2025 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION

UNITED STATES

SAN JOSE, CA, USA
FEBRUARY 24-27, 2025

Life is too short for bad or boring training!

Sequencer Containers - A Unified and Simple Technique to Execute Both Sequences and Virtual Sequences

Clifford E. Cummings
Paradigm Works, Inc.

Mark Glasser
Paradigm Works, Inc.

PARADIGM WORKS

PARADIGM WORKS

accellera SYSTEMS INITIATIVE

Agenda

- Hierarchical independence ← **And why you should care**
- Agent & sequencer interfaces ← **Which interface causes problems?**
- Typical techniques for starting sequences and virtual sequences
  - starting sequences: **`sequence.start(...)`**
  - virtual sequences: **Typically uses a `vseqeuncer`**
- Sequencer containers ← **A simple technique for storing and retrieving sequencer handles**
- Sequencer Pool (`sqr_pool`) ← **Simple and efficient container**
- Sequencer Aggregator (`sqr_aggregator`) ← **More advanced container for larger testbenches**
- Conclusions

Please read the paper for more information and details

Hierarchical Independence

- Hierarchical independence definition:
  - A component does not have to "know" where it is hierarchically instantiated
  - It may be instantiated anywhere in the hierarchy

Component functionality is independent of hierarchical location

  - ***Sequencers*** are locked into a set location in the hierarchy
  - ***Virtual sequencers*** are locked into a set location in the hierarchy

Sequencers **themselves are hierarchically independent**

Tests and sequences must know where ***sequencers*** **are hierarchically located**

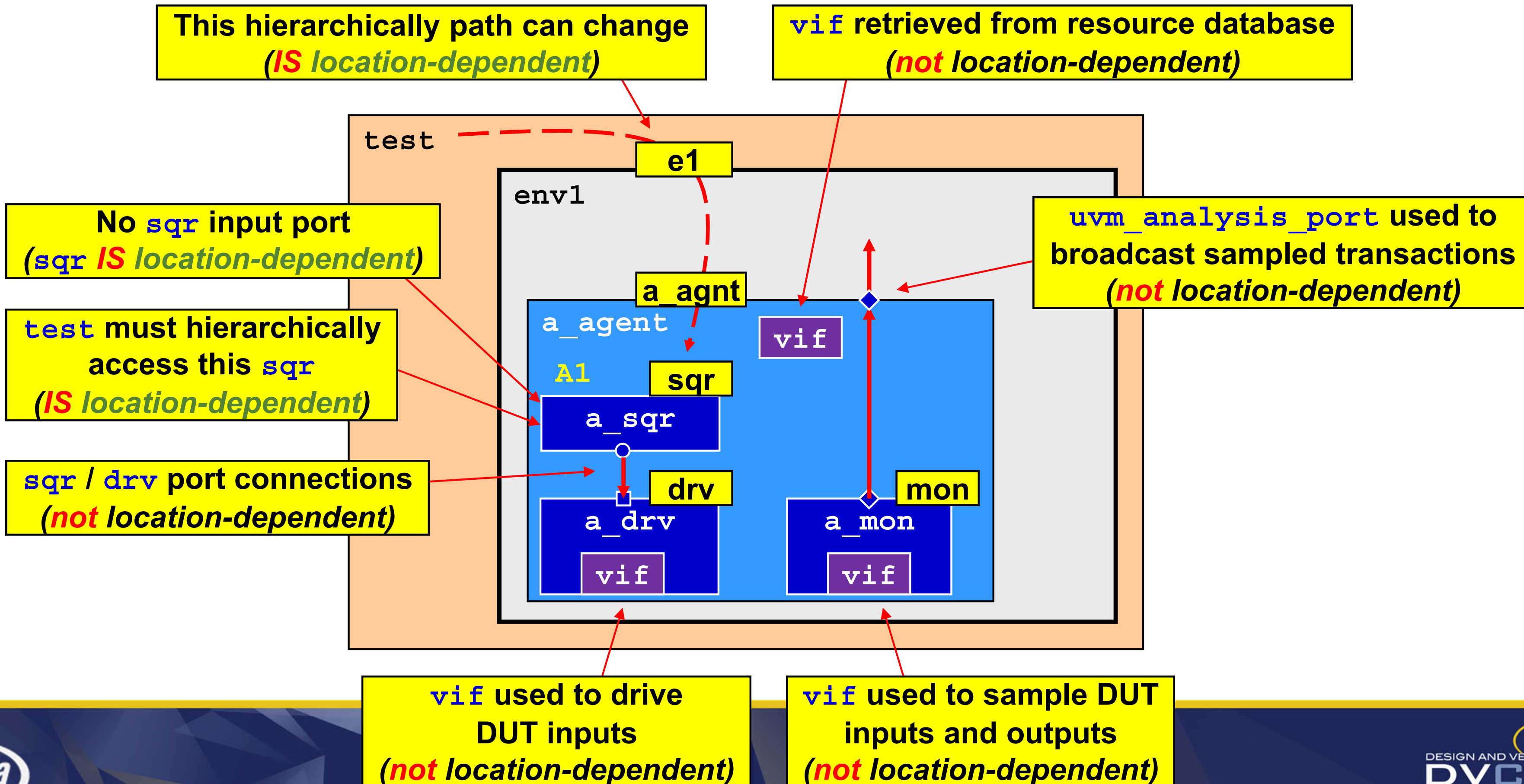
- *Virtual sequencers* contain *sequencer* handles
  - The ***sequencer*** handles are assigned relative hierarchical paths to ***subsequencers***

- Components that require a component path *are not hierarchically independent*

Partial or complete path

Including ***virtual sequencers***

Agent & Sequencer Interfaces



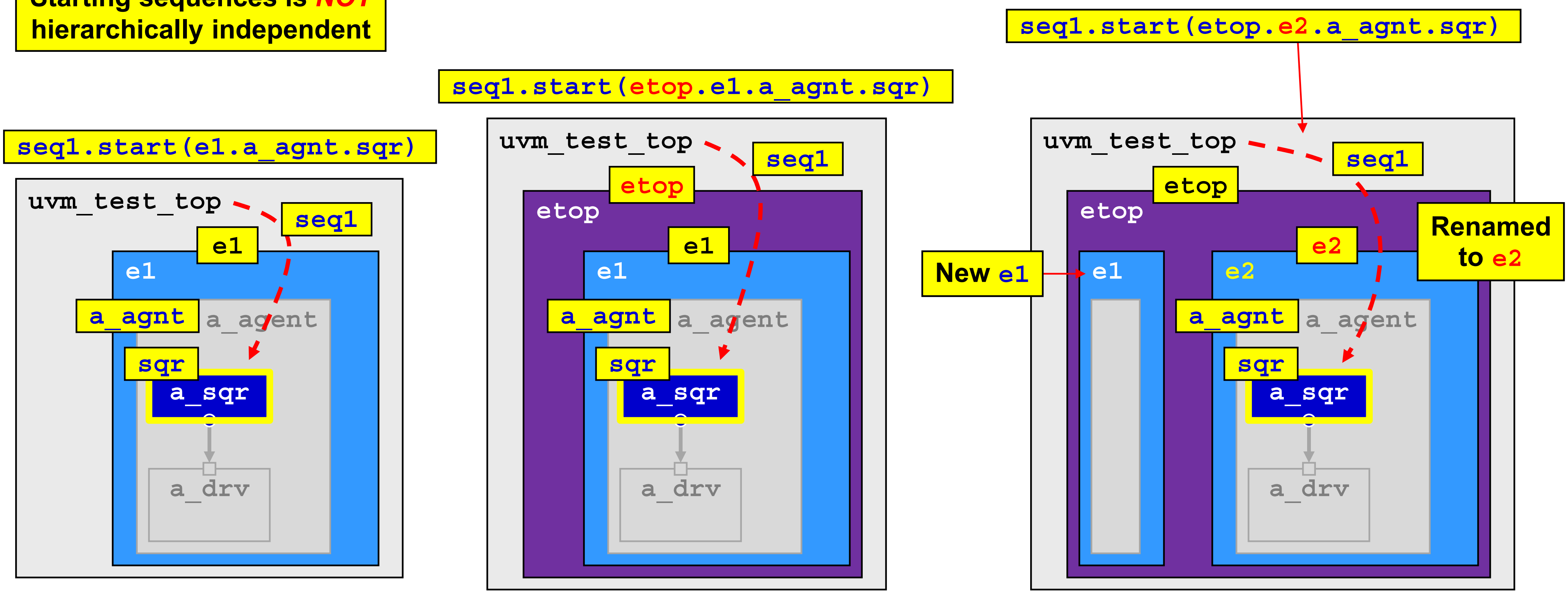
Starting Sequences

Typical Technique

Starting sequences is *NOT* hierarchically independent

Not hard … *as long as you remember!*

Subtle bugs are hardest to find and fix



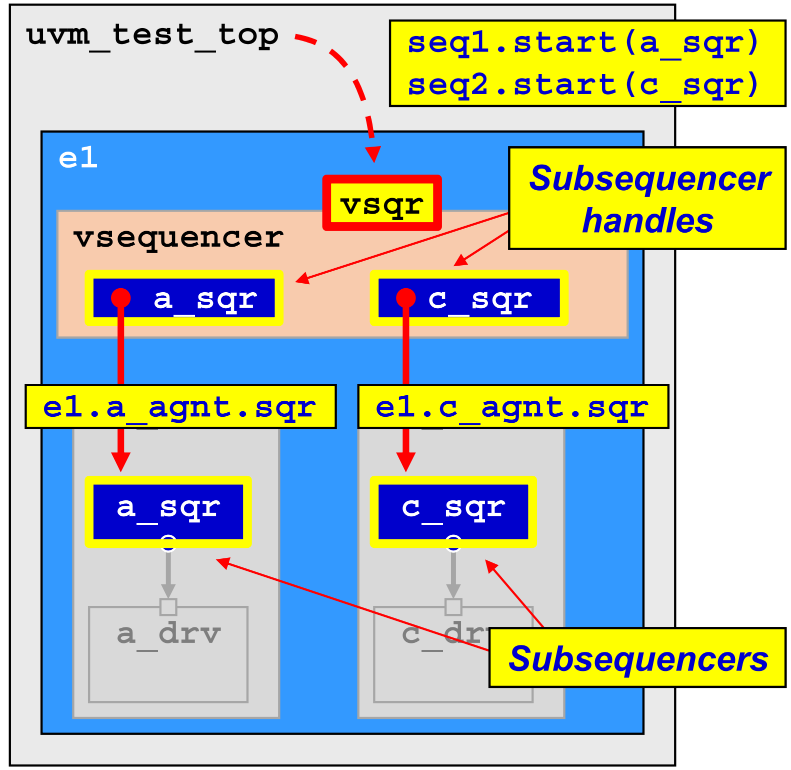
Starting Virtual Sequences

Typical - Uses Virtual Sequencer

Virtual sequences coordinate single-interface sequences across multiple *subsequencer handles*

- Requires `vsequencer` container
- `vsequencer` declares *subsequencer* handles
- Environment stores relative *subsequencer* paths in the `vsequencer`
- `vseq.start(`*path_to_vsequencer*`)`
- `vseq_base` retrieves handles from `vsequencer`
- *virtual*-seqs extend `vseq_base`
- *virtual*-seqs started on *subsequencer* handles
- When `vsequencer` or *subsequencer* locations changes …

paths_to_sequencers **must also be updated**



2025
DESIGN AND VERIFICATION™
DVCON
CONFERENCE AND EXHIBITION
UNITED STATES
SAN JOSE, CA, USA
FEBRUARY 24-27, 2025

Sequencer Containers

Why Use Sequencer Containers?

accellera
SYSTEMS INITIATIVE

Sequencer Container

Introduction

Commonly used in UVM testbenches

- Virtual sequencers served as traditional containers
  - Virtual sequencers served as pseudo-config object

Not hierarchically independent

Held handles to other sequencers

- Introducing *Sequencer Containers*
  - Associative array(s) that map names to sequencer handles
  - Designed as a container to hold sequencer handles
  - *Virtual sequences* retrieve the sequencer handles by name

Hierarchically independent

name	handle
"A1"	a1_sqr
"C"	c_sqr
"A2"	a2_sqr
"B"	b_sqr

- Paper describes two sequencer container implementations
  - Sequencer Pool (`sqr_pool`)
  - Sequencer Aggregator (`sqr_aggregator`)

Singleton derived from `uvm_pool`

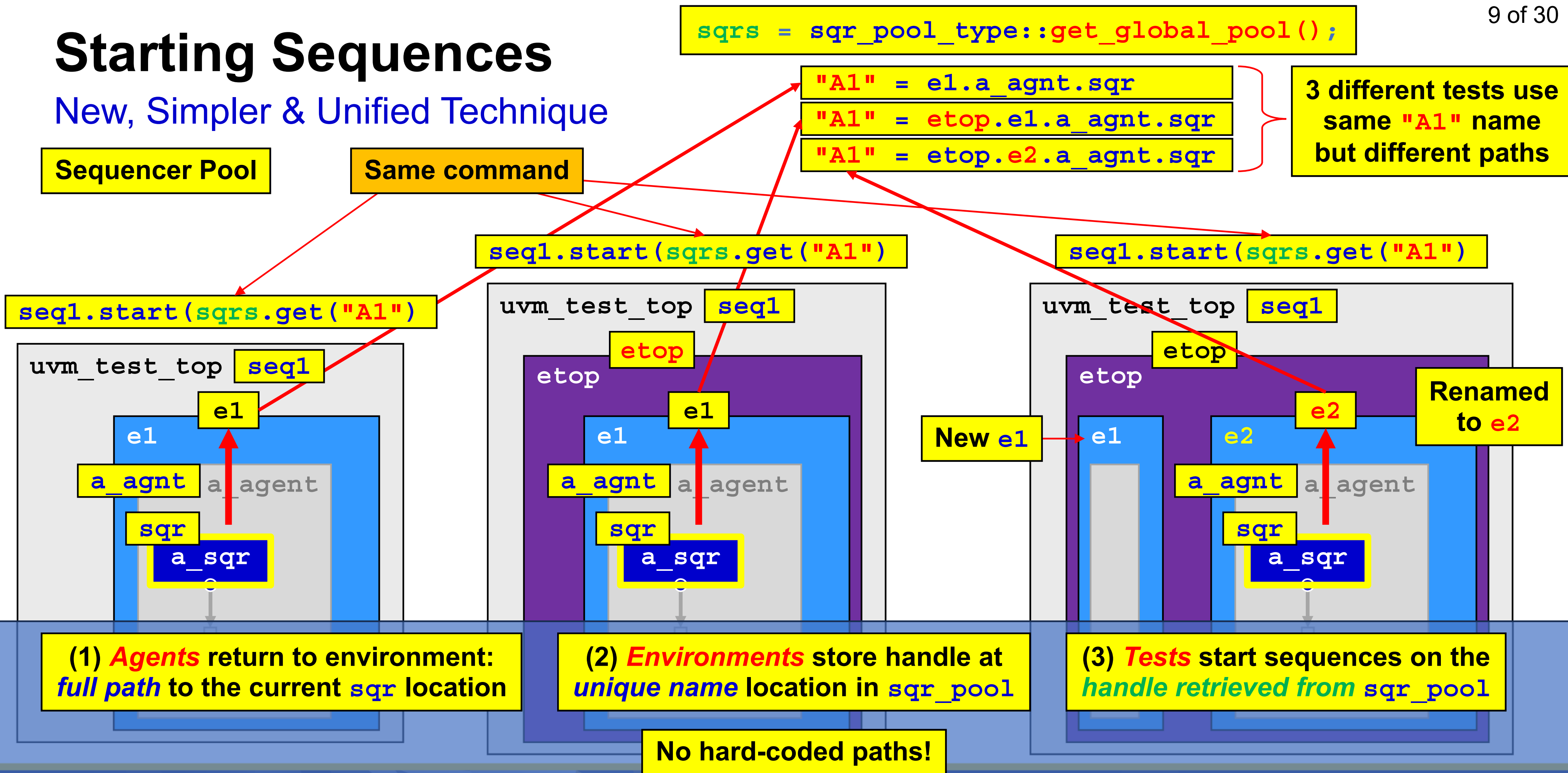
Uses multiple associative arrays

More advanced

Provides way to locate sequencer handles singly or in groups

Starting Sequences

New, Simpler & Unified Technique



Starting Sequences

Typical -vs- Improved Techniques

Topic of this presentation

Typical

- `seq.start(`***path_to_sequencer***`)`
- When sequencer location changes …

_path_to_sequencer_ must also be updated

Improved

- ***named_sequencer*** handles are stored in a *sequencer container*

Special associative array

- `seq.start(`***named_sequencer***`)`
- When sequencer location changes …
  - ***named_sequencer*** handle locations are *automatically updated*

Continue to run on the same *named_sequencer*

No modification required

Starting Virtual Sequences

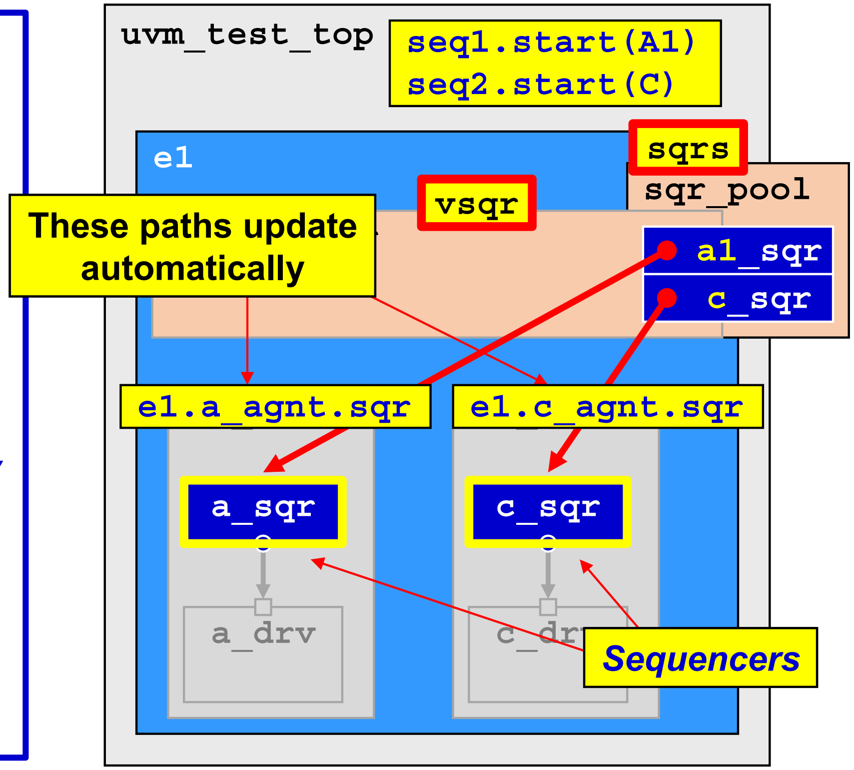
Improved Technique - Uses sqr_pool

Virtual sequences coordinate single-interface sequences across multiple *subsequencer handles*

- ***named_sequencer*** handles are stored in a *sequencer container*
- `seq.start(`***named_sequencer***`)`
- When sequencer locations change …
  - ***named_sequencer*** handle locations are *automatically updated*

Continue to run on the same *named_sequencer*

No modification required



Sequencer Pool

sqr_pool

sqr_pool Functionality

Introduction

- Sequencer pool (`sqr_pool`) is a sequencer container

Special associative array that can hold *any* sequencer handle

- `sqr_pool` features:
  - Singleton class derived from `uvm_pool`
  - Maps string ***names*** to ***sequencer handles***

Much like UVM RAL uses register *names* to map to register *addresses*

  - Has method to `add()` new ***sequencer handles*** to container

The environment *names* and *stores* the sequencer handles into the `sqr_pool`

  - Has method to `get()` any ***sequencer handle*** by name

The sequence retrieves the sequencer handles from the `sqr_pool`

  - As a singleton, it is available to all virtual sequences

No need to store handles in a virtual sequencer

sqr_pool Singleton Class

Extends uvm_pool Base Class

uvm_pool is a UVM base class that creates an associative array

```
class uvm_pool #(type KEY=int, T=uvm_void) extends uvm_object;
  const static string type_name = "uvm_pool";
  typedef uvm_pool #(KEY,T) this_type;

  static protected this_type m_global_pool;
  protected T pool[KEY];
```

1st parameter **2nd parameter**

```
class sqr_pool #(type T=uvm_sequencer_base) extends uvm_pool #(string,T);
```

sqr_pool is an extension of uvm_pool

- **indexed by KEY=string**

  -and uses -
- **type T=uvm_sequencer_base**

Why not use a parameterized version of the uvm_pool base class ??

UVM Base Class: uvm_pool

Extend uvm_pool to create sqr_pool

- `uvm_pool` base class defines an associative array with:
  - `type KEY int` ← `type KEY string`
  - `type T uvm_void` ← `type T uvm_sequencer_base`

- `uvm_pool` ← **Required: two methods should be modified**
- `get()`
  - returns the item with the given key ***-or-***
  - creates a new item if one does not exist
- `add()`
  - Adds the given item to the associative array
  - ***AND*** quietly overwrites the contents

	`uvm_pool` actions	**`sqr_pool` actions**
returns the item with the given key	**At the given `KEY` location**	**Same**
creates a new item if one does not exist	**Creates null item at that location (*BAD!*)**	**`` `uvm_fatal `` ****
Adds the given item to the associative array	**At the given `KEY` location**	**Same**
AND quietly overwrites the contents	**If there is already an item at that location (*BAD!*)**	**`` `uvm_fatal `` ****

**** Nothing good will happen!**

sqr_pool (Part 1 of 3)

```
class sqr_pool #(type T=uvm_sequencer_base) extends uvm_pool #(string,T);

  typedef sqr_pool #(T) this_type;

  static protected this_type m_global_pool;
  // protected T pool[KEY];

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

  static function this_type get_global_pool();
    if (m_global_pool==null)
      m_global_pool = new("pool");
    return m_global_pool;
  endfunction

  static function T get_global (KEY key);
    this_type gpool;
    gpool = get_global_pool();
    return gpool.get(key);
  endfunction
  ...
```

Defines a sqr_pool type parameterized to the uvm_sequencer_base type

All sequencers are derivatives of the uvm_sequencer_base

Any parameterized sequencer can be added to the sqr_pool

Constructs and returns the m_global_pool singleton handle

sqr_pool (Part 2 of 3)

```
...
virtual function T get (string key);
  if (pool.exists(key)) return pool[key];
  else begin
    dump();
    `uvm_fatal("SQR_POOL",
    $sformatf("No pool entry exists for sqr name %s", key))
  end
endfunction

virtual function void add(string key, uvm_sequencer_base item);
  if(key != "") begin
    if(pool.exists(key))
      `uvm_fatal("SQR_POOL",
      $sformatf("Duplicate name_table entry: name %s", key))
    pool[key] = item;
  end
endfunction
...
```

Returns a handle of a parameterized sequencer stored as a uvm_sequencer_base in the sqr_pool

The sequencer handle stored at the key-*string* location in the sqr_pool

If no handle is stored at the key location : `uvm_fatal

add a *sequencer handle* to the sqr_pool

If there is already a handle stored at the key location : `uvm_fatal

Store a *sequencer handle* at the key-*string* location in the sqr_pool

sqr_pool (Part 3 of 3)

```
  ...

  virtual function void dump();
    $display("\n--- SEQUENCER POOL ENTRIES -------");

    foreach(pool[name]) begin
      uvm_sequencer_base sqr = pool[name];
      $write  ("%10s : ", name);
      $display("%s", sqr.get_full_name());
    end

    $display("--- END SEQUENCER POOL     -------\n");
  endfunction

endclass
```

`dump()` a concise list of the named sequencers in the `sqr_pool`

`foreach` loop walks through each `sqr_pool` entry and prints its `name` and ***full-path***

Agent & Environment

Preview

```
function uvm_sequencer_base get_sequencer();
  return sqr;
endfunction
```

Each agent includes a `get_sequencer()` method

Returns full path to *sequencer*, *no matter where it exists* in a UVM testbench

```
function void get_sequencers();
  sqrs.add("A1", a_agnt.get_sequencer());
  sqrs.add( "C", c_agnt.get_sequencer());
endfunction
```

Unique names - index into `sqr_pool`

Each environment includes a `get_sequencers()` method

Calls each agent's `get_sequencer()` method and `adds` the `sqr`-handle to the `sqr_pool` (`sqrs`) with a unique name

The environment's location does not matter

Agent Code

Returns Sequencer Handle

The agent does not need to know about the `sqr_pool`

Each agent includes a `get_sequencer()` method

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

  a_driver    drv;
  a_sequencer sqr;

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

  function void build_phase(uvm_phase phase);
    drv =     a_driver::type_id::create("drv", this);
    sqr = a_sequencer::type_id::create("sqr", this);
  endfunction

  function void connect_phase(uvm_phase phase);
    drv.seq_item_port.connect(sqr.seq_item_export);
  endfunction

  virtual function uvm_sequencer_base get_sequencer();
    return sqr;
  endfunction
endclass
```

Returns full path to *sequencer*, *no matter where it exists* in a UVM testbench

Environment Code

Names & Stores Handles

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

  typedef sqr_pool #(uvm_sequencer_base) sqr_pool_type;

  a_agent  a_agnt;
  c_agent  c_agnt;
  sqr_pool_type sqrs = sqr_pool_type::get_global_pool();

  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);
    get_sequencers();
  endfunction

  virtual function void get_sequencers();
    sqrs.add("A1", a_agnt.get_sequencer());
    sqrs.add("C",  c_agnt.get_sequencer());
  endfunction
endclass
```

Each environment retrieves the `sqr_pool` singleton

Each environment calls the `get_sequencer()` method for each agent

The returned *agent-sqr* handles are added to the `sqr_pool`

These names must be unique in the `sqr_pool`

Test Base Code

Declare & Create sqr_pool

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

  typedef sqr_pool #(uvm_sequencer_base) sqr_pool_type;

  uvm_factory factory = uvm_factory::get();
  env_top     e_top;
  sqr_pool_type sqrs = sqr_pool_type::get_global_pool();
  ...
  function void start_of_simulation_phase(uvm_phase phase);
    super.start_of_simulation_phase(phase);
    if (uvm_report_enabled(UVM_HIGH)) begin
      this.print();
      factory.print();
      sqrs.dump();
    end
  endfunction

  function void final_phase(uvm_phase phase);
    if (uvm_report_enabled(UVM_HIGH)) sqrs.dump();
  endfunction
endclass
```

First `get_global_pool()` call will create the `sqr_pool`

Declares & creates `sqr_pool` singleton

***Pre-run:* display the following:**
- **testbench structure**
- **contents of factory**
- **`dump()` the contents of the `sqr_pool`**

Printing happens with command line: `+UVM_VERBOSITY=HIGH` *(or higher)*

***At end of simulation*: `dump()` the contents of the `sqr_pool`**

vseq_base Code

Sets Sequencer Handles

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

  typedef sqr_pool #(uvm_sequencer_base) sqr_pool_type;

  sqr_pool_type sqrs = sqr_pool_type::get_global_pool();

  uvm_sequencer_base A1;
  uvm_sequencer_base A2;
  uvm_sequencer_base B;
  uvm_sequencer_base C;

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

  task body();
    A1 = sqrs.get("A1");
    A2 = sqrs.get("A2");
    B  = sqrs.get("B");
    C  = sqrs.get("C");
  endtask
endclass
```

Declares & retrieves the `sqr_pool` singleton handle

Sequencer handles declared to be of type `uvm_sequencer_base`

Retrieve the handles stored in `sqrs` (the `sqr_pool`)

Assign the retrieved handles to the handles declared above

MARK TODO: Should this be `get_handles()` instead of `body()` task ??

These names were assigned by the environment

2025 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION
UNITED STATES
SAN JOSE, CA, USA
FEBRUARY 24-27, 2025

Sequencer Aggregator

sqr_aggregator

accellera SYSTEMS INITIATIVE

Sequencer Aggregator

Advanced Sequencer Container

Hierarchically independent

- *Sequencer Aggregator* (`sqr_aggregator`)
- Advanced sequencer container
  - Not a singleton
  - Can aggregate a collection of sequence handles
  - Has multiple associative arrays

Created during `build` and `connect` phases

Multiple `sqr_aggregators` possible

Allows for multiple domains & namespaces

Associative arrays indexed by `string` type

```
class sqr_aggregator;
  typedef uvm_sequencer_base sqr_q_t[$];

  local uvm_sequencer_base sqr_table [string];
  local uvm_sequencer_base name_table[string];
  local sqr_q_t            kind_table[string];
```

Stores sequencer handles differently

Sequencer Aggregator - add()

sqr handle

```
function void add(uvm_sequencer_base sqr, string name, string kind);
  sqr_q_t q;
  string path = sqr.get_full_name();
  sqr_table[path] = sqr;

  if(kind != "") begin
    if(kind_table.exists(kind))
      q = kind_table[kind];
    q.push_back(sqr);
    kind_table[kind] = q;
  end

  if(name != "") begin
    if(name_table.exists(name))
     `uvm_info("SQR_AGGREGATOR",
          $sformatf("replacing sequencer with name %s", name),
          UVM_NONE)
    name_table[name] = sqr;
  end
endfunction
```

Sequencer handles can be stored by:
1. **handle-`path`**
2. **user-defined `kind` (`string`)**
3. **name (`string`)**

`kind_table` enables access to groups of sequencers by assigned `kind`

Associative array indexed by user-chosen `name` (`string`)

Aggregator - lookup Methods (Part 1 of 2)

```
...

function uvm_sequencer_base lookup_path(string path);
  if(sqr_table.exists(path))
    return sqr_table[path];
  else
    return null;
endfunction


function uvm_sequencer_base lookup_name(string name);
  if(name_table.exists(name))
    return name_table[name];
  else
    return null;
endfunction

...
```

Lookup by `sqr path` from the `sqr_table`

Lookup by `string name` from the `name_table`

Aggregator - lookup Methods (Part 2 of 2)

```
...

function sqr_q_t lookup_path_regex(string regex);
  sqr_q_t q = {};
  foreach(sqr_table[path]) begin
    if(uvm_re_match(regex, path))
      q.push_back(sqr_table[path]);
  end
  return q;
endfunction


function sqr_q_t lookup_kind(string kind);
  return kind_table[kind];
endfunction

...
```

Lookup by `sqr` *regular expression* `path` from the `sqr_table`

Lookup by `string kind` from the `kind_table`

Conclusions

Sequencer Containers Simplify UVM Testbenches

Common / old style

- Virtual sequencers are not hierarchically independent
  - **Makes reuse difficult**
  - **Makes debug difficult!**
- Sequencer containers eliminate virtual sequencer deficiencies
  - **Makes accessing sequencers hierarchically independent**
- Two sequencer containers described in this presentation
  - `sqr_pool` — **Singleton and simple to use**
  - `sqr_aggregator` — **Allows multiple containers for advanced UVM testbench environments**
  - Add `get_sequencer()` method to every agent — **Enables sequencer container usage**
  - Add `get_sequencers()` method to every environment — **Let environment name the sequencer handles**
- Sequencer containers simplify and unify sequence execution
  - **Tests can execute sequences & virtual sequences using a common technique that reduces usage mistakes**

Questions?

2025 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION
UNITED STATES
SAN JOSE, CA, USA
FEBRUARY 24-27, 2025

accellera SYSTEMS INITIATIVE

2025

DESIGN AND VERIFICATION™

DVCON

CONFERENCE AND EXHIBITION

UNITED STATES

SAN JOSE, CA, USA

FEBRUARY 24-27, 2025

Life is too short for bad or boring training!

Sequencer Containers - A Unified and Simple Technique to Execute Both Sequences and Virtual Sequences

Clifford E. Cummings
Paradigm Works, Inc.

Mark Glasser
Paradigm Works, Inc.