Advanced UVM Tutorial
Taking Reuse to the Next Level

Mark Litterick
Jason Sprott
Jonathan Bromley
Agenda

• Vertical & Horizontal Reuse of UVM Environments

• Self-Tuning Functional Coverage
  - Strategy & Implementation

• Adaptive Protocol Checks
  - Configuration Aware Assertions

• Configuration Object Encapsulation &
  Appropriate config_db Usage

• Parameterized Classes, Interfaces & Registers
Vertical & Horizontal Reuse of UVM Environments

Mark Litterick
What is Verification Reuse?

“Code reuse, is the use of existing software, or software knowledge, to build new software” Wikipedia

• Verification Reuse:
  – achieving verification goals through effective code reuse

• Specifically:
  – reuse verification components between environments
  – reuse verification components between hierarchies
  – reuse verification environments between projects
  – also reuse of base-classes, patterns & methodology

• Several different paradigms:
  – referred to as horizontal and vertical reuse
Horizontal Reuse

- Reusing verification code *without changing role*
  - similar responsibilities, structure & use-case
- Reuse *stimulus, checks, coverage & messages*
  - same stimulus API presented to user or test writer
- **Horizontal reuse** usually means *predictable use-case*
  - typically achieved by configuration, adaption and flexibility
  - code is designed for more than one known use-case
Horizontal Reuse examples

• Reuse **verification component** in multiple envs

![Diagram](image)

SAME ROLE, DIFFERENT ENVIRONMENT

DIFFERENT DUT

• Reuse **verification environment** in multiple projects

![Diagram](image)

SAME ROLE & ENV, DIFFERENT CONFIG

DERIVATIVE DUT
Vertical Reuse

• Reusing verification code with a different role
  – different responsibilities, structure & use-case

• Reuse stimulus, checks, coverage & messages, but...
  – stimulus nested or layered inside high-level scenario
    (with different user API)
  – or stimulus not reused at all
    (but checks, coverage & messages are)

• Vertical reuse usually means unknown use-cases
  – typically achieved by encapsulation and extensibility
  – code is designed to allow the user to do other things
Vertical Reuse Examples

- **Reuse passive** block-level in **top-level** environment
  - DIFFERENT STRUCTURE, NO STIMULUS
    (STIMULUS COMES FROM ANOTHER UVC)

- **Reuse active** component in **top-level context**
  - DIFFERENT ROLE, STIMULUS IS PART OF HIGH-LEVEL SCENARIO (NOT SO RANDOM)
Reuse in UVM

• **UVM enables reuse**, but does *not guarantee* it
  – SystemVerilog **object-oriented** program paradigm (OOP)
  – UVM provides reusable **base-classes & methodology**
  – UVM reuses standard **software patterns** for utilities:
    factory, configuration database, event pools, phases, ...

• **Reuse affects all aspects** of environments, including:
  – architecture
  – configuration
  – stimulus
  – checks
  – coverage
  – modeling
Tutorial Content

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

VERTICAL REUSE
Top-Level Verification Requirements

• **Top-Level** has different **concerns** to block-level
  – functional **correctness** & **performance** of full **application**
  – **interaction** of modules, sub-systems & **shared resources**
  – operation with all **clock, power & test** domains
  – **connectivity** of all blocks & sub-systems

• Cannot **achieve closure** on all these by looking only at external pins in a complex top-level SoC
  – require **checks, coverage & debug messages** at all levels

---

**NOT ENOUGH TO PROVE OPERATION OF THE PERFECT CHIP**
(ALSO NEED TO **ISOLATE & DEBUG FAILURES** EFFECTIVELY)

© Verilab & Accellera
Vertical Reuse

• Three observations:
  – vertical reuse is hard to prepare for due to change of role and unknown use-cases
  – top-level environment often must be developed in parallel with block-level
  – vertical reuse is effort for implementer and only adds value for the re-user

• Reality check:
  – don’t try to second guess everything the user might need
  – get structure right, validate operation & allow for adaption
Active/Passive Operation

- **Active** components provide or **affect the stimulus** driven to the DUT, influencing the stimulus flow
- **Passive** components do **not** provide or **affect the stimulus** in any way, they just observe
- **Active/Passive mode** affects:
  - run-time architecture
  - functional capability
  - error tolerance
  - debug capability
Active Block-Level

- **Stimulus** is provided by sequencers and drivers
  - proactive master generates request stimulus
  - reactive slave generates response stimulus
- **Checks** performed by interfaces, monitors & scoreboards
- **Coverage** is collected by monitors & scoreboards
- **Messages** are generated by all components
Passive Block-Level

- **No stimulus** is performed by passive components
  - sequencers and drivers are not even present
- **Checks** are still performed by interfaces, monitors & scoreboards
- **Coverage** is still collected by the monitors & scoreboards
- **Messages** are generated by the remaining components
Passive Reuse in Top-Level

ALL CHECKS, COVERAGE & MESSAGES REUSED TO ENSURE:
• EFFECTIVE VALIDATION OF BLOCK IN TOP-LEVEL CONTEXT
• COMPREHENSIVE TOP-LEVEL OPERATION AND DEBUG
Active/Passive Misuse

- low-level UVC OK but `env` ignores active/passive
- drivers post `sequence items` to `scoreboard`
- `drivers` doing functional `timeout checks`
- `coverage` defined in `active` components
- `configuration updates` from sequence or driver
- important `messages` from drivers
- `error injection` traffic reported as a `warning`
- `passive` components control `end-of-test` schedule
- uncontrollable `end-of-test` scoreboard checks
- ...
**Active/Passive Guidelines**

- Complete env must consider active/passive
- Do *not* connect scoreboards to active components
- Perform functional checks in passive comps
- Collect functional coverage in passive comps
- Generate important messages in passive comps
- Update configuration only from passive comps
- Promote warnings to errors in passive mode
- Do *not* control end-of-test from passive comps
- Allow disable for scoreboard end-of-test check

---

**Notes:**

- (Obviously) active component is not present in passive mode
- (Not so obviously) active stimulus is also not present
- Internal bus may still be active when top scenario completes
- Top-level scenario decides when to end the test

---

*Update pseudo-static config:*
- *Not sequence, but monitor*
- *Not reg write, but predict*
Problems of Scale

• **Top-level** environment is already **complex**
• Integrating *many block-level* environments introduces additional requirements:
  – build and integration **encapsulation**
  – **configuration** containment and **consistency**
  – **namespace** isolation and **cohabitation**

**BLOCK-LEVEL SUPPLIERS NEED TO MAKE THINGS AS EASY AS POSSIBLE TO INTEGRATE AND REUSE**
Problems of Scale

- environment pulled together only in **base-test**
- **multiple** dispersed **config** objects and fields
- **multiple** agent **interfaces** required for top-level
- **macro definitions** have global scope
- **enumeration literals** and types without prefix
- inflexible **inappropriate** low-level **coverage**
- incorrect **message verbosity** resulting in **clutter**
- ...

© Verilab & Accellera 21
 encapsulate all sub-components in env not test
use hierarchical configuration objects for env
combine multiple I/Fs into hierarchical interfaces
avoid namespace collisions by using scope prefix
encapsulate coverage in separate class for overload
apply more rigorous message verbosity rules
...
Un-Encapsulated Reuse

- Many inconsistent config objects & fields to maintain
- Many small interfaces to instantiate & set in config_db
- Un-encapsulated reuse is error-prone, hard to maintain & too much effort
- Many components to instantiate & connect

© Verilab & Accellera
Encapsulated Reuse

HIERARCHICAL CONFIG (TOP CONTROLS BLOCKS, BLOCKS CONTROL UVCS)

HIERARCHICAL INTERFACE (1 INST & 1 SET IN CONFIG_DB)

PROPER ENCAPSULATION MAKES VERTICAL REUSE OF MANY BLOCK-LEVEL ENVS A FEASIBLE PROPOSITION
What is the Cost of Reuse?

• Additional **effort** in **verification components**
  – architectural concepts are provided by methodology
  – shortcuts & quick fixes save time in initial project...
  – ...but in the long term all bad coding style costs money
  – effort here **benefits** both **supplier** and **reuser**

• Additional **effort** in **verification environments**
  – packaging environment, config, interfaces all costs effort
  – effort here **only** really **benefits reuser**

• Total **cost** is relatively **low** but not **zero**
Reality Check

• Observation: since reuse does not come for free...
  • ... it will not be correctly implemented in first instance!

• Specifically:
  • first project implementing block-level has other priorities
  • top-level project needs to start in parallel with block-level
  • second project using the block should factor in the costs
  • trade-off reuse costs verses design from scratch
  • trade-off reuse costs verses quality & debug improvements
  • architectural fixes for reuse should be retrofitted to source (since working regression provides the best cross-check)
What is Retrofitting?

“Retrofitting refers to the addition of new technology or features to older systems”  Wikipedia

• Retrofitting Reuse:
  • adding reuse capability to an existing component or environment that does not yet support reuse

• Specifically:
  • fixing a verification component to comply with guidelines
  • re-architecting a block-level environment for vertical reuse
  • …attempting reuse of an environment for the first time!
Strategy for Retrofitting

• First, determine **what** you have
  • probably supplied documentation will not be good enough
  • print & review the topology, config and factory settings

• Next, determine **scope** of rework
  • active/passive build control or major scoreboard reconnect?

• Implement **changes** in block-level environment
  • run **passing** block-level **regressions** throughout
  • validate **passive** operation at block-level with **shadow** instance

• **Use** in the top-level environment
  • ensure top-level continues to run with **no bad side-effects**
  • ensure block-level checks, coverage and messages work

**VALIDATE PASSIVE MODE USING BLOCK-LEVEL REGRESSIONS**
Passive Shadow

TWO INSTANCES OF THE SAME ENVIRONMENT
ONE IN ACTIVE MODE, ONE IN PASSIVE MODE

SHADOW PASSIVE ENVIRONMENT

BLOCK-LEVEL BASE TEST

NORMAL ACTIVE ENVIRONMENT

PROVE FUNCTIONALITY USING A PASSIVE SHADOW ENV
FOR UP-FRONT OR RETROFITTED PASSIVE OPERATION
References and Further Reading

• “Advanced UVM Tutorial”
  Verilab, DVCon Europe 2014, www.verilab.com/resources

• “Advanced UVM Register Modeling – There’s More Than One Way To Skin A Reg”

• “Pragmatic Verification Reuse in a Vertical World”
  M.Litterick, DVCon 2013, www.verilab.com/resources
Questions
Self-Tuning Coverage

Jonathan Bromley
Overview

• Coverage reuse needs flexibility, configurability

SELF TUNING in response to configuration, parameters etc

• Coverage can mislead!

See Verilab papers DVCon-US 2015 and suggestions in this tutorial

• SV covergroups are not easy to configure

More detail and recommendations later in this tutorial

• SV covergroups are not easy to reconfigure

More detail and recommendations later in this tutorial
COVERAGE GUIDELINES
Coverage Encapsulation

• Coverage should be encapsulated for maintenance
  – isolate coverage code
  – separate class for coverage
  – each coverage class should be in its own file
  – minimize SV assertion-based coverage

• **but** architecture of coverage classes needs care
  – more details in later sections
Implementation Options in SV

• Coverage in a subscriber class
• Extend a component to add coverage groups
• Instantiate a coverage class in some component
Implementation Options in SV

- Coverage in a subscriber class
  
  ```
  class ... extends uvm_subscriber#(dvcon_txn);
  
  dvcon_txn cov_txn;  // no need for deep copy
  
  covergroup cg;
  ...
  coverpoint cov_txn.size {...}
  ...
  endgroup
  
  function void write(dvcon_txn t);
  cov_txn = t;
  cg.sample();
  endfunction
  ```

- Extend a component to add coverage
- Instantiate a coverage class in some component

```
Implementation Options in SV

accellera

no need for deep copy

Environment structure

Coverage subscriber
```
Implementation Options in SV

- Coverage in a subscriber class

- Extend a component to add coverage groups

```
class dvcon_cov_monitor extends dvcon_monitor;
`uvm_component_utils(dvcon_cov_monitor)

covergroup cg;
  ...
endgroup
  ...
```

- Can be sampled at any point in the code
- Has access to all class members

```
uvm_factory f = uvm_factory::get();
f.set_type_override_by_type (dvcon_monitor::get_type(),
dvcon_cov_monitor::get_type());
```

no override, no coverage!
Implementation Options in SV

- Coverage in a subscriber class
- Extend a component to add coverage groups
- Instantiate a coverage class in some component

```verilog
class dvcon_cov extends uvm_component;
    `uvm_component_utils(dvcon_cov)
    dvcon_monitor p_monitor;
    covergroup cg;
    coverpoint p_monitor.value {...}

    function new(string name, uvm_component parent);
    super.new(name, parent);
    $cast(p_monitor, parent);
    cg = new();
    endfunction
```

No instance, no coverage!
Implementation Options - review

• Coverage in a subscriber class
  – restricted to contents of a single object (transaction, transaction-set)
  – but otherwise provides good isolation

• extend monitor class to add
  – flexible, but limits TB structure
  – instantiate correct monitor type, else coverage missed!

• instantiate coverage class in another component
  – coverage class has handle to component, sees all contents
  – allows all coverage implementation including control/timing
  – most flexible solution!
Coverage Is Distributed

• Coverage of individual transactions: easy but insufficient
  – cover each txn from monitor – *protocol coverage*

• DUT behaviour coverage is also important, but sampling and data gathering likely to be distributed across...
  – parts of verif env
    • activity on other interfaces
    • register state and changes
    • end-to-end matching
  – time
    • what else happened during the life of this txn/instruction/...?
    • DUT state at start, end, other key points in txn lifetime?
Coverage Is Distributed

- Coverage from *many places* contributes to verif plan

assertion coverage

Coverage of end-to-end behaviours and checks

cross with system-wide concerns (reset, congestion, mode changes...)

configuration coverage

protocol coverage

DUT internals and register coverage

protocol coverage
Coverage Is Distributed

• Coverage from *many places* contributes to verif plan
• Encapsulate!
• Many connections may be required

---

cross with system-wide concerns (reset, congestion, mode changes...)

---
Coverage Tuning

• Modify the coverage model to suit our requirements
  – for example exhaustive block-level coverage not appropriate for top-level verification requirements, so reduce scope

• Reuse the same mechanism for sampling coverage
  – this is built into the component implementation
  – fully identifies what cover points are sampled and when

• Redefine the coverage groups and coverpoints
  – modify the number of coverpoints in existing cover group
  – modify range and conditions of existing coverpoint bins
  – uses AOP to redefine existing coverpoints
  – SystemVerilog uses OOP factory to replace existing class
IMPLEMENTING FLEXIBLE COVERAGE
Making Coverage Flexible

- Traditional approaches to configurable points/bins
  - sample() wrapper – map sampled data to coverage-friendly value
  - sample() – multiple sample() can be useful
  - coverpoint expression is the result of a function
  - covergroup constructor args used to shape bins
  - module parameters used to shape bins

- SV gotchas:
  - "if" doesn’t remove a bin, just suppresses a sample
  - Effects of illegal, ignore, default – often misunderstood

```
enum {normal, fizz, buzz, fizzbuzz} fb;

bit this_bit;
int bit_num;
covergroup cg;
  coverpoint this_bit;
  coverpoint bit_num;
  cross this_bit, bit_num;
endgroup

function sample(int x);
  if (x%15 == 0)
    fb = fizzbuzz;
  else if (x%3==0)
    fb = fizz;
  else if (x%5==0)
    fb = buzz;
  else
    fb = normal;
  cg.sample();
endfunction
```

```
typedef enum {normal, fizz, buzz, fizzbuzz} fb_e;

int x;
covergroup cg;
  coverpoint this_bit;
  coverpoint bit_num;
endgroup

function to_fb(int x);
  if (x%15 == 0)   return fizzbuzz;
  else if (x%3==0) return fizz;
  else if (x%5==0) return buzz;
  else             return normal;
endfunction
```

```
bit this_bit;
int bit_num;
covergroup cg;
  coverpoint this_bit;
  coverpoint bit_num;
cross this_bit, bit_num;
endgroup

function sample (bit [31:0] x);
  for (bit_num = 0; bit_num < 32; bit_num++) begin
    this_bit = x[bit_num];
    cg.sample();
  end
endfunction
```
Recent Language Enhancements

- Available SV-2012 options:
  - bin `with()` function (value filter)
  - bin set expression (value list specification)
  - cross bin `with()`/`matches` (tuple filter, match threshold)
  - cross bin set expression (queue of value tuples)
- Many scope visibility gotchas
  - good examples are hard to find
  - meanwhile, read LRM carefully and try small test examples
- Patchy tool support
  - `with()` functions are widely supported
  - set expressions in some tools
Bin Specification Improvements

- Illegal cross bins example taken from a real-world problem:
  - Illegal if `active==0` or `active==ready`
  - Illegal if `active` has any bits set that are not set in `ready`

```
covergroup cg_active_ready(int Nbits);
  active_X_ready: cross active, ready {
    function CrossQueueType all_illegals(int n);
      for (int R=0; R<(1<<n); R++)
        for (int A=0; A<(1<<n); A++)
          if ( A==0 || A==R || (A & ~R)!=0 )
            all_illegals.push_back( '{A,R} );
      endfunction
      illegal_bins bad_active = all_illegals(Nbits);
  }
endgroup
```

- `binof` expressions: messy, inflexible

```
illegal example
```

```
N bits

active

0 1 0 1

ready

0 1 1 0

Easy in SV-2012!

self-tuning for Nbits

illegal example

Easy in SV-2012!

self-tuning for Nbits
RECONFIGURABLE COVERAGE
Configurable Coverage

• Any good reusable VC or env has a **configuration object**
• Can we use this config object to tune coverage?
• YES, but there are challenges:
  – covergroup must be created in a class's constructor
  – this is too early in a UVM component; config is not yet known
• Following slides offer two solutions
  1. wrap covergroup in uvm_object, get config from UVM config DB
     • straightforward
  2. embedded class constructed much later, after config is
Making It Configurable

- Covergroup in a component can't be configured:

```plaintext
class dvcon_monitor extends uvm_monitor;
  covergroup dvcon_cg(int max);
  ...
endgroup

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

configuration not yet available
```

- Config information is available much later:

```plaintext
function void build_phase(uvm_phase phase);
  super.build_phase(phase);
  dvcon_cg = new(...);
  ...
endfunction
```

- We need a workaround...

```plaintext
get automatic config
illegal! must be in constructor
```
Configurable Coverage Roadblock

- Key problem:
  - CG must be created in enclosing class's `new()` but...
  - UVM classes have fixed constructor arguments

- Solution 1: constructor gets info from `config_db` before CG `new`
  - OK for `uvm_object`, supports factory
  - Parent object must prepare `config_db` entries before creation
  - Unhelpful for `uvm_component`

- Solution 2: Encapsulate CG in a non-UVM class
  - pass config in constructor arguments
  - factory cannot replace a non-UVM class
Workaround 1: Coverage Object

class my_component...
  ...
  dvcon_cov cov_wrap;
  string cov_name = "cov_wrapper";
  function void start_of_simulation_phase(...);
  dvcon_cov_cfg cfg = new("cov_cfg");
  ...
  uvm_config_db#(dvcon_cov_cfg)::set(
    null, cov_name, "cfg", cfg);
  cov_wrap = dvcon_cov::type_id::create(cov_name);
endfunction
...

late config and creation

component needs configurable coverage

coverage object can be factory-overridden

class dvcon_cov extends uvm_object;
  `uvm_object_utils(dvcon_cov)
  covergroup dvcon_cg(int max); ... endgroup
  dvcon_cov_cfg cfg;
  function new(string name = "");
    super.new(name);
    uvm_config_db#(dvcon_cov_cfg)::get(
      null, name, "cfg", cfg);
    dvcon_cg = new(cfg.max);
  endfunction
endclass

coverage wrapper class
Workaround 2: Coverage Wrapper

• For easy factory replacement, component is

```vhdl
class dvcon_txn_cvg extends uvm_subscriber #(dvcon_txn);
`uvm_component_utils(dvcon_txn_cvg)

class cov_wrapper;
  covergroup dvcon_cg(int max); ...; endgroup
  function new(string name, dvcon_config cfg); ...
  ...
endclass
dvcon_config cfg;
cov_wrapper cov;
virtual function void create_coverage();
  cov = new({get_full_name(), ".cov"}, cfg);
endfunction
virtual function void write(dvcon_txn txn);
  cov.sample(txn);
endfunction
...
```

- register with factory
- nested (local) class definition
- `cfg` set from above in `build_phase` or later
- do not call until `cfg` is ready
Coverage Wrapper Details

- Delegate sampling to virtual methods for easy extension.

```vhls
class cov_wrapper;
  dvcon_txn txn;
  covergroup dvcon_cg(int max),
    cp_len: coverpoint txn.length {
    bins tiny[] = {[0 :3 ]};
    bins mid = {[4 :max-4]};
    bins limit[] = {[max-3:max ]};
  }
  function new(...); ...
  virtual function void sample(dvcon_txn t);
  txn = t;
  dvcon_cg.sample();
  endfunction
endclass
```

- Arbitrary covergroup arguments used to configure bin shapes.

- No need for object copy – `txn` is used only during `sample()`.
Configurable Coverage Component

• nested (wrapper) class contains covergroup(s)
• not a uvm_object – arbitrary constructor signature
  OK
• nested-class object can be constructed any time
  – postpone until config is fully known
• component encapsulates responsibility for:
  – understanding and preparing configuration
  – constructing nested-class object
  – data collection and sampling
• prepare for extension, factory applicability
Reference

• Guidance on coverage design:
  – avoid lies
  – maximise effectiveness
Adaptive Protocol Checks
Configuration Aware Assertions

Mark Litterick
Introduction

• Motivation
  – inconsistent SVA setup across clients & projects
  – many protocols where adaptive SVA required

• SVA encapsulation in UVM
  – why, what & where?

• Techniques for making SVA configuration-aware
  – why & how?

• Techniques for making SVA phase-aware
  – automatically self-configuring SVA checks
  – initial build and dynamic run-time configuration
Distributed UVC Checks

- Types of UVC checks:
  - `signal protocol` and timing
  - `transaction content` and functional correctness
  - `transaction comparison` and relationships
- Each is handled by different component

All checks *belong* to UVC

Concurrent assertions are *not allowed* in SystemVerilog classes
Check Configuration & Control

• All checks can be affected by:
  – control knobs (e.g. checks_enable)
  – config fields (e.g. cfg.mode)

• Configuration object fields can be:
  – constant after build-phase
  – dynamic during run-phase

```plaintext
class my_monitor ...
if (condition)
    // update config
    cfg.speed_mode = FAST;

class my_test ...
    uvm_config_db#(my_config)::set
        (this,"*","cfg",cfg);

    uvm_config_db#(bit)::set
        (this,"*","checks_enable",0);

class my_monitor ...
    if (checks_enable)
        trans.check_crc();

class my_monitor ...
    trans.check_parity(
        cfg.odd_even);
```
sequence s_fast_transfer;
  REQ #1 !REQ[*1:4] ##0 ACK;
endsequence

sequence s_slow_transfer;
  REQ #1 !REQ[*3:10] ##0 ACK;
endsequence

property p_transfer;
  @(posedge CLK)
  disable iff (!checks_enable)
  REQ |->
  if (cfg_speed_mode == FAST)
    s_fast_transfer;
  else
    s_slow_transfer;
endproperty

a_transfer:
  assert property (p_transfer)
  else $error("illegal transfer");
SVA Interface

... but what about SVA?

This interface is required for UVC-DUT signals

© Verilab & Accellera
interface my_interface;

// local signal definitions
// modport declarations
// clocking blocks
// bus-functional methods

// support code for SVA
// property definitions
// assertion statements

endinterface

SVA code is verbose, complex & not related to signal communication

=> isolate and encapsulate SVA

SVA checks create clutter in interface construct

Lines of Code

- Signal Connections
- SVA Checks

© Verilab & Accellera
Interface in Interface

interface myInterface;
  // local signals
  logic CLK;
  logic REQ;
  logic ACK;
  logic [7:0] DATA;
  logic OTHER;
  ...
  // modports, etc.
  ...
  // protocol checker
  my_sva_checker sva_checker(.*);
endinterface

interface my_sva_checker(
  // signal ports
  input logic CLK,
  input logic REQ,
  input logic ACK,
  input logic [7:0] DATA
);
  // support code
  // properties
  // assertions
endinterface

optional signal subset well encapsulated

module not allowed inside interface

=> use interface
interface my_sva_checker
  // control knobs
  bit checks_enable = 1;
  // config object
  my_config cfg;
  // local variables for SVA
  my_speed_enum cfg_speed_mode;
  int unsigned cfg_max_value;
  bit cfg_data_en;

  // properties and assertions...
  // update local vars from cfg...
endinterface

SVA Configuration
Method API

push CFG from class environment to SVA interface

manual operation and only partially encapsulated by interfaces
interface my_sva_checker(...);

function void set_config (my_config cfg);
    cfg_speed_mode = cfg.speed_mode;
    cfg_max_value = cfg.max_value;
    cfg_data_en = cfg.data_en;
endfunction

function void set_checks_enable (bit en);
    checks_enable = en;
endfunction

endinterface

interface my_interface;

function void set_config (my_config cfg);
    sva_checker.set_config(cfg);
endfunction

function void set_checks_enable (bit en);
    sva_checker.set_checks_enable(en);
endfunction

endinterface

user API via VIF methods

required fields well encapsulated

internal SVA details well hidden

accellera

© Verilab & Accellera
API – Initial Configuration

could be done in **agent** or **monitor**

call set_* methods via **virtual interface** after build and connect

```verilog
class my_monitor extends uvm_monitor;
    ...
    function void end_of_elaboration_phase(...);
    ...
    // set interface config after build
    vif.set_config(cfg);
    vif.set_checks_enable(checks_enable);
endfunction
    ...
endclass
```

**must be after VIF assigned**
API – Dynamic Configuration

must be **PASSIVE** component

class my_monitor extends uvm_monitor;
  ...
  task run_phase(...);
  forever begin
    ...
    if (cfg updated) **vif.set_config**(cfg);
  end
  endtask
endclass

**call set_config via virtual interface** when update required

additional work for monitor hard to debug if done wrong
Phase-Aware Interface

reference CFG from class environment inside SVA interface

automatic and fully encapsulated in interface
Class Inside SVA Interface

- **class** is UVM phased component
- locally declared **class** can see all local **variables** in **interface**

```verilog
interface my_sva_checker(...);
  // declare local variables (cfg, checks_enabled, etc.)
  ...
  class checker_phaser extends uvm_component;
    // use local variables (cfg, checks_enabled, etc.)
    // use UVM phases (build, connect, run, etc.)
  endclass

// construct unique instance of phaser class
// (at the top-level under uvm_top)
checker_phaser m_phase = new($psprintf("%m.m_phase"));
endinterface
```

- **no component_utils** if multiple instances required (can’t register same type multiple times with factory)
- **unique name required** for multiple instances
- **unique name** provides better debug messages
Parallel Phases

**SVA interface class in parallel with main env under uvm_top**

UVM phases run in parallel

both build phases complete before either connect phase starts

... but order of build phase completion cannot be predicted!

potential race scenario

© Verilab & Accellera
class checker Phaser extends uvm_component;
  ...
  function void end_of_elaboration_phase(...);
  ...
  if (!uvm_config_db#(my_config)::get(this,"","cfg",cfg))
    `uvm_fatal("CFGERR","no my_config cfg in db")
  void'(uvm_config_db#(bit)::get
    (this,"","checks_enable",checks_enable));
  cfg_speed_mode = cfg.speed_mode;
  cfg_max_value = cfg.max_value;
  cfg_data_en = cfg.data_en;
  endfunction
endclass

use end_of_elaboration_phase to avoid race condition

cfg must be provided by db
checks_enable may be overloaded by db

copy required cfg fields to local variables for use in SVA

required cfg fully encapsulated

no demands on monitor class
Class – Dynamic Configuration

interface my_sva_checker(...);

always @(cfg.speed_mode or cfg.data_en) begin
    cfg_speed_mode = cfg.speed_mode;
    cfg_data_en = cfg.data_en;
end
endinterface

class checker_phaser ...;

task run_phase(...);

forever begin
    @(cfg.speed_mode or cfg.data_en) begin
        cfg_speed_mode = cfg.speed_mode;
        cfg_data_en = cfg.data_en;
    end
    `uvm_info("CFG","cfg updated",UVM_LOW)
end
endclass

always@ and always_comb in do not work since cfg is null at start of simulation

detect change in cfg object

use run_phase to check for dynamic cfg changes

only required dynamic cfg fields

forever @* and @(*) do not work (cfg object does not change only fields inside the object)
Conclusion

• Practical suggestions for **SVA encapsulation**
  – basic encapsulation inside interface

• Demonstrated **configuration-aware SVA**
  – method API with demands on UVC class
  – **phase-aware SVA** checker with embedded class

• Successfully used in many **UVM projects**

• Validated with **multiple tools** in different clients
References and further reading

• “SVA Encapsulation in UVM – Enabling Phase and Configuration Aware Assertions”
  M.Litterick, DVCon 2013, www.verilab.com/resources
Questions
Configuration Object Encapsulation & uvm_config_db Usage

Jason Sprott
Agenda

• Basic syntax refresher
• Encapsulation of configuration data
• Automatic configuration revisited
• Accessing configuration entries (more examples)
• Care with configuration objects
• Conclusion & References
BASIC SYNTAX AND USAGE REFRESHER
Basic Syntax

typedef uvm_config_db#(uvm_bitstream_t) uvm_config_int;
typedef uvm_config_db#(string) uvm_config_string;
typedef uvm_config_db#(uvm_object) uvm_config_object;
typedef uvm_config_db#(uvm_object_wrapper) uvm_config_wrapper;

class uvm_config_db#(type T=int) extends uvm_resource_db#(T)

uvm_config_db#(T)::set(...)
uvm_config_db#(T)::get(...)
uvm_config_db#(T)::exists(...)
uvm_config_db#(T)::wait_modified(...)

set() only modifies database
get() modifies target variables

automatic configuration is done by uvm_component::build_phase() not the database
ENCAPSULATION OF CONFIGURATION DATA
• Focused on UVC functionality not register fields
• Usability: amount of work to configure, synchronize and manage
• Extensibility: make it easy to add new functionality
• Maintainability: modify without breaking user code

• Easy to configure from a test or other part of environment (e.g. callbacks)
• Manage diverse configurations
• Manage functional/structural changes
• Manage cross cutting configurations
class prx_cfg extends ...
  // encapsulate UVC vars
  // partition nicely
  // maybe add some methods
  ...

// maybe add some methods
...

// encapsulate UVC vars

// partition nicely

// maybe add some methods
...

// encapsulate UVC vars

// partition nicely

// maybe add some methods
...

In ENV config we'd probably wrap it
  Hide the gazillions of set() calls
  Limit exposure to implementation vars
  And make a more useful API

Nice if someone had done that here in the first place

uvm_component_utils_begin(top_env)
  'uvm_field_int(m_active_ch,...)
  'uvm_field_int(m_max_ch,...)
  'uvm_field_int(m_twait_min,...)
  'uvm_field_int(m_trespto_max,...)
  // and 50 others like that
  'uvm_component_utils_end

PLEASE NO ... 😞
A Better Config API Example

```plaintext
if (field.write(val))
    rx = extracted_from(val);
    cfg.set_active_rx_chans(rx);
```

Takes care of lower level details and limits exposure to implementation (config variables)

```plaintext
// do some checking
if (rx > this.max_chans)
    uvm_fatal(...)
```

Check using DUT param

```plaintext
// distribute the data
prx_cfg.set_active_chans(rx);
stxrx_cfg.set_active_rx_chans(rx);
```

// Optional: notify change
Configuration objects at the same level often have common configuration data e.g. DUT parameter derived values
Events and Pool Example

Using the event pool on lower levels (e.g. in the agent) may expose context sensitive names

```cpp
m_global_ev_pool.get("env2_hrst")
```

May be appropriate for upper level to extract entries from pool for lower levels

```cpp
uvm_event_pool m_env2_ev_pool
```

Using the event pool on lower levels (e.g. in the agent) may expose context sensitive names

```cpp
cfg.m_hard_reset_ev; // events set by
cfg.m_soft_reset_ev; // Env2 from event
cfg.m_frame_end_ev; // event pool
```

uvm_event and uvm_event_pool can be shared via uvm_config_db
Using Discrete (Non-object) Entries

Rules of Thumb

• When there is an existing standard
  – e.g. UVC is_active (active/passive) flag

• Simple standalone functional mode or feature
  – e.g. UVC's physical interface or protocol type
  – e.g. UVC performance mode (fast with no frills)
  – Affects other dependent configurations

• Quick wildcard setup required
  – Not as easy with objects, as entries part of a collection

• When run-time modification unlikely
  – Avoid need for run-time ::get() uvm_config_db calls
AUTOMATIC CONFIGURATION
OBJECTS AND Enums REVISITED
Some Important Config DB Points

• Look-ups come down to a string match

• Entries distinguished **strictly** by the type used in set()

• For auto config base types must be used in set()
  
  – some "hacked" exceptions, e.g. is_active

• Build phase treated differently
  
  – priority to highest set() in hierarchy
### Setting and Getting Objects

<table>
<thead>
<tr>
<th></th>
<th>::set() type</th>
<th>::get() type</th>
<th>auto config</th>
<th>explicit ::get()</th>
</tr>
</thead>
<tbody>
<tr>
<td>uvm_object</td>
<td>uvm_object</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>uvm_object</td>
<td>CONCRETE</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CONCRETE</td>
<td>CONCRETE</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>CONCRETE</td>
<td>uvm_object</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **set() using base type required for auto-config**
- **set() using base type both auto and get() work**
- $\text{cast to concrete type required for explicit get()}$
- **set() using derived type get() must use derived**
- Only auto-config works
- Only get() works

**Recommend using uvm_config_object typedef**
Use Objects Like This

```c
my_config m_cfg;
...
```

```c
uvm_config_object::set(..., "m_config", m_cfg);
```

*set() and get() use uvm_object type]*

In the target component

```c
'uvm_component_utils_begin(top_env)
'   uvm_field_object(m_config, UVM_DEFAULT)
'   uvm_component_utils_end
```

```c
uvm_object tmp;
uvm_config_object::get(..., "m_config", tmp);
$cast(m_config, tmp); // back to original type
```

Explicit *get()* needs a cast to concrete type

© Verilab & Accellera
Parameterized Classes Work Too

```
my_config#(XYZ) m_cfg;
...
uvm_config_object::set(..., "m_config", m_cfg);
```

*set() and get() use uvm_object type!*

In the target component

```
my_config#(XYZ) m_config;
'uvm_component_utils_begin
  'uvm_field_object(m_config, UVM_DEFAULT)
'uvm_component_utils_end
```

```
uvm_object tmp;
uvm_config_object::get(..., "m_config", tmp);
$cast(m_config, tmp); // back to original type
```

*Explicit get() needs a cast to concrete type!*

© Verilab & Accellera 2015
Wildcard Matching By Name+Type

Config entries with the **same name but different types**

```plaintext
cfg_a m_cfg_a;
cfg_b m_cfg_b;
```

```plaintext
uvm_config_db#(cfg_a)::set(null, "*", "m_cfg", m_cfg_a);
uvm_config_db#(cfg_b)::set(null, "*", "m_cfg", m_cfg_b);
```

**Concrete types**

```plaintext
uvm_config_db#(cfg_a)::get(this, ",", "m_cfg", m_cfg);
```

Possible but **not recommended**: breaks auto-config and a bit risky

```plaintext
uvm_config_db#(cfg_a)::get(this, ",", "m_cfg", m_cfg);
```
### Setting and Getting Enums

**typedef uvm_config_db#(uvm_bitstream_t) uvm_config_int**

<table>
<thead>
<tr>
<th>config_int</th>
<th>USER</th>
</tr>
</thead>
<tbody>
<tr>
<td>config_int</td>
<td>✓</td>
</tr>
<tr>
<td>USER</td>
<td>✓</td>
</tr>
<tr>
<td>USER</td>
<td>✓</td>
</tr>
<tr>
<td>config_int</td>
<td>✓</td>
</tr>
</tbody>
</table>

- **::set()**
  - config_int: ✓ ✓ ✓
  - USER: ✓ x ✓
- **::get()**
  - config_int: ✓ ✓ ✓
  - USER: x ✓ ✓
- **set() using integral type**
  - Both auto and get() work
- **Cast to USER enum type**
  - Required for explicit get()
- **Wrong**

**Recommend using uvm_config_int typedef**
Use Enums Like This

```c
uvm_config_int::set(..., "m_bus_sz", S216);
```

- `set()` and `get()` use `integral` type

In the target component

```c
'uvm_component_utils_begin(top_env)
  'uvm_field_enum(my_enum_t, m_bus_sz, UVM_DEFAULT)
'uvm_component_utils_end
```

```c
uvm_bitstream_t tmp;
 uvm_config_int::get(..., "m_bus_sz", tmp);
 m_bus_sz = my_enum_t'(tmp); // back to original type
```

- Explicit `get()` needs a cast to enum type

- Using convenience types is typically less hassle
ACCESSING CONFIG ENTRIES (MORE EXAMPLES)
Link Visibility to a Component

- We can link visibility to a sequencer (e.g., from a test)

\[
\text{uvm_config_db}(T)::\text{set(this, } \text{"env.agent1.seqr"}, \text{ "err_cfg", m_err_cfg})
\]

\[
\text{uvm_config_db}(T)::\text{get(p_sequencer, "", "err_cfg", m_err_cfg})
\]
Globally Visible Tag

```
ulm_config_db#(T)::set(null,"ERRINJ::","err_cfg",m_err_cfg)
```

Auto config not possible

```
ulm_config_db#(T)::get(null,"ERRINJ::","err_cfg",tmp)
```

Creates a pseudo namespace anyone in hierarchy can use for lookup

```
$cast(m_err_config,tmp);
```

Lookup using this tag

Limited wildcard options due to ulm_config_db issues
Link Visibility to Self

• Dynamically created instance – no parent available

Example: setting the value from the test using the full path

```c
uvm_config_int::set(this,"env.agent1.obj1", "m_val", m_someval)
```

Fragile due to path/instance names if set outside enclosing component.
### Config Changes: Run-time & Reset

<table>
<thead>
<tr>
<th>build_phase</th>
<th>after build_phase</th>
</tr>
</thead>
</table>
| • **Highest in hierarchy wins**  
• Auto configuration  
• **Done once** | • **Last written wins**  
• This applies in run_phase()  
• Run-time phases may repeat  
• Config changes possible  
• User must police changes  
• Using objects is easier |

- e.g. A repeated phase get() could fetch a value out of sync with a register state OR race with a register callback

- Nothing reset specific in uvm_config_db

© Verilab & Accellera
CARE WITH CONFIG OBJECTS
REFERENCES AND CONTENTS
Care with Handles – Local Copies

First set() in `build_phase()`

env_cfg1

Later set() in `run_phase()`

env_cfg2

m_cfg points to `env_cfg1` after first get() in `build_phase()`

```
m_copy_of_var = m_cfg.var1
my_callback = new("...", m_cfg.agent1_cfg)
```

m_cfg members variables used

m_cfg pointing to `env_cfg2` after later get() is **not enough**!

Users of original content may be out of date
Care with Handles – Multiple Users

// originally set up multiple agents
set(this,"agent*","m_cfg", m_agent_cfg)

We can replace m_agent_cfg reference for Agent2 but where?

Multiple people may be using the same reference. It could be important to stay in sync.
Conclusions

• The uvm_config_db is quite simple, but can be used many different ways – consistency is advisable
• Thought required to organize and encapsulate your configuration data
• There are advantages to using objects vs discrete entries
• Automatic configuration can simplify things so try to keep it working
• There's nothing reset specific in uvm_config_db
• Some funky things are possible, but no always advisable
Additional Reading & References

- Accellera
  - [http://www.accellera.org](http://www.accellera.org)
- DVCON2014: Advanced UVM Register Modelling:
- DVCON Europe 2014: Advanced UVM Tutorial
- Hierarchical Testbench Configuration Using uvm_config_db:
- Configuration in UVM: The Missing Manual (DVCON India 2014), Mark Glasser
- Verification Academy:
  - [https://verificationacademy.com/forums/uvm/uvmconfigdb-usage-big-confusion](https://verificationacademy.com/forums/uvm/uvmconfigdb-usage-big-confusion)
  - [https://verificationacademy.com/cookbook/testbench/build#Sub-Component_Configuration_Objects](https://verificationacademy.com/cookbook/testbench/build#Sub-Component_Configuration_Objects)
Parameterized Classes, Interfaces & Registers

Mark Litterick
Introduction to Parameters

- In *SystemVerilog parameters* enable flexibility
  - compile-time specialization of code-base
  - e.g. RTL module with variable FIFO depth, bus width, etc.
  - e.g. verification component with variable channels, etc.

- Parameterization enables horizontal reuse
  - e.g. same DUT RTL code in different project derivative
  - e.g. same verification code adapting to different projects

- Parameterized code trades flexibility with complexity
  - in RTL parameters are usually easy to deal with and simple
  - in verification code the benefits are less clear...
Parameterized Classes

- Parameterized classes are a good fit for many cases:
  - extensive horizontal reuse for a set of derivatives
  - UVM base classes (e.g. TLM, sequencer, driver, monitor,...)
- But parameterization is intrusive...
  - creates verbose code base (harder to read and maintain)
  - parameter proliferation is ubiquitous (all over the code)
  - all files need parameters just to identify specialized types
- When is it worth the effort?
  - parameterization is a lot of effort for the developers
  - ... but can be very good for the user of the code!
Non-Parameterized Setup

- DERIVATIVE-SPECIFIC REGISTER SEQUENCES (CLONE & MODIFY)
- DERIVATIVE-SPECIFIC REGISTER MODEL
- MY_ENV
- MY_REG_MODEL
- seq lib
- VS
- VS
- BASE MODEL
- P
- A
- UVC_BUS
- S
- D
- M
- C
- UVC_STXRX
- TX
- RX
- 1-4
- RX
- TX
- MY_DUT
- UVC_PRX
- TX
- RX
- D
- S
- M
- NOP_ENV
- VS
- seq lib
- UVC_PTX
- TX
- RX
- D
- S
- M
- UVC_STRXRX
- SERIAL
- PARALLEL
- CONFIG-AWARE AGENTS & SEQUENCES
- MAX-FOOTPRINT INTERFACES
- SHARED ENVIRONMENT
- ACTIVE CHANNELS HELD IN CONFIG
- SHARED SEQUENCES HAVE NO ACCESS TO REGISTER FIELDS (SINCE FIELDS ONLY IN DERIVED CLASS)
- DERIVATIVE-SPECIFIC ENV
- Derived-Specific REGISTER SEQUENCES (CLONE & MODIFY)
Parameterized Environment

Shared Sequences have access to register fields (parameterized)

Register model of parameter type

Shared environment

Parameters (register type, num channels)

Parameterized agents & sequences

Parameterized interfaces
Reuse of Parameterized Env

**A FEW DERIVATIVE-SPECIFIC REGISTER SEQUENCES**

**DERIVATIVE-SPECIFIC ENVS (SPECIALIZE PARAMETERS)**

- **e.g. B12_ENV SETS:**
  - REG TYPE = REG_B,
  - NUM SERIAL TX = 1
  - NUM SERIAL RX = 2

**SHARED ENVIRONMENTS** (IDENTICAL CODE BASE)

**MASSIVE REUSE & AUTOMATIC TUNING:** WHAT’S NOT TO LIKE?

**A FEW DERIVATIVE-SPECIFIC REGISTER SEQUENCES**

**DERIVATIVE-SPECIFIC ENVS (SPECIALIZE PARAMETERS)**

- **e.g. B12_ENV SETS:**
  - REG TYPE = REG_B,
  - NUM SERIAL TX = 1
  - NUM SERIAL RX = 2

**SHARED ENVIRONMENTS** (IDENTICAL CODE BASE)

**MASSIVE REUSE & AUTOMATIC TUNING:** WHAT’S NOT TO LIKE?
Generic `p_env` Environment

```vhdl
class p_env #(type REG=uvm_reg_block, int T=1, int R=1)
    extends uvm_env;

    type_id::create
    reg_model;    // register block base
    sequencer;   // virtual sequencer
    stxrx;       // serial UVC

    function void build_phase(...);

    reg_model.build();

    uvm_config_object::set(this, "*", "reg_model", reg_model);
```

**PARAMETER DECLARATION**

**PARAMETER PROPAGATION**

**MUST USE `*_PARAM_UTILS`**

**MUST USE PARAMETERIZED TYPES IN UTILS (OR YOU GET THE WRONG TYPE)**

**MUST USE PARAMETERIZED TYPES IN CREATE (OR YOU GET THE WRONG TYPE)**
class stxrx_env #(int NT=1, int NR=1) extends uvm_env;
  stx_agent #(NT) tx_agent;
  srx_agent #(NR) rx_agent;
  ...

class stx_agent #(int N=1) extends uvm_agent;
  stx_driver #(N) driver;
  s_monitor #(TX,N) monitor;
  virtual s_interface #(N) vif;
  ...

class s_monitor #(s_dir_enum D=RX, int N=1) extends uvm_monitor;
  ...
  s_transaction m_trans[N];
  ...
  if (D==TX)
    `uvm_warning(...,"error injected in Tx traffic to DUT")
  else
    `uvm_error(...,"error observed in Rx traffic from DUT")
  ...

USE PARAMETERS AT LOW LEVELS
(e.g. ARRAY SIZE, MESSAGES, ETC.)

INTRODUCE NEW PARAMETERS
AS REQUIRED (e.g. SHARED MONITOR FOR TX AND RX)

PROPAGATE PARAMETERS THROUGHOUT HIERARCHY
**p_env – Virtual Sequencer**

PARAMETERIZE SEQUENCER TO GIVE SEQUENCES ACCESS TO REG_MODEL OF CORRECT TYPE

```java
class p_env_sequencer #(type REG=uvm_reg_block)
    extends uvm_sequencer;

    REG reg_model; // handle to register model

    `uvm_component_param_utils_begin(p_env_sequencer #(REG))
    `uvm_field_object(reg_model, UVM_ALL_ON | UVM_NOPRINT)
    ...
    function void build_phase(...);
    super.build_phase(...);
    if (reg_model == null)
        `uvm_fatal("NOREG", "null handle for reg_model")
    ...
```

DON’T FORGET 😊
p_env – Sequences

class p_env_base_seq #(type REG=uvm_reg_block) extends uvm_sequence;

\u0027uvm_declare_p_sequencer(p_env_sequencer #(REG))
\u0027uvm_object_param_utils(p_env_base_seq #(REG))
...

class p_env_init_seq #(type REG=uvm_reg_block) extends p_env_base_seq #(REG);

p_env_reset_seq #(REG) reset_seq; // drive reset
p_env_wait_cfg_seq #(REG) cfg_seq; // wait cfg ack
p_env_wait_ready_seq #(REG) ready_seq; // wait ready

\u0027uvm_object_param_utils_begin(p_env_init_seq #(REG))
...

PSEQUENCER MUST BE CORRECT TYPE OR YOU GET RUN-TIME CAST FAIL ERROR “...SEQ CANNOT RUN ON SEQUENCER TYPE…”

SEQUENCES MUST EXTEND CORRECT BASE TYPE => SEQUENCES MUST BE PARAMETERIZED

DON’T EVER FORGET ☺
DUT-Specific Environment

specialize p_env environment by setting actual parameter values

class a21_env extends uvm_env;
p_env#(reg_a,2,1) env; // generic environment

`uvm_component_utils_begin(a21_env)
  ...
function void build_phase(...);
  env = p_env#(reg_a,2,1)::type_id::create("env", this);
  ...
`uvm_component_utils_end

don't forget this either 😊

typedef p_env#(reg_a,2,1) a21_env;

... or replace all of that with convenience type definition
DUT-Specific Sequences

**MUST EXTEND CORRECT TYPE**

```
class a21_example_seq extends p_env_base_seq #(reg_a,2,1);
  `uvm_object_utils(a21_example_seq)
...
```

**DEFINE CONVENIENCE TYPE**

```
typedef p_env_base_seq #(reg_a,2,1) a21_base_seq;
```

**USE CONVENIENCE TYPE**

```
class a21_example_seq extends a21_base_seq;
  `uvm_object_utils(a21_example_seq)
  ...
  p_sequencer.reg_model.TX2_FIELD.write(status, 1'b0);
  ...
```

**SHARED SEQUENCES MUST ALSO BE SPECIALIZED TO RUN ON SPECIALIZED ENVIRONMENT SEQUENCER => DEFINE AND USE CONVENIENCE TYPES**

```
typedef p_env_init_seq #(reg_a,2,1) a21_init_seq;
typedef p_env_config_seq #(reg_a,2,1) a21_config_seq;
...```
DUT-Specific Tests

```plaintext
class test_example_seq extends a21_base_seq;
    a21_init_seq init_seq; // init sequence
    a21_config_seq cfg_seq; // cfg handshake
...
task seq_body();
    `uvm_do(init_seq)
    `uvm_do_with(cfg_seq, {...})
p_sequencer.reg_model.TX2_CFG_STAT.read(status, m_val);
```

**NON-PARAMETERIZED OBJECTS & COMPONENTS AT THE TOP-LEVEL**

```plaintext
class test_example extends a21_base_test;
    test_example_seq example_seq = new();
    virtual task run_phase(...);
    example_seq.start(tb.env.sequencer);
    tb.env.reg_model.RX1_OFFSET_FIELD.write(status, 0);
...
```

**SEQUENCES ARE ALREADY SPECIALIZED BY TYPEDEFS**

**SEQUENCES ALL RUN ON ENVIRONMENT SEQUENCER**
Parameterization Tips

• Don’t do it!
  – avoid parameterization if possible
  – ...but sometime it is an ideal fit for horizontal reuse

• It is **hard to implement** first time round but relatively **easy to retrofit**
  – errors are all related to types specialization
  – bugs are all caused by bad parameter proliferation
  – so get the initial version working, then parameterize

• **Practice** makes **perfect**...
  – clone a working environment right now...
  – ...and retrofit parameterization just for fun!
PARAMETERIZED INTERFACES
Normal Interfaces

class my_comp extends uvm_component;
  virtual my_intf vif;
  function void build_phase(...);
  uvm_config_db#(virtual my_intf)
      ::get(this, "", "cif", vif);

module testbench;
  my_intf mif;
  my_dut dut(.ADDR(mif.addr),...);
  initial begin
    uvm_config_db#(virtual my_intf)
        ::set(null, "*", "cif", mif);
    run_test();
  end

interface my_intf();
  logic [31:0] addr;
  ...
Parameterized Interfaces

- RTL Parameters often affect module ports
  - e.g. signal width (e.g. `input logic [WIDTH-1:0] ADDR`)
  - (but not the presence or absence of signal ports)
- Tempting to parameterize the interface to match RTL ...

```verilog
interface my_intf #(int WIDTH=1)();
  logic [WIDTH-1:0] addr;
  ...
module testbench;
  my_intf#(32) mif;
  my_dut dut(.ADDR(mif.addr),...);
  initial begin
    uvm_config_db#(virtual my_intf#(32)) ::set(null,"*","cif",mif);
    run_test();
  end
```

**SIMPLE CHANGE TO ADD PARAMETER TO INTERFACE**

**SIMPLE CHANGES TO SPECIALIZE INTERFACE AND SEND TO CONFIG_DB**

**NOTE THE INTERFACE TYPE IS SPECIALIZED IT IS NOT JUST my_intf**
Using Parameterized Interfaces

- Parameterized interfaces create a lot of superfluous code
  - in structural components (monitor, driver, agent, env, ...), and in
  - functional objects (transaction, sequences, scoreboard, predictor, ...)

```
class monitor#(int W=1) extends uvm_monitor;
  `uvm_component_param_utils(monitor#(W))
  uvm_analysis_port #(trans#(W)) aport;
  virtual my_intf#(W) vif;

class agent#(int W=1) extends uvm_agent;
  `uvm_component_param_utils(agent#(W)) mon;

class env#(int W=1) extends uvm_env;
  `uvm_component_param_utils(env#(W))
  agent#(W) agt;

class test_env extends uvm_env;
  `uvm_component_utils(test_env)
  env#(32) env;
```

PARAMETERIZED INTERFACE

PARAMETER PROLIFERATION

e.g. ADD ONE PARAMETER TO INTERFACE CHANGED 43 LINES IN 11 FILES

PARAMETER SPECIALIZATION
Accessor Class

- **Alternative** for parameterized interfaces...
- Uses **object handle** instead of **virtual interface**
  - classes support inheritance, interfaces do not!

**BASE CLASS** Defines Accessor API:
- set/get signal values
- get parameter values
- wait for N clocks
- ...

**CONCRETE CLASS** Implements API:
- scoped within the interface instance
- direct access to interface signals

Handles of **base type** can reference object of any derived type regardless of interface parameterization.
```c
#define MAX_WIDTH 256
... 
typedef bit [MAX_WIDTH-1:0] t_uvc_data;

virtual class my_uvc_accessor extends uvm_object{
  pure virtual function int get_data_width();
  pure virtual task wait_cycles(int N=1);
  pure virtual function t_uvc_data get_data();
... 

class monitor extends uvm_monitor#(my_uvc_txn);
  my_uvc_accessor accessor;
  task get_data();
    accessor.wait_cycles();
    if (accessor.get_strobe()) begin
      t_uvc_data data = accessor.get_data();
    ... 
```
Accessor Class Interface

interface my_uvc_intf #(int DATA_WIDTH=32);
  import my_uvc_pkg::*;
  bit [DATA_WIDTH-1:0] data;
  clocking cb @(posedge clock);
    input #1step data;
  endclocking
  class accessor extends my_uvc_accessor;
    function t_uvc_data get_data(); return cb.data; endfunction
    function int get_data_width(); return DATA_WIDTH; endfunction
  endclass
  function accessor get_acc(string name);
    get_acc = new(name);
    uvm_config_db#(uvm_object)::set(null, name, "accessor", get_acc);
  endfunction
  accessor acc = get_acc($sformatf("%m"));
endinterface
Accessor Class Comments

• Advantages:
  – supports **polymorphic extensions** to **signal** interaction
  – highlights the **limitations** of **virtual interfaces**!

• Disadvantages:
  – **restricts** how driver & monitor can access signals (API)
  – **lot of code** setup, duplication & maintenance
  – **not** the **standard UVM** approach

• Suitability?
  – some **protocols** that have to support **multiple diverse** I/Fs
  – wrapper methodology for **legacy HDL** transactors
  – ... but **not appropriate** for **most UVM** environments
Maximum Footprint

- Avoid parameterization using maximum sized interface
  - often referred to as *maximum footprint* interface

```verilog
interface my_intf();
logic [31:0] addr, data;

module testbench;
my_intf mif;
my_dut16 dut(
  .ADDR(mif.addr[15:0]),
  .DATA(mif.data[15:0]),...);
initial begin
  uvm_config_db#(virtual my_intf)
    ::set(null,"","cif",mif);
  run_test();
end

class my_config extends uvm_object;
  rand int width;
  constraint wc {soft width == 32;}

class test_env extends uvm_env;
cfg.randomize() with {width == 16;}
```

**INTERFACE NOT PARAMETERIZED**

**INTERFACE SUPPORTS MAX WIDTH**

**UVC USES CONFIG TO CONSTRaining OUTPUT VALUES & CHECK INPUT VALUES (CAN BE UGLY BIT-SLICE CODE)**

**DUT IGNORES UNUSED INPUT BITS & UNUSED OUTPUT BITS ARE ‘Z**

**DUT CONNECTS TO REQUIRED SLICE**
Interface Wrapper

- Maximum footprint can be enhanced with wrapper
  - parameterized interface wrapper for testbench module
  - contains a maximum footprint interface for class world

```verilog
interface my_intf_wrapper
  #(int WIDTH=8, string INST="*", string FIELD="cif")();
import uvm_pkg::*;
logic [WIDTH-1:0] addr, data;
my_intf mif();
assign mif.data[WIDTH-1:0] = data; // from DUT to UVC
assign addr = mif.addr[WIDTH-1:0]; // from UVC to DUT
initial
  uvm_config_db#(virtual my_intf)::set(null, INST, FIELD, mif);
module testbench;
  my_intf_wrapper#(16) ifw;
  my_dut16 dut(.ADDR(ifw.addr), .DATA(ifw.data), ...);
initial
  run_test();
```
Interface Tips

• Parameterized interfaces look **good in theory**, but **implementation is complex**

• If interface adaption is only reason for parameters...
  – recommend you **do not use parameterized interfaces**
  – **use the maximum footprint** approach with **wrapper**
  – (or in some cases consider the **accessor class** solution)

• If classes are parameterized for other reasons...
  – then **use parameterized interfaces** as well!
PARAMETERIZED REGISTERS
Generic Registers

• Two approaches for register generation:
  
  – generate **DUT-specific register model on-demand**
    (separate model for each derivative)
    • registers are specialized in all representations
      (RTL, documentation, UVM register model)
    • code is simpler, but needs to be generated for each derivative
  
  – generate **generic register model once** for all DUTs
    (one model that adapts to each derivative)
    • registers are generic in all representations
      (RTL, documentation, UVM register model)
    • code is more complex, harder to read, but is only generated once

© Verilab & Accellera
Normal Registers

- Fields, registers & blocks are generated on-demand
  - they are specialized for DUT-specific requirements

```verilog
class my_reg extends uvm_reg;
`uvm_object_utils(my_reg)
virtual function void build();
    my_field = uvm_reg_field::type_id::create("my_field");
    my_field.configure(this, 32, 0, "W1R", 1, 'h0,1,1,1);

class my_reg_block extends uvm_reg_block;
`uvm_object_utils(my_reg_block)
...
rand my_reg my_reg;
...
function void build();
    my_reg = my_reg::type_id::create("my_reg");
    my_reg.configure(this, null, "my_reg");
    my_reg.build();
```

**CREATE & CONFIGURE EACH FIELD**

**FIELD WIDTH USES A GENERATED NUMBER**

**CREATE, CONFIGURE & BUILD EACH REG**

**ALL GENERATED CODE**
Parameterized Registers

- Fields, registers and blocks are classes
  - they can be parameterized (e.g. width, default, etc.):

  ```
  class my_reg #(int WIDTH=1) extends uvm_reg;
  `uvm_object_param_utils(my_reg#(WIDTH))
  virtual function void build();
  my_field = uvm_reg_field::type_id::create("my_field");
  my_field.configure(this, WIDTH, 0, "W1R",1,'h0,1,1,1);
  ```

  ```
  class my_reg_block #(int WIDTH=1) extends uvm_reg_block;
  `uvm_object_param_utils(my_reg_block#(WIDTH))
  ... rand my_reg#(WIDTH) my_reg;
  ... function void build();
  my_reg = my_reg#(WIDTH)::type_id::create("my_reg");
  my_reg.configure(this, null, "my_reg");
  my_reg.build();
  ```
Using Parameterized Registers

- Parameterization is **invasive & ubiquitous**
  - all classes referencing model become parameterized or specialize the required type by fixing the parameter

```
class reg_sequencer#(int W=1) extends uvm_sequence;
  my_reg_block#(W) reg_model;
  `uvm_component_param_utils(reg_sequence);

class reg_base_seq#(int W=1) extends uvm_sequence;
  `uvm_object_param_utils (reg_base_seq#(W));
  `uvmDeclare_p_sequencer(reg_sequence#(W))

class top_vsequencer extends uvm_sequence;
  reg_sequencer#(32) reg_sequencer;
  `uvm_component_utils(top_vsequencer)

class my_reg_seq extends reg_base_seq#(32);
  `uvm_object_utils (my_reg_seq);
```

- **Parameter Proliferation** in middle layers (SEQ, SEQR, AGT, ENV, ...)
- **Parameter Specialization** in upper layer (SEQ, SEQR, ENV, ...)

# Example
- e.g. **ADD ONE FIELD PARAMETER CHANGED**
  - **30 LINES IN 6 FILES**

© Verilab & Accellera
Configuration-Aware Registers

• Alternative generic register model
  – build & configure based on configuration object fields
  – no parameters or compile-time defines

• Problem:
  – `uvm_reg_block` is a `uvm_object` not a `uvm_component`
  – so it does not follow UVM phasing for safe configuration

• Solution:
  – implement `uvm_component wrapper` around model
  – extract configuration during build phase & use in model
Normal Register Build

class my_env extends uvm_env;
    my_reg_block reg_model;
    function void build_phase(...);
        reg_model = my_reg_block::type_id::create("reg_model", this);
        reg_model.configure(null, "reg_model");
        reg_model.build();
        reg_model.lock_model();

class my_reg_block extends uvm_reg_block;
    function new (...);
        super.new(); ...
        virtual function void build();
            reg_a = my_reg_a::type_id::create("reg_a");
            reg_a.configure(this, null, "reg_a");
            reg_a.build();

class my_reg_a extends uvm_reg;
    function new (...);
        super.new(); ...
        virtual function void build();
            field_x = uvm_reg_field::type_id::create("field_x");
            field_x.configure(this, 32, 0, "RW", 1, 'h0, 1, 1, 1);

NOTE: REG_MODEL IS CONSTRUCTED BUT EMPTY AFTER CREATE
CREATE THE REG_BLOCK
BUILD THE REG_BLOCK
CREATE & BUILD EACH REGISTER
CREATE & CONFIGURE EACH FIELD
FIELD WIDTH USES A GENERATED NUMBER
configurable register build

```cpp
class my_reg_a extends uvm_reg;
function new(...);
  field_x = my_reg_field::type_id::create("field_x");
  virtual function void build();
  field_x.configure(this, field_x.width, 0, "RW", 1, 'h0,...);
endfunction

function void build();
  field_x.configure(this, field_x.width, 0, "RW", 1, 'h0,...);
endfunction
```

```cpp
class my_reg_block extends uvm_reg_block;
function new(...);
  reg_a = my_reg_a::type_id::create("reg_a");...
  virtual function void build();
  reg_a.configure(this, null, "reg_a");
  reg_a.build();
endfunction
```

```cpp
class my_reg_wrapper extends uvm_component;
my_reg_block reg_model;
my_config cfg;
function void build_phase(...);
  reg_model = my_reg_block::type_id::create("reg_model");
  reg_model.reg_a.field_x.set_width(cfg.width);
  reg_model.configure(null, "reg_model");
  reg_model.build();
  reg_model.lock_model();
endfunction
```

- **Use config to set knobs in field class.**
- **Create reg_block, registers & fields.**
- **Create moved build to new.**
- **Field width comes from config knob.**
- **Build the reg_block.**
- **Build each register.**
- **Configure each field.**

Note: reg_model is constructed and not empty after create.
Using Register Wrapper

- All functionality is contained inside wrapper class
  - just instantiate and configure like a normal component

```
class my_env extends uvm_env;
  ...
  my_reg_wrapper reg_wrapper;
  my_config   cfg;
  ...
  `uvm_component_utils_begin (my_env)
    `uvm_field_object (cfg, UVM_ALL_ON)
  ...
  function void build_phase(...);
    reg_wrapper = my_reg_wrapper::type_id::create(...);
    uvm_config_object::set(this, "*", "cfg", cfg);
  ...
```
Register Tips

• Generate **derivative-specific registers** on-demand
  – if possible, since it makes life easy for everyone!
  – ...but it is not always appropriate
    (e.g. many DUT parameters in registers, bundled IP&VIP)

• If class environment is already parameterized
  – then generate and **use parameterized registers**

• If class environment is **not** parameterized
  – then **do not use parameterized registers**
  – **use configuration-aware registers** and encapsulate
    in a **wrapper component**
References and further reading

• “Abstract BFMs Outshine Virtual Interfaces for Advanced SystemVerilog Testbenches”

• “Parameterized Interfaces and Reusable VIP”

• “On SystemVerilog Interface Polymorphism and Extendability”
  T.Timi, Blog, [http://blog.verificationgentleman.com](http://blog.verificationgentleman.com)

• “Pragmatic Verification Reuse in a Vertical World”
  M.Litterick, DVCon 2013, [www.verilab.com/resources](http://www.verilab.com/resources)