Be a Sequence Pro to Avoid Bad Con Sequences

Presenter: Mark Litterick
Contributors: Jeff Vance, Jeff Montesano
UVM Overview

Verification Environment

Universal Verification Methodology (UVM)

SystemVerilog IEEE1800

Questa™  Xcelium™  VCS™

- Application-specific code
- Uses UVM & SystemVerilog

- Open source (Apache)
- Class library & methodology
- Facilitates interoperability
- Uses SystemVerilog

- Supported by all simulators

- Multi-language simulators
  - VHDL, Verilog, SV, SC

UVM provides:
- base class library
- methodology for using it

but methodology allows flexibility
- it is necessary
- best practices
Tutorial Content

- Behind the Scenes of the UVM Factory
- Advanced UVM Register Modeling & Performance
- Demystifying the UVM Configuration Database
- Effective Stimulus & Sequence Hierarchies
- Vertical & Horizontal Reuse of UVM Environments
- Configuration Object & config_db Usage
- Parameterized Classes, Interfaces & Registers
- Adaptive Protocol Checks - Config Aware Assertions
- Self-Tuning Functional Coverage

[1] DVCon EU 2014 – Advanced UVM
[3] DVCon EU 2018 – UVM Audit
DVCon EU 2019 – UVM Sequences
Problem Statement

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
Tutorial Outline

• Background to tutorial
• Introduction to sequences – *what are they & why do we care*
• Sequence execution – *masters, reactive slaves, streaming data*
• Sequence guidelines – *improve control, complexity, & reuse*
• 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?

UVM sequence class encapsulates constrained-random stimulus

class access_seq extends base_seq;
    rand master_enum source;
    rand cmd_enum cmd;
    rand bit[15:0] addr;
    bit[15:0] data;

    constraint legal_c {
        cmd != NOP;
        addr <= MAX_ADDR_C;
    }

    task body();
    ...
    if (p_sequencer.cfg.mode == ACTIVE)
        ...

Sequences can be reused, extended, randomized, and combined sequentially and hierarchically in interesting ways to produce realistic stimulus… [UVM ref. man.]

Random control knobs for users
Constraints on random options
Procedural body
Access resources via sequencer
Comparing Sequences and Tasks

Both **sequences** and **tasks**:
- encapsulate a **scenario**
- provide control **options**
- have **procedural** body

```verilog
class access_seq extends base_seq;
    rand master_enum source;
    rand cmd_enum cmd;
    rand bit[15:0] addr;
    bit[15:0] data;
    constraint legal_c {...}
    task body();
    ...

task access_task (    master_enum source,
            cmd_enum cmd,
            bit[15:0] addr,
            ref bit[15:0] data);

    // task body
    ...
```
Executing Sequences and Tasks

In the most **basic** cases, tasks and sequences can be equivalent...
Why Bother Using Sequences?

<table>
<thead>
<tr>
<th>Tasks</th>
<th>Sequences</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arguments are mandatory (or fixed by default)</td>
<td>Arguments are optional (random by default)</td>
</tr>
<tr>
<td>Users must randomize args</td>
<td>Legal randomization is built-in</td>
</tr>
<tr>
<td>No built-in arg validity checks</td>
<td>Constraints validate user options</td>
</tr>
<tr>
<td>Awkward to return data (use ref arguments)</td>
<td>Caller can access any data (with sequence handle)</td>
</tr>
<tr>
<td>No built-in access to resources</td>
<td>Access to resources via sequencer</td>
</tr>
<tr>
<td>Adding args breaks existing code</td>
<td>Adding options has minimal impact</td>
</tr>
</tbody>
</table>

© Verilab & Accellera Systems Initiative
Orchestrating Stimulus

- Test Scenario Constraints
- Virtual Sequencer
  - Sequence A
  - Sequence B
  - Sequence C
    - Driver A
    - Driver B
    - Driver C
Sequence API Strategy

Virtual sequencers:
- vbus
- sqr
- driver
- vbus
- seq
- lib
- agent#1

UVC env
- UVC env sequencer
- uvc_env
- seq
- lib

"test sequences"

"top environment sequences"

Physical sequencers:
- vbus
- sqr
- driver
- vbus
- seq
- lib
- agent#1

- vbus
- sqr
- driver
- vbus
- seq
- lib
- agent#2

- i2c
- sqr
- driver
- i2c
- seq
- lib
- agent

"UVC sequences"

"test sequences"

© Verilab & Accellera Systems Initiative
### Sequence Layer Roles

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

- **Each layer resolves a subset of random options**
- **Benefits both directed and random tests**
- **Reduce complexity at each layer**
- **Control with intuitive APIs**
- **Sequences decoupled and reusable**
- **Existence of some layers is application dependent**

© Verilab & Accellera Systems Initiative
How different types of sequences are executed

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 stream description
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

© Verilab & Accellera Systems Initiative
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

© Verilab & Accellera Systems Initiative 2019
Sequence Types

• Virtual Sequences: Control other sequences

• Normal Sequences: Generate a single transaction

• Streaming Sequences: Generate autonomous stimulus
Virtual Sequences

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

• **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)

• high-level scenarios
• parallel transactions
• sequential transactions
• multiple agents/UVCs
• constraint wrapper
• resource encapsulation

© Verilab & Accellera Systems Initiative
class my_virtual_seq extends uvm_sequence;
   // rand fields ...;
   // constraints ...;

   task body();
       bus_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
Normal Sequences

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

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

• **bus transactions**
• **data packet**
• **power on/reset**

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

© Verilab & Accellera Systems Initiative
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);
          ...
        })
    ...
  endcase
endtask

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

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

generate response item based on observed request
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
Streaming Sequences

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

• **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**

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

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

call item_done before drive_item to pass control back to sequencer

blocking peek waits for 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
... // saturation & looping
endtask

drive request pattern forever (unless new item received)

kill drive_item task when new item
How to maximize the benefits of using sequences

SEQUENCE GUIDELINES
Sequence Constraint Guidelines

- Produce legal stimulus by default
- Use sequence layers to isolate constraints
- Minimize the number of control knobs
- Use dedicated constraint blocks to support extensibility\[^5\]
- Use soft constraints carefully and sparingly\[^5\]
- Use descriptor objects to encapsulate complex constraint sets\[^5\]
- Use policy classes to redefine constraints and bypass layers\[^6\]

\[^5\] *Use the Sequence, Luke* – Verilab, SNUG 2018

\[^6\] *SystemVerilog Constraint Layering via Reusable Randomization Policy Classes* – John Dickol, DVCon 2015
Produce legal stimulus by default

class bus_burst_seq extends base_seq;
   rand int addr;   // start address
   rand int length; // burst length
   ...               // other fields (direction, data, etc.)

constraint legal_c {
   addr%4 == 0;
   length inside {16,64,256,1024};
   addr + length <= p_sequencer.cfg.get_max_addr();
}

// user code examples
`uvm_do(burst_seq)                         // any burst
`uvm_do_with(burst_seq, {addr == 'h4321;}) // any length
`uvm_do_with(burst_seq, {length == 256;})   // any address
`uvm_do_with(burst_seq, {addr == 'hFFF0; length == 1024;})

Provide 0, 1 or more inline constraints
Legal stimulus guaranteed
Constraint solver detects illegal values

address misaligned? illegal combination?
Use sequence layers to isolate constraints

```plaintext
class bus_burst_seq extends base_seq;
    rand int    addr;  // start address
    rand burst_t burst;  // burst type
    ...  // other fields (direction, ...

constraint legal_c {
    burst inside {WRAP,INCR};
    (burst == INCR) -> addr%4 == 0;  // INCR must be aligned
    ...
}

`uvm_do_with(burst_seq, {addr==`h4321; burst inside {WRAP,INCR};})

rand burst_t burst;
constraint burst_c {burst inside {WRAP,INCR};}
`uvm_do_with(burst_seq, {addr==`h4321; burst == local::burst;})
```

Two-Step Randomization:
1. randomize local variables
2. randomize lower sequences
   • easy to debug

Without layer isolation:
• unexpected distribution
• illegal combinations
• hard to debug

only does WRAP bursts!

constraint violation detected!
Minimize the number of control knobs

```plaintext
class master_write_seq extends base_seq;
  rand int addr; // address
  ...
protected rand int slave; // derived field
  ...
constraint slave_c {
  slave == p_sequencer.cfg.get_slave_id(addr);
}
  ...

  task body();
    slave_bus_seq bus_seq;
    `uvm_do_with(bus_seq, {
      bus_seq.dir == WRITE;
      bus_seq.prot == p_sequencer.cfg.get_prot();
      bus_seq.addr == local::addr;
      bus_seq.slave == local::slave;
    })
```

- **exposed control knob**
- **hidden control knob**
- **fixed lower-level knob**
- **background config**

Sequences are easy to use

Sequences are hard to abuse

Typically lower-level sequences have more control knobs
Sequence Reuse Guidelines

• Make test sequences independent of testbench architecture
• Use configuration objects and accessor methods to adapt to setup
• Use utility methods to support self-tuning sequences
Make tests independent of architecture

```verilog
class test_feature_seq extends base_test_seq; // test seq
...
  `uvm_do_with(config_seq, {mode == FAST; crc_en == 0;})
  `uvm_do_with(write_seq, {addr == 'hFFF0; data == 'h5555;})

class dut_config_seq extends base_seq; // mid-level seq
...
  regmodel.block.CTRL_REG.SPEED.set(int'(mode));
  regmodel.block.CTRL_REG.CRCEN.set(crc_en);
  regmodel.block.CTRL_REG.update(status);

class bus_write_seq extends base_seq; // mid-level seq
...
  `uvm_do_on_with(cmd_send_seq, p_sequencer.master_sequencer[0], {
    cmd_send_seq.addr == local::addr;
    cmd_send_seq.data == local::data;
  })
```

- Test sequences decoupled from testbench architecture
- Tests are generic and reusable on derivative projects
Use config & access methods to adapt seqs

```verilog
class bus_config extends uvm_object;
  rand int fifo_depth;
  rand time clk_period;
  ...
  constraint fifo_c {fifo_depth inside {[1:`MAX_DEPTH]});
  constraint clk_c {clk_period inside {[10ns:100ns]});
  ...
  function int get_fifo_depth(); // current fifo depth
  function time get_transfer_time(int len); // len * clk_period

class fifo_seq extends test_base_seq; // test or mid-level seq
  ...
  repeat (p_sequencer.cfg.get_fifo_depth())
    ... // do something
    `uvm_do(start_seq)
    # (p_sequencer.cfg.get_transfer_time(length));
    `uvm_do_with(status_seq, {exp_flag == 1; exp_error == 0;})
```

Keep project-specific configuration constraints outside of sequences

Use config access methods in sequences

Sequence adapts to current configuration

Sequence is generic and reusable
Use utility methods to support self-tuning

Define utility methods in package scope, base seq, or config (depends on who needs to use it – e.g. driver, monitor, or just seq)

```plaintext
function automatic int calc_base_address(int addr);
  return (addr / DATA_WORDS_PER_ADDR_C);

function automatic int calc_data_offset(int addr);
  return ((addr / DATA_WORD_SIZE_C) % DATA_WORDS_PER_ADDR_C);
```

Be aware – default lifetime:
- in class is automatic
- in package is static

class write_word_seq extends base_seq;
  rand bit[31:0] addr;
  ...
  task body();
  `uvm_do_with(write_single_seq, {
    write_single_seq.addr == calc_base_address(addr);
    write_single_seq.word_sel == calc_data_offset(addr);
  })
  ...

Avoid code duplication between sequences
Sequences adapt to changes in calculations

Use utility methods in sequences
Sequence Library Tips

• Use typedef header at top of sequence library file
• Provide random and fixed versions of configuration sequences
• Implement start and end messages in base sequence
• Group related sequences into sequence library files\(^5\)
• Construct sequences using inheritance or composition\(^5\)
• Use enumerated types for improved clarity\(^5\)
Use typedef header at top of seq lib file

typedef class init_seq;    // initialization
typedef class power_on_seq; // power ramp
typedef class reset_seq;   // hard reset

class init_seq extends base_seq;
    power_on_seq pwr_seq;
    reset_seq rst_seq;
...
endclass

class power_on_seq extends base_seq;
...
endclass

class reset_seq extends base_seq;
...
endclass

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

typically **many lines of code per file**
(sequence libraries are large files)

✅ documents content

✅ allows sequences used in any order
Provide random and fixed config seqs

class config_random_seq extends base_seq; // mid-level seq
    rand speed_mode_t mode;
    rand bit crc_en;
    rand int trim;
...
constraint mode_c {mode inside {FAST, SLOW};}
constraint trim_c {trim inside {[1:15]};}
...
regmodel.block.CTRL_REG.SPEED.set(int'(mode));
regmodel.block.CTRL_REG.CRCEN.set(crc_en);
regmodel.block.CTRL_REG.update(status);

class config_fixed_seq extends config_random_seq;
constraint fixed_c {
    soft mode == SLOW;
    soft crc_en == 1;
    soft trim == 1;
    `uvm_do(random_seq)
    `uvm_do(fixed_seq)
    `uvm_do_with(fixed_seq, {mode == FAST;})
}
Implement start and end messages in base seq

```verilog
class base_seq extends uvm_sequence;

    task pre_start();
        super.pre_start();
        `uvm_info(..., "Starting...", UVM_MEDIUM)
        `uvm_info(..., "{Sequence:\n",this.sprint}, UVM_HIGH)
    endtask

    task post_start();
        super.post_start();
        `uvm_info(..., "Completed.", UVM_HIGH)
    endtask
```

- **Use** `pre/post_start()`

- **Do not use** `pre/post_body()`
  - not called by `uvm_do*(seq)`

- **Do not use** `pre/post_do()`
  - not called by `seq.start()`

**Generic** non-verbose messages inherited by all sequences

**Use** `pre/post_start()` called by `seq.start()` and `uvm_do*(seq)`
Strategies to apply sequence API to reach project goals

VERIFICATION PRODUCTIVITY
How do we leverage our Sequence API?

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

**Project Challenges**
- Massive evolving state space to verify
- Need to make & track progress despite bugs
- Need to adapt to changing requirements & priorities

Sequence architecture is key part of strategy to meet goals
Feature Group Isolation

<table>
<thead>
<tr>
<th>Test Type</th>
<th>Configuration</th>
<th>Data</th>
<th>Timing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smoke</td>
<td>FIXED</td>
<td>FIXED</td>
<td>FIXED</td>
</tr>
<tr>
<td>Bug-Hunt</td>
<td>RAND</td>
<td>RAND</td>
<td>RAND</td>
</tr>
<tr>
<td>Feature</td>
<td>RAND</td>
<td>RAND</td>
<td>FIXED (Low)</td>
</tr>
<tr>
<td>Feature</td>
<td>RAND</td>
<td>RAND</td>
<td>FIXED (High)</td>
</tr>
<tr>
<td>Feature</td>
<td>FIXED</td>
<td>RAND</td>
<td>RAND</td>
</tr>
<tr>
<td>Feature</td>
<td>RAND</td>
<td>FIXED</td>
<td>RAND</td>
</tr>
<tr>
<td>Corner</td>
<td>MAX</td>
<td>RAND</td>
<td>MAX</td>
</tr>
<tr>
<td>Use-Case</td>
<td>TYPICAL</td>
<td>RAND</td>
<td>TYPICAL</td>
</tr>
</tbody>
</table>

Virtual Sequence Control Knob Options

Coarse-grained Isolation

© Verilab & Accellera Systems Initiative
Mapping features to control knobs

Identify design features that partition *major* DUT functionality

<table>
<thead>
<tr>
<th>Configurations</th>
<th>Data Patterns</th>
<th>Timing</th>
</tr>
</thead>
<tbody>
<tr>
<td>• number of channels</td>
<td>• directed/random</td>
<td>• directed / random</td>
</tr>
<tr>
<td>• mode of operation</td>
<td>• corner cases</td>
<td>• corner / use-cases</td>
</tr>
<tr>
<td>• memory size</td>
<td>• use-case</td>
<td>• flow patterns</td>
</tr>
</tbody>
</table>

Map features to sequence control knobs (enum types)

- **Configurations**
  - CH_ALL, CH_1, CH_2
  - FAST_MODE, ...
  - M256, M512, M1024

- **Data Patterns**
  - FIXED, RAND
  - ADC_MAX, ADC_MIN
  - DEFAULT, CASE1, ...

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

© Verilab & Accellera Systems Initiative

Stress features in isolation

Combine features in groups

We can't isolate *every* feature. Choose strategically!
# Test Suite Example

<table>
<thead>
<tr>
<th>Test Name</th>
<th>Configuration</th>
<th>Data</th>
<th>Timing</th>
</tr>
</thead>
<tbody>
<tr>
<td>seq_flow_test</td>
<td>DUT_MODE</td>
<td>CH_CFG</td>
<td>Input</td>
</tr>
<tr>
<td>par_flow_test</td>
<td>DUT_MODE</td>
<td>CH_CFG</td>
<td>Input</td>
</tr>
<tr>
<td>fast_mode_test</td>
<td>DUT_MODE</td>
<td>CH_CFG</td>
<td>Input</td>
</tr>
<tr>
<td>basic_data_test</td>
<td>DUT_MODE</td>
<td>CH_CFG</td>
<td>Input</td>
</tr>
<tr>
<td>max_thput_test</td>
<td>DUT_MODE</td>
<td>CH_CFG</td>
<td>Input</td>
</tr>
<tr>
<td>use_case_test</td>
<td>DUT_MODE</td>
<td>CH_CFG</td>
<td>Input</td>
</tr>
</tbody>
</table>

- **Test scope is obvious**
- **Bugs less likely to block progress**
- **Debug issues rapidly:** Timing, Data, or Config bug?
- **Can adapt to changing requirements and schedules**
- **Easy to target corner-cases and use-cases**

- **Regression results are implicitly mapped to features**
- **Easy to allocate regressions to close feature coverage**
- **Easy status reporting with test-naming conventions**

© Verilab & Accellera Systems Initiative
Where does portable stimulus fit in?

PORTABLE STIMULUS
What is Portable Stimulus?

• Portable Test and Stimulus Standard (PSS) [7]

“[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

- **vertical 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
  – some sub-system and full-system scenario virtual sequences replaced by PSS

• **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
Sequence execution, guidelines and productivity still apply

PSS replaces all UVM

PSS replaces some UVM

PSS potentially duplicates UVM

All UVM checks and functional coverage still available
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

© Verilab & Accellera Systems Initiative
References

1. *Advanced UVM Tutorial* – Verilab, DVCon Europe 2014
2. *UVM Reuse Tutorial* – Verilab, DVCon Europe 2015
3. *UVM Audit Tutorial* – Verilab, DVCon Europe 2018
5. *Use the Sequence, Luke* – Verilab, SNUG 2018

Q & A

mark.litterick@verilab.com
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**