2019 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

Be a Sequence Pro to Avoid Bad Con Sequences

Presenters: Jeff Vance, Jeff Montesano

Contributors: Mark Litterick, Jason Sprott

accellera SYSTEMS INITIATIVE

Introduction

UVM sequences are vital for verification success

Need Control
- Reach scenarios
- Find and isolate bugs
- Close coverage

Manage Complexity
- Debug constraint failures
- Reduce mistakes
- Transfer knowledge

Need Reuse
- Within a sequence library
- With derivative projects
- For generic VIP

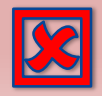
UVM sequences are often not applied appropriately

Insufficient API
- Can't control from tests
- Can't isolate features

Too Complex
- Intractable constraint failures
- Invalid stimulus

Not Reusable
- Copy/pasted routines
- Tied to a specific DUT

Poor visibility of project status



No risk-management of features

Outline

- **Introduction to sequences**
- **Sequence guidelines –** *improve control, complexity, & reuse*
- **Sequence execution** – *masters, reactive slaves, streaming data*
- **Verification productivity** – *strategies to manage features*
- **Portable Stimulus Considerations** – *how PSS impacts sequences*
- **Conclusion & references**

What are UVM sequences and why do we care?

INTRODUCTION TO SEQUENCES

What is a Sequence?

A sequence encapsulates a scenario

```
class access_seq extends base_seq;
  rand master_enum   source;
  rand cmd_enum      cmd;
  rand bit[31:0]     addr;
  rand bit[31:0]     data[];

  constraint legal_c{ ...}


  task body();
    ...
    if(p_sequencer.cfg.chmode == ENABLED)
      ...
```

Random control knobs for users

Constraints on random options

Procedural body()

Access resources via sequencer

Why Bother Using Sequences?

```
class access_seq extends base_seq;
  rand master_enum    source;
  rand cmd_enum       cmd;
  rand bit[31:0]      addr;
  rand bit[31:0]      data[];

  constraint legal_c{ ...}

  task body();
    ...
```

Both provide **options**

```
task access_dut (master_enum    source,
                 cmd_enum       cmd,
                 bit[31:0]      addr,
                 ref bit[31:0]  data[]);
    ...
   //Task body
```

Both ***sequences*** and ***tasks*** encapsulate a scenario

Both have procedural body

Why Bother Using Sequences?

```
class read_test extends uvm_test;
  task run_phase(uvm_phase phase);

    `uvm_do_with(access_seq,
                 {source == PORT_A;
                  cmd    == WRITE;
                  addr   == 'hA0;
                  data   == 'h55;})
    ...
```

start sequence

```
class access_seq extends base_seq;
  rand master_enum    source;
  rand cmd_enum       cmd;
  rand bit[31:0]      addr;
  rand bit[31:0]      data[];

  constraint legal_c{ ...}

  task body();
    ...
```

```
class read_test extends uvm_test;
  task run_phase(uvm_phase phase);

    access_dut(PORT_A, WRITE,
               'hA0, 'h55);
    ...
```

call task

```
task access_dut (master_enum    source,
                 cmd_enum       cmd,
                 bit[31:0]      addr,
                 ref bit[31:0]  data[]);
    ...
   //Task body
```

In the most **basic** cases, tasks and sequences can be equivalent

Why Bother Using Sequences?

Tasks

- ✗ Arguments are mandatory (or fixed by default)
- ✗ Users must randomize args
- ✗ No built-in arg validity checks
- ✗ Awkward to return data (use ref arguments)
- ✗ No built-in access to resources
- ✗ Adding args breaks existing code

Sequences

- ✓ Arguments are optional (random by default)
- ✓ Legal randomization is built-in
- ✓ Constraints validate user options
- ✓ Caller can access any data (with sequence handle)
- ✓ Access to resources via sequencer
- ✓ Adding options has minimal impact

With guidelines

test configuration constraints

virtual sequencer

Sequence A

Sequence B

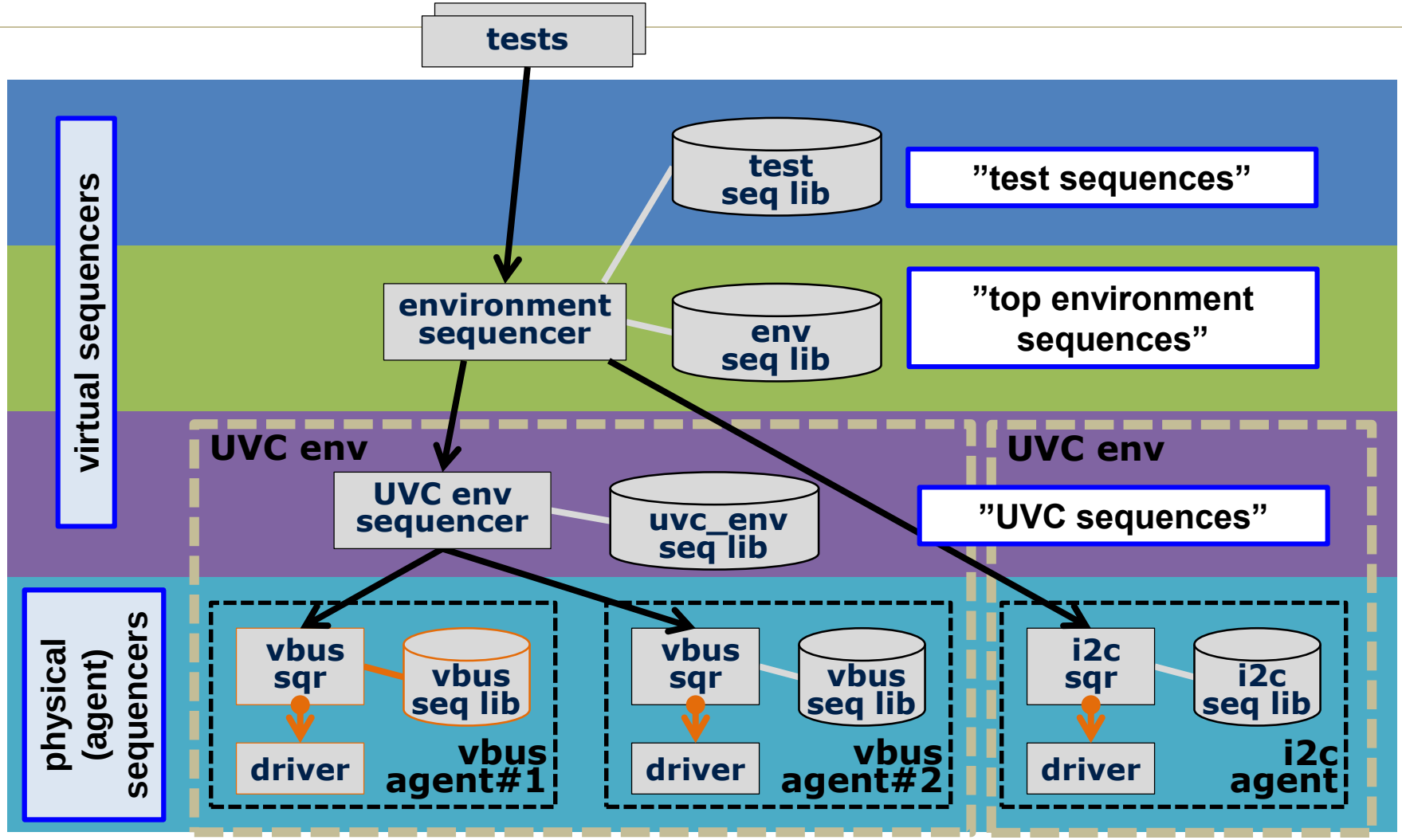
Sequence C

Driver A

Driver B

Driver C

Sequence API Strategy



Sequence Layer Roles

LAYER		CONSTRAINTS	PRIMARY PURPOSE
TEST		Test scenario	Highest-level sequence
TOP	**User**	DUT use cases/scenarios	API for test writer
	Lower	System requirements	Scenario building blocks
UVC	**User**	Protocol use cases	Encapsulates sequencer(s)
	Middle	Protocol operations	Encapsulates basic operations
	Low	Low-level requirements	Data formatting
	Item	Enforce legality	Support all possible scenarios

- **Reduce complexity at each layer**
- **Control with intuitive APIs**
- **Sequences decoupled and reusable**

- **Each layer resolves a subset of random options**
- **Benefits both directed and random tests**

Existence of some layers is application dependent

How to maximize the benefits of using sequences

SEQUENCE GUIDELINES

Legality Guideline

Produce legal stimulus by default

```
class top_seq extends base_seq;
  …
  task body();
    `uvm_do(ahb_burst_seq_inst)
```

No knobs provided

start sequence

```
class ahb_burst_seq extends base_seq;

  constraint legal_c{
    dir inside {WRITE, READ};
    addr + length < p_sequencer.cfg.get_max_addr();
    …
  }
```

Enforce legal stimulus

Users can provide 0 or more inline constraints

Legality Guideline

```
class top_seq extends base_seq;
  …
  task body();
    `uvm_do_with(ahb_burst_seq_inst,
                 {addr == 'hFF00;})
```

Address set by user

start sequence

```
class ahb_burst_seq extends base_seq;

  constraint legal_c{
    dir inside {WRITE, READ};
    addr + length < p_sequencer.cfg.get_max_addr();
    …
  }
```

Length still random and legal

Inline constraints are optional, but legality is always guaranteed

Legality Guideline

```
class top_seq extends base_seq;
  …
  task body();
    `uvm_do_with(ahb_burst_seq_inst,
           {addr   == 'hFF0;
            length ==  1024;})
```

start sequence

```
class ahb_burst_seq extends base_seq;
  constraint legal_c{
    dir inside {WRITE, READ};
    addr < p_sequencer.cfg.get_max_addr();
    …
  }
```

No constraint on **length**

Without this guideline, we risk wasting significant time!

Users must manage legal rules in higher sequences

What if length is **invalid**? (User error or bug)

May produce illegal burst length

Wasted time debugging invalid simulations

Legality Guideline

```
class top_seq extends base_seq;
  …
  task body();
    `uvm_do_with(ahb_burst_seq_inst,
          {addr   == 'hFFFF_FFF0;
           length ==  256;})
```

Illegal address and length

start sequence

```
class ahb_burst_seq extends base_seq;
  constraint legal_c{
    dir inside {WRITE, READ};
    addr + length < p_sequencer.cfg.get_max_addr();
    …
  }
```

- **Protect users from illegal stimulus**
- **Constraint solver fails on illegal options**

Control Knob Debug Guideline

Constrain control knobs with class constraints, then pass results with inline constraints

```
class ahb_write_burst_seq extends ahb_base_seq;
  …
  `uvm_do_with(ahb_seq,
               {ahb_seq.hwrite == HWRITE_WRITE;
                ahb_seq.hburst inside {HBURST_SINGLE, HBURST_INCR};}
```

Don't pass inline

start sequence

```
class ahb_burst_seq extends ahb_base_seq;
   rand write_enum hwrite;
   rand burst_enum hburst;
   …
  }
```

All constraints solved concurrently, making solver failures hard to debug

Control Knob Debug Guideline

```
class ahb_write_burst_seq extends ahb_base_seq;
  rand hburst_t  hburst;

  constraint c_hburst {
    hburst inside {HBURST_SINGLE, HBURST_INCR};
  }
  task body();
    `uvm_do_with(ahb_seq,
                 {ahb_seq.hwrite == HWRITE_WRITE;
                  ahb_seq.hburst inside {HBURST_SINGLE, HBURST_INCR};
                  ahb_seq.hburst == local::hburst;}
```

Use class constraint

Pass result to sequence

start sequence

```
class ahb_burst_seq extends ahb_base_seq;
   rand hwrite_t hwrite;
   rand hburst_t hburst;
```

Two-Step Randomization:

1. Randomize class variables
2. Run **body()** to randomize lower sequences

Debug randomization in isolated steps

API Guideline

Minimize the number of control knobs

```
class ahb_master_write_seq extends ahb_base_seq;
  rand int slave_num;
  …
  protected rand int  slave_id;

  constraint id_c {
    slave_id == p_sequencer.cfg.get_slave_id(slave_num);
  }
  virtual task body();
     ahb_seq_item req;
    `uvm_do_with(req,{req.hwrite == HWRITE_WRITE;
                      req.hprot3 == p_sequencer.cfg.get_hprot3();
                      req.haddr[31:24] == local::slave_num;
                      req.id == local::slave_id})
```

Exposed control knob

Hidden control knob

Keep fixed and derived variables in body()

Users can't control these

☑ **Sequences are easier to use**

☑ **Users can't misuse sequence and cause unexpected errors**

Reuse Guideline

Make tests **independent** of testbench architecture

```
class ahb_fabric_write_seq extends base_seq;
  task body();
    ahb_fabric_master_write_seq master_write_seq;
    `uvm_do(master_write_seq)
  endtask: body
```

Test-level sequence decoupled from testbench architecture

start mid-level sequence

```
class ahb_fabric_master_write_seq extends ahb_base_seq;
  ...
  task body();
    ahb_master_write_seq ahb_master_write_seqs[string];
    ...
    `uvm_do_on_with(ahb_master_write_seqs[b],
                    p_sequencer.agent_sequencer[b],
                    {…})
```

Only mid-level sequences reference sequencers

Test sequences are generic and reusable on derivative projects

Adaptability Guideline

Use **configuration objects** and **accessor methods** to adapt to project-specific configurations

```
class ahb_cfg extends uvm_object;
  rand int slv_fifo_depth;
  ...
  constraint {
    slv_fifo_depth inside {[1:`MAX_FIFO_DEPTH]};
  };
  function int get_fifo_depth();
    return(this.slv_fifo_depth);
  endfunction
```

Keep project-specific configuration constraints *outside* of sequences

```
class fifo_test_seq extends fabric_base_seq;
  ...
  task body();
    for(int i=0; i<=p_sequencer.cfg.get_fifo_depth();
i++) begin
      `uvm_do_with(master_seq, {
                   hsize == HSIZE_32;
                   hburst == SINGLE;})
```

- ☑ **Sequence is generic and reusable**
- ☑ **Changes in spec are transparent**

Self-tuning Guideline

Use **utility methods** to support self-tuning sequences

```
function automatic int calc_data_offset_from_address(ADDR_t addr);
    return(addr / DATA_WORD_SIZE) % DATA_WORDS_PER_ADDR);
endfunction
```

Package-scope methods perform common calculations

```
class write_word_seq extends base_seq;
  rand bit[31:0] addr;

  task body();
    bit[31:0] ram_addr = addr / DATA_WORDS_PER_ADDR;

    `uvm_do_with(write_single_seq, {
                 addr == local::ram_addr;
                 word_sel == calc_data_offset_from_addr(local::addr)})
```

```
package ahb_pkg;
  `include "ahb_common.sv"
  ... //etc
endpackage
```

- Derive values using formulas
- Calculate delays for transactions
- Calculate timeouts for waiting

Avoid code duplication between sequences

Sequences adapt to changes in calculations

Constraint Placement Guideline

Constraint Strategy	Ideal Purpose
class constraints	legal requirements
inline constraints	scenarios
configuration objects	configuration register dependencies
descriptor objects[1]	bundle sets of control knobs
policy classes [3]	dynamically redefine constraints or impose constraints that bypass many layers

[3] ***SystemVerilog Constraint Layering via Reusable Randomization Policy Classes*** – John Dickol, DVCon 2015

Sequence Library Tip

Use **typedef header** at top of sequence library file

```
typedef class power_on_seq; // powers on DUT
typedef class reset_seq;    // hard reset of DUT
typedef class por_seq;      // powers on and hard resets DUT
...
class power_on_seq extends base_seq;
  ...
endclass

class reset_seq extends base_seq;
  ...
endclass

class por_seq extends base_seq;
  power_on_seq power_seq;
  reset_seq    rst_seq;
  ...
endclass
```

typically **multiple classes per file**
(normal UVM has one class per file)

☑ **documents content**

☑ **allows sequences used in any order**

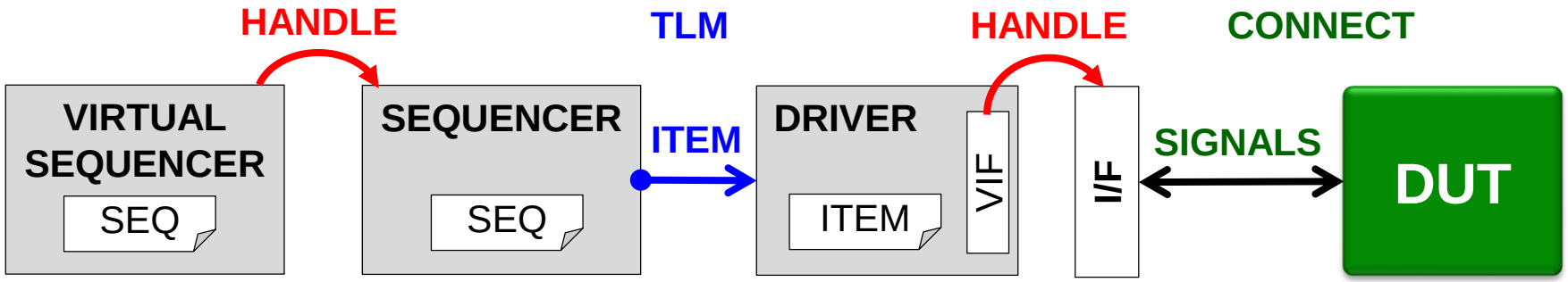
More Guidelines

Guidelines	
Use Dedicated Constraint Blocks	Inheritance vs. Composition
Use Soft Constraints Carefully	Manage Control Knobs Hierarchically
Use Enumerated Types	Provide Random and Directed Flavors
Use Descriptor Objects	Messaging at Sequence Start and End

[1] ***Use the Sequence, Luke*** – Verilab, SNUG 2018

SEQUENCE EXECUTION

Sequence Execution Overview



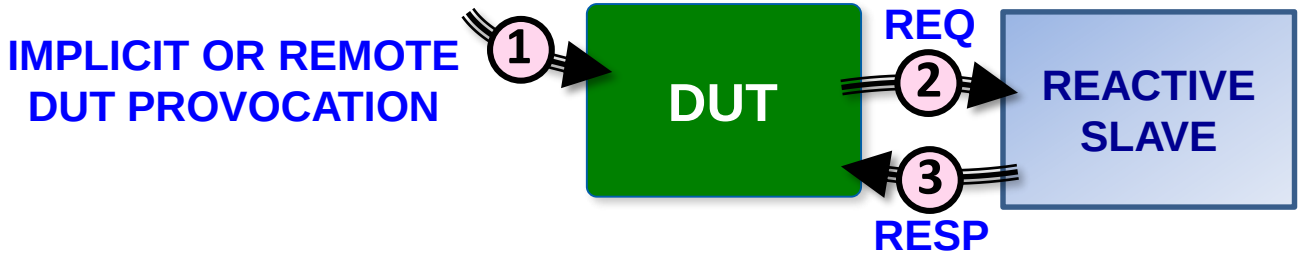
- **Sequences execute on sequencers to control stimulus**
  - virtual sequences coordinate and execute one or more sequences
  - physical sequences generate items which are passed to drivers
  - drivers interact with DUT via signal interface
- **Sequence execution affected by:**
  - verification component role - proactive or reactive
  - sequencer type - virtual (no item) or physical (item)
  - item content - single transaction or streams of data

Proactive Masters & Reactive Slaves

- **Proactive Masters**:

  - Test controls when sequences are executed on the UVC and timing of requests to DUT
  - Stimulus blocks test flow waiting for DUT response

- **Reactive Slaves**:



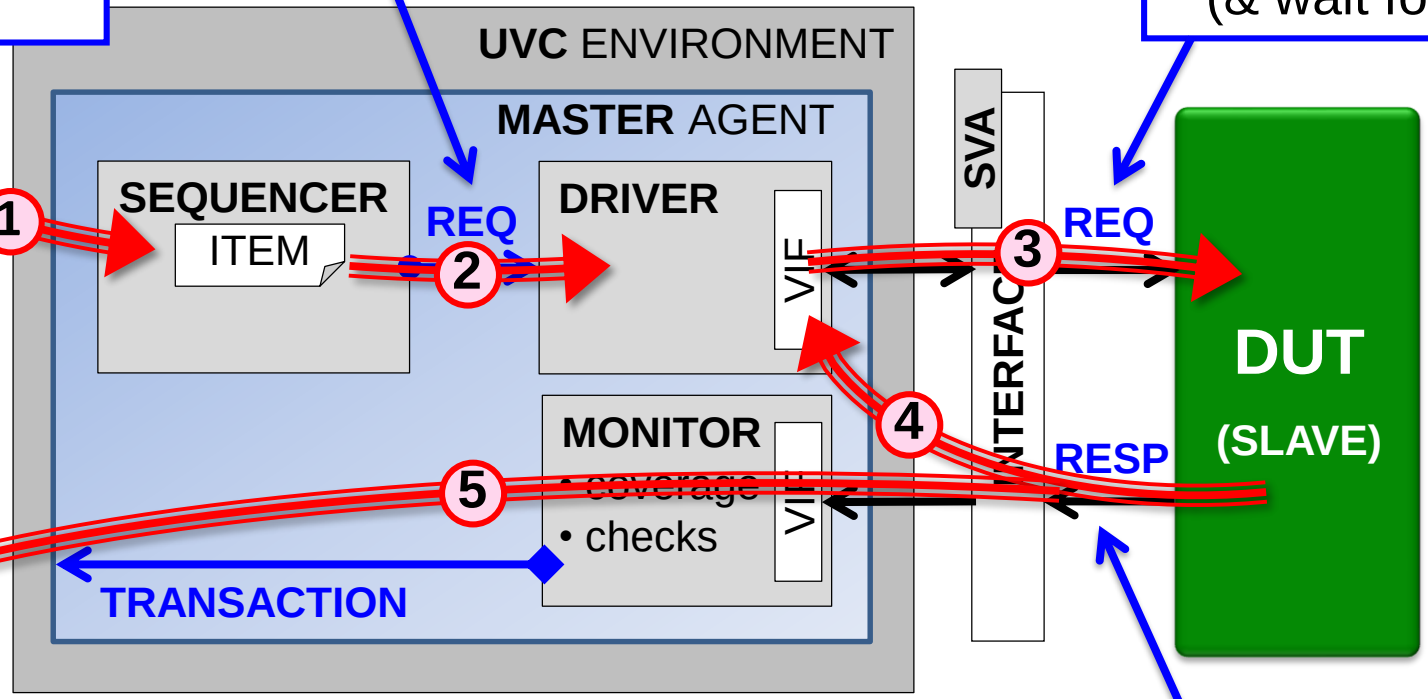
  - Timing of DUT requests is unpredictable (e.g. due to embedded FW execution)
  - UVC must react to request and respond autonomously without blocking test flow

Proactive Master Operation

test or higher-level:
start / do sequence
on sequencer

generate sequence item
& pass to driver via TLM

drive request signals
according to protocol
(& wait for response)



monitor publishes full transactions
via TLM analysis port (REQ & RESP)

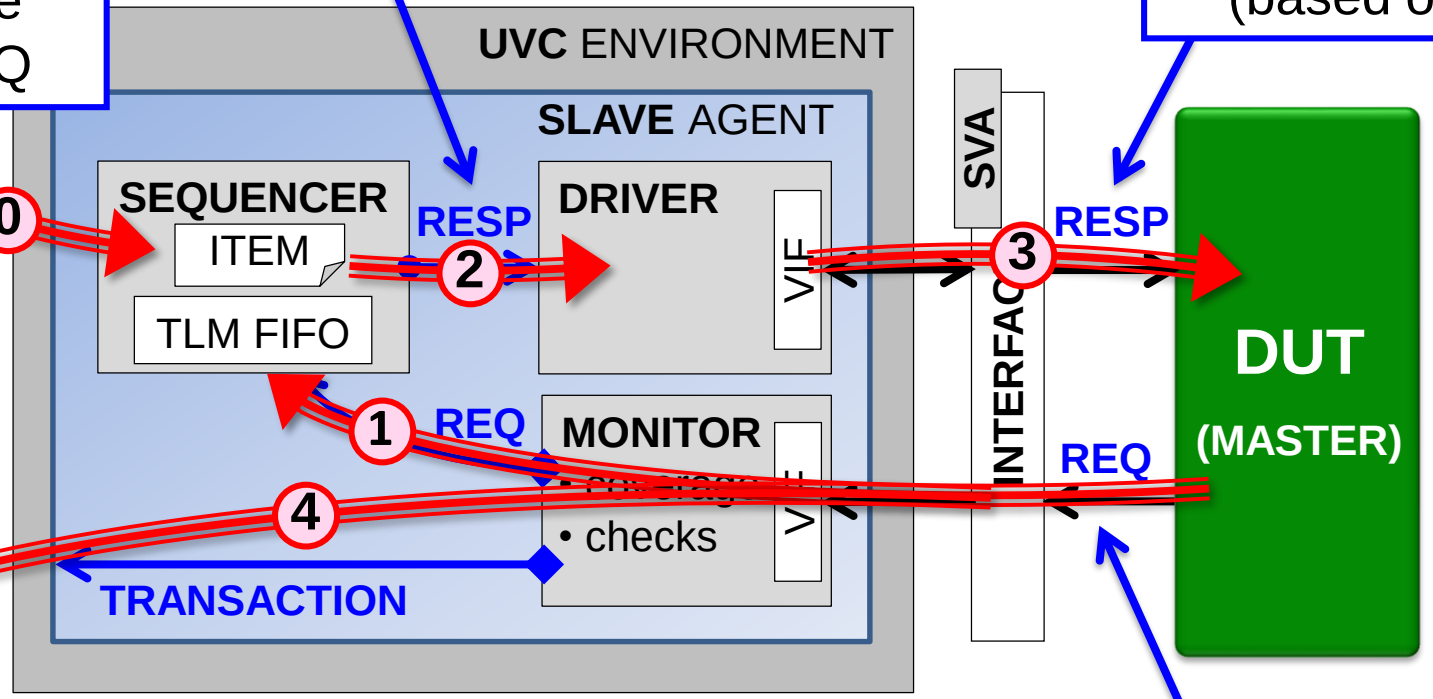
decode response
& return to sequence

Reactive Slave Operation

test or higher-level: start / do / default forever sequence that waits for REQ

generates sequence item & passed to driver via TLM

drive response signals according to protocol (based on request)



monitor publishes full transactions via TLM analysis port (REQ & RESP)

decode request whenever DUT initiates it & publish to sequencer

Sequence Types

- **Normal Sequences:** Generate a ***single*** transaction
- **Virtual Sequences:** Start ***other*** sequences
- **Streaming Sequences:** Generate ***autonomous*** stimulus

Normal Sequences

- **Normal sequences use a sequence item to:**
  - Generate stimulus via a driver
  - Describe required transaction-level stimulus
  - Define a single finite transaction

- bus transactions
- data packet
- power on/reset

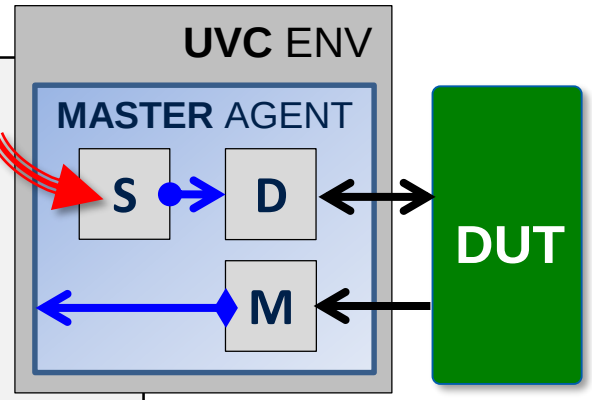
- Key **characteristics**:
  - Driver is not autonomous
  - Fully controllable from virtual sequences
  - Sequence handshake is blocking
  - Sequence items handled consecutively

Return after **complete** transaction (& response)

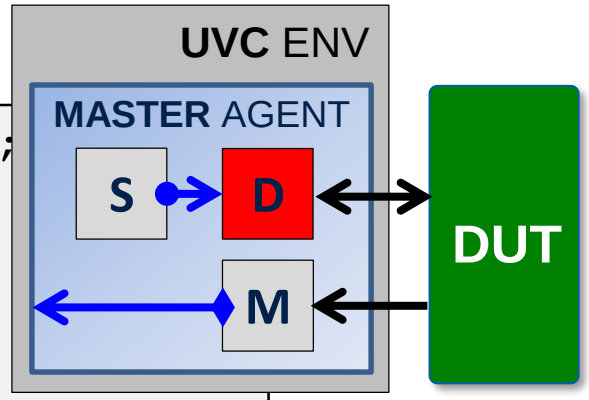
Proactive Master Sequence

```
class my_master_request_seq extends
                              uvm_sequence #(my_master_seq_item);
  rand cmd_enum      cmd;
  rand bit[31:0]     addr;
  rand bit[31:0]     data[];
  ...
  my_master_seq_item m_item;
  ...
  task body();
    `uvm_do_with(m_item,{
      m_item.m_cmd == local::cmd;
      m_item.m_addr == local::addr;
      foreach (local::data[i]) m_item.m_data[i] == local::data[i];
      ...
    })
  ...
```

generate request item based on sequence knobs



Proactive Master Driver




```
class my_master_driver extends uvm_driver#(my_master_seq_item);
  my_master_seq_item m_item;
  ...
  task run_phase(...);
    ...
    forever begin
      seq_item_port.get_next_item(m_item);
      drive_item(m_item);
      seq_item_port.item_done();
    end
  endtask

  task drive_item(my_master_seq_item item);
    ...
  endtask

endclass
```

standard driver-sequencer interaction

drive request signals to DUT
(based on sequence item fields)

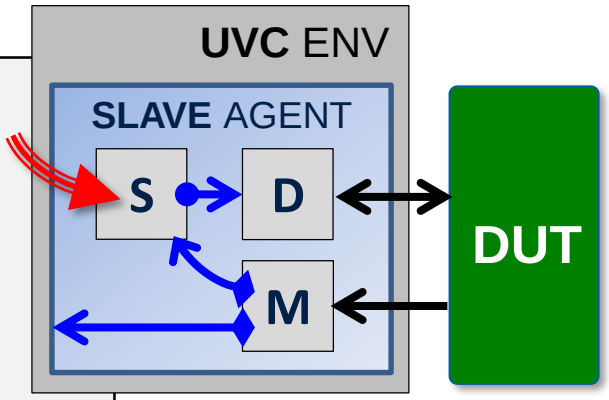
Reactive Slave Sequence

```
class my_slave_response_seq extends
                              uvm_sequence #(my_slave_seq_item);
  my_slave_seq_item m_item;
  my_transaction m_request;
  ...
  task body();
    forever begin
      p_sequencer.request_fifo.get(m_request);
      case (m_request.m_direction)
        READ :
          `uvm_do_with(m_item,{
            m_item.m_resp_kind == READ_RESPONSE;
            m_item.m_delay <= get_max_delay();
            m_item.m_data == get_data(m_request.m_addr);
            ...
          })
  ...
```

call forever loop inside sequence (sequence runs throughout phase: ...do not raise and drop objections!)

wait for a transaction request (*fifo.get* is blocking)

generate response item based on observed request

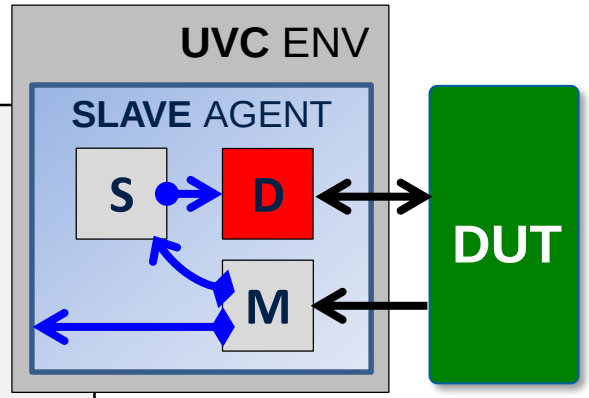


Reactive Slave Driver

```
class my_slave_driver extends uvm_driver #(my_slave_seq_item);
  my_slave_seq_item m_item;
  ...
  task run_phase(...);
    ...
    forever begin
      seq_item_port.get_next_item(m_item);
      drive_item(m_item);
      seq_item_port.item_done();
    end
  endtask

  task drive_item(my_slave_seq_item item);
    ...
  endtask

endclass
```



standard driver-sequencer interaction

identical code structure to proactive master

drive response signals to DUT (based on sequence item fields)

[2] ***Mastering Reactive Slaves in UVM*** – Verilab, SNUG 2016

Virtual Sequences

- **Virtual sequences**:
  - do not directly generate an item
  - coordinate & execute other sequences
  - define scenarios, interaction & encapsulation

- high-level scenarios
- parallel transactions
- multiple agents/UVCs

- **Key characteristics**
  - full control over all child sequences
  - may be blocked by time-consuming sequences
  - multiple virtual sequences may run at same time (nested or parallel) on same virtual sequencer

Must target different resources (not possible for physical sequencers)

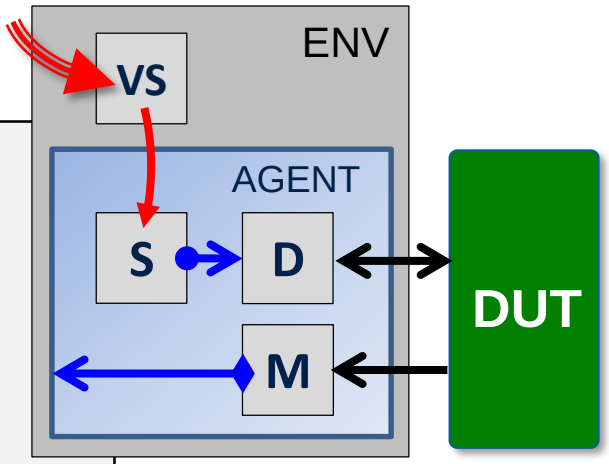
Virtual Sequence

```
class my_virtual_seq extends uvm_sequence;
  // rand fields ...;
  // constraints ...;
  ...
  task body();
    my_poll_fifo_seq poll_fifo_seq;
    i2c_send_data_seq send_data_seq;
    fork
      `uvm_do_with(poll_fifo_seq, {
        timeout == 1ms;
      })
      `uvm_do_on_with(send_data_seq, p_sequencer.i2c_sequencer,{
        slave == 1;
        data == local::data;
      })
    join
    ...
```

fork multiple sequences in parallel

execute on ***this*** **virtual** sequencer (targets different physical sequencer)

execute on referenced **physical** sequencer



Streaming Sequences

- **Streaming is a stimulus pattern where:**
  - Item defines repetitive autonomous stimulus
  - Driver generates derived patterns on its own

- clock generators
- background traffic
- analog waveforms (**real number** models)

- **Key characteristics for successful streaming include:**
  - Sequences (& config) control the autonomous behavior
  - Sequence handshake must be non-blocking
  - Operation can be **interrupted** by a new operation

Sequences can run **forever**

Safely stopped and started again

Streaming Sequence

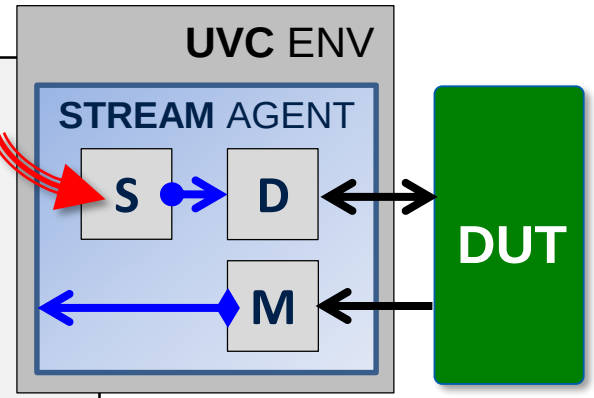
```
class my_ramp_seq extends uvm_sequence #(my_analog_seq_item);
  rand real start;
  rand real step;
  rand time rate;
  ...
  my_analog_seq_item m_item;
  ...
  // constraint start inside {[0.0:0.75]};
  // constraint rate inside {[1ps:100ns]};
  ...
  task body();
    `uvm_do_with(m_item,{
      m_item.m_start == local::start;
      m_item.m_step  == local::step;
      m_item.m_rate  == local::rate;
    })
  ...
```

random real and time control knobs[*]

real and time constraints

totally normal physical sequence structure

generate normal sequence item constrained by control knobs



[*] Or use random integers with scaling factor for real and time values

Streaming Driver

```
class my_analog_driver extends uvm_driver #(my_analog_seq_item);
  my_analog_seq_item m_item;
  ...
  task run_phase(...);
    ...
    forever begin
      seq_item_port.get_next_item(m_item);
      seq_item_port.item_done();
      fork
        seq_item_port.peek(m_item);
        drive_item(m_item);
      join_any
      disable fork;
    end
  endtask
  ...
end
```

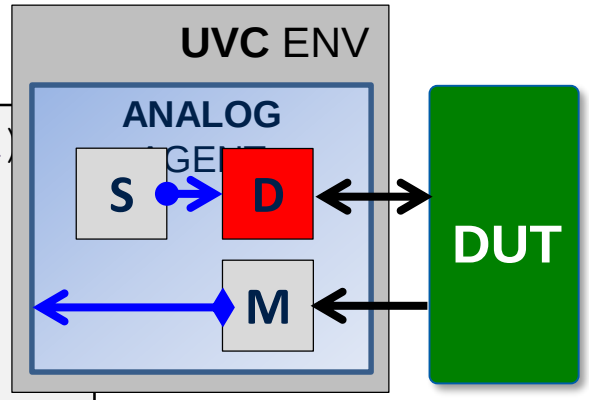
call **item_done** *before* **drive_item** to pass control back to sequencer

blocking peek waits for new item

kill **drive_item** task when new item

```
task drive_item(my_analog_seq_item item);
  real value = item.start;
  forever begin // ramp generation
    vif.data = value;
    #(item.rate);
    value += item.step;
    ... // saturation & looping
  end
  ...
endtask
```

drive request pattern **forever** (unless new item received)



Strategies to apply sequence API to reach project goals

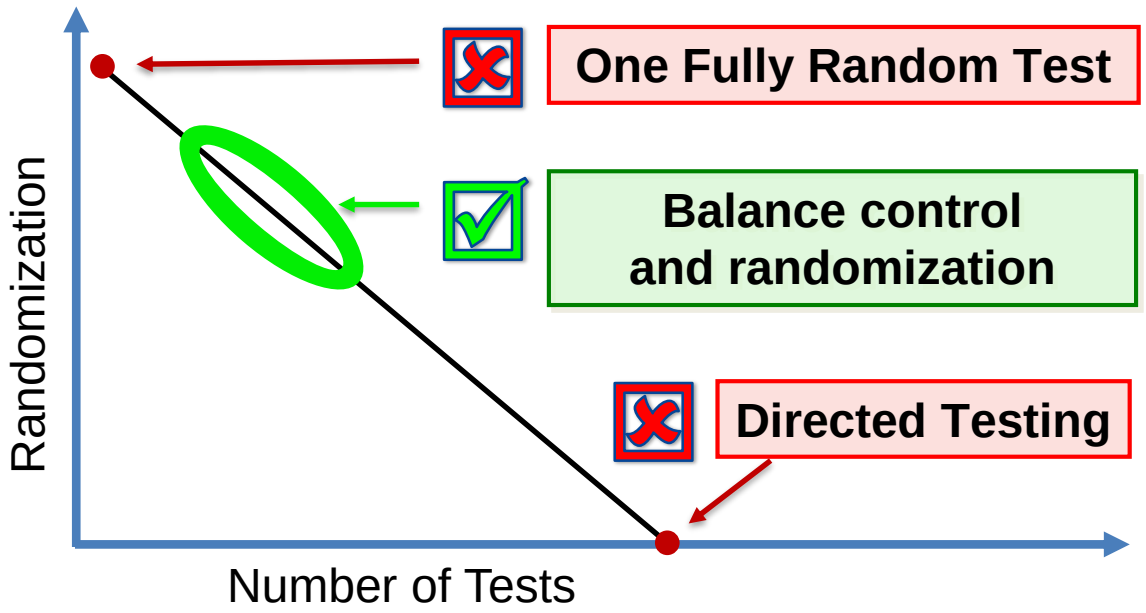
VERIFICATION PRODUCTIVITY

How do we use our Sequence API?

Project Goals
- Meet milestone deadlines
- Maximize chance of finding bugs
- Report status to stakeholders

Project Challenges
- Massive state space to verify
- Need to track progress
- Need a **strategy** to meet goals

Project Risks
- Features ready at later milestones
- Bugs block progress
- Changes in requirements
- Changes in priorities

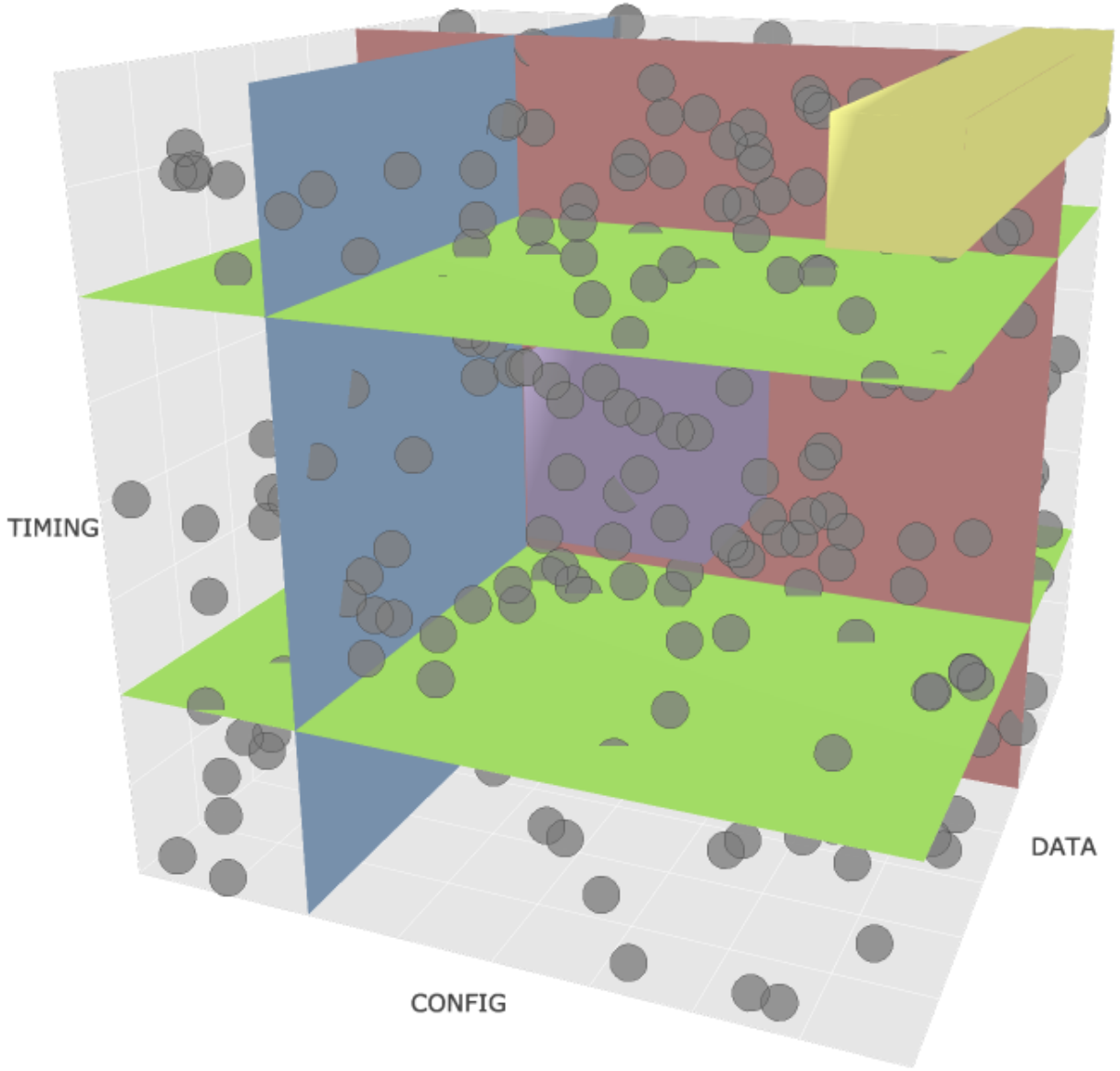


Our sequence API is a powerful tool for solving this

Feature Group Isolation

Test Type	Configuration	Data	Timing
Smoke	**FIXED**	**FIXED**	**FIXED**
Bug-Hunt	RAND	RAND	RAND
Feature	RAND	RAND	**FIXED (Low)**
Feature	RAND	RAND	**FIXED (High)**
Feature	**FIXED**	RAND	RAND
Feature	RAND	**FIXED**	RAND
Corner	MAX	RAND	MAX
Use-Case	TYPICAL	RAND	TYPICAL

Virtual Sequence Control Knob Options



☑ **Coarse-grained Isolation**

Feature Group Isolation

Identify design features that partition *major* DUT functionality

Configurations
- number of channels
- mode of operation
- memory size

Data Patterns
- directed/random
- corner cases
- use-case

Timing
- directed / random
- corner / use-cases
- flow patterns

Map features to sequence control knobs (enum types)

- CH_ALL, CH_1, CH_2
- FAST_MODE, …
- M256, M512, M1024

- FIXED, RAND
- ADC_MAX, ADC_MIN
- DEFAULT, CASE1, …

- FIXED, RAND
- DELAY_MAX, DELAY_MIN
- SEQUENTIAL, PARALLEL

☑ **Can stress any feature in isolation per test**

☑ **Can control mix of features per test**

We can't isolate ***every*** feature.
Choose strategically!

Test Suite Example

Test Name	Configuration		Data	Timing	
	DUT_MODE	CH_CFG	Input	TR_DELAY	DUT_FLOW
seq_flow_test	RAND	RAND	RAND	FIXED	SEQUENTIAL
par_flow_test	RAND	RAND	RAND	FIXED	PARALLEL
fast_mode_test	FAST_MODE	SINGLE	RAND	RAND	RAND
basic_data_test	RAND	RAND	FIXED	RAND	RAND
max_thput_test	FAST_MODE	ALL_CHAN	RAND	MIN_DELAY	PARALLEL
use_case_test	NORM_MODE	TYPICAL	RAND	TYPICAL	PARALLEL

- ☑ **Test scope is obvious**
- ☑ **Bugs less likely to block progress**
- ☑ **Debug issues rapidly: Timing, Data, or Config bug?**
- ☑ **Regression results are implicitly mapped to features**
- ☑ **Easy to allocate regressions to close feature coverage**
- ☑ **Easy status reporting with test-naming conventions**
- ☑ **Can adapt to changing requirements and schedules**
- ☑ **Easy to target corner-cases and use-cases**

Only possible with properly designed sequence API

Where does portable stimulus fit in?

PORTABLE STIMULUS

What is Portable Stimulus?

- **Portable Test and Stimulus Standard (PSS)** [5]

> "[**PSS**] defines a specification for creating a single representation of stimulus and test scenarios ... enabling the generation of different implementations of a scenario that run on a variety of execution platforms..."

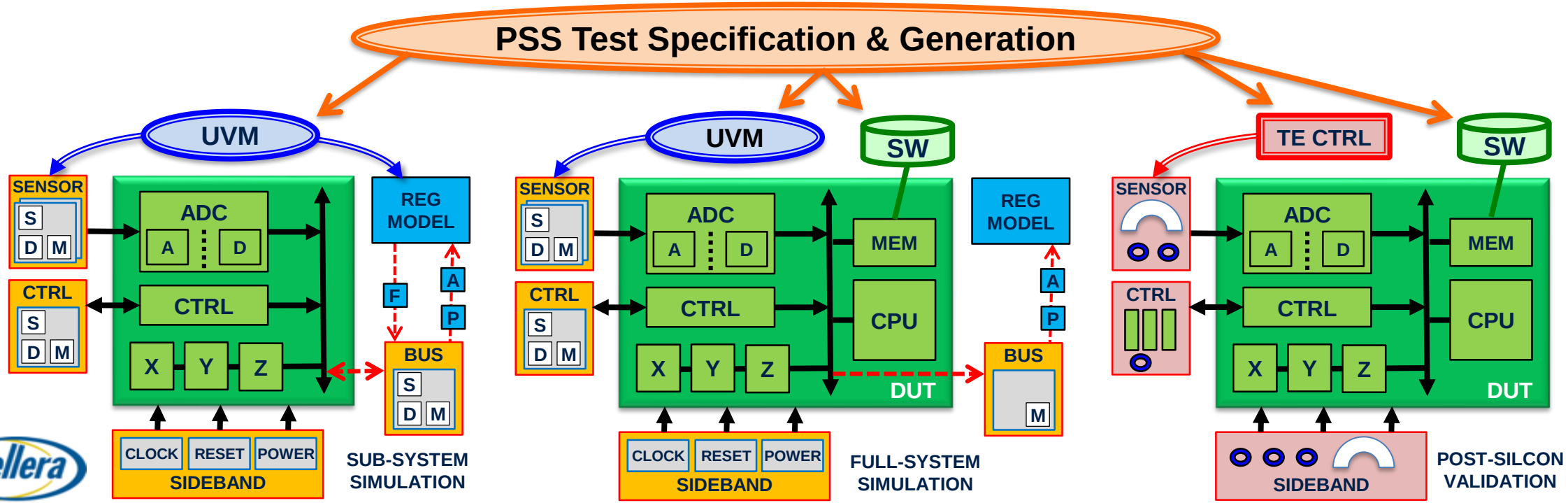
- **Key features:**
  - **higher level of abstraction** for describing test intent
  - test intent is **decoupled** from implementation details
  - declarative domain-specific system modeling language
  - allows test portability between **implementations** and **platforms**
  - executes implementation-specific methods and sequences

- **Does PSS replace all our UVM sequences and stimulus?**
  - no, but it can replace the test layer and some virtual sequences

Test Reuse

PSS addresses reuse of test intent

- reuse from (block to) sub-system to full-system (*within UVM*)
- reuse of tests on different **target implementations** (*e.g. UVM or SW*)
- reuse of tests on different **target platforms** (*e.g. simulation or hardware*)

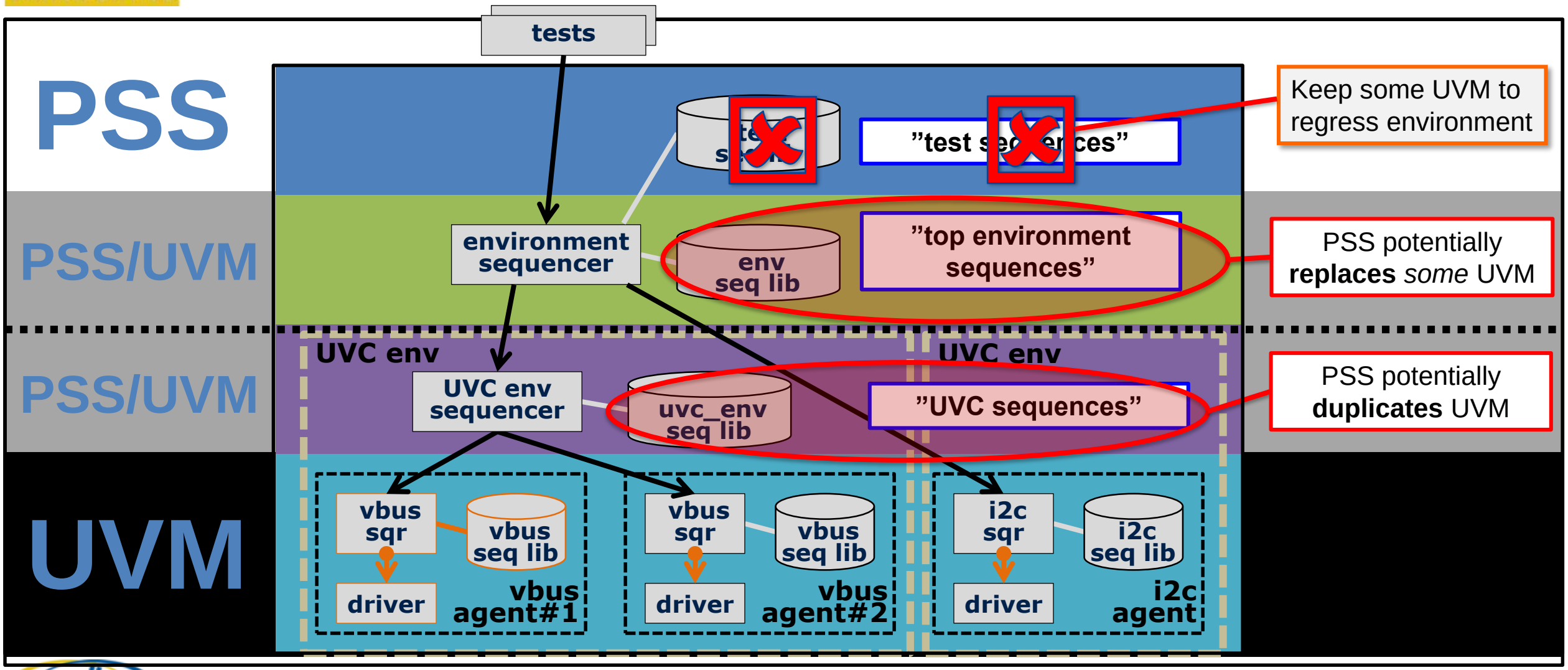


What Changes

- High-level test scenarios & use-cases **delegated to PSS**
  - almost[*] all test sequences & test components **replaced** by PSS
  - many sub-system and full-system scenario virtual sequences **replaced**
- We do not implement these tests in UVM
  - we generate UVM tests from the PSS tools
  - we conceive test scenarios using PSS modeling paradigm
- PSS tests are responsible for corresponding high-level checks
- PSS also has built-in (stimulus) functional coverage capability

[*] Retain some pure UVM tests to sign-off & regress UVM environment

PSS Tests & UVM Sequences



Sequence **guidelines**, **execution**, and **productivity** strategies still apply

CONCLUSION

Conclusion

Common Problems We All Face

- Insufficient API
- Too Complex
- Not Reusable
- Poor visibility of project status
- No risk-management of features

Apply Sequence API Guidelines

- Achieved Control
- Managed Complexity
- Enabled Reuse

Control Advanced Scenarios

- Control Reactive Slaves
- Autonomous Streaming Sequences

Conclusion (cont.)

Project Challenges

- Poor visibility of project status
- No risk-management of features

Apply Sequence API Strategically

- Prioritize features efficiently
- Track and report status easily
- Anticipate and manage project risks

UVM Sequences Remain Vital

- PSS benefits from high quality sequences

References

1 ***Use the Sequence, Luke*** – Verilab, SNUG 2018

2 ***Mastering Reactive Slaves in UVM*** – Verilab, SNUG 2016

3 ***SystemVerilog Constraint Layering via Reusable Randomization Policy Classes,*** John Dickol, DVCon 2015

4 ***Advanced UVM Tutorial*** – Verilab, DVCon 2014

5 ***Portable Test and Stimulus Standard***, Version 1.0, Accellera

Verilab **papers** and **presentations** available from:
http://www.verilab.com/resources/papers-and-presentations/

2019 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

Q & A

jeff.vance@verilab.com

jeff.montesano@verilab.com

accellera SYSTEMS INITIATIVE

The following images are licensed under CC BY 2.0

- Verdi_Requiem 008 by Penn State
- Sheet music by Trey Jones
- Flutist by operaficionado
- Cello Orchestra by Malvern St James
- Folk_am_Neckar_4915 by Ralf Schulze
- Flute by Bmeje
- Cello by Lori Griffin