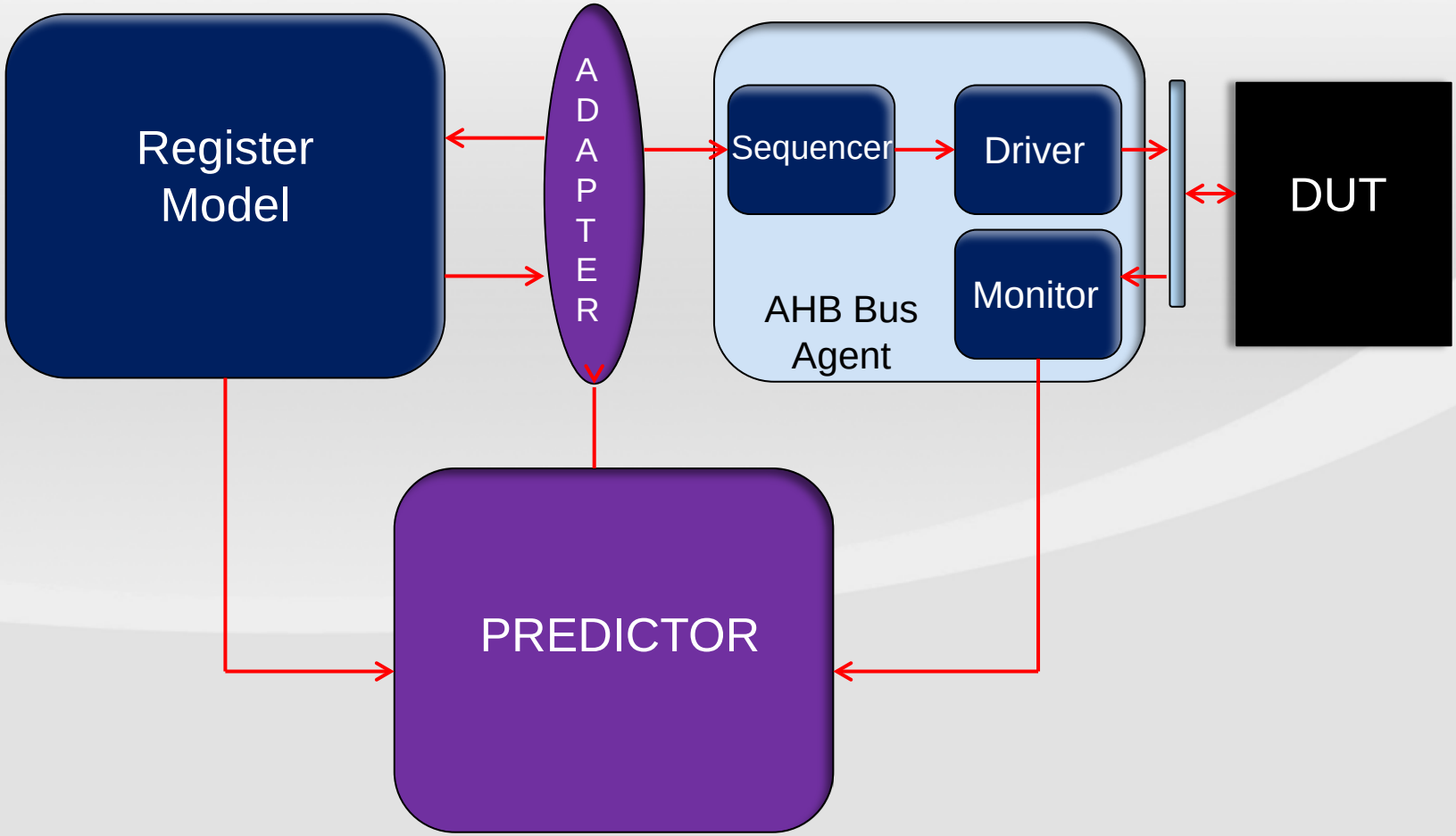
- **Use the Register Model with the pipeline AHB bus**
- **Capture read data accurately**



Register Model: UVM Reg Predictor



Register Model: UVM Reg Predictor



Register Model: UVM Reg Predictor

- **build_phase**
  - Create the predictor with the bus uvm_sequence_item parameter in your env
- **connect_phase**
  - Set the predictor map to the register model map
  - Set the predictor adapter to the register adapter
  - Connect the predictor to the monitor

Register Model: UVM Reg Predictor

ENV

Declare

```
uvm_reg_predictor#(ahb_transfer) reg_predictor;
```

Create

```
reg_predictor = uvm_reg_predictor#(ahb_transfer)::
                type_id::create("reg_predictor", this);
```

Map

```
reg_predictor.map = master_regs.default_map;
reg_predictor.adapter = reg2ahb_master;
```

Connect

```
ahb_env.agent.monitor.item_collected_port.
                          connect(reg_predictor.bus_in);
```

Register Model: UVM Reg Predictor

```
master_regs.default_map.set_auto_predict(0);
```

Implicit

Explicit

Passive

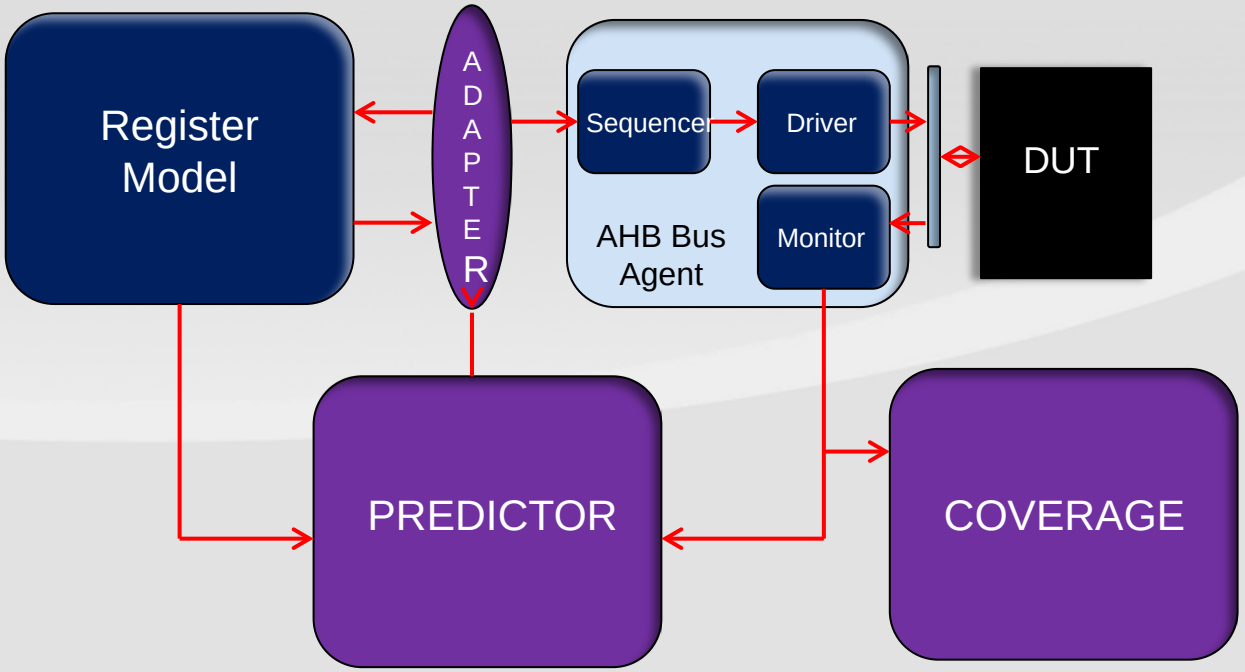
Agenda

- Testbench Architecture
- Using the Configuration Database
- Connecting the Scoreboard
- Register Model: UVM Reg Predictor
- **Register Model: Coverage**

Register Model: Coverage

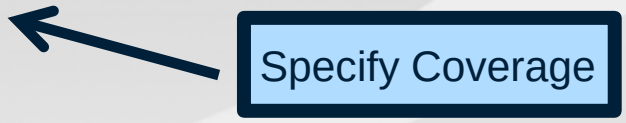
PROBLEM

- How do I enable coverage with my Register Model



Register Model: Coverage

```
class regs_control_reg extends uvm_reg;

   rand uvm_reg_field control;

   function new(string name = "regs_control_reg");
       super.new(name, 32, build_coverage(UVM_CVR_ALL));
    endfunction: new

   virtual function void build( );
      ...
   endfunction: build

   `uvm_object_utils(regs_control_reg)

endclass: regs_control_reg
```

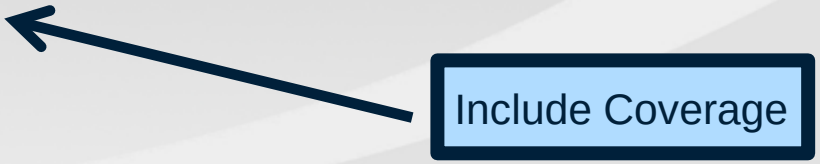


Register Model: Coverage

```
211 #include_coverage not located
212 #  did you mean disable_scoreboard?
213 #  did you mean dut_name?
214 #include_coverage not located
215 #  did you mean disable_scoreboard?
216 #  did you mean dut_name?
```

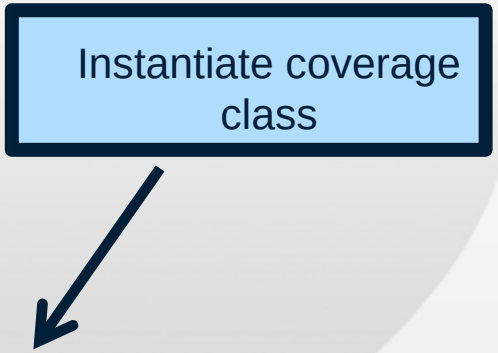
Register Model: Coverage

```
class base_test extends uvm_test;
   ...

   `uvm_component_utils(base_test)

   function new(string name, uvm_component parent);
      super.new(name, parent);
      uvm_reg::include_coverage("*", UVM_CVR_ALL);
   endfunction: new

   ...
endclass: base_test
```



Register Model: Coverage

```
class dut_regs extends uvm_reg_block;
   ...
   reg_coverage reg_cov;

   virtual function void build( );
      if(has_coverage(UVM_CVR_ALL)) begin
         reg_cov = reg_coverage::type_id::create("reg_cov");
         set_coverage(UVM_CVR_ALL);
      end
      ...
   endfunction: build

   `uvm_object_utils(dut_regs)
endclass: dut_regs
```



Register Model: Coverage

```
class dut_regs extends uvm_reg_block;

   ...

   function void sample(uvm_reg_addr_t offset, bit is_read,
                        uvm_reg_map map);

      if(get_coverage(UVM_CVR_ALL)) begin
         if(map.get_name( ) == "default_map") begin
            reg_cov.sample(offset, is_read);
         end
      end
   endfunction: sample

endclass: dut_regs
```

Automatically Called

Call sample in coverage class

Register Model: Coverage

```
class reg_coverage extends uvm_object;
   ...
   covergroup reg_cg(string name) with function
              sample(uvm_reg_addr_t addr, bit is_read);

      //COVERPOINTS HERE
      ...

   endgroup: reg_cg

   ...

   function void sample(uvm_reg_addr_t offset, bit is_read);
      reg_cg.sample(offset, is_read);
   endfunction: sample

endclass: reg_coverage
```

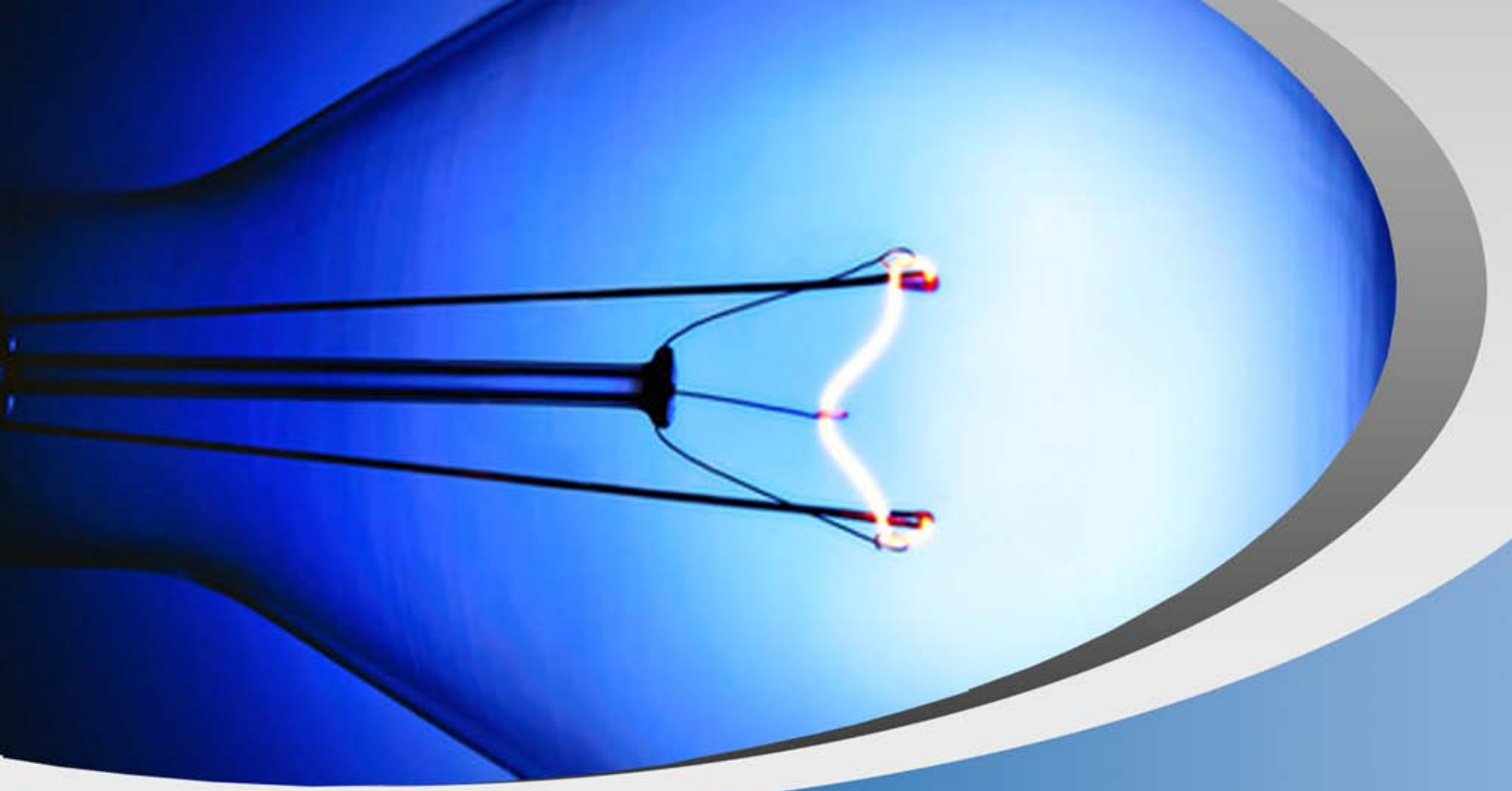
Register Model: Coverage

Covergroup

```
ADDR: coverpoint addr {
    bins mode = {'h00};
    bins cfg1 = {'h04};
    bins cfg2 = {'h08};
    bins cfg3 = {'h0C};
    }

RW: coverpoint is_read {
    bins RD = {1};
    bins WR = {0};
    }

ACCESS: cross ADDR, RW;
```

Questions

- Testbench Architecture
- Using the Configuration Database
- Register Model: UVM Reg Predictor
- Register Model: Coverage
- Connecting the Scoreboard



UVM: Ready, Set, Deploy!

UVM

DVCon 2012
Design & Verification Conference & Exhibition

accellera
SYSTEMS INITIATIVE

Stacking Verification Components in UVM

Stephen D'Onofrio

Peter D'Antonio

Overview

- **Background**
- **UVM Motivation**
- **Reuse Requirements**
- Stacking UVCs Requirements & Architecture
- Code Examples
- Conclusion

Background

- **Verification at MITRE**
  - Effective verification flow for DSP combines analysis, system modeling and RTL verification
  - Worked well in previous technologies
  - File based, directed tests; capacity does not scale with technology
- **Project need**
  - 45nm ASIC, DSP datapath design
  - Schedule/resource constrained, 1st pass success
  - Step function verification flow improvement needed to meet increased design complexity

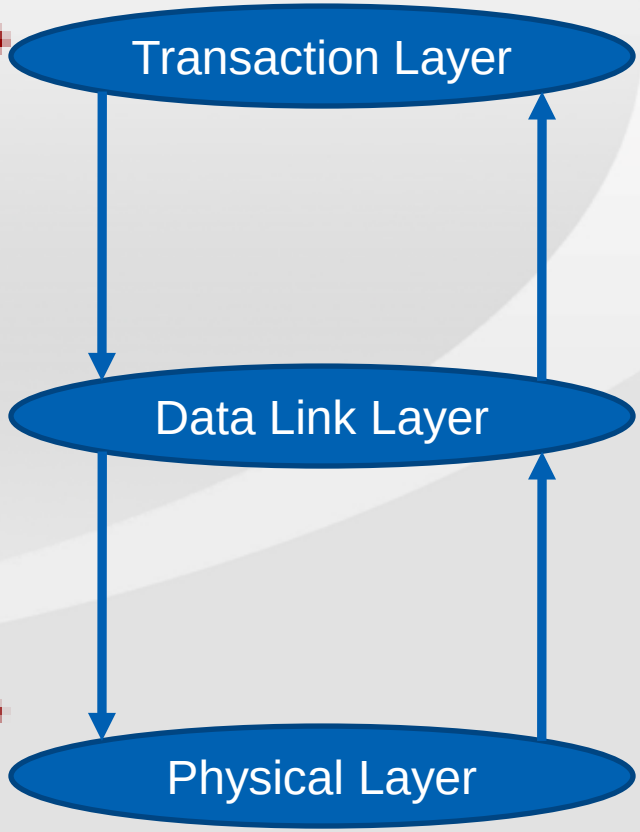
UVM Motivation

- **Quality**
  - Improve functional verification and ultimately product
- **Efficiency**
  - Return on investment through reuse
  - Recommended structure gives us a foundation to build
- **Leverage**
  - Broad industry support
  - User momentum
  - EDA vendor independent

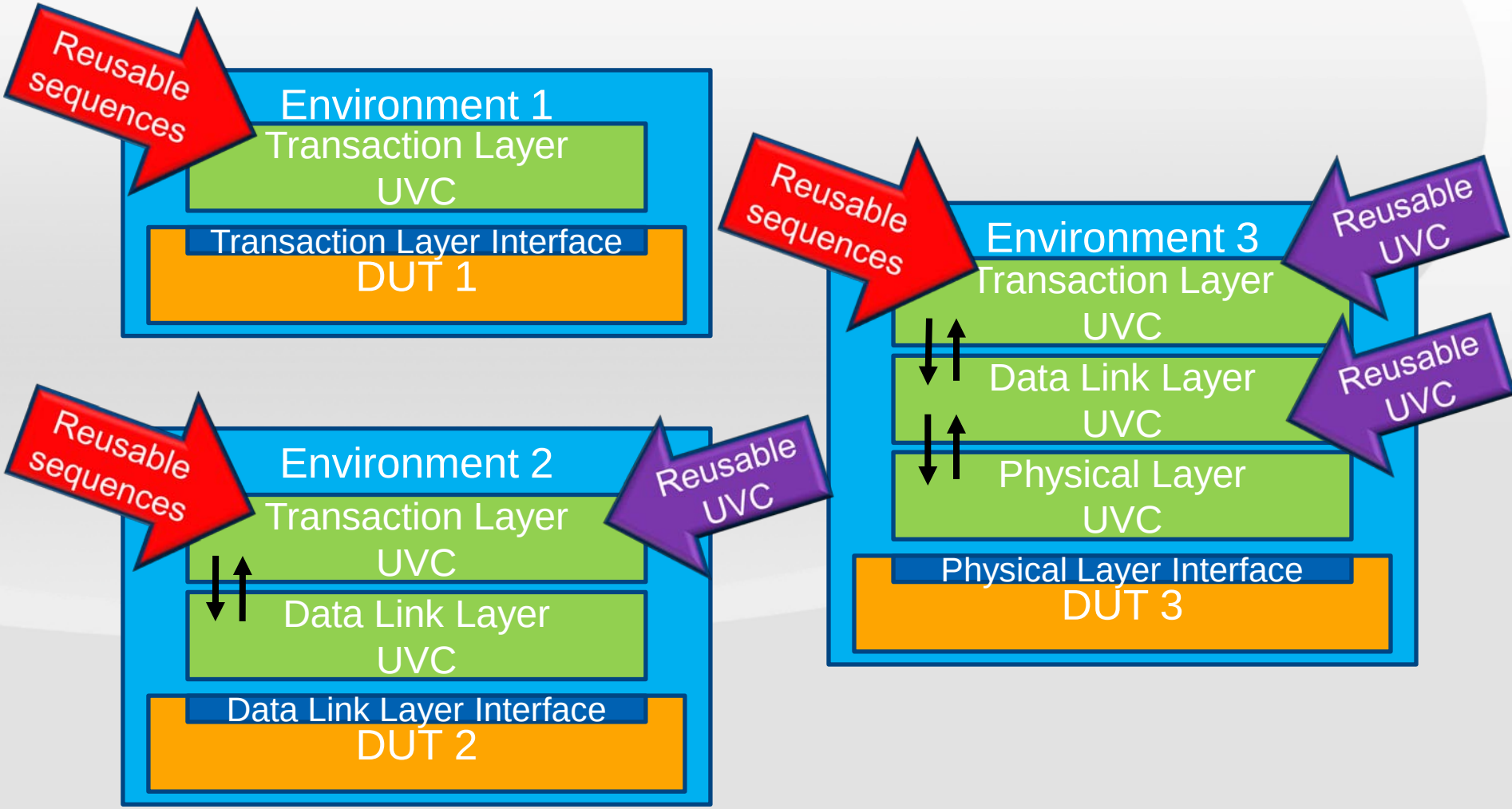
Reuse Requirements

- **Reuse as much as possible!**
- **UVCs**
  - Configurable behavior
- **DUT Expect models**
  - Reuse between unit and system
  - Components with TLM ports
- **Sequences**
  - Abstract UVCs
  - Virtual sequences

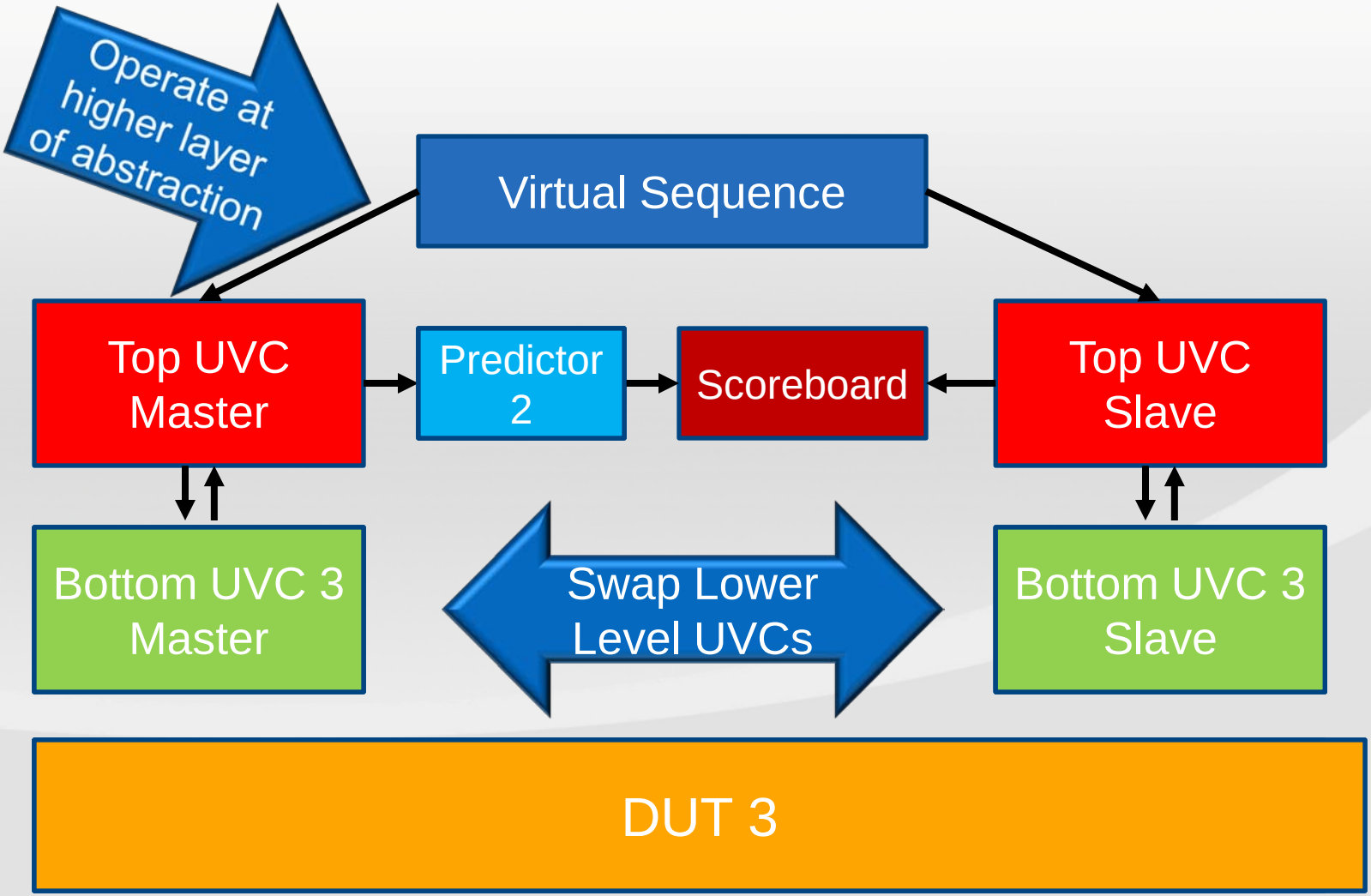
Reuse Problem



Vertical Reuse Solution



Horizontal Reuse Solution



Overview

- Background
- UVM Motivation
- Reuse Requirements
- **Stacking UVCs Requirements & Architecture**
- **Code Examples**
- Conclusion

UVM Layering Shortcomings

- User Guide missing layering monitor path
- No methodology for layering across UVCs
- No UVCs (agents) TLM interconnection strategy

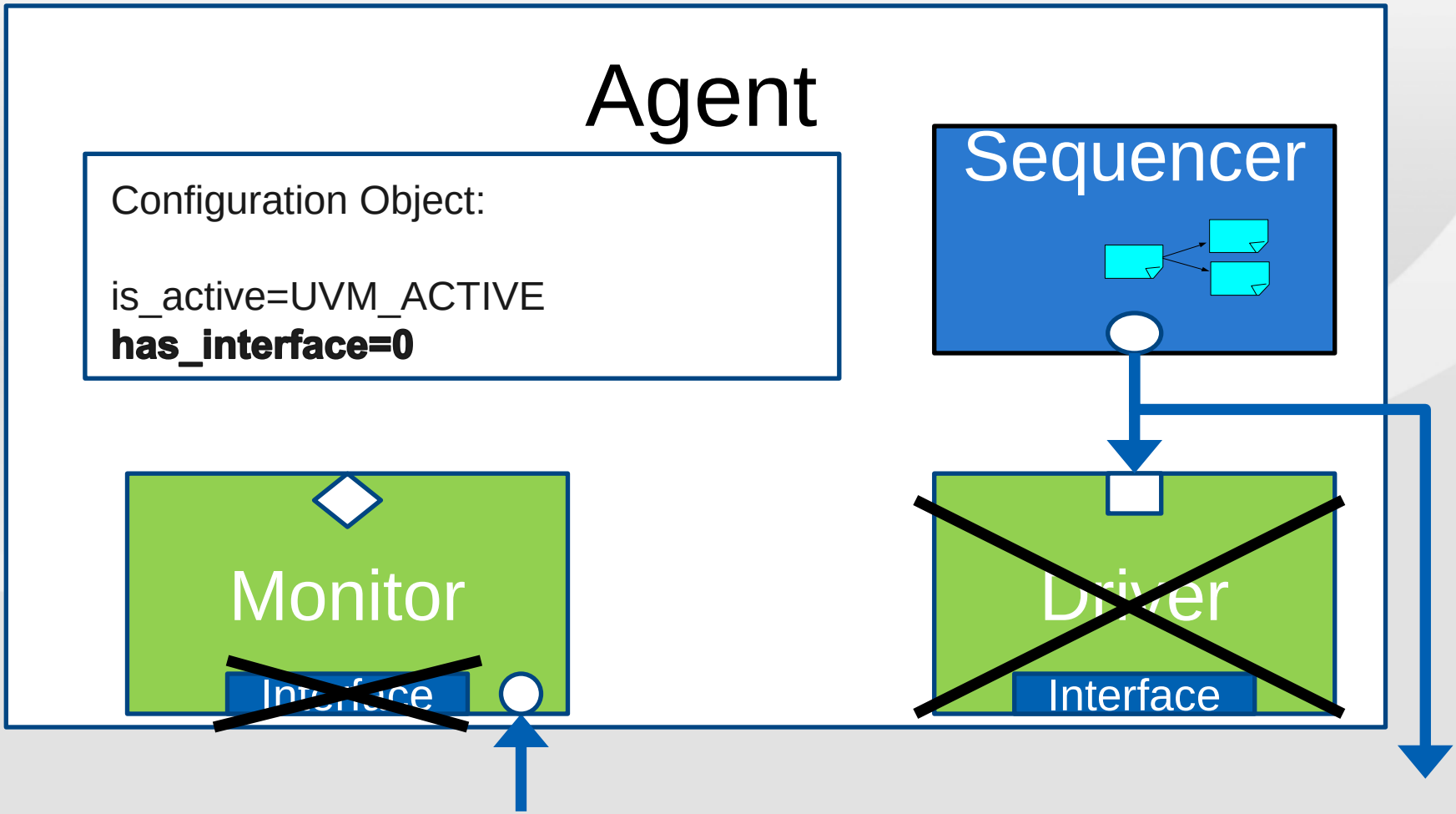
What Is Stacking UVCs?

- A framework for UVCs to handle layering
- Connecting or stacking one or more UVCs
- Easily swap UVCs anywhere in a stack
- Arbitrary UVC to UVC connection

Stacking Requirements

- **UVCs**
  - Optionally include interface
  - No knowledge of other UVCs
- **Adapter package**
  - **Converts** transactions
  - Connects 2 or more stacked UVCs together
  - Must be reusable
    - Operate in both Active/Passive modes
    - Includes a configuration object

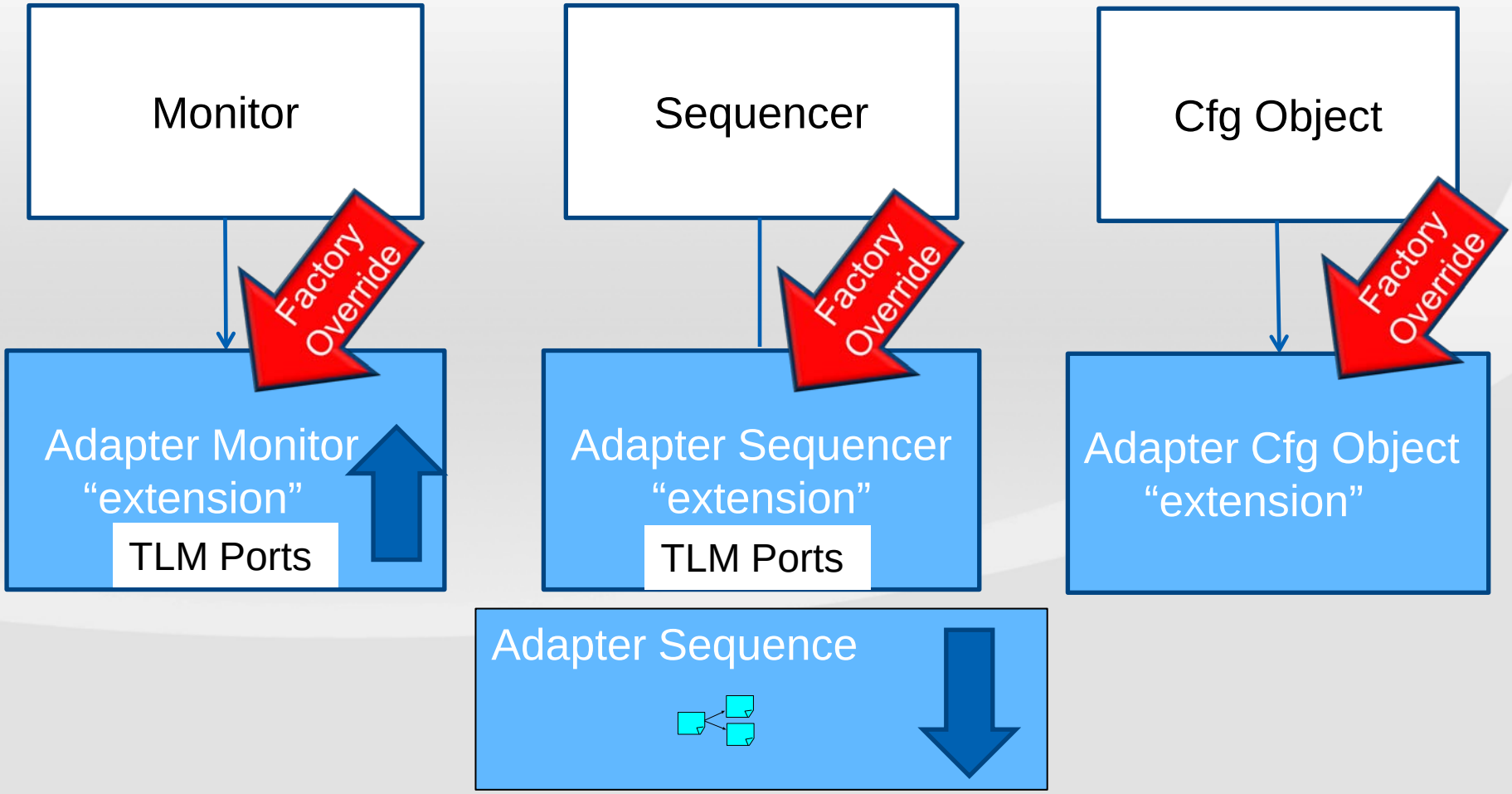
UVC


Agent
Configuration Object:
is_active=UVM_ACTIVE
has_interface=0
Sequencer
Monitor
Interface
Driver
Interface

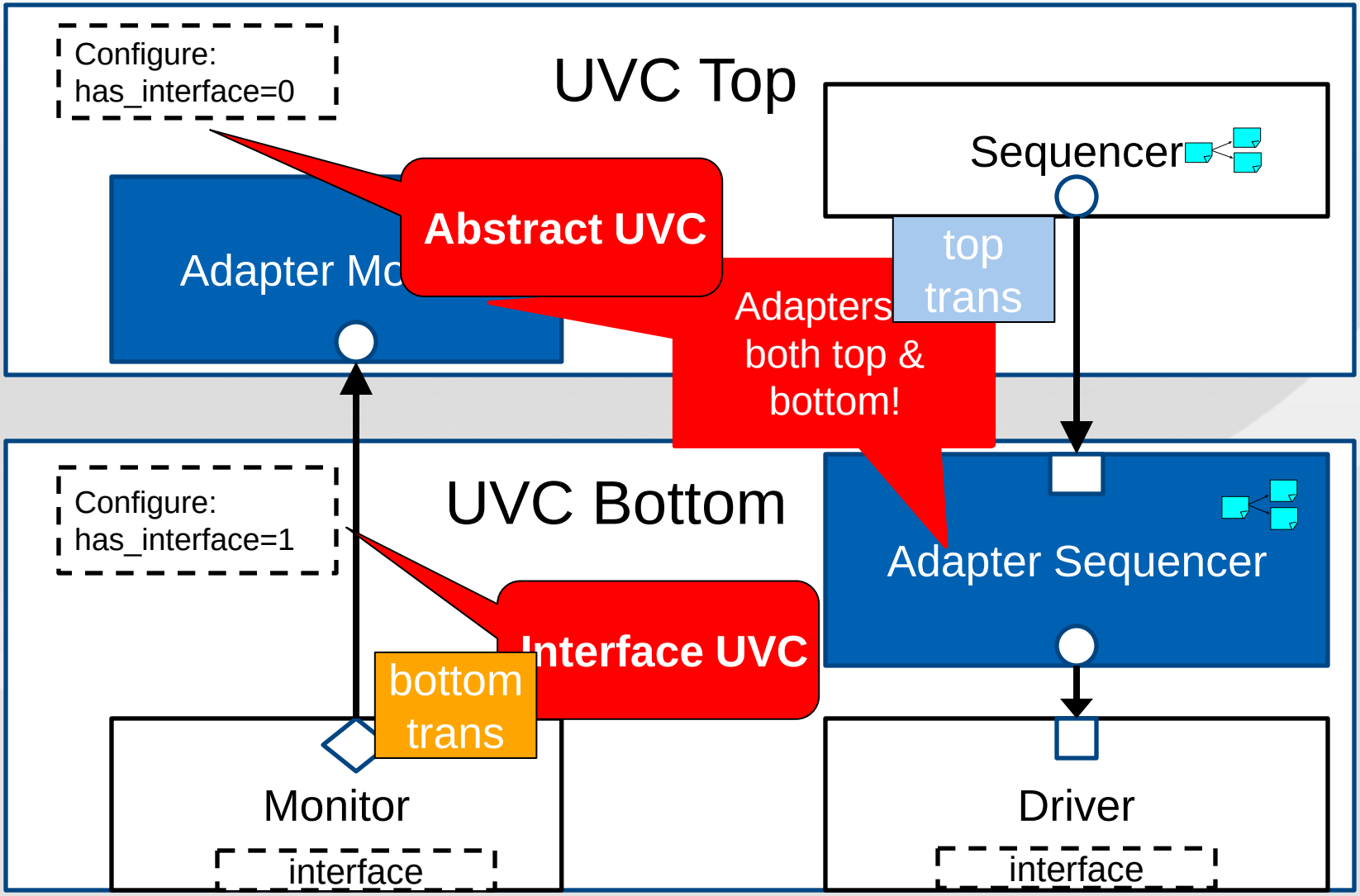
UVC Agent Code Snippet

```
class top_agent extends uvm_agent;

  bit has_interface = 1;

  function void build_phase(uvm_phase phase);
    void'(uvm_config_db#(bit)::get(this, "", "has_interface", has_interface));

    monitor = top_in_monitor::type_id::create("monitor",this);
    if(get_is_active() == UVM_ACTIVE) begin
      sequencer = top_in_sequencer::type_id::create("sequencer",this);
      if (has_interface)
        driver = top_in_driver::type_id::create("driver",this);
    end
  endfunction

  function void connect_phase(uvm_phase phase);
    if(get_is_active() == UVM_ACTIVE && has_interface)
      driver.seq_item_port.connect(sequencer.seq_item_export);
  endfunction
```

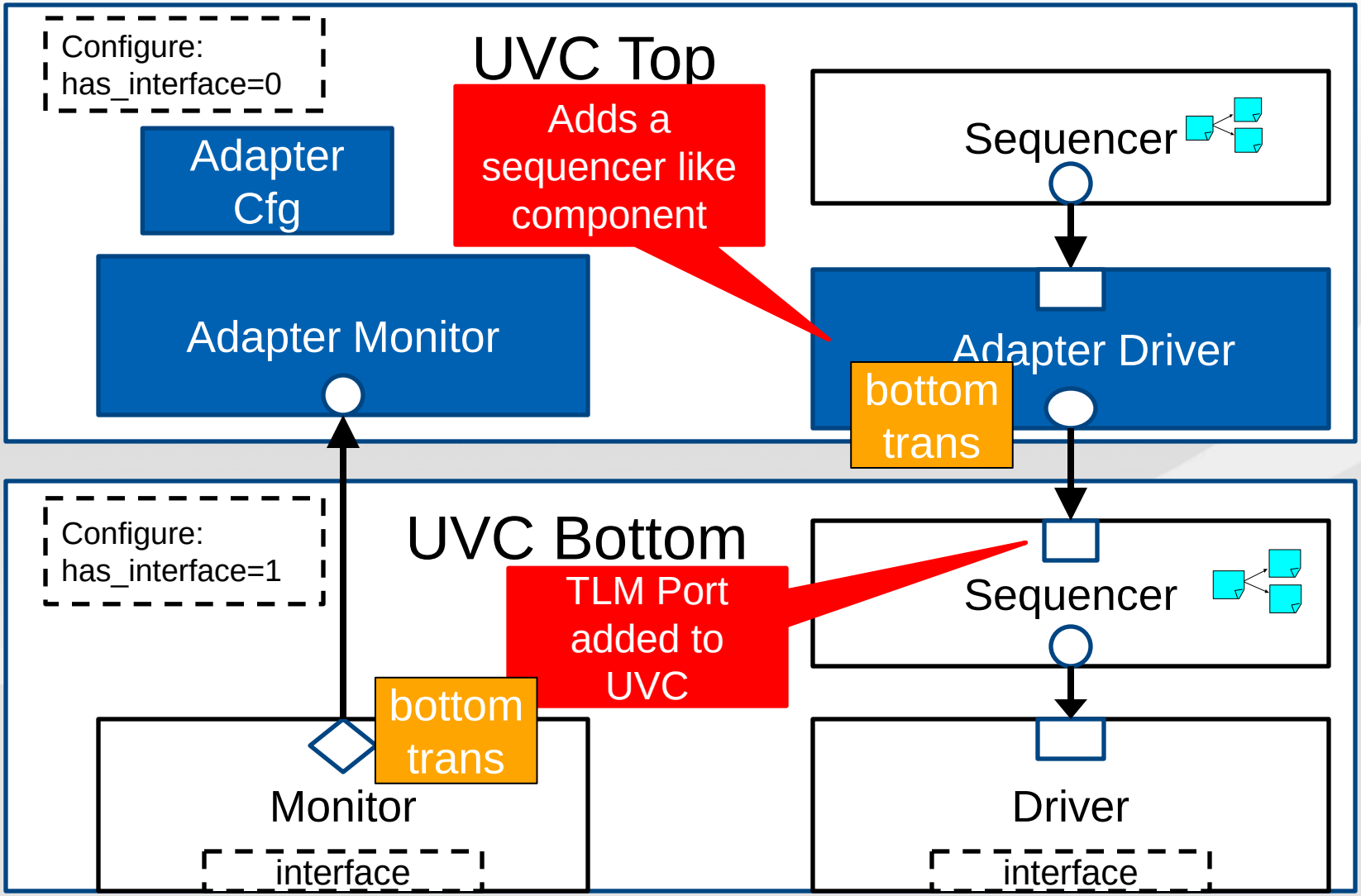
Adapter Package



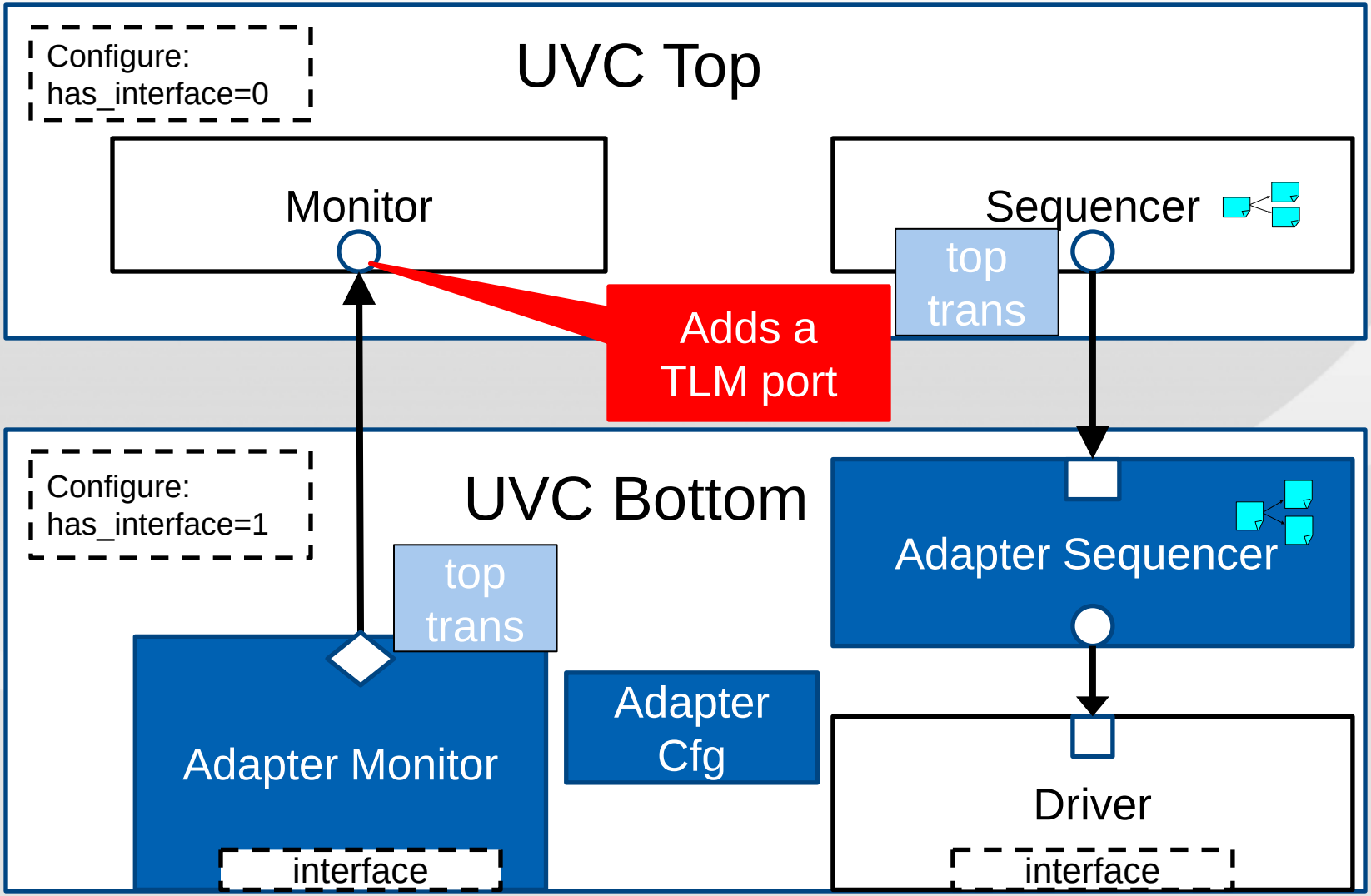
Implementation Option A



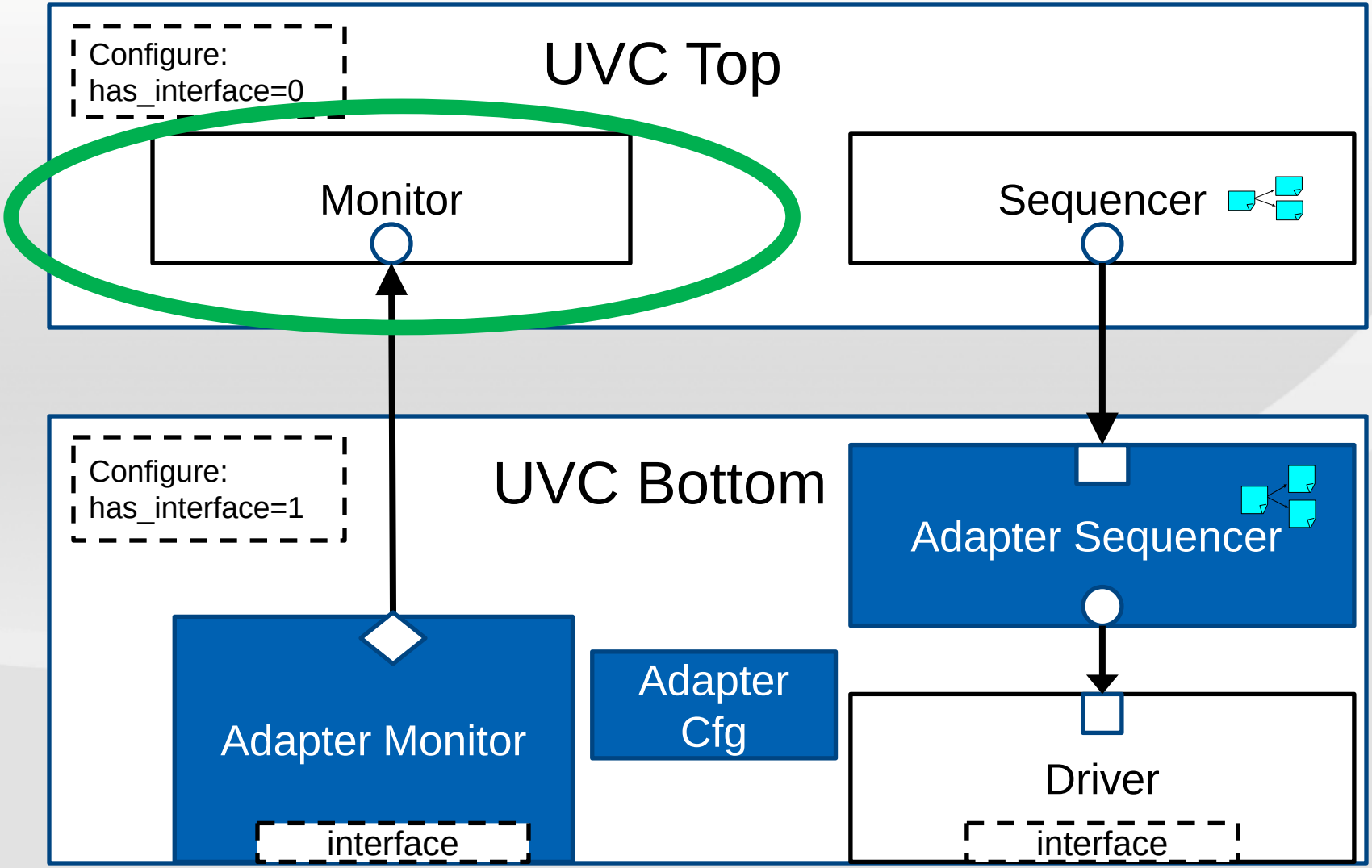
Implementation Option B



Implementation Option C



Monitor Code Snippet



Monitor Code - Top UVC

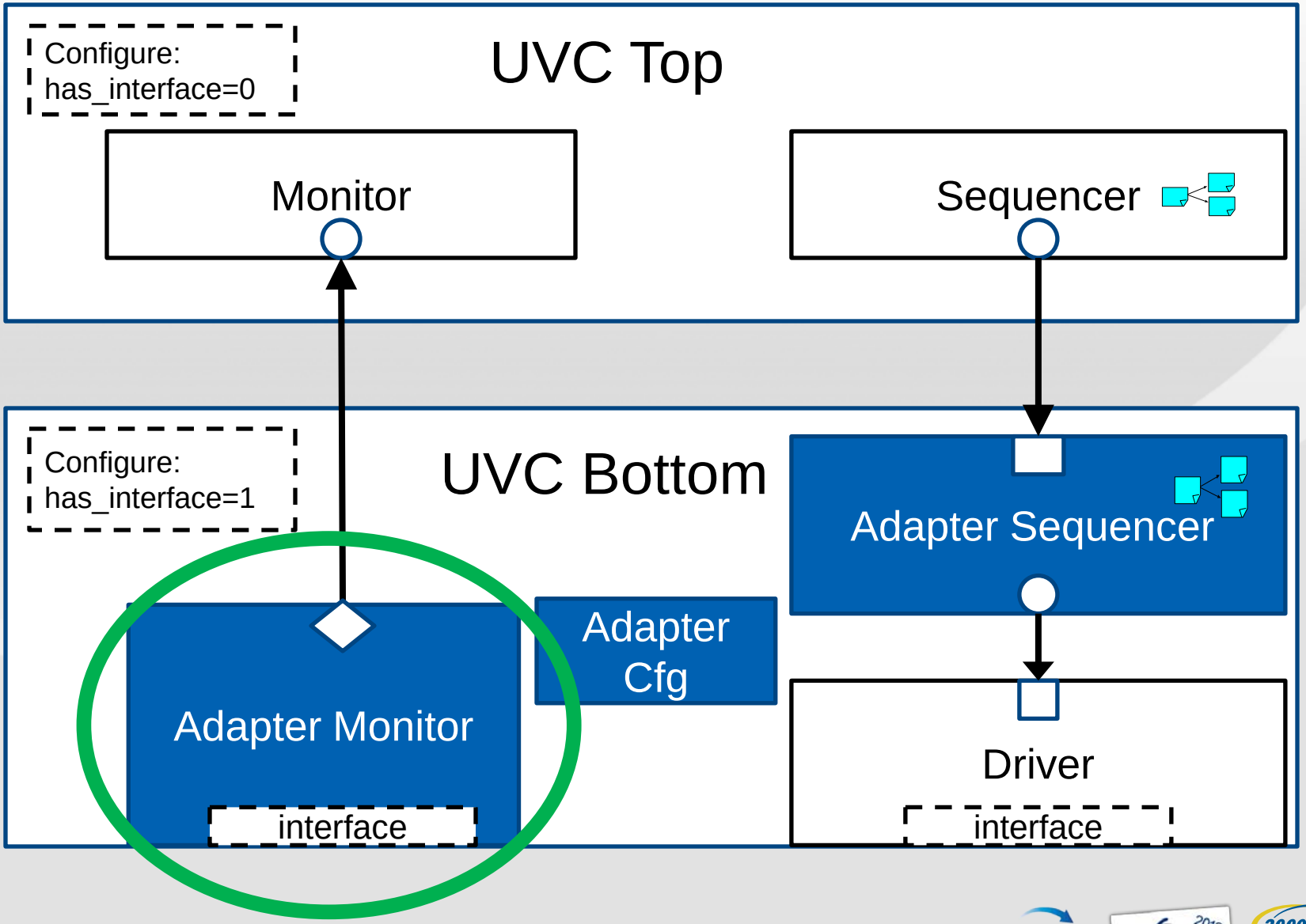
```
class top_monitor extends uvm_monitor;

uvm_analysis_imp#(top_trans, top_monitor) top_item_collected_imp;

function void write(top_trans trans);

  // Perform checks on the transaction & coverage
  …
  // Send this transaction to other components
  publish_transaction(trans);

endfunction : write

virtual function void publish_transaction(trans);
  $cast(cloned_trans, trans.clone());
  item_collected_port.write(cloned_trans);
endfunction : publish_transaction
```

Add TLM analysis imp port to allow lower layer UVC to send transactions to this UVC

Typical UVM Monitor Procedure code

Broadcast transactions to SB and other components

Monitor Adapter Code Snippet



Monitor Adapter Code - Bottom Adapter UVC extension

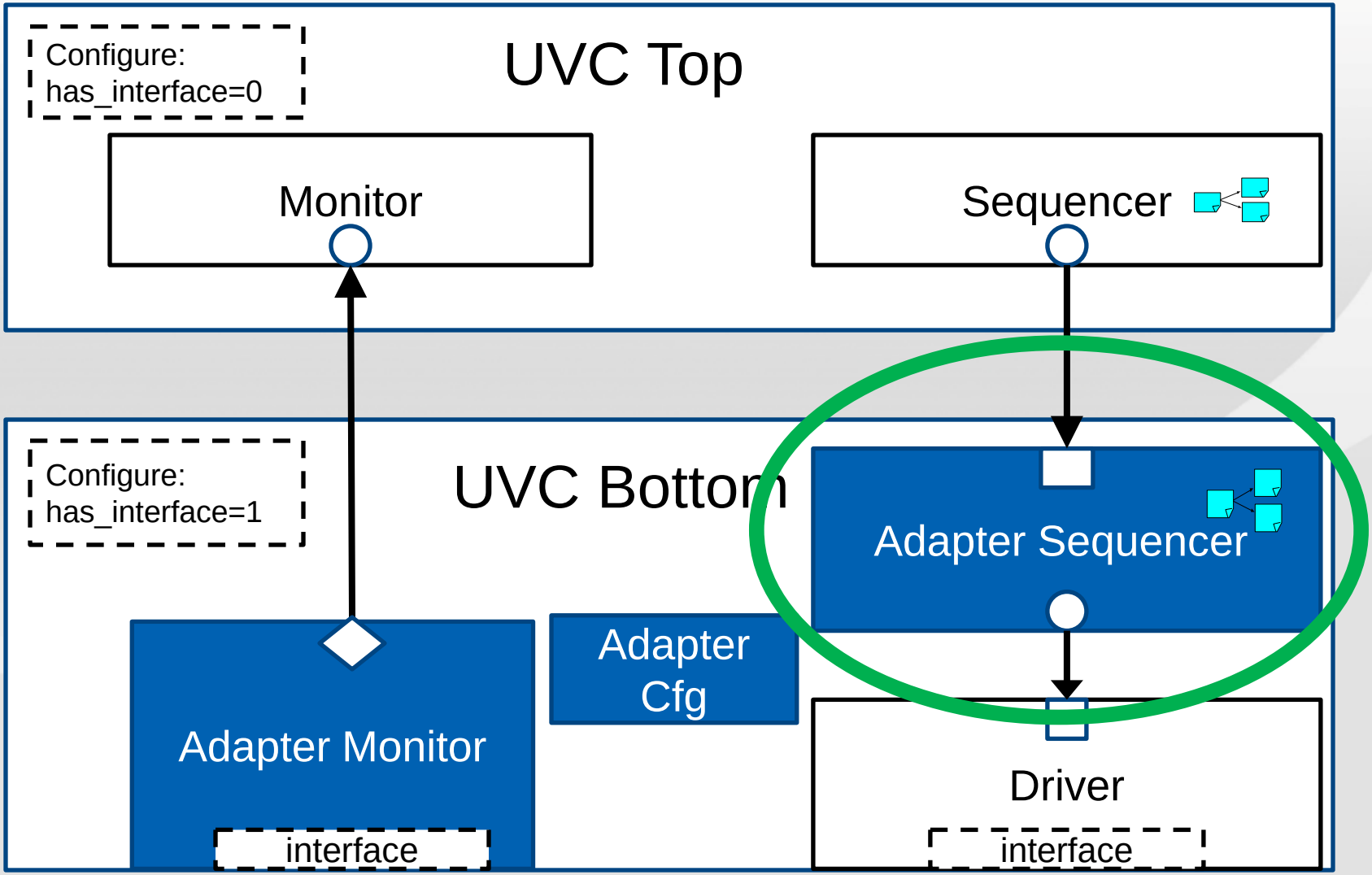
```
class btm_adapter_monitor extends btm_monitor;

uvm_analysis_port #(top_trans) top_item_collected_port;

virtual function void publish_transaction();
  super.publish_transaction();

  // top layer transactions
  top_trans = new;

  // USER: Add your convert code here
  …
  $cast(cloned_top_trans, top_trans.clone());
  top_item_collected_port.write(top_trans);
endfunction
…
```

Analysis Port that broadcasts **top transaction** to top UVC

Convert bottom transaction to top transaction and send it to top UVC

Send the top transaction to the top UVC

Sequencer Adapter Code Snippet



Sequencer Code - Bottom Adapter UVC Extension

```
class btm_adapter_sequencer extends btm_sequencer;

  //! This is the TLM pull port that allows this UVC to communicate with a top UVC
  uvm_seq_item_pull_port #(top_trans) top_seq_item_port;

  …
```

Use inheritance to extend the bottom UVC and add a top sequence item pull port

Sequence Code- Bottom Adapter UVC Sequence

```
class btm_adapter_sequence extends uvm_sequence #(btm_trans);

  btm_transaction btm_trans;

  top_transaction top_trans;

  `uvm_declare_p_sequencer(btm_adapter_sequencer)

task body();
 forever begin : repeat_block
   p_sequencer.top_seq_item_port.get_next_item(top_trans);
   // USER: Add your top/bottom conversion here
  `uvm_send(btm_trans)
   p_sequencer.top_seq_item_port.item_done();
  end
endtask
```

Declare bottom & top transaction

Forever convert top transaction to bottom transaction

Sequence Code - Bottom Adaptor with Response

```
class btm_adapter_sequence extends uvm_sequence #(btm_trans);
  btm_transaction  btm_req_trans;
  top_transaction   top_req_trans;
  btm_transaction  btm_rsp_trans;
  top_transaction   top_rsp_trans;

  `uvm_declare_p_sequencer(btm_adapter_sequencer)

  task body();
    forever begin : repeat_block
      p_sequencer.top_seq_item_port.get_next_item(top_req_trans);
      // USER: Convert top request to bottom request here
      `uvm_send(btm_req_trans)
      get_response(btm_rsp_trans)
      top_rsp_trans = new;
      // USER: Convert bottom response to top response here
      p_sequencer.top_seq_item_port.item_done(top_rsp_trans);
    end
  endtask
```

Declare bottom & top

Declare bottom & top response transactions

Same forever loop

Get bottom response, convert to top

Pass top response to item_done()

Testbench Override Code

```
class testbench extends uvm_env;

function void build_phase(uvm_phase phase);

  set_type_override_by_type (
                        btm_cfg::get_type(),
                        btm_adapter_cfg::get_type());
  set_inst_override_by_type("btm.*",
                        btm_sequencer::get_type(),
                        btm_adapter_sequencer::get_type());
  set_inst_override_by_type("btm.*",
                        btm_monitor::get_type(),
                        btm_adapter_monitor::get_type());
endfunction
…
```

Use factory to override UVC bottom components with “bottom adapter” components.

Testbench Override Code (Cont)

```
connect_phase(uvm_phase phase);
  btm_adapter_monitor monitor;
  btm_adapter_sequencer sequencer;

  $cast(monitor, btm.monitor);
  monitor.top_item_collected_port.connect(top.monitor.top_item_collected_imp);

  if (top.is_active == UVM_ACTIVE && btm.is_active == UVM_ACTIVE) begin
    $cast(sequencer, btm.sequencer);
    sequencer.top_seq_item_port.connect(top.sequencer.seq_item_export);
  end
...
```

Connect the monitor adapter TLM ports

Connect the sequencer adapter TLM ports

Testbench Override Code (Cont)

```
connect_phase(uvm_phase phase);
  …
  uvm_config_db#(uvm_object_wrapper)::set(this,
                                          (".run_phase",
                                           "default_sequence",
                                            btm_adapter_sequence::type_id::get());
  …
```

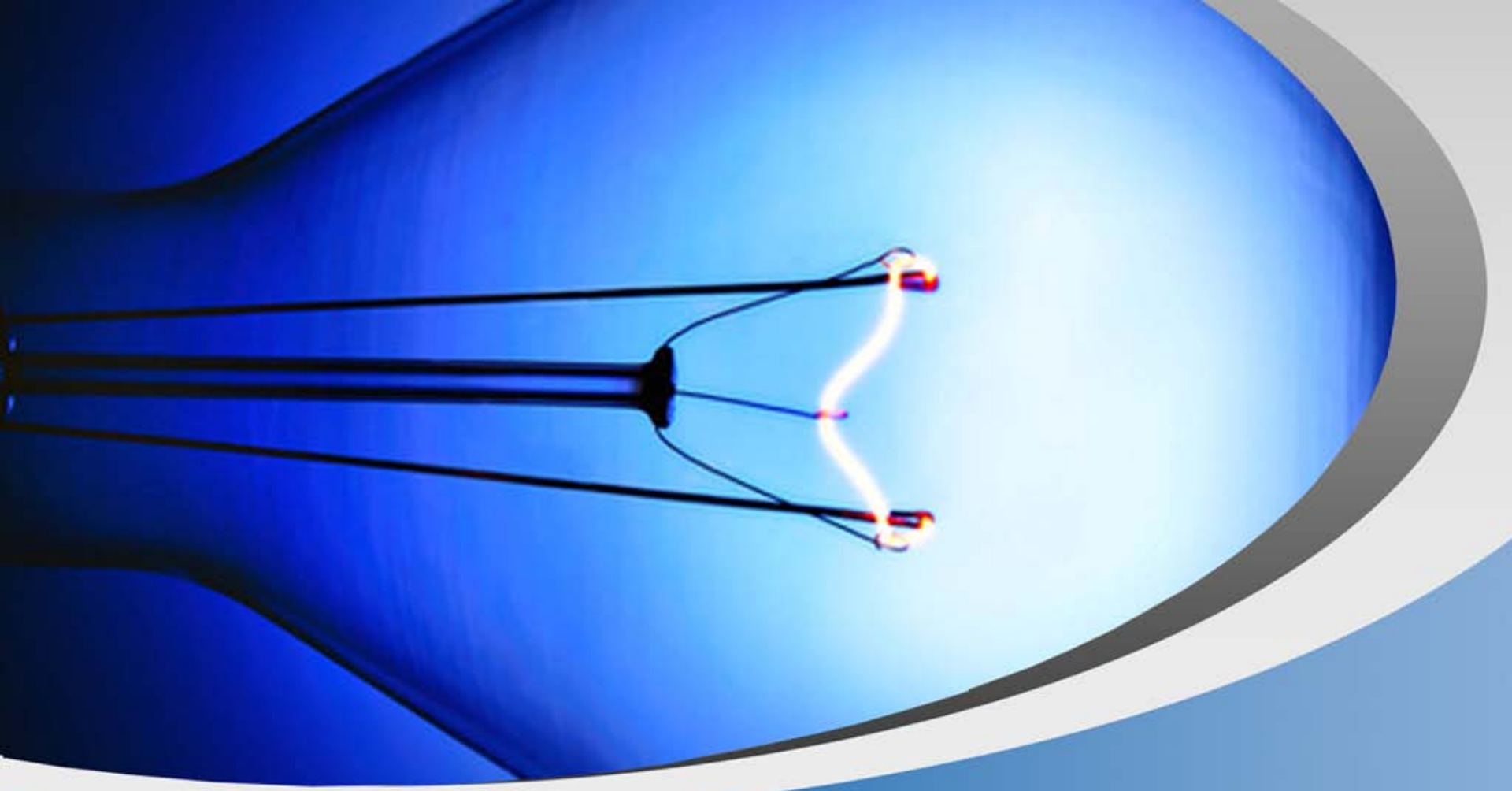
Start the adapter sequence – it will run continuously during all the UVM runtime phases

MITRE Results

- **14 unit and one system testbench**
  - 8 engineers over 6 months
- **Heavy stimulus reuse with abstract UVCs**
  - Easier to understand, create, and share
  - 2, 3 and 4 stacked UVCs
- **Full bottom UVC reuse between unit level**
  - Every unit had a different logical use
- **Reused unit environments entirely in system**
  - Modified connections for system topology

Conclusion

- **UVM User Guide provides no UVC (Agent) layering methodology**
- **Stacking UVCs == reusable UVCs and stimulus**
  - Simple API and implementation
- **Proven results with this and successive projects at MITRE and beyond**
- **Full example contributed on uvmworld.org**



UVM: Ready, Set, Deploy!

UVM

DVCon 2012
Design & Verification Conference & Exhibition

accellera
SYSTEMS INITIATIVE

OVM to UVM Transition

John Fowler

Fellow Design Engineer

Justin Refice

MTS Design Engineer

Problem

How does one transition a company which heavily uses OVM into using UVM, with the least amount of end-user pain?

Where We Were

- **Where we were:**
  - AMD has many interface, module, and system OVCs for multiple SoCs
    - Thousands of lines of code actively used in multiple real-world projects
  - The OVM Library is provided to the entire company from a central location
    - Version 2.1.2 + Bug fixes
    - Additional commonly reused elements available via an optional package. “The OVM Kit”
- **Where we’d like to be:**
  - UVM (Obviously!)
    - Register Library (RAL)
    - Resources
    - Industry Support

Conversion Issues - 1

- **Areas where the users can be bitten by a simple “O->U” conversion**
  - The UVM Library removed methods and parameters which were ‘deprecated’ within the OVM Library (Such as sequence constructor parameters).

```
function ovm_sequence::new(string name = "ovm_sequence",
                           ovm_sequencer_base sequencer_ptr = null,
                           ovm_sequence_base parent_seq = null);
```

~VS~

```
function uvm_sequence::new(string name = "uvm_sequence");
```

Conversion Issues - 2

- **Areas where the users can be bitten by a simple “O->U” conversion**
  - The UVM Library changed the ordering of parameters in non-deprecated methods

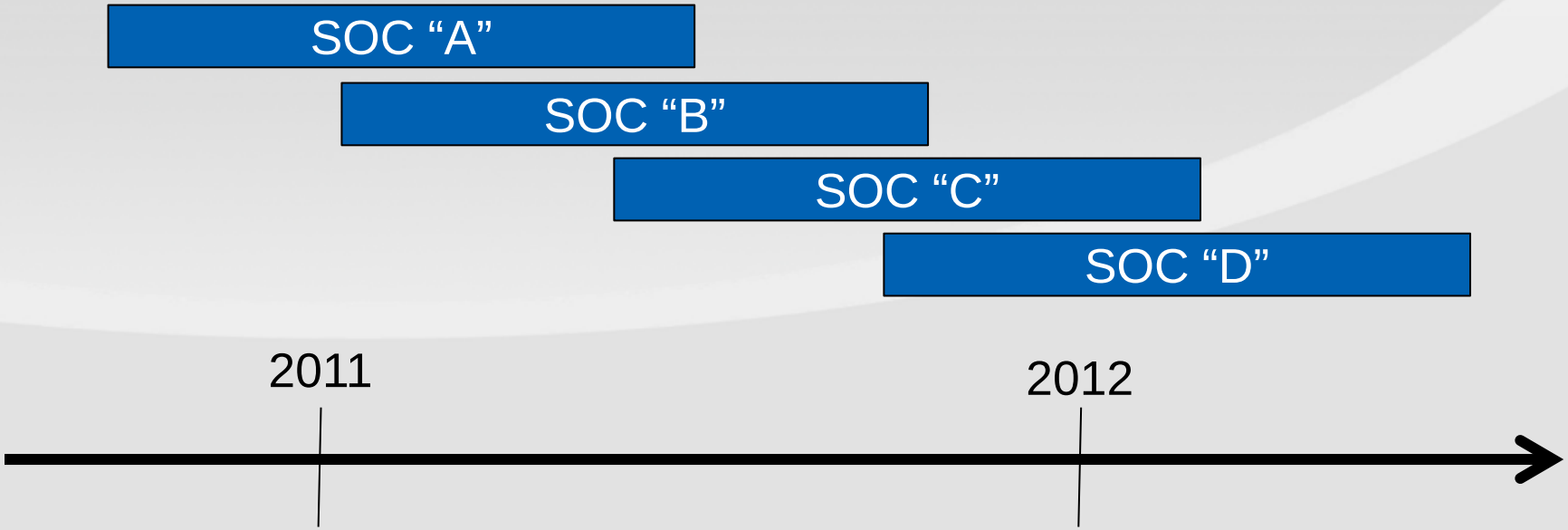
```
function ovm_objection::raise_objection(ovm_object obj = null,
                                        int count = 1);
```

~VS~

```
function uvm_objection::raise_objection(uvm_object obj = null,
                                        string description = "",
                                        int count = 1);
```

Conversion Issues - 3

- **Areas where the users can be bitten by a simple “O->U” conversion**
  - The sheer number of SoCs in flight makes it unrealistic to simply “convert” overnight



Co-existence

- **What about co-existence?**

  - UVM 1.0/1.1 can’t even be compiled with OVM 2.1.2 due to shared file names
    - base.svh, tlm.svh, etc.

  - The two libraries use different base classes
    - Connecting an OVM TLM port to a UVM TLM port would require specially written translation layers (for translating ovm_object derivatives to uvm_object derivatives)

Co-existence (cont.)

- **What about coexistence? (cont.)**

  - With 1.1a you _*can*_ create a “Walled Garden” approach
    - Limited re-use potential
    - TLM Translation work is “throw-away” work
    - Need to synchronize the OVM and UVM phases
      - Also “throw-away” work…

First Step

- **The First Step – Attack deprecation in user code**

  - A new version of the OVM Kit (2.1.2-TRANS) was created, with a modified version of the OVM Base Class Library.
    - Any deprecated method which is not available in UVM was removed
      - Sequence constructors were modified to only accept one parameter (name)
      - Deprecated functionality from the compatibility layer was removed
      - Etc.

OVM does not make this easy, as there is no `OVM_NO_DEPRECATED switch

First Step (cont.)

- **The First Step – Attack deprecation in user code (cont.)**
  - This cleaned up all of the user code in existing projects.
  - With rare exception, all of the projects were willing to move forward in the name of removing deprecation.

Code which runs on the new version was guaranteed to run on the old version… this allowed individual IPs to advance ahead of SoCs.

Translation

- **You're translatable (but we're not done yet!)**
  - Once users have gotten past this first step, their code can be translated to UVM using a script like the one provided inside of the UVM library
  - The script isn't perfect, but it's a great template for companies to use to create their own
    - Script catches raise_objection/drop_objection, but not raised/dropped/all_dropped callbacks
  - This still doesn't solve the issue of SoCs (and IPs) which are not able/willing to move to UVM right now.

Second Step

- **The Second Step – Whet their appetites for UVM**
  - A “UVM Backports” package is created, providing:
    - Resources (but NOT config db)
    - RAL (courtesy of a contribution on ovmworld)
    - Plus callbacks and a few other features…
  - IPs and SOCs which are not yet ready to fully deploy UVM can begin to see _*some*_ of the newer features
  - Hooks were written which locked backports users into the 2.1.2-TRANS version of the kit
    - You don’t get any of the new functionality w/o removing deprecated code from your environment
    - A spoonful of sugar helps the medicine go down…

Third Step

- **The Third Step – Heterogeneous Flows**
  - IP code is translatable, but IPs are delivering into multiple SoCs
  - Provide an “on the fly” translation script
    - Meant to be run when IP publishes into SoC
    - Translates IP code from OVM (2.1.2-TRANS) to UVM (1.1a)
    - IPs can run the script locally, just to make sure they _*really*_ are translatable

Third Step (cont.)

- **The Third Step – Heterogeneous Flows (cont.)**

  - SoCs get to decide whether or not they will:

    A. Stay at OVM until tape-out

    – No changes to SoC or IP Delivered Code

    B. Transition to UVM before tape-out

    – SoC runs UVM, while the IPs are OVM->Translated

    C. Start at UVM

    – SoC and New IPs are UVM, old IPs are OVM->Translated

Speed Bumps

- **Speed Bumps…**

  - It turns out that some code in OVM, which was perfectly legal and non-deprecated, simply will _*not*_ work when translated to UVM

  - Yes **`ovm_phase`**, we're looking at you…
    - Working with the end-users, an objection-based phasing mechanism was developed, which would work in both standard OVM, and translated "O->UVM".
    - In the future, additional work will be required to migrate this mechanism to the UVM phasing mechanism

Speed Bumps (cont.)

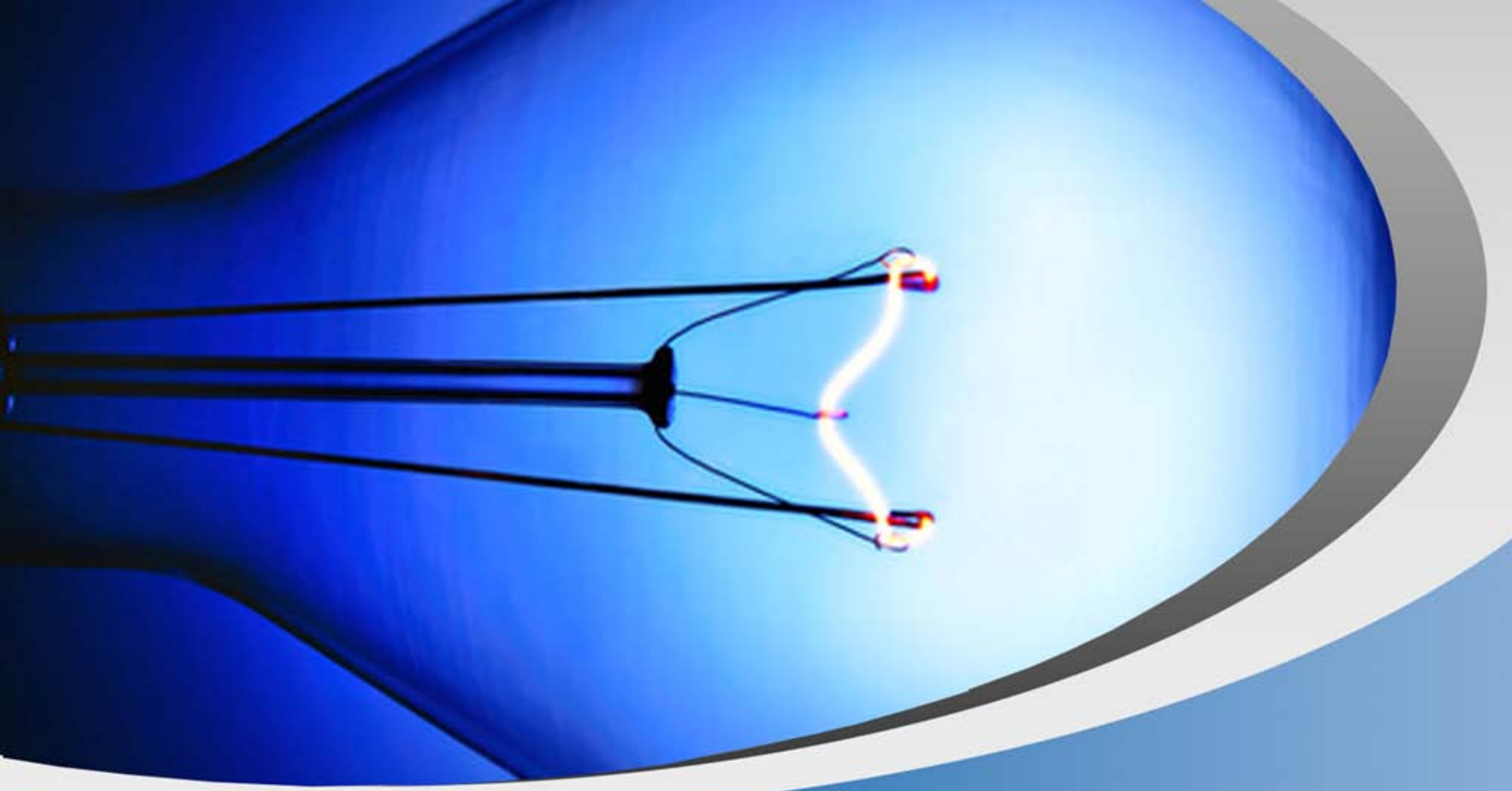
- **Speed Bumps… (cont.)**
  - Where'd the run phase go?
    - Unless an OVM test was objecting to ovm_test_done within one non-blocking assignment region of **`ovm_component::run`**() starting, the translated code will get killed by the UVM phasing system.
  - Provide a "run-phase prolonger" inside the UVM Kit
    - Automatically raises an objection to the run_phase ending
    - Drops the objection once it sees any other objections get raised
    - Basically makes uvm_test_done work like ovm_test_done
    - Cleaner than using +UVM_USE_OVM_RUN_SEMANTIC
      - Don't need to modify testbench run scripts…

Final Step

- **The Final Step – Enjoy your UVM environment!**
  - Eventually, all SoCs will be UVM, either via transition, or having simply started as such
  - New IPs for SoCs will UVM from scratch
  - Old IPs which are only delivered into UVM SoCs can be converted locally using the same script

Thank You!

If you would like to download a copy of the presentation slides in PDF format, click the Attachments link at the top of the presentation viewer.



UVM: Ready, Set, Deploy!

UVM

DVCon 2012
Design & Verification Conference & Exhibition

accellera
SYSTEMS INITIATIVE

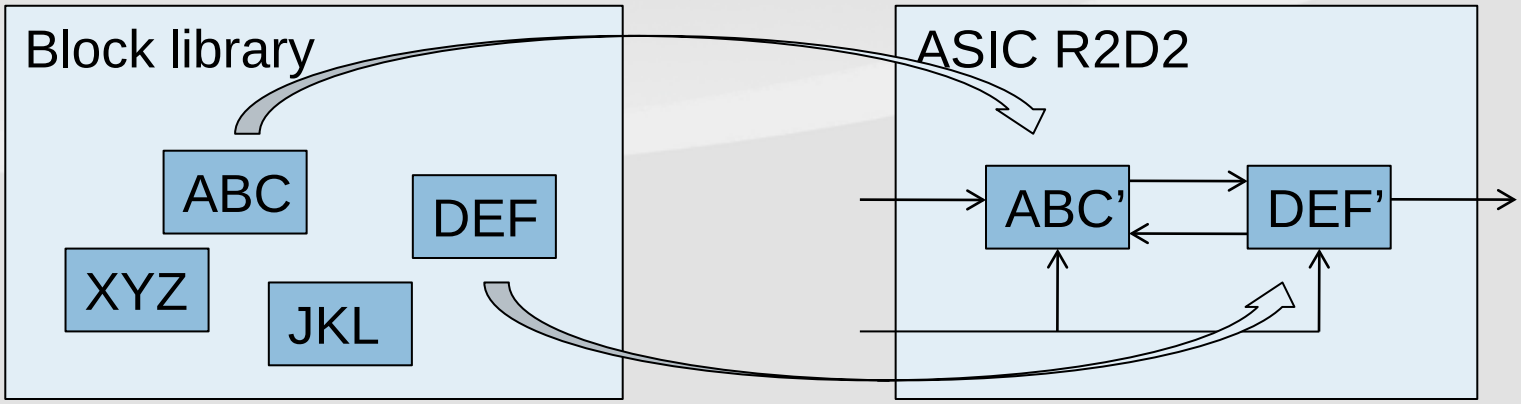
VC Building Blocks with UVM

Mark Strickland

Verification Technical Lead

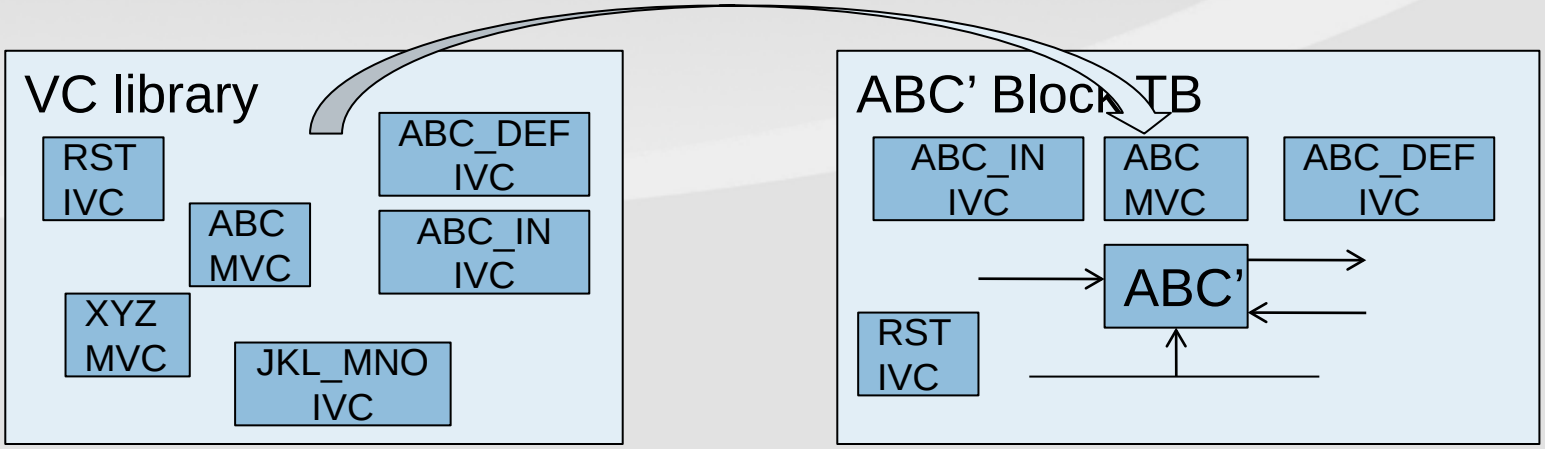
Design Paradigm

- **We do a number of ASICs with the same architecture but different performance levels.**
- **To efficiently support this design, we create a library of RTL blocks and then construct an ASIC out of this library with parameterization or modification.**



DV Matching Paradigm

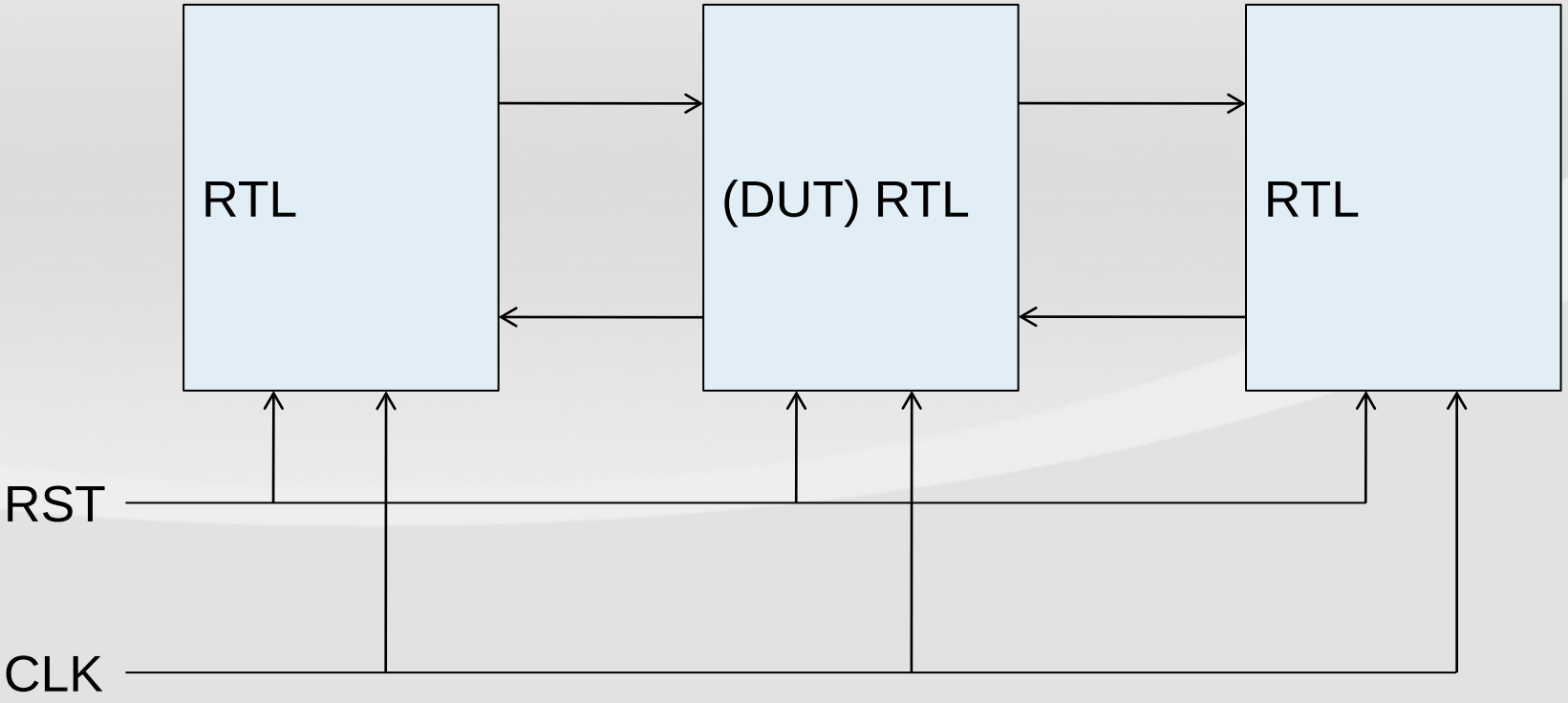
- DV has a library of interface verification components (IVC) and module verification components (MVC)
- A Testbench is constructed by instancing / connecting / customizing some IVCs and MVCs from the library
- Testcases are written by a combination of constraining or setting config and choosing or adding sequences



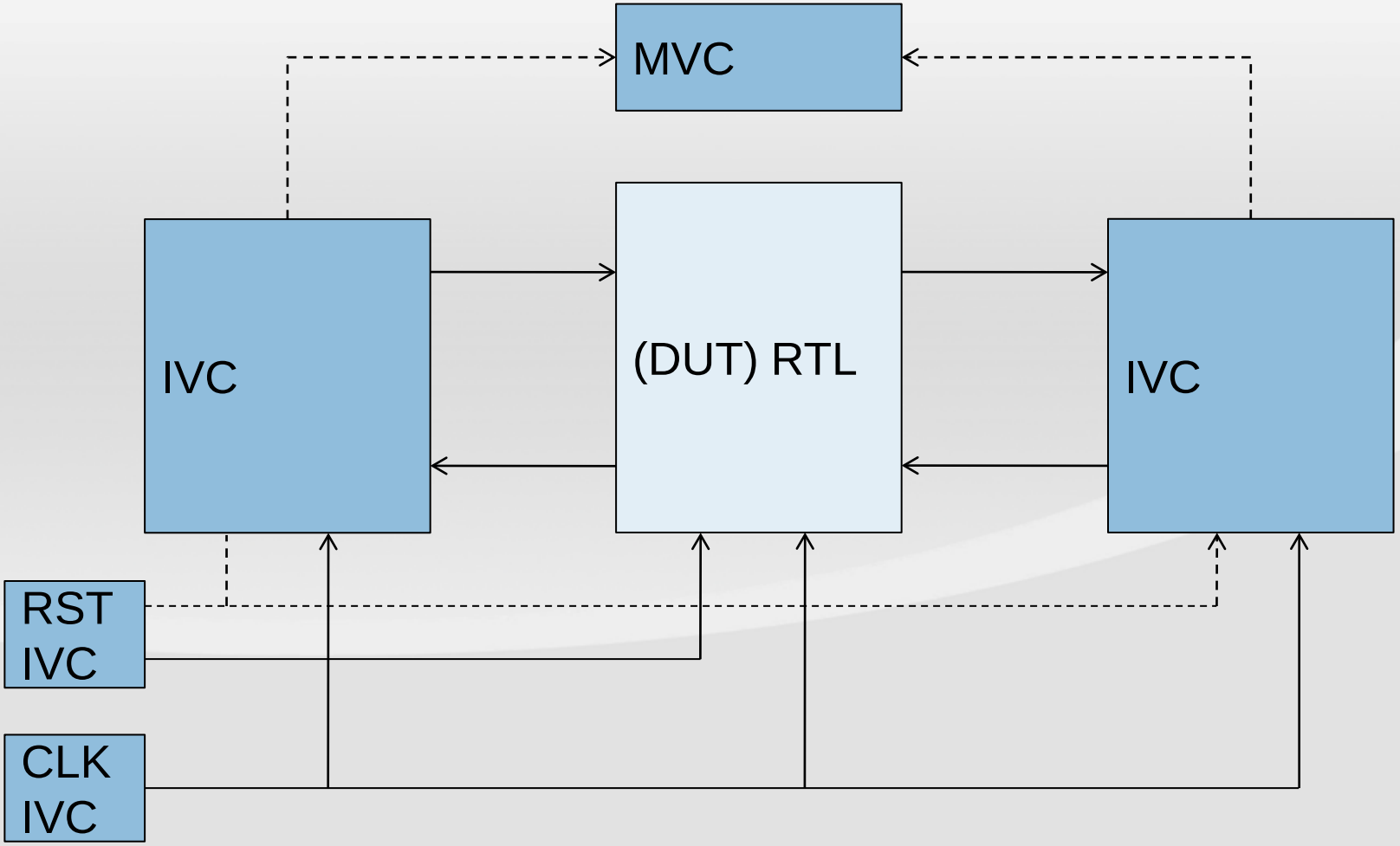
Requirements To Support Paradigm

- **These are the requirements that the presentation will cover**
  - Flexible IVC / MVC structure to match possible DUT topology
    - Want to avoid multiple instance statements for each VC
  - A standard checking scheme that can be easily set up and/or changed to match DUT behavior
  - A means to ensure coherency among configuration of library elements
  - A standard means to synchronize behavior of library elements

Our Typical DUT Block Environs

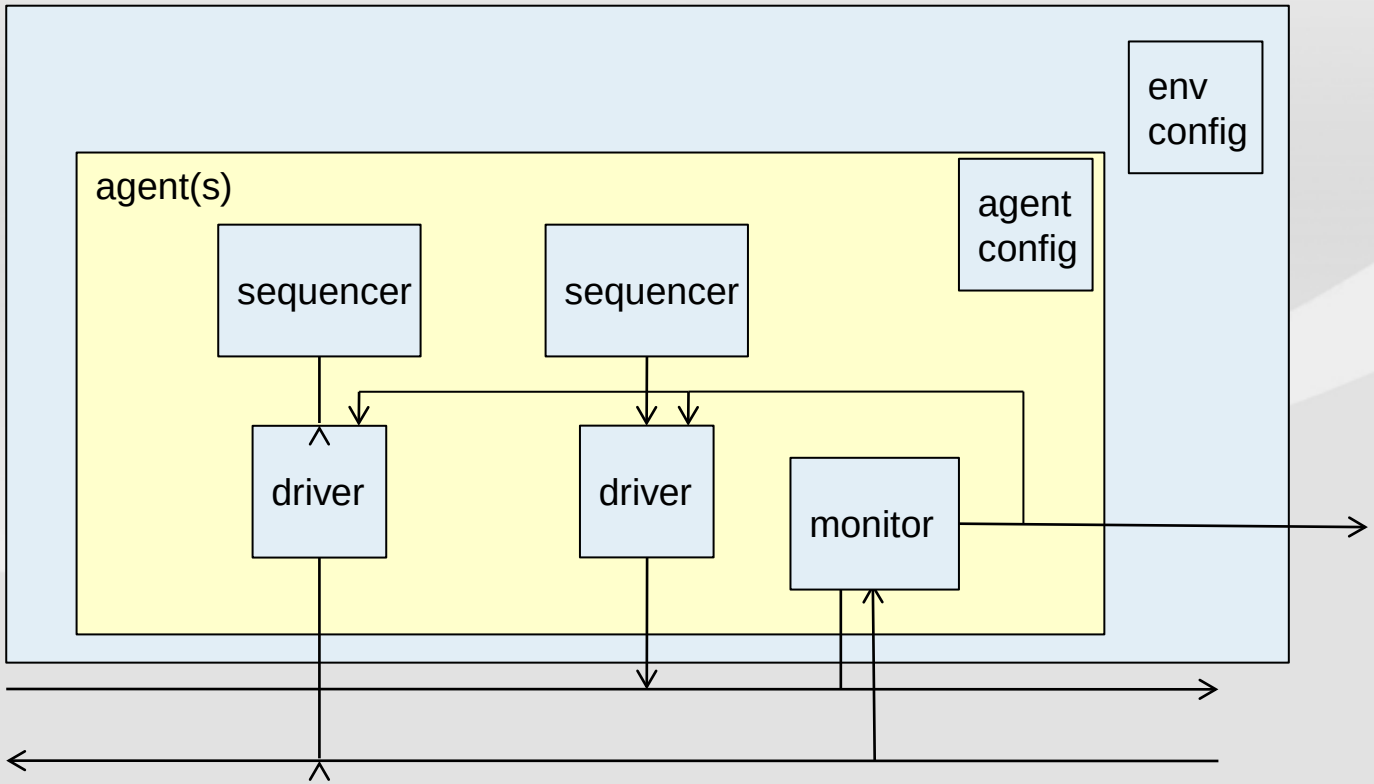


Corresponding Testbench Elements



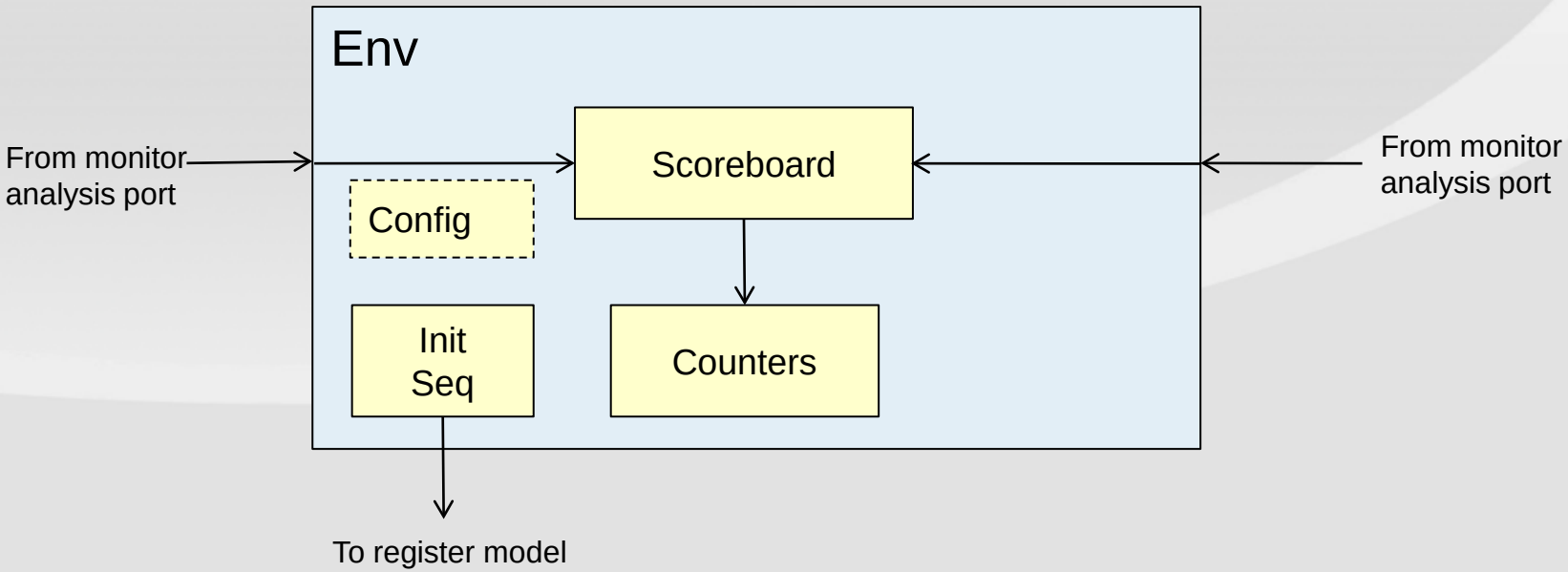
Inside IVC

- **Two sets of driver/generator**



Inside MVC

- **Checking and initialization for corresponding block**



Code Template Engines

- If engineers code IVC and MVC from scratch, they will look different in subtle ways, reducing the ease of using library elements
- Inheritance does not provide a full solution because there will be different configurations required
- A code generator template provides the initial code for all VCs and testbench
- Template toolkit http://template-toolkit.org

Sample IVC Template Code

```
function void myivc_agent::build_phase(uvm_phase phase);
  super.build_phase(phase);

  // Get a handle to virtual interface from the config db
  if(!uvm_config_db#(virtual my_intf)::get(this, "", “my_intf", sigs) ||
     (sigs == null))
    `uvm_fatal("CFGERR", "Virtual interface (my_intf) not set“)

  if(m_cfg.m_active == UPSTREAM) begin
    seqr =`csco_create_child(myivc_sequencer,"seqr",m_cfg);
    drv = `csco_create_child(myivc_driver,"drv",m_cfg);
    // Pass virtual interface to driver and sequencer (for clock)
    uvm_config_db#(virtual my_intf)::set(this, "drv", “my_intf", sigs);
    uvm_config_db#(virtual my_intf)::set(this, "seqr", “my_intf", sigs);
end

  if(m_cfg.m_active == DOWNSTREAM)
    `uvm_fatal("NOIMPL", "this agent does not have DOWNSTREAM functionality");

   if(m_cfg.m_active != DISABLE) begin
      // Create and configure monitor
      mon = `csco_create_child(myivc_monitor,"mon",m_cfg)
      uvm_config_db#(virtual my_intf)::set(this, "mon", “my_intf", sigs);
   end

endfunction : build_phase
```

Standard name for interface

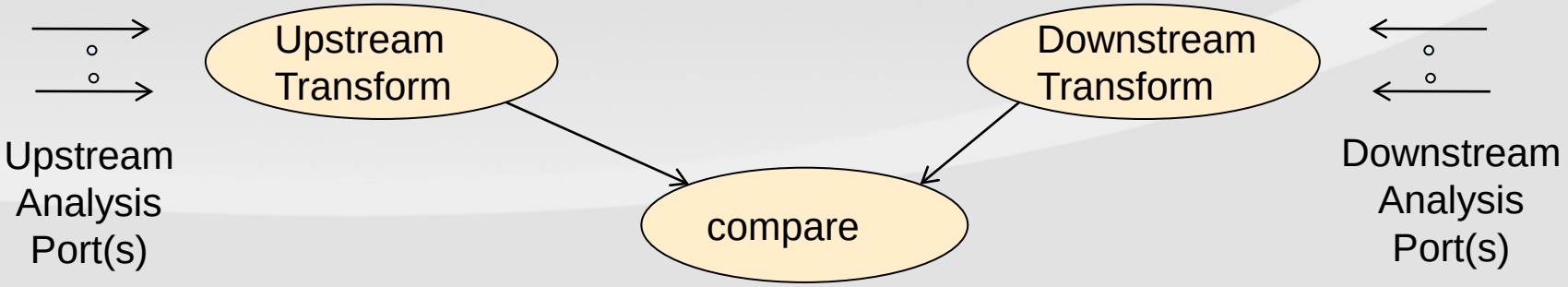
Option: list of drivers

Ensure sequences have a standard path to clock

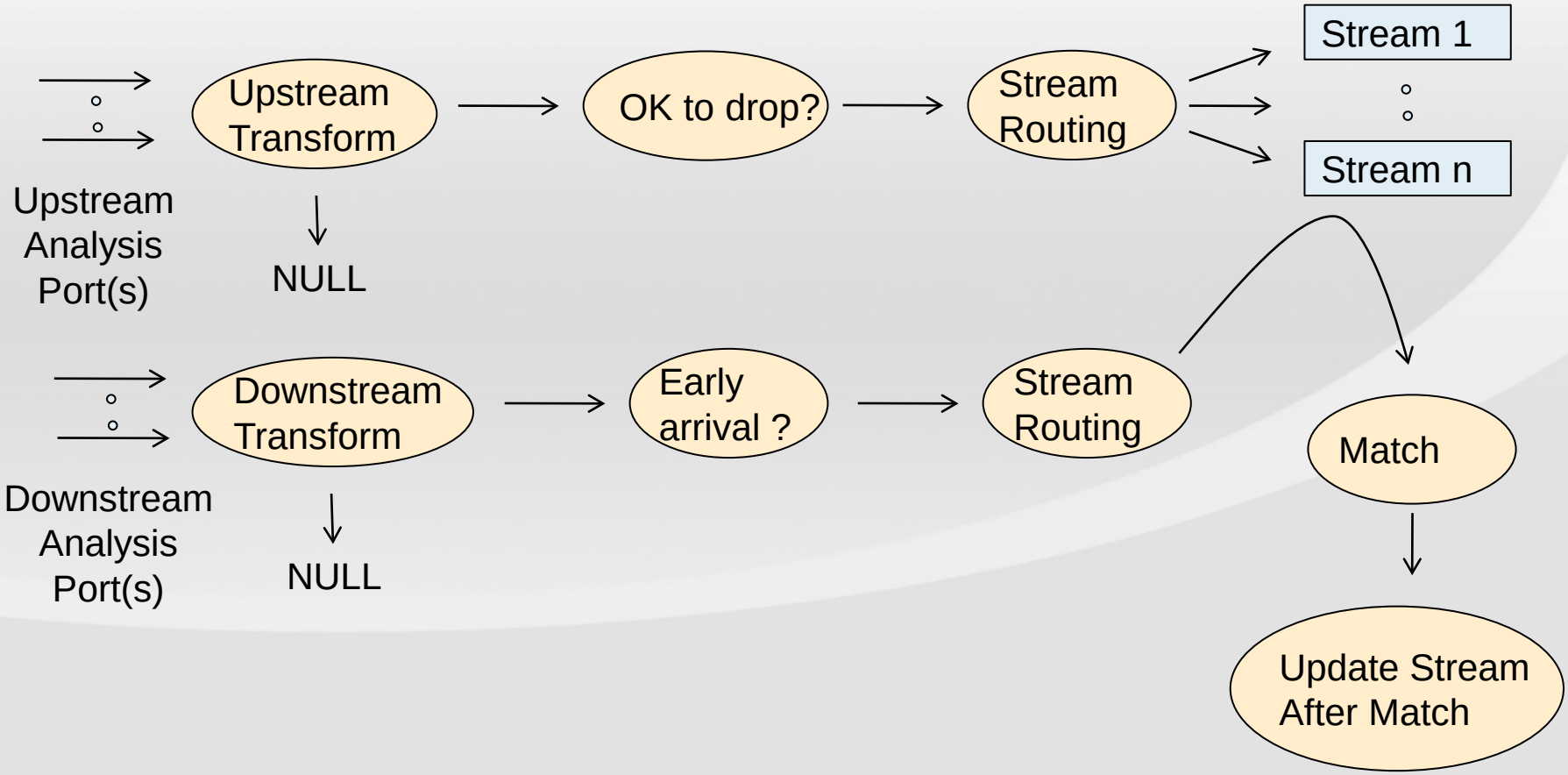
Enforce use of common config

Standard Checker

- **Most of our blocks take a packet, modify it, pass it on**
- **This DUT functionality lends itself to scoreboard**
- **For people to be able to modify an existing scoreboard in the library, we need a standard structure**



Scoreboard Virtual Functions



Modifying the Scoreboard

- **Example**

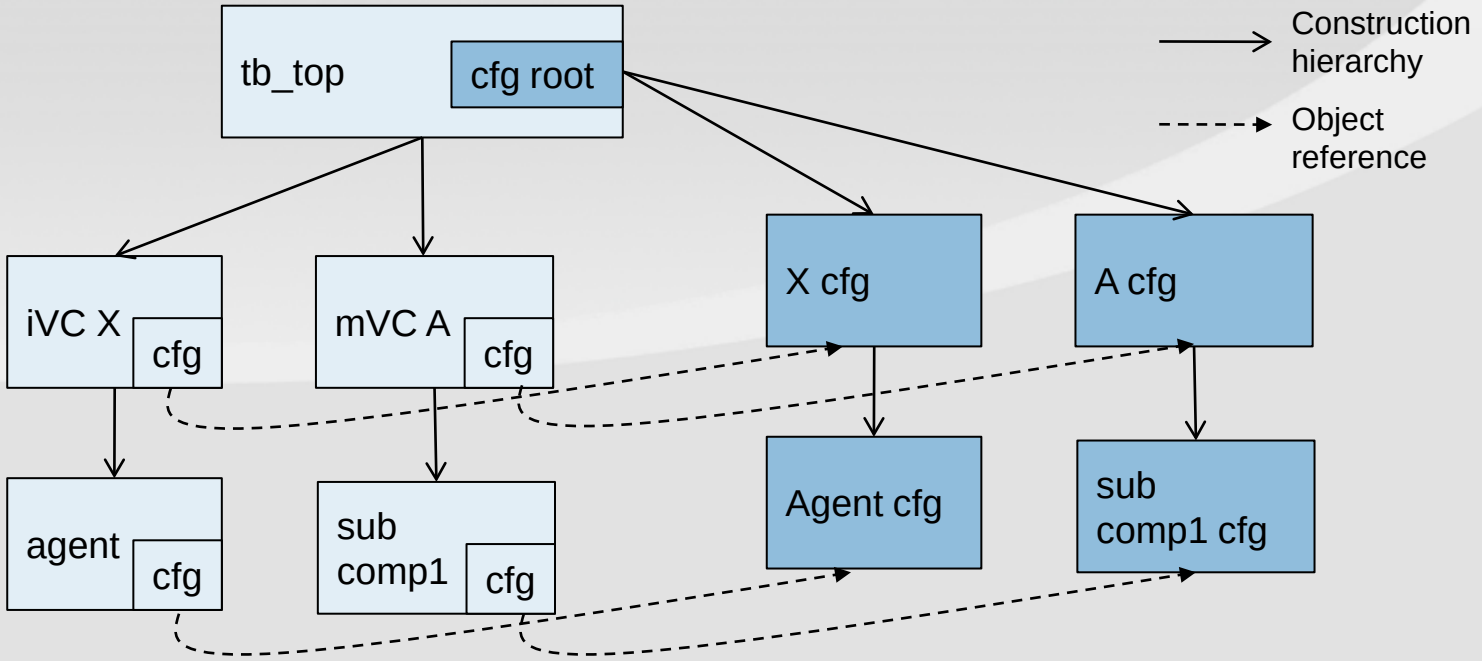
```
function my_pkg::my_data my_sb::ds_transform(
           my_pkg::my_data ds, int unsigned port_num);
  //super behavior is return d
  // instead, for this dut, modify data based on port_num
  my_pkg::my_data result = new ;
  result.copy(d);
  result.modify_by_port_num(port_num);
  return result;
endfunction
```

Config Coherence Problem

- Each element in the library needs its config, but when multiple elements are put together some of the config values must be coherent.
- It would be error prone to have each test maintain coherence.
- Need a way to specify coherency constraints in the testbench.

Tree of Config Structs

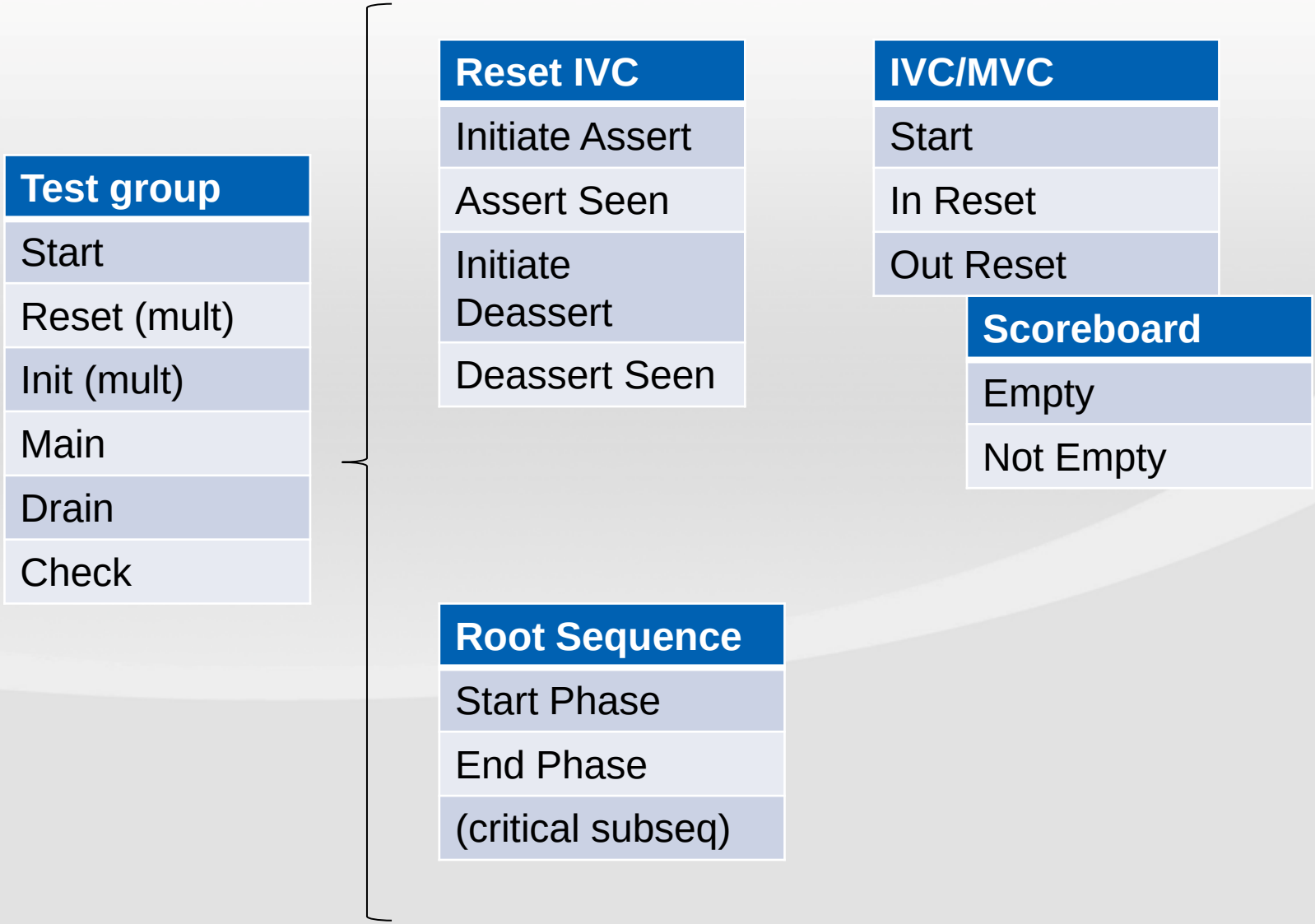
- **During build phase, entire configuration tree is generated in single randomize step. When each component is built, it establishes a reference to its configuration instance in the tree.**



Configuration Implementation

- **When parent instances child in build_phase**
  - my_type::type_id::create(name,this)
  - uvm_config_db#(csco_cfg)::set(this, name, “cfg”, config_obj)
- **In component base class build_phase**
  - If (!uvm_config_db#(csco_cfg)::get(this, “”,“cfg”,m_cfg) begin
    // top level, must create and randomize
    end
- **Optionally mid-simulation, the full tree or a branch could be re-randomized**
  - If a branch, set rand_mode(0) for all other branches
  - top_tb.cfg.randomize();

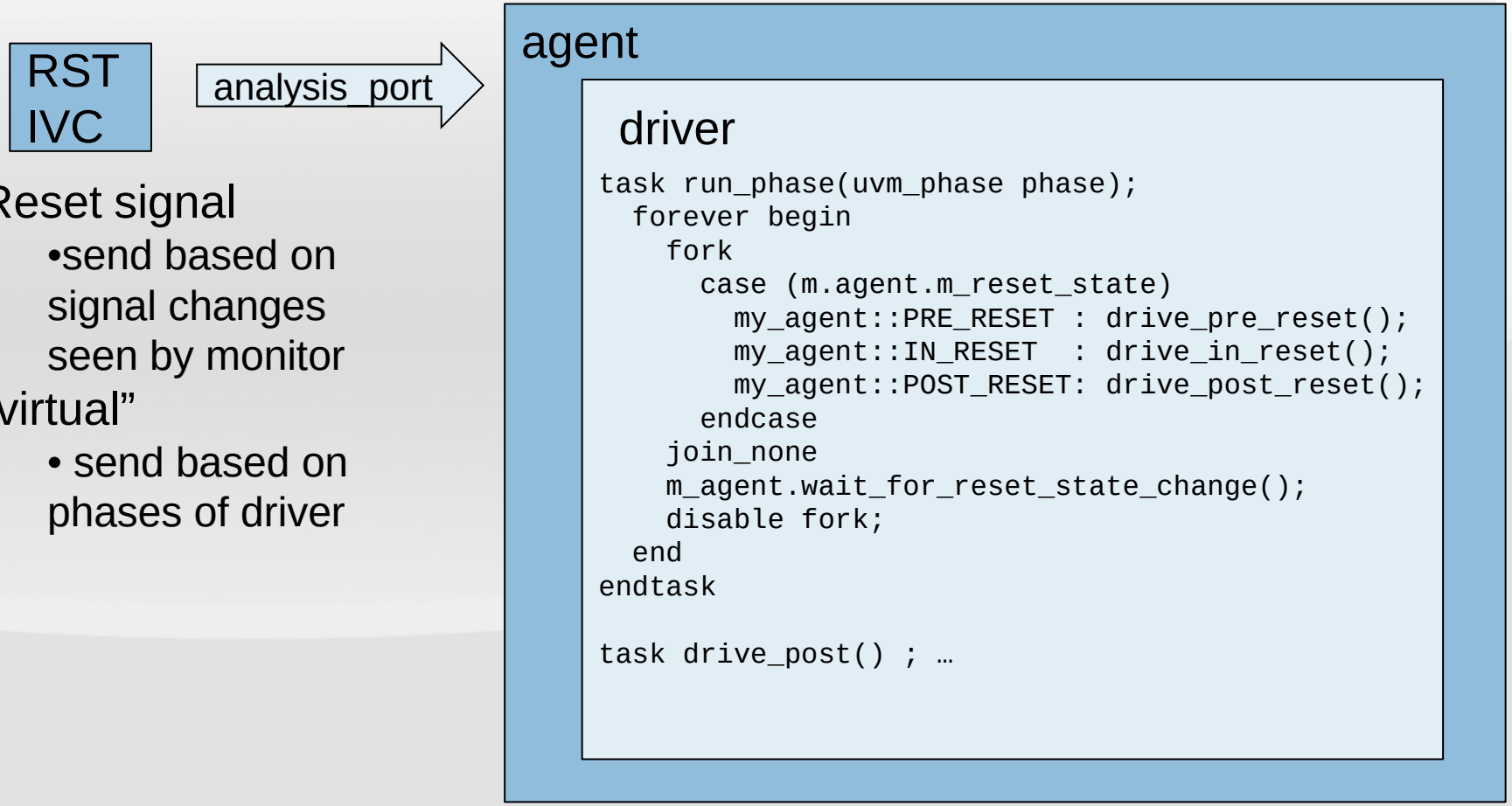
Element Synchronization



Synchronization Implementation

- **Reset IVC broadcasting pre-/in-/post-reset, components calling appropriate tasks**
  - One reset IVC per reset domain
- **UVM run-time phasing for stimulus and scoreboard objections**
  - Chosen over virtual sequences due to ability to have integration-level code only for exceptional VCs
  - Added user-defined phases for project-specific reset and config staging
  - Avoided hard-coding phases in clients
    - Each config has variables for relevant phases (default or constrained values)
    - Each element uses phase_started() to compare phase to config phase, call task, etc.

Reset Status



- Reset signal
  - send based on signal changes seen by monitor
- "virtual"
  - send based on phases of driver

Representative Phasing

Phase	Test	Reset Driver	CSR Seq	Data Seq	MVC
pre_reset	↕				
reset		All assert			↕
post_ reset		All deassert	(↕)		
configure			↕		
main	↕		Back-ground	↕	
shutdown			Back-ground		↕
post_ shutdown			↕		

Phasing Workarounds

- **Sequences are not phase-aware**
  - No virtual sequences working on sequencers in different phase domains
- **Phasing process control is not careful with sequences**
  - Terminating sequences with extra code to avoid handshake lock-up
    - Introduced our own csco_sequence and csco_sequencer classes
    - Require sequences that span phases to start in run_phase, so we can re-start them
- **Jump action is immediate, not a request to get ready**
  - Added user phase to allow “ready for restart”

Summary

- **To support “library element”-based testbench creation**
  - Flexible IVC / MVC structure to match possible DUT topology
    - Want to avoid multiple instance statements for each VC
  - A standard checking scheme that can be easily set up and/or changed to match DUT behavior
  - A means to ensure coherency among configuration of library elements
  - A standard means to synchronize behavior of library elements