Understanding the UVM m_sequencer, p_sequencer Handles, and the `uvm_declare_p_sequencer Macro

Cliff Cummings - VP of Training
Paradigm Works

Life is too short for bad or boring training!

Agenda

- Sequences run on sequencers ← How sequences are started
- Sequence-related UVM base classes ← `m_sequencer` inheritance
- The *sequence*`.start()` method ← What does the `start()` method do?
- `` `uvm_declare_p_sequencer `` macro ← What does this macro do? Why does it exist?
- Typical **`vsequencer`** example ← Using `` `uvm_declare_p_sequencer `` and `p_sequencer`
- Using **`uvm_resource_db`** API from a sequence ← Efficient resource access API

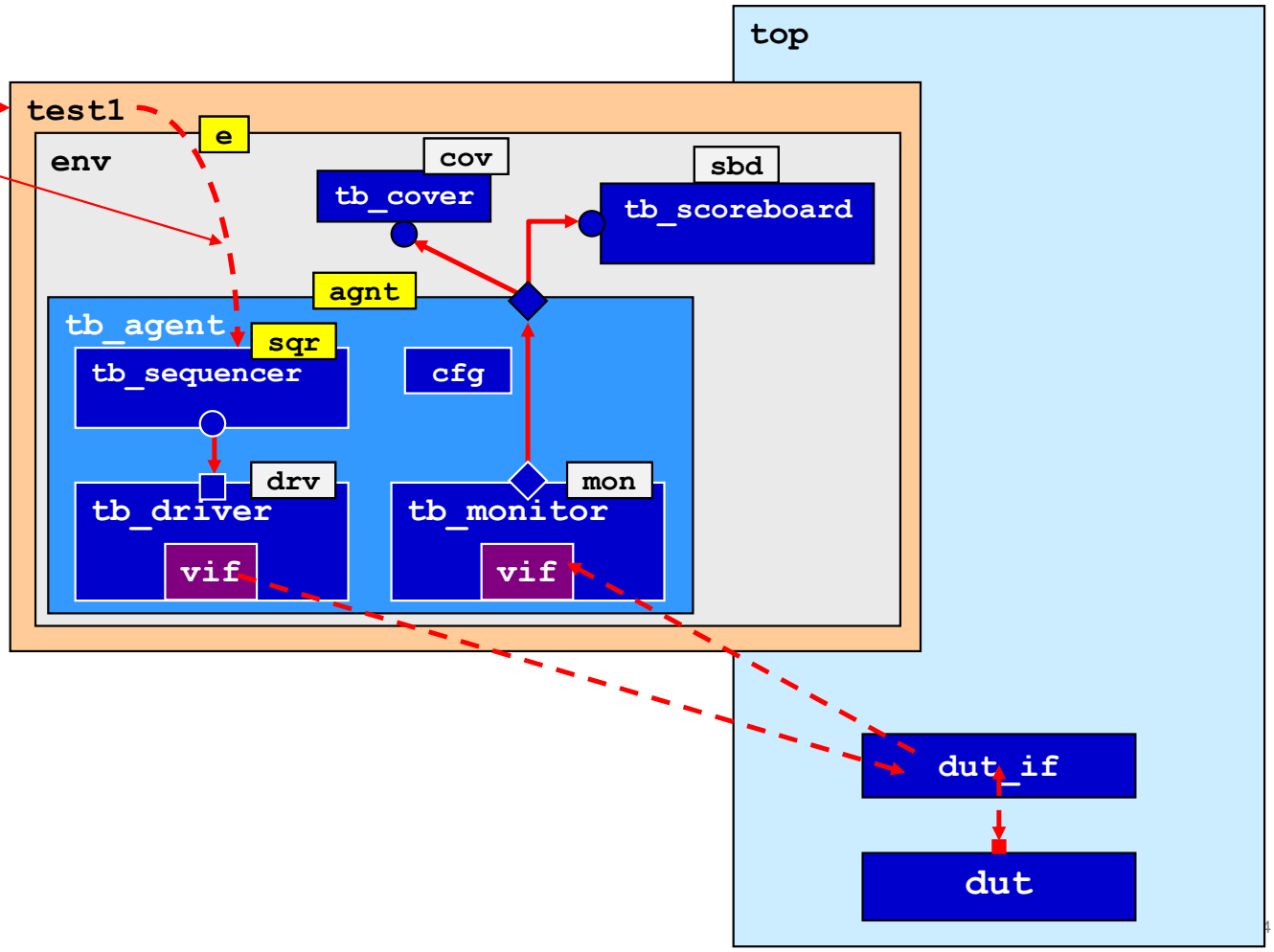
More examples and details in the paper

Sequences Are Started on Sequencers

Tests start sequences on a sequencer
Example: `seq.start(e.agnt.sqr)`

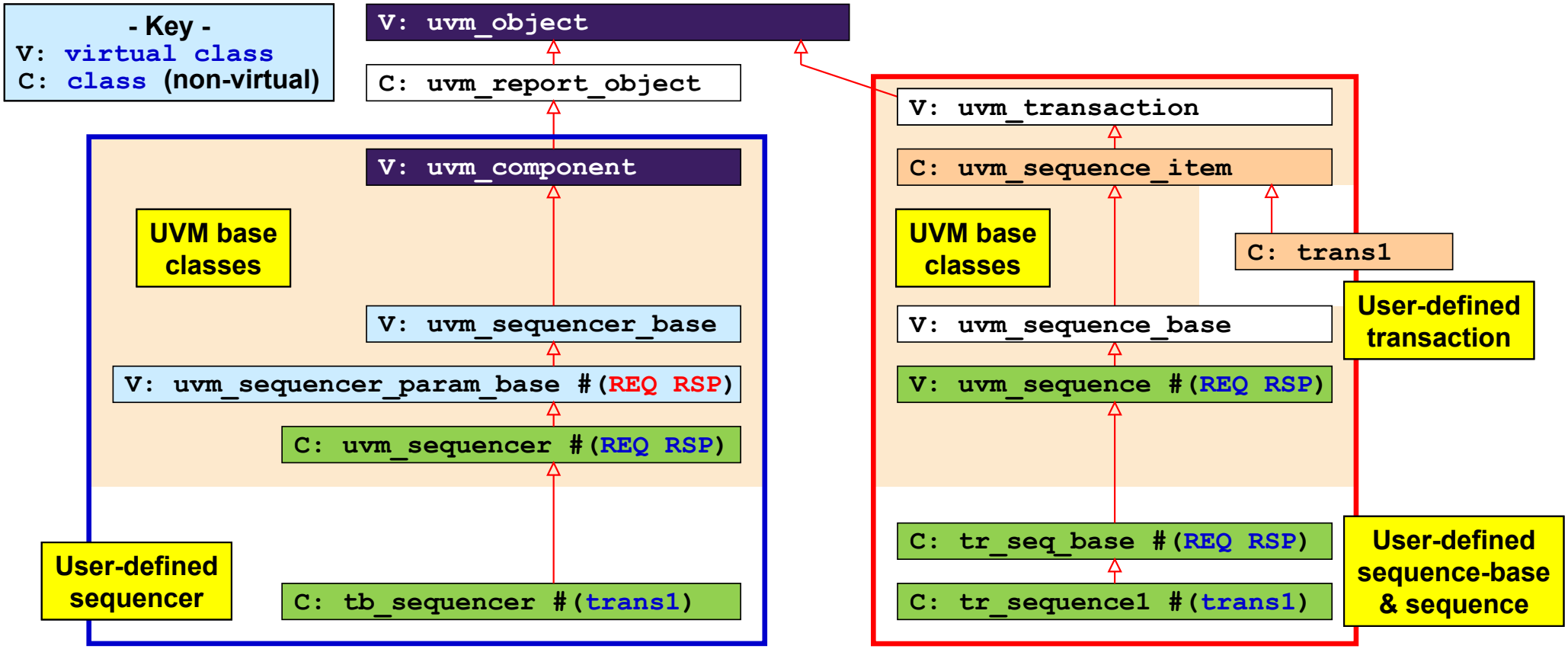
Sets the `m_sequencer` handle *in the sequence*

The sequence now has a handle to the sequencer where it is running



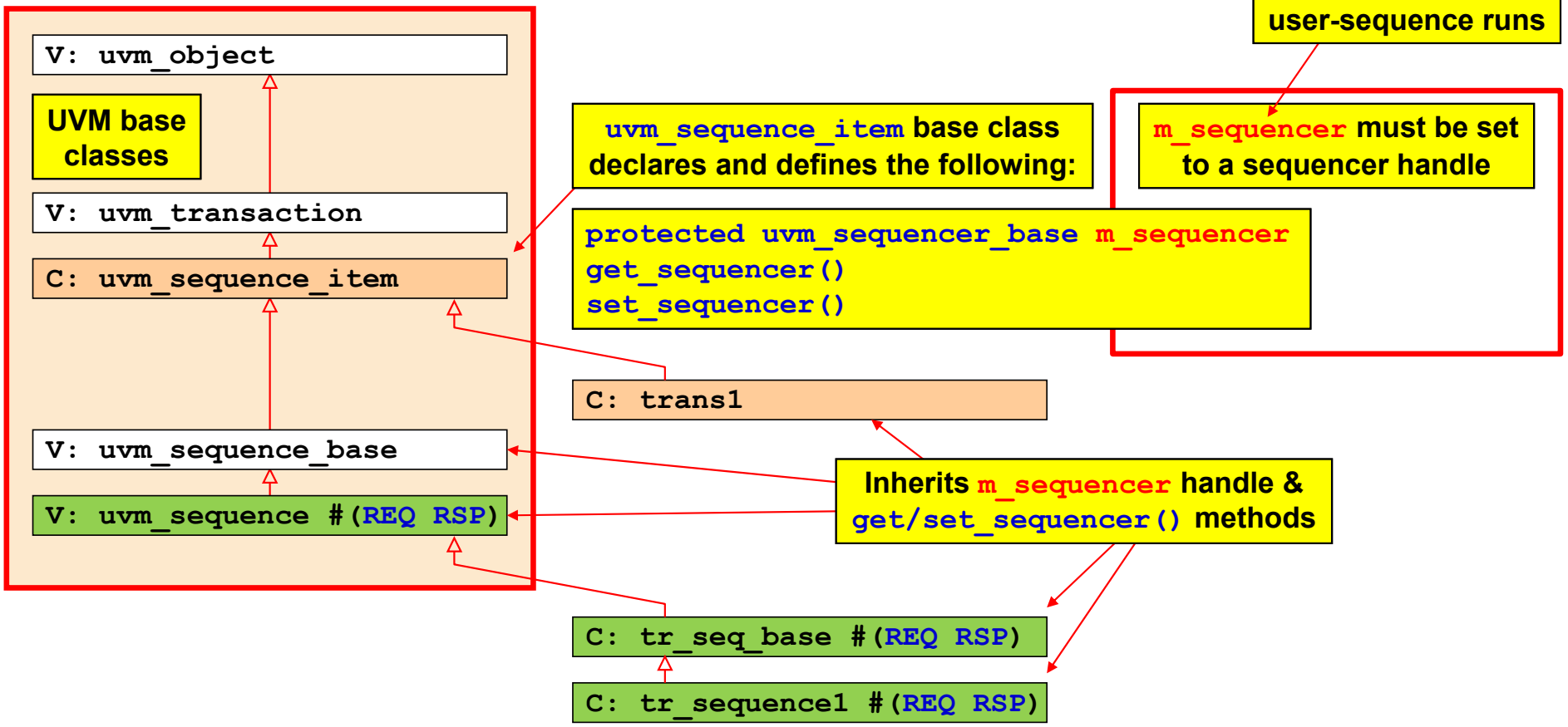
User Sequencer, Sequences, Transactions

Derived from UVM Base Classes



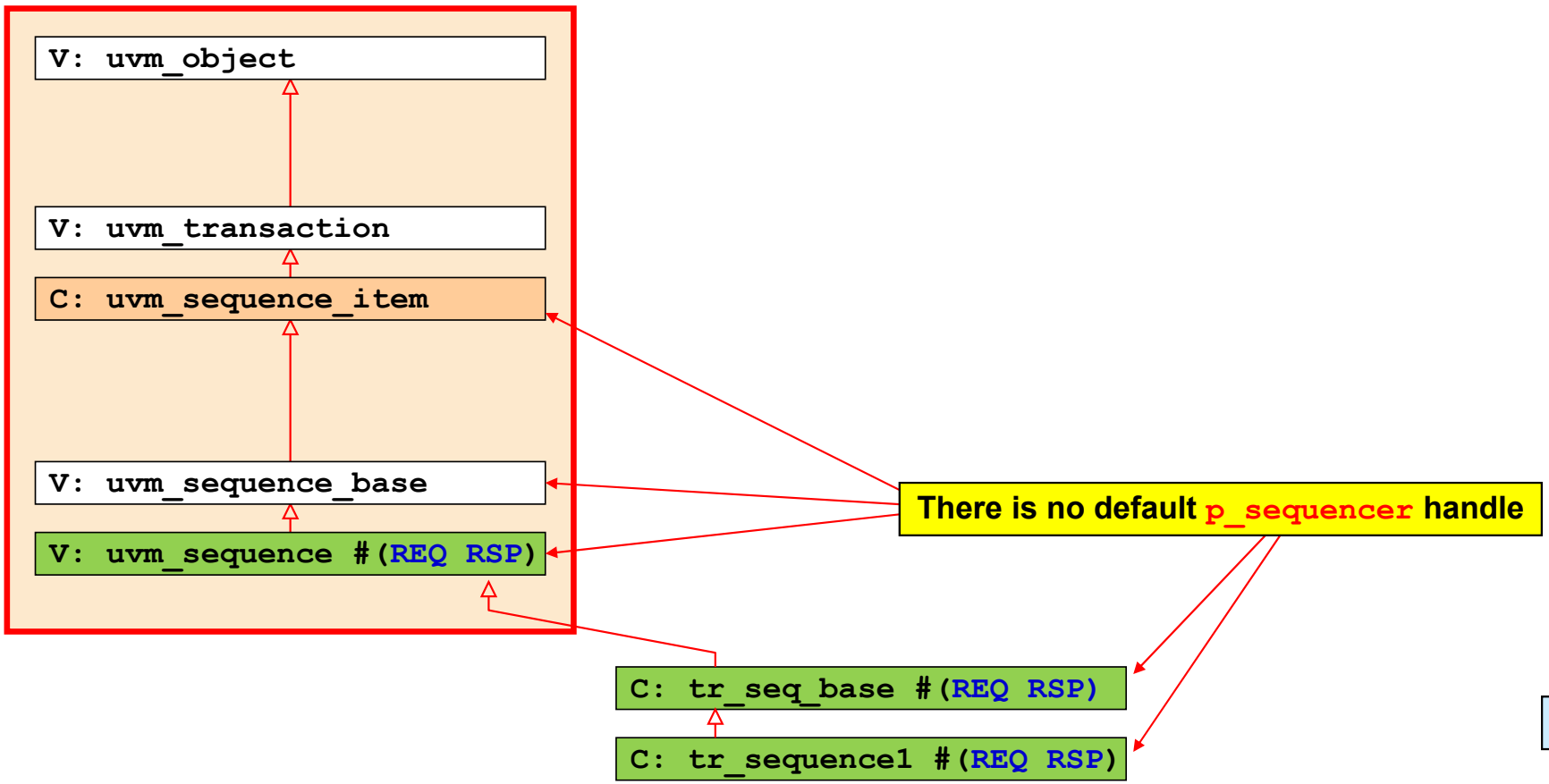
User Transactions and Sequences

m_sequencer, set_sequencer() & get_sequencer()



Sequences Must Run on the Correct Sequencer

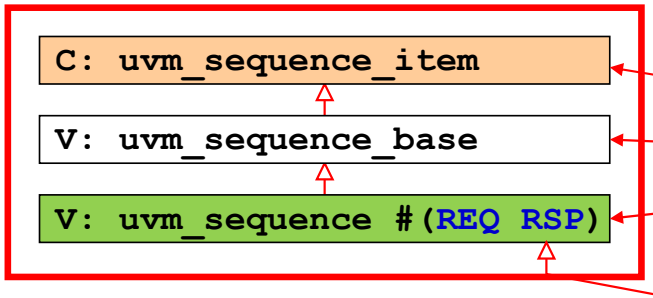
m_sequencer, `uvm_declare_p_sequencer, p_sequencer



Next slide

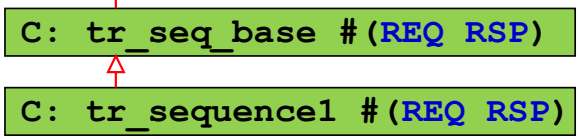
Sequences Must Run on the Correct Sequencer

m_sequencer, `uvm_declare_p_sequencer, p_sequencer



There is no default `p_sequencer` handle

Sequence base classes call `` `uvm_declare_p_sequence(SEQUENCER) `` macro to create and test a `p_sequencer` handle



`$cast(p_sequencer, m_sequencer)` *tests* to ensure `p_sequencer` & `m_sequencer` point to the same sequencer

Frequently used with virtual sequencers

If a virtual sequence is pointing to the wrong virtual sequencer, test-aborting `null` pointer errors can occur *(difficult to debug)*

Virtual Sequences

Virtual Sequencer Technique

Good reference paper:
Using UVM Virtual Sequencers & Virtual Sequences
www.sunburst-design.com/papers/CummingsDVCon2016_Vsequencers.pdf

UVM Virtual Sequence - vsequencer

- A **`vsequencer`** component declares the subsequencer handles

The environment copies the subsequencer handles to point to the real subsequencers

- A **`vseq_base`** class calls the **`` `uvm_declare_p_sequencer ``** macro

The macro creates a `p_sequencer` handle that points to the `vsequencer`

- The **`vseq_base`** uses the **`p_sequencer`** handle

To copy the subsequencer handles from the `vsequencer` to the `vseq_base` class

- The disadvantages of this technique?
  - It adds a **`vsequencer`** component to the top-level environment
  - The **`vsequencer`** component is sometimes a dumping ground

Misguided verification engineers sometimes store "*stuff*" in the `vsequencer`

- This technique is described in Cliff's DVCon 2016 paper

Virtual Sequencers & Sequences

Requirements Overview

- Create a virtual sequencer
  - Declare subsequencer handles
- Environment copies subsequencer handles

Copy real subsequencer handles to the subsequencer handles declared in the virtual sequencer

Testbench structure

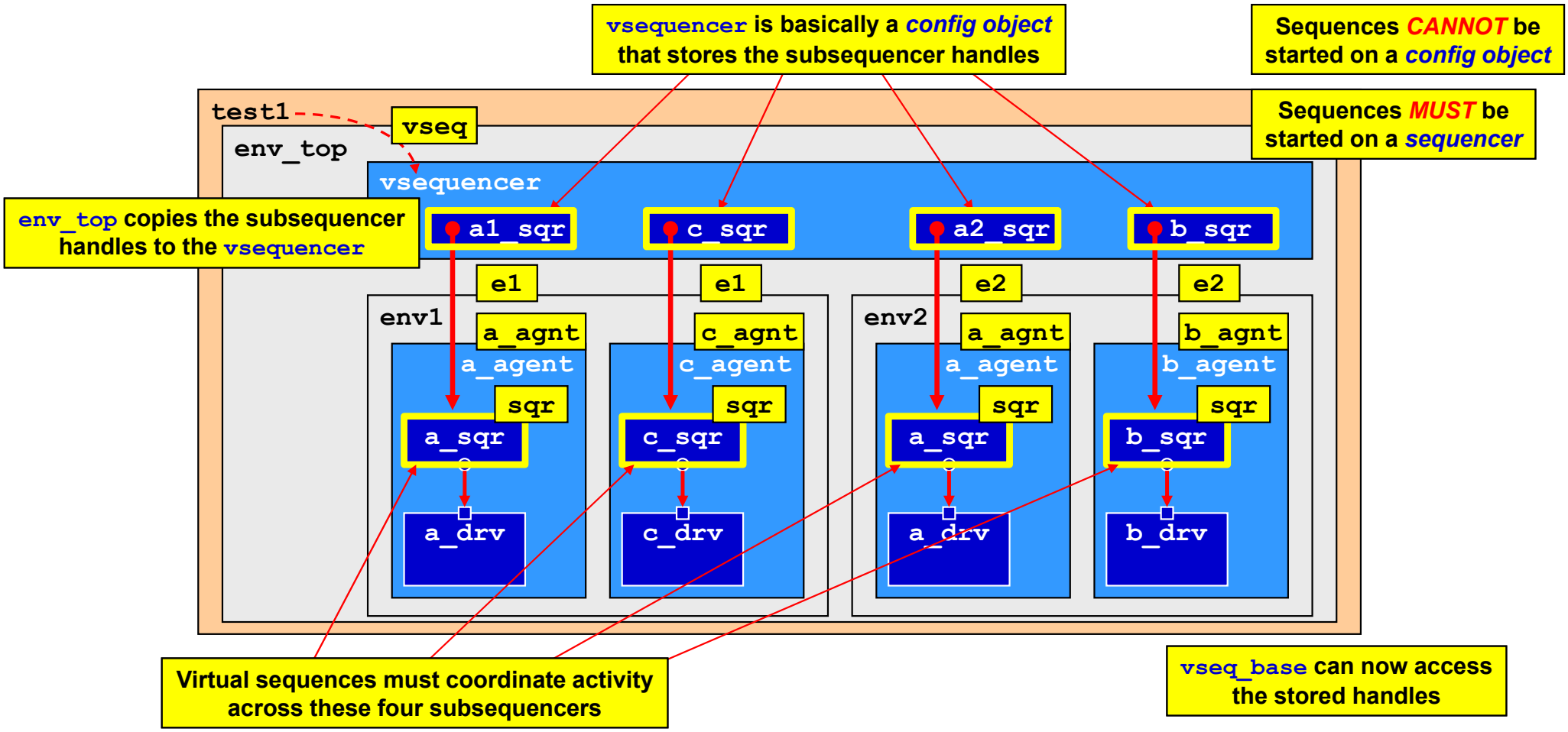
- Create a **`vseq_base`** class
  - Use **`` `uvm_declare_p_sequencer() ``** macro to set vsequencer handle
  - Use the **`body()`** task to set the subsequencer handles
- Create virtual sequences by extending **`vseq_base`**
  - Start sub-sequences on subsequencers
  - Coordinate execution order of sub-sequences
- Test starts virtual sequences on the virtual sequencer

Virtual sequence base class

Sequences & tests

UVM Virtual Sequence - vsequencer

Block Diagram



vsequencer Component


```
class vsequencer extends uvm_sequencer;
  `uvm_component_utils(vsequencer)

  a_sequencer a1_sqr;
  a_sequencer a2_sqr;
  b_sequencer  b_sqr;
  c_sequencer  c_sqr;

  function new(string name, uvm_component parent);
    super.new(name, parent);
  endfunction
endclass
```

`vsequencer` is a wrapper-class that declares (*holds*) the subsequencer handles

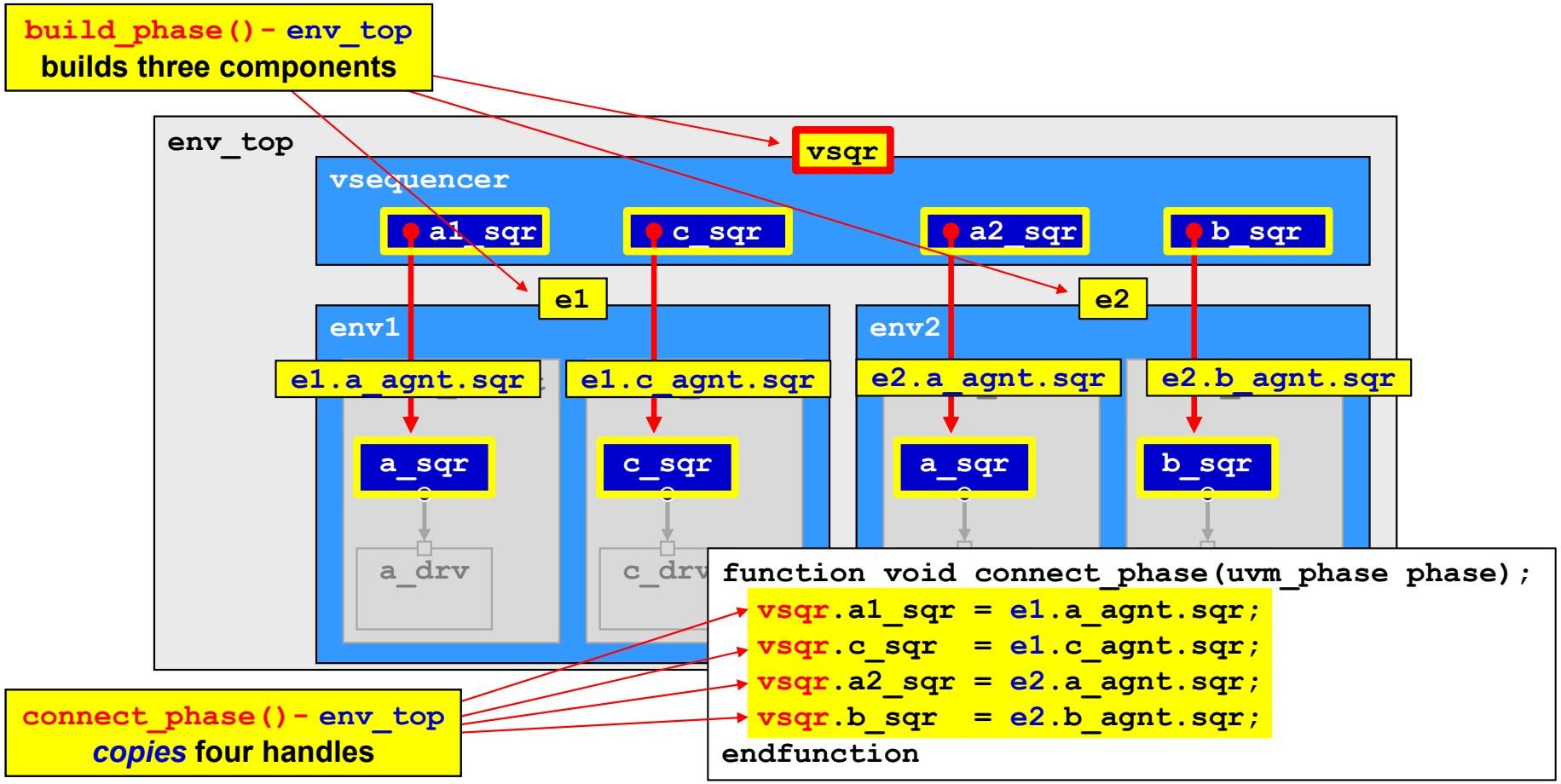
This is basically a *config object* to hold subsequencer handles

This *config object* must be a derivative of `uvm_sequencer`

These subsequencer handles will be set in the `connect_phase()` by the top-level environment class (`e_top`)

UVM Virtual Sequence - vsequencer

vseq_base Accesses Subsequencer Handles



Top Environment Component


```
class env_top extends uvm_env;
  `uvm_component_utils(env_top)

  env1       e1;
  env2       e2;
  vsequencer vsqr;

  function new(string name, uvm_component parent); ...

  function void build_phase(uvm_phase phase);
    e1   =       env1::type_id::create("e1",   this);
    e2   =       env2::type_id::create("e2",   this);
    vsqr = vsequencer::type_id::create("vsqr", this);
  endfunction

  function void connect_phase(uvm_phase phase);
    vsqr.a1_sqr = e1.a_agnt.sqr;
    vsqr.c_sqr  = e1.c_agnt.sqr;
    vsqr.a2_sqr = e2.a_agnt.sqr;
    vsqr.b_sqr  = e2.b_agnt.sqr;
  endfunction
endclass
```

`vsequencer` is a wrapper-class that declares (*holds*) the subsequencer handles

The top-environment component copies the real subsequencer handles to the subsequencer handles declared in the `vsequencer`

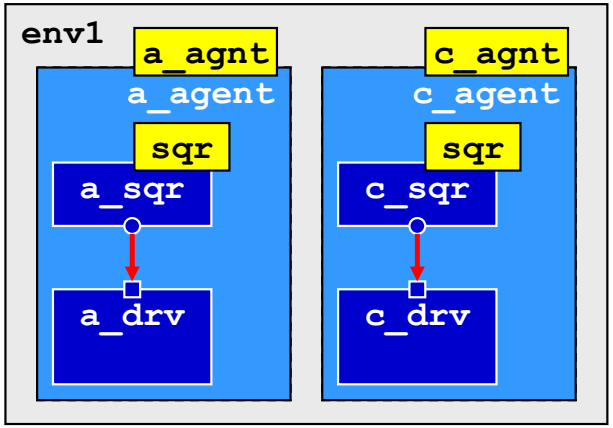
Two Environments

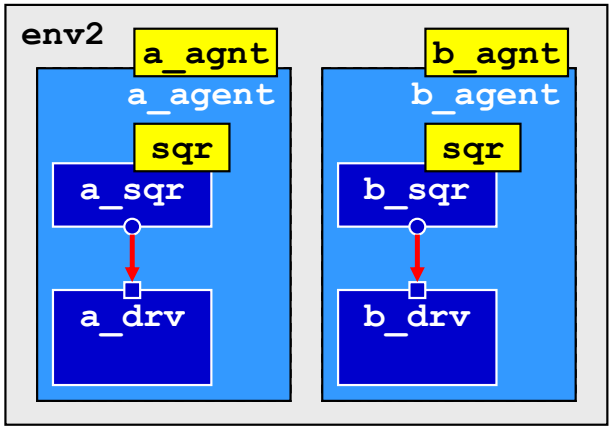
```
class env1 extends uvm_env;
  `uvm_component_utils(env1)
  a_agent a_agnt;
  c_agent c_agnt;
  ...

  function void build_phase(uvm_phase phase);
    a_agnt = a_agent::type_id::create("a_agnt", this);
    c_agnt = c_agent::type_id::create("c_agnt", this);
  endfunction
endclass
```

```
class env2 extends uvm_env;
  `uvm_component_utils(env2)
  b_agent b_agnt;
  a_agent a_agnt;
  ...

  function void build_phase(uvm_phase phase);
    b_agnt = b_agent::type_id::create("b_agnt", this);
    a_agnt = a_agent::type_id::create("a_agnt", this);
  endfunction
endclass
```





Test Base Class

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

  env_top e_top;

  function new(string name, uvm_component parent);
    super.new(name, parent);
  endfunction

  function void build_phase(uvm_phase phase);
    e_top = env_top::type_id::create("e_top", this);
  endfunction
endclass
```

`test_base` declares and builds the top environment (`e_top`)

m_sequencer, p_sequencer & `uvm_declare_p_sequencer macro

Lots of confusion about these

m_sequencer & p_sequencer

`uvm_declare_p_sequencer macro

- **`m_sequencer`**
  - A handle inside each sequence that points to its controlling sequencer
  - Set automatically

Users should never directly access `m_variables`

Set by calling `sequence.start(path_to_sequencer)` from the test

- **`p_sequencer`**
  - A handle created by calling the **`` `uvm_declare_p_sequencer ``** macro
  - User-settable handle to access local sequencer variables from the test

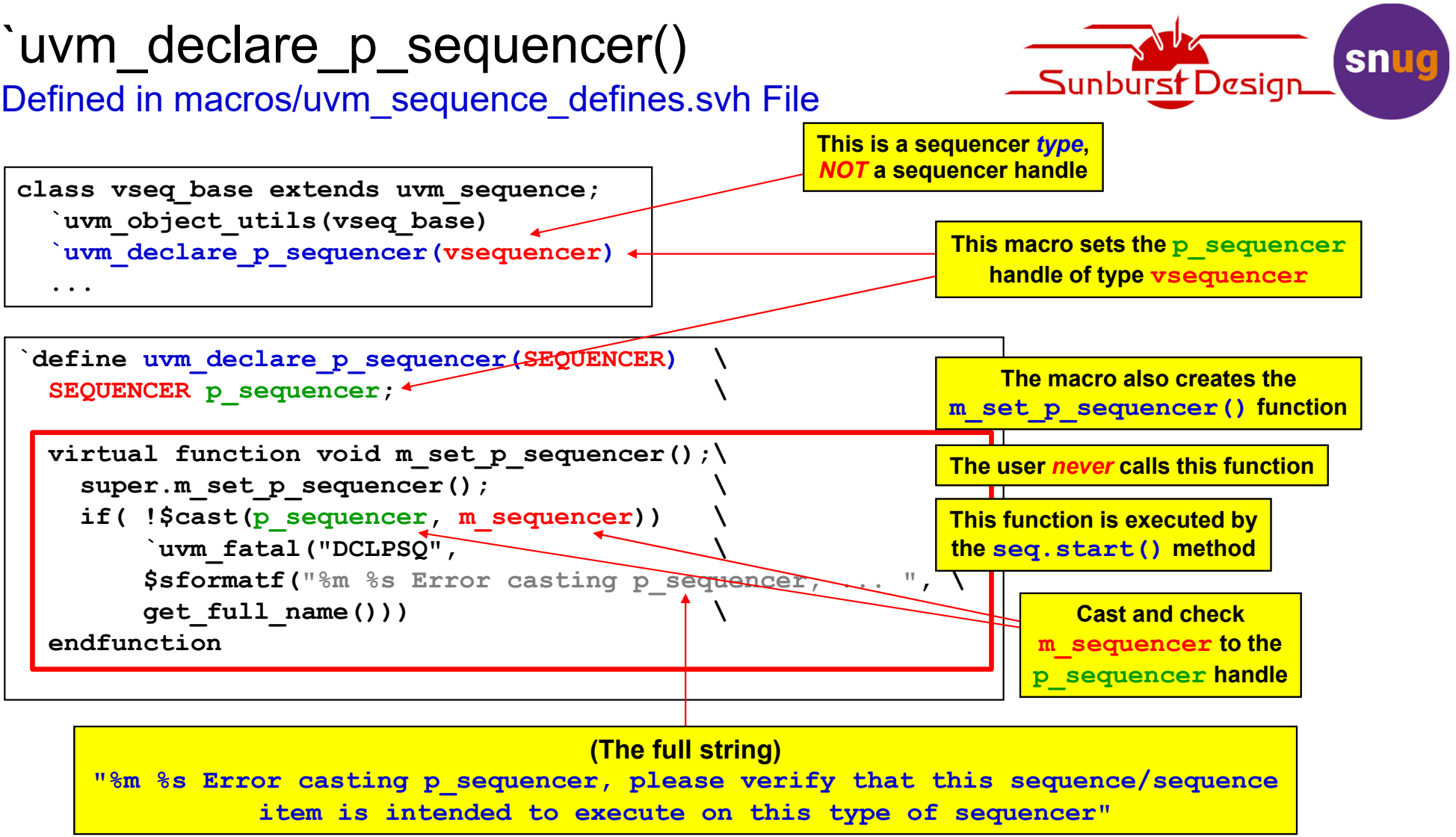
Setting the `p_sequencer` handle can help set sub-sequencer handles

- **`` `uvm_declare_p_sequencer ``** macro:
  - Used to set the **`p_sequencer`** handle

What does the `` `uvm_declare_p_sequencer `` macro do? *(next slide)*

`uvm_declare_p_sequencer()

Defined in macros/uvm_sequence_defines.svh File




```
class vseq_base extends uvm_sequence;
  `uvm_object_utils(vseq_base)
  `uvm_declare_p_sequencer(vsequencer)
  ...
```

```
`define uvm_declare_p_sequencer(SEQUENCER)   \
  SEQUENCER p_sequencer;                     \

  virtual function void m_set_p_sequencer();\
    super.m_set_p_sequencer();               \
    if( !$cast(p_sequencer, m_sequencer))    \
        `uvm_fatal("DCLPSQ",                 \
        $sformatf("%m %s Error casting p_sequencer, ... ", \
        get_full_name()))                    \
  endfunction
```

Virtual Sequence Base Class


```
class vseq_base extends uvm_sequence #(uvm_sequence_item);
  `uvm_object_utils(vseq_base)

  `uvm_declare_p_sequencer(vsequencer)

  a_sequencer A1;
  a_sequencer A2;
  b_sequencer B;
  c_sequencer C;

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

  task body;
    A1 = p_sequencer.a1_sqr;
    A2 = p_sequencer.a2_sqr;
    B  = p_sequencer.b_sqr;
    C  = p_sequencer.c_sqr;
  endtask
endclass
```

Call this macro to set the `p_sequencer` handle

`vseq_base` class declares the subsequencer handles

Declaring subsequencer handles is easy!

These subsequencer handles will be set when extended virtual sequences call `super.body()`

`p_sequencer` handle was set above

First Virtual Sequence

```
class vseq_A1_B_A2_A1 extends vseq_base;
  `uvm_object_utils(vseq_A1_B_A2_A1)

  function new(string name="vseq_A1_B_A2_A1");
    super.new(name);
  endfunction

  task body();
    a_seq  a = a_seq::type_id::create("a");
    b_seq  b = b_seq::type_id::create("b");
    a_seq a2 = a_seq::type_id::create("a2");

    super.body();

    a.start(A1);
    fork
       b.start(B);
      a2.start(A2);
    join
    a.start(A1);
  endtask
endclass
```

Inherits the subsequencer handles from the `vseq_base` class

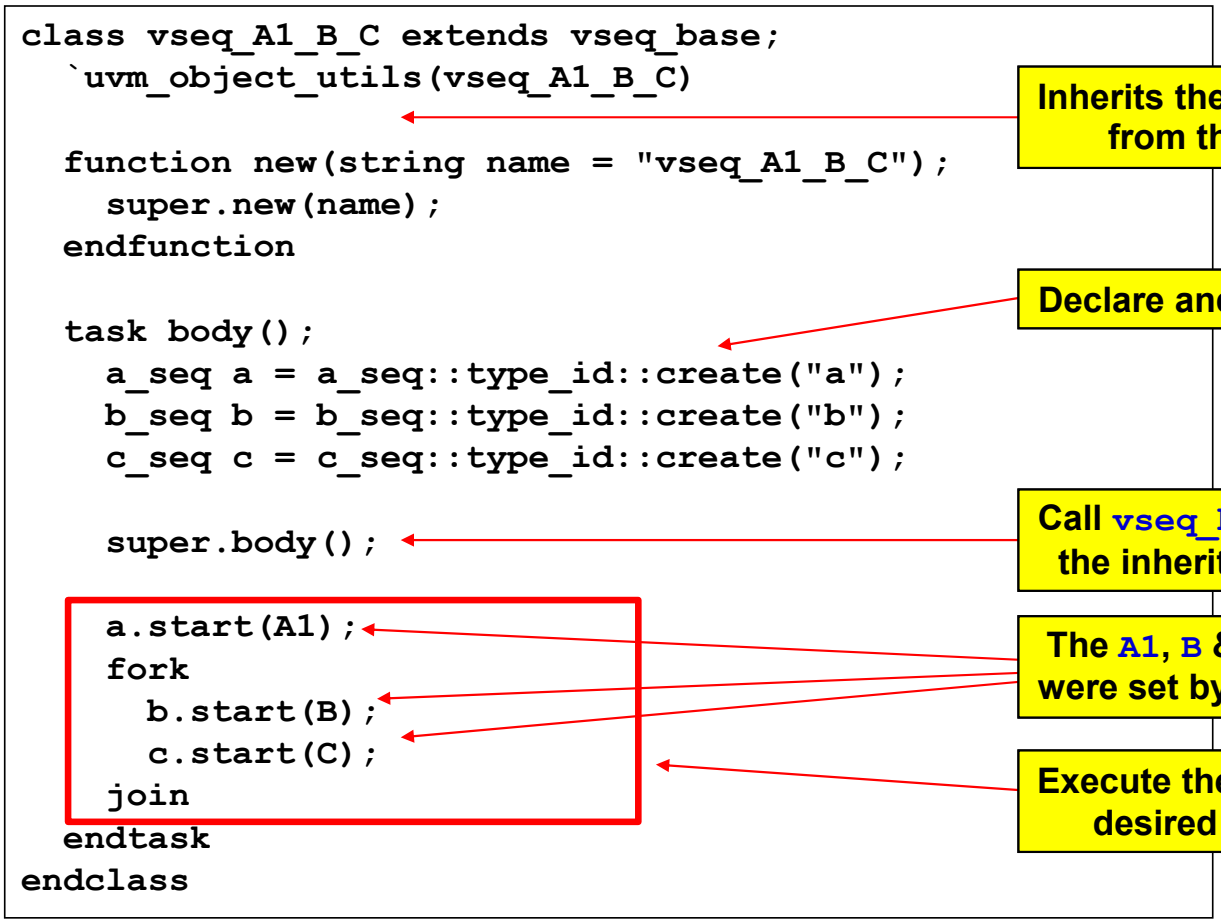
Declare and factory-create the sequences

Call `vseq_base body()` method to set the inherited subsequencer handles

The `A1`, `A2` & `B` subsequencer handles were set by the call to `super.body()`

Execute the sequences on the desired subsequencers

Second Virtual Sequence



Inherits the subsequencer handles from the `vseq_base` class

Declare and factory-create the sequences

Call `vseq_base body()` method to set the inherited subsequencer handles

The `A1`, `B` & `C` subsequencer handles were set by the call to `super.body()`

Execute the sequences on the desired subsequencers

test1 Extends From test_base

```
class test1 extends test_base;
  `uvm_component_utils(test1)

  function new(string name, uvm_component parent);
    super.new(name, parent);
  endfunction

  task run_phase(uvm_phase phase);
    vseq_A1_B_C     vseq1 =    vseq_A1_B_C::type_id::create("vseq1");
    vseq_A1_B_A2_A1 vseq2 = vseq_A1_B_A2_A1::type_id::create("vseq2");

    phase.raise_objection(this);
    vseq1.start(e_top.vsqr);
    vseq2.start(e_top.vsqr);
    phase.drop_objection(this);
  endtask

endclass
```

test1 inherits the e_top handle and the build_phase() from the test_base

Declare and create the virtual sequences

start the virtual sequences on the virtual sequencer (vsqr)

Why p_sequencer?

`uvm_declare_p_sequencer macro

- Sequences require a handle to a sequencer
  - **`seq.start(`***path_to_sequencer***`)`** sets the **`m_sequencer`** handle

Where the sequence is running

- Virtual sequences require handles to all subsequencers
  - Virt-seqs must retrieve subsequencer handles from *somewhere*
  - Virt-seqs retrieve handles stored in the virtual sequencer

- **`uvm_declare_p_sequencer`** creates & sets **`p_sequencer`** handle
  - **`p_sequencer`** will be set to point to **`vsqr`**
  - Subsequencer handles are stored in the **`vsequencer`**
  - Virtual sequence will retrieve the subsequencer handles
  - Virtual sequence will coordinate execution of sequences

The `vsequencer` handle

On subsequencer handles

Virtual Sequences the Easy Way

Use the uvm_resource_db Resources API

- UVM added resources
  - Environments can store subsequencer handles as `uvm_resources`
  - Subsequencer handles do NOT have to be stored inside a virtual sequencer
  - Virtual sequences can retrieve subsequencer handles from the resources database
  - Removes unnecessary `vsequencer` storage
  - This technique NOT possible using `uvm_config_db` API

Maintained in a resource pool

Virtual sequencer not required

- `uvm_config_db` is a secondary and inferior API into UVM resources

`uvm_config_db` does not work with sequences

- `uvm_resource_db` is the primary API into UVM resources

`uvm_resource_db` works with sequences

Easier than `uvm_config_db` and more powerful

See DVCon 2023 paper by Cliff Cummings & Mark Glasser

Environments Store Subsequencer Handles

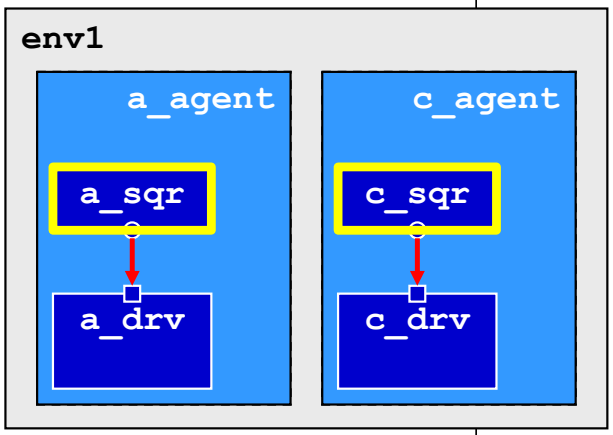
```
class env1 extends uvm_env;
  `uvm_component_utils(env1)

  a_agent a_agnt;
  c_agent c_agnt;

  function new(string name, uvm_component parent); ...

  function void build_phase(uvm_phase phase);
    a_agnt = a_agent::type_id::create("a_agnt", this);
    c_agnt = c_agent::type_id::create("c_agnt", this);
  endfunction

  function void connect_phase(uvm_phase phase);
    uvm_resource_db#(a_sequencer)::set("E1::a_sqr", "a_handle", a_agnt.sqr, this);
    uvm_resource_db#(c_sequencer)::set("E1::c_sqr", "c_handle", c_agnt.sqr, this);
  endfunction
endclass
```



Wait for the `build_phase()` to complete, then store the subsequencer handles

This is just a password that must be wildcard matched when retrieving the resource!

Virtual Sequence Base Class

This technique is not possible using `uvm_config_db` API

```
class vseq_base extends uvm_sequence #(uvm_sequence_item);
  `uvm_object_utils(vseq_base)

  a_sequencer A1;
  a_sequencer A2;
  b_sequencer B;
  c_sequencer C;

  function new(string name = "vseq_base"); ...

  task body;
    if (!uvm_resource_db#(a_sequencer)::read_by_name("E1::*", "a_handle", A1, this)) VSEQB_ERR("A1");
    if (!uvm_resource_db#(a_sequencer)::read_by_name("E2::*", "a_handle", A2, this)) VSEQB_ERR("A2");
    if (!uvm_resource_db#(b_sequencer)::read_by_name("E2::*", "b_handle",  B, this)) VSEQB_ERR( "B");
    if (!uvm_resource_db#(c_sequencer)::read_by_name("E1::*", "c_handle",  C, this)) VSEQB_ERR( "C");
  endtask

  function void VSEQB_ERR (string SQR);
    `uvm_fatal("VSEQB_ERR", {SQR," sequencer handle not found in resource_db"})
  endfunction
endclass
```

`vseq_base` class declares the subsequencer handles

These subsequencer handles will be inherited by every extended virtual sequence

Read the handles directly into the `vseq_base` using the `uvm_resource_db` API

String-matching *passwords*

Retrieve the subsequencer handles

If the resource is not available, report a `uvm_fatal` error

Virtual Sequence Base Class

Improved Efficiency!

```
class vseq_base extends uvm_sequence #(uvm_sequence_item);
  `uvm_object_utils(vseq_base)

  a_sequencer A1;
  a_sequencer A2;
  b_sequencer B;
  c_sequencer C;

  function new(string name = "vseq_base"); ...

  task body;
    if (A1 == null) begin
      if (!uvm_resource_db#(a_sequencer)::read_by_name("E1::*", "a_handle", A1, this)) ...;
      if (!uvm_resource_db#(a_sequencer)::read_by_name("E2::*", "a_handle", A2, this)) ...;
      if (!uvm_resource_db#(b_sequencer)::read_by_name("E2::*", "b_handle",  B, this)) ...;
      if (!uvm_resource_db#(c_sequencer)::read_by_name("E1::*", "c_handle",  C, this)) ...;
    end
  endtask

  function void VSEQB_ERR (string SQR); ...
endclass
```

vseq_base class declares the subsequencer handles

These subsequencer handles will be inherited by every extended virtual sequence

Only read the uvm_resource_db handles if the subsequencer handles are null

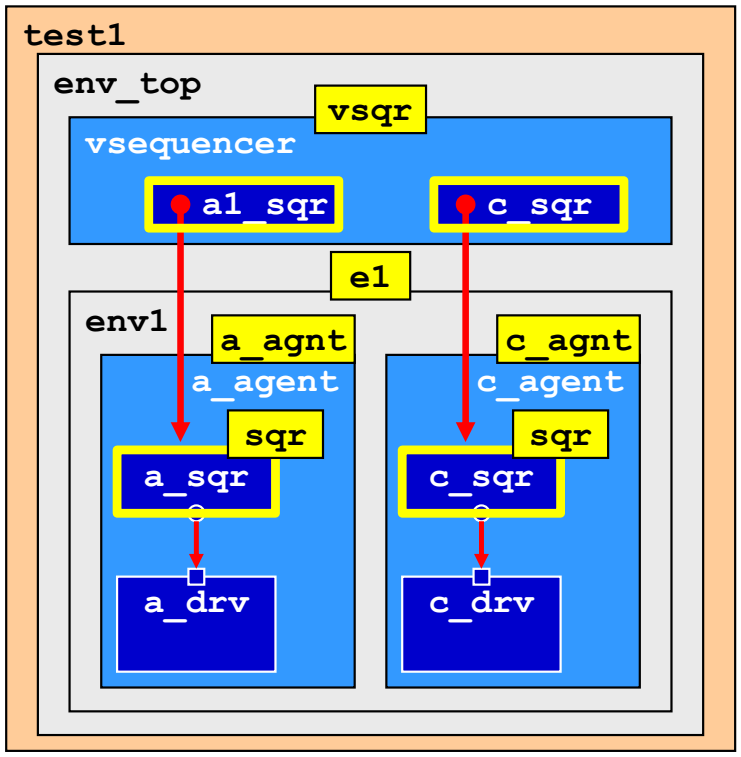
Same uvm_resource_db::read_by_name commands as previous slide

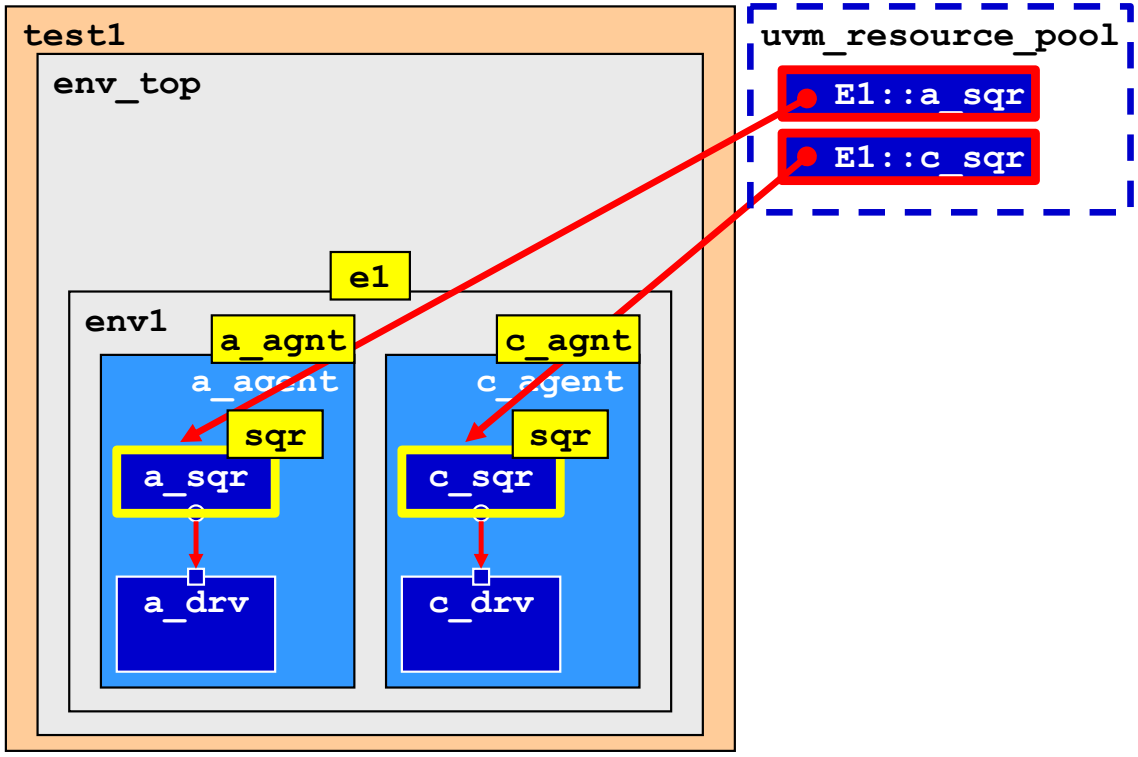
Summarizing Two Virtual Sequence Techniques

Comparing Both Methods

Using a virtual sequencer
(next slide)

Using the `uvm_resource_db` API
(in two slides)





Running Virtual Sequences

Using vsequencer

Env declares/builds `vsequencer`

`vsequencer` declares subsequencer handles

Env copies subsequencer handles into `vsequencer`

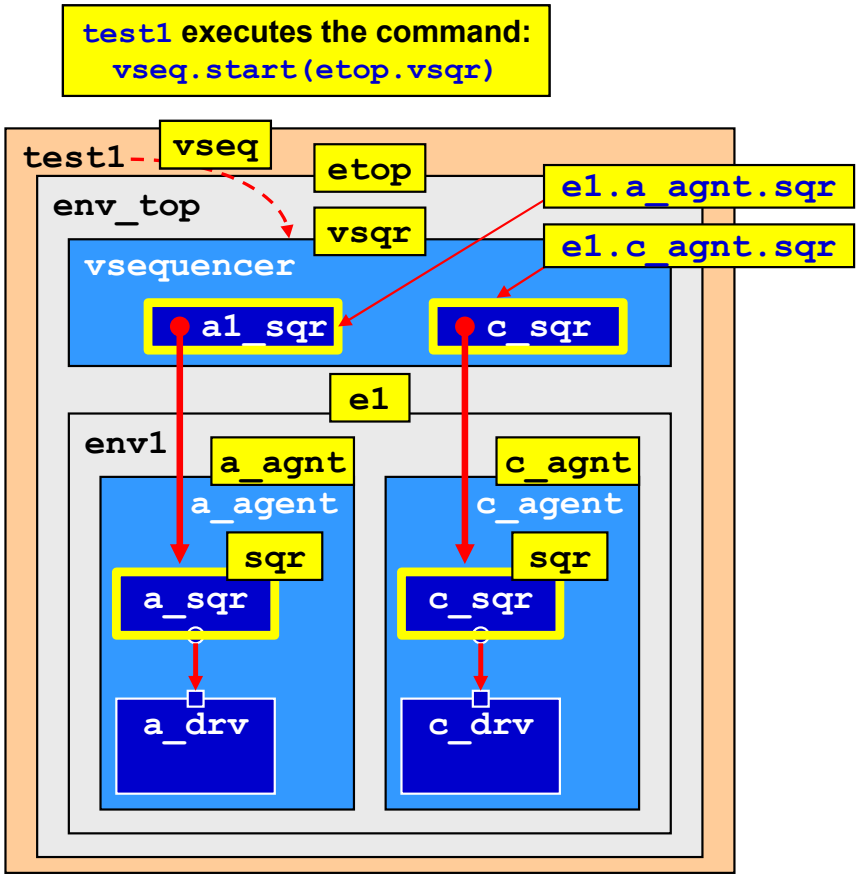
Test starts virtual sequence on `vsequencer`

`m_sequencer` handle points to `vsqr`

`vseq_base` uses `` `uvm_declare_p_sequencer `` to declare `p_sequencer` and `$casts` `m_sequencer` to `p_sequencer` handle

`vseq_base` retrieves subsequencer handles from `vsequencer` using `p_sequencer` handle

Virtual sequences extend `vseq_base` and start execution of sequences on subsequencers



Running Virtual Sequences

Using UVM Resources

Env stores subsequencer handles into resource pool using `uvm_resource_db::set()`

`vseq_base` retrieves subsequencer handles using `uvm_resource_db::read_by_name()`

Virtual sequences extend `vseq_base` and start execution of sequences on subsequencers

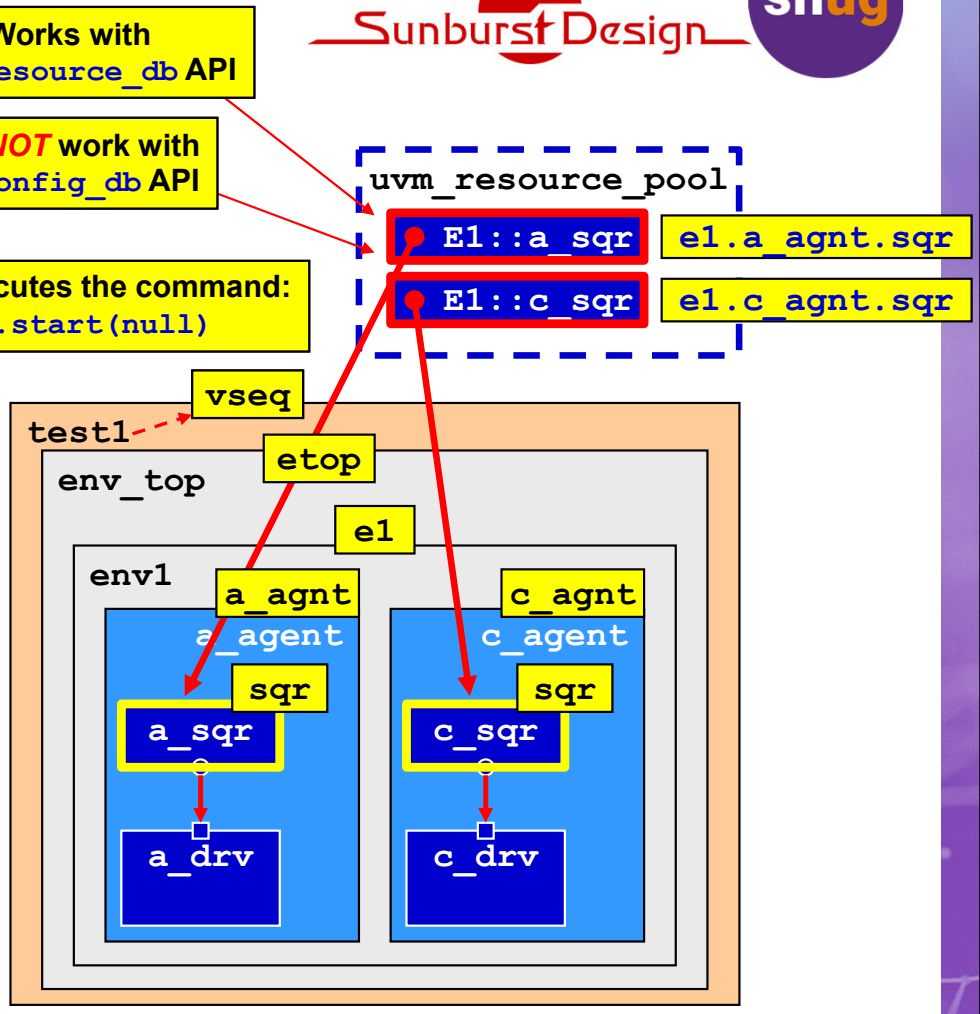
Test starts virtual sequences on `null`

Works with `uvm_resource_db` API

Does *NOT* work with `uvm_config_db` API

`test1` executes the command: `vseq.start(null)`

Virtual sequences coordinate subsequence activity by doing *sequence*`.start()` on retrieved subsequencer handles



Conclusions

m_sequencer Is Required by Sequences

- Every sequence has an **`m_sequencer`** handle

Most often set using the command `sequence.start(path_to_sequencer)`

- The **`p_sequencer`** handle does not exist in sequences

Unless explicitly declared -OR- the `` `uvm_declare_p_sequencer `` macro is used

- Sequences use **`p_sequencer`** handle to retrieve information stored in a sequencer

To pass required testbench information to a sequence

- The **`p_sequencer`** handle is not necessary

Used most often with traditional `vsequencer` storage

- The **`` `uvm_declare_p_sequencer ``** macro is not necessary

- Passing testbench info to sequences can be done using **`uvm_resource_db`** API

Info can be stored and retrieved from anywhere using `uvm_resource_db` API

… and *Thank You* to my friend & colleague Jeff Montesano for his review and valuable feedback on the paper and presentation slides

Understanding the UVM m_sequencer, p_sequencer Handles, and the `uvm_declare_p_sequencer Macro

Cliff Cummings - VP of Training
Paradigm Works

Life is too short for bad or boring training!