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*

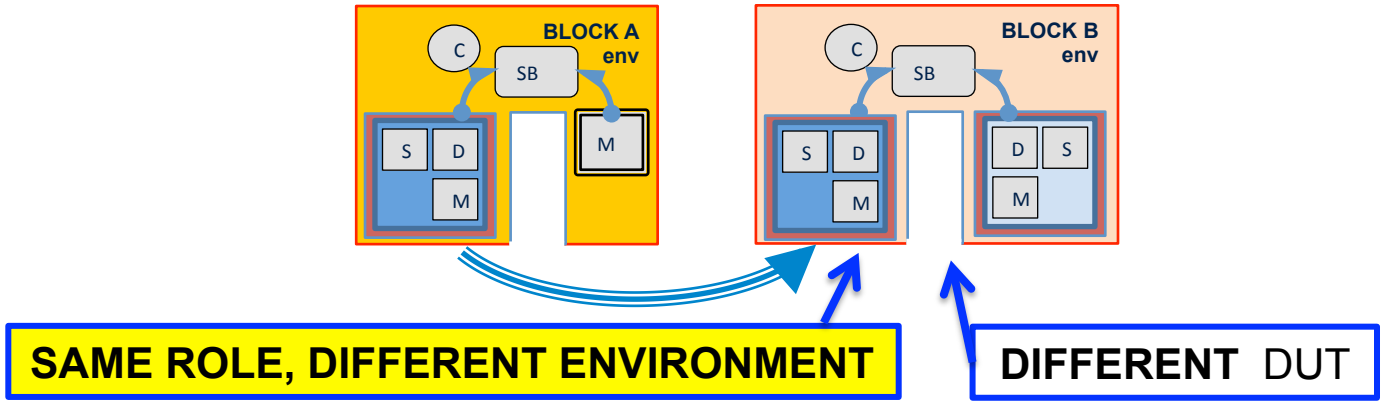
- Verification Reuse:
  - achieving **verification** goals through ***effective*** **code reuse**
- Specifically:
  - reuse verification components between environments
  - reuse verification components between hierarchies
  - reuse verification environments between projects
  - also reuse of base-classes, patterns & methodology
- Several different paradigms:
  - referred to as ***horizontal*** and ***vertical*** reuse

Horizontal Reuse

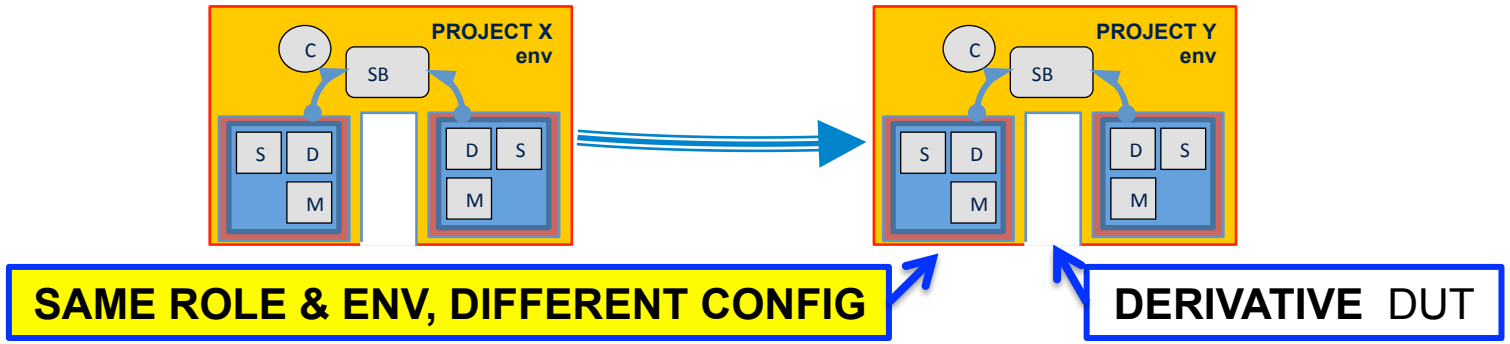
- Reusing verification code ***without changing role***
  - similar responsibilities, structure & use-case
- Reuse **stimulus**, **checks**, **coverage** & **messages**
  - same stimulus API presented to user or test writer
- **Horizontal reuse** usually means **predictable use-case**
  - typically achieved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

Horizontal Reuse examples

- Reuse **verification component** in multiple envs



- Reuse **verification environment** in multiple projects

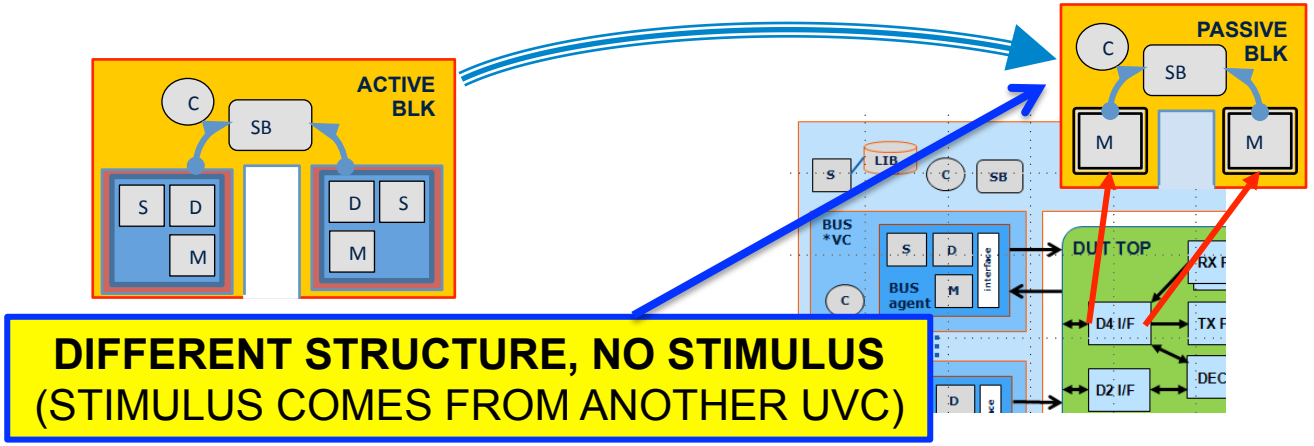


Vertical Reuse

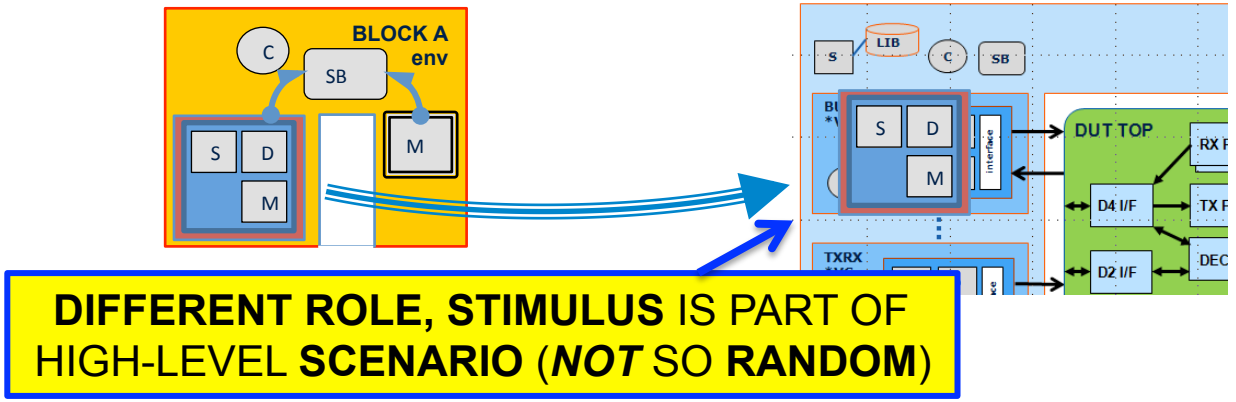
- Reusing verification code **with a *different* role**
  - different responsibilities, structure & use-case
- Reuse **stimulus**, **checks**, **coverage** & **messages**, but…
  - stimulus nested or layered inside **high-level scenario** (with different user API)
  - *or* stimulus **not reused** at 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



- **Reuse active** component in **top-level context**



accellera SYSTEMS INITIATIVE

2015 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

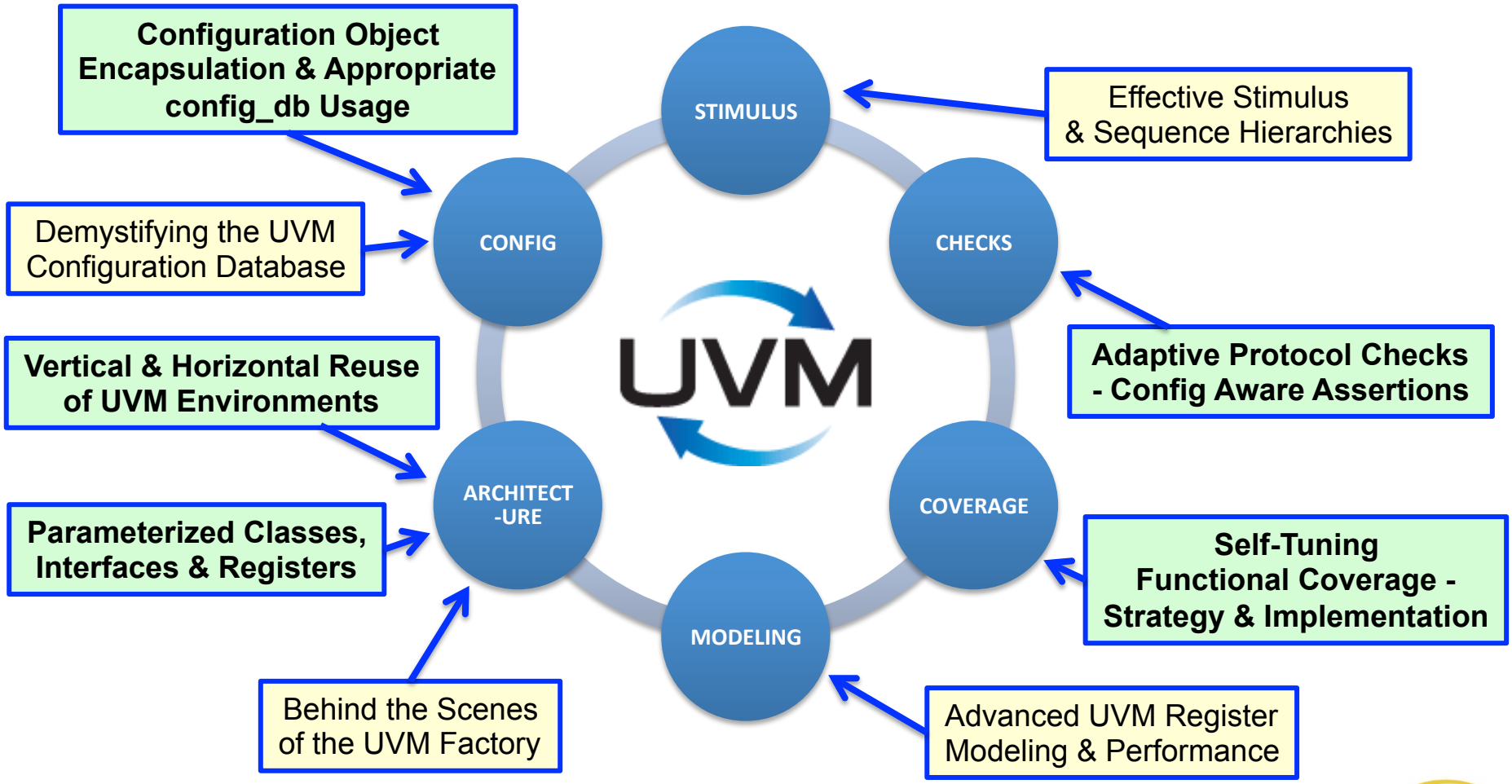
Reuse in UVM

- **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

DVCon EU 2014

DVCon EU 2015



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)

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

EFFORT: HOW MUCH?

TIME: SCHEDULE IMPACT?

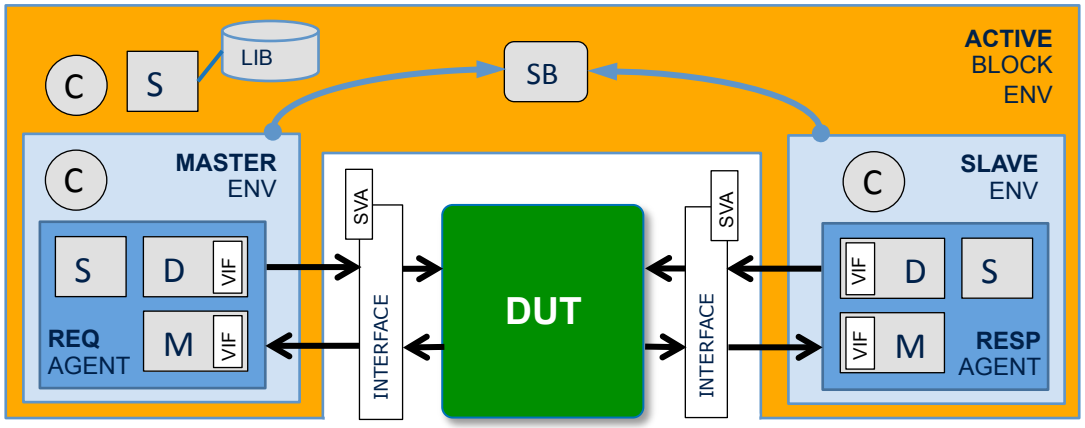
COST: WHO PAYS?

REUSE GUIDELINES **RETROFITTING REUSE**

Active/Passive Operation

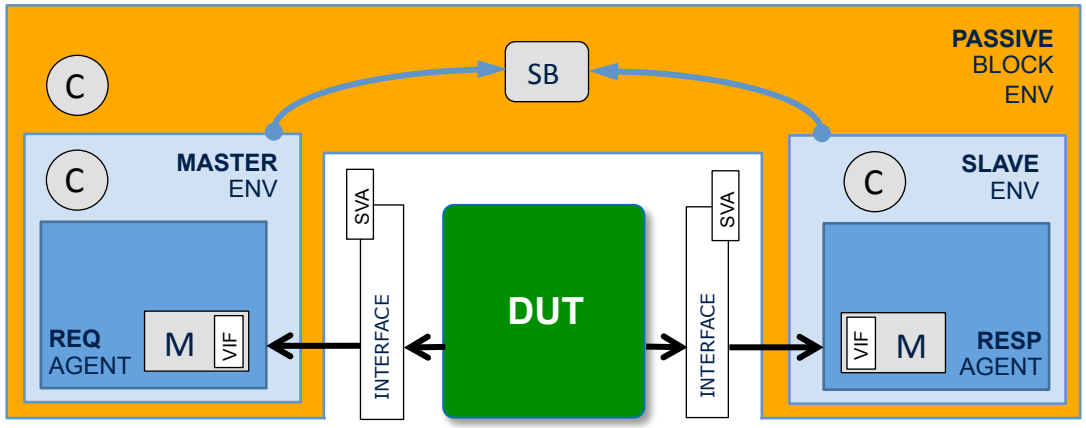
- **Active** components provide or **affect the stimulus** driven to the DUT, influencing the stimulus flow
- **Passive** components **do not** provide or **affect the stimulus** in any way, they just observe
- **Active/Passive mode** affects:
  - run-time architecture
  - functional capability
  - error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

Active Block-Level



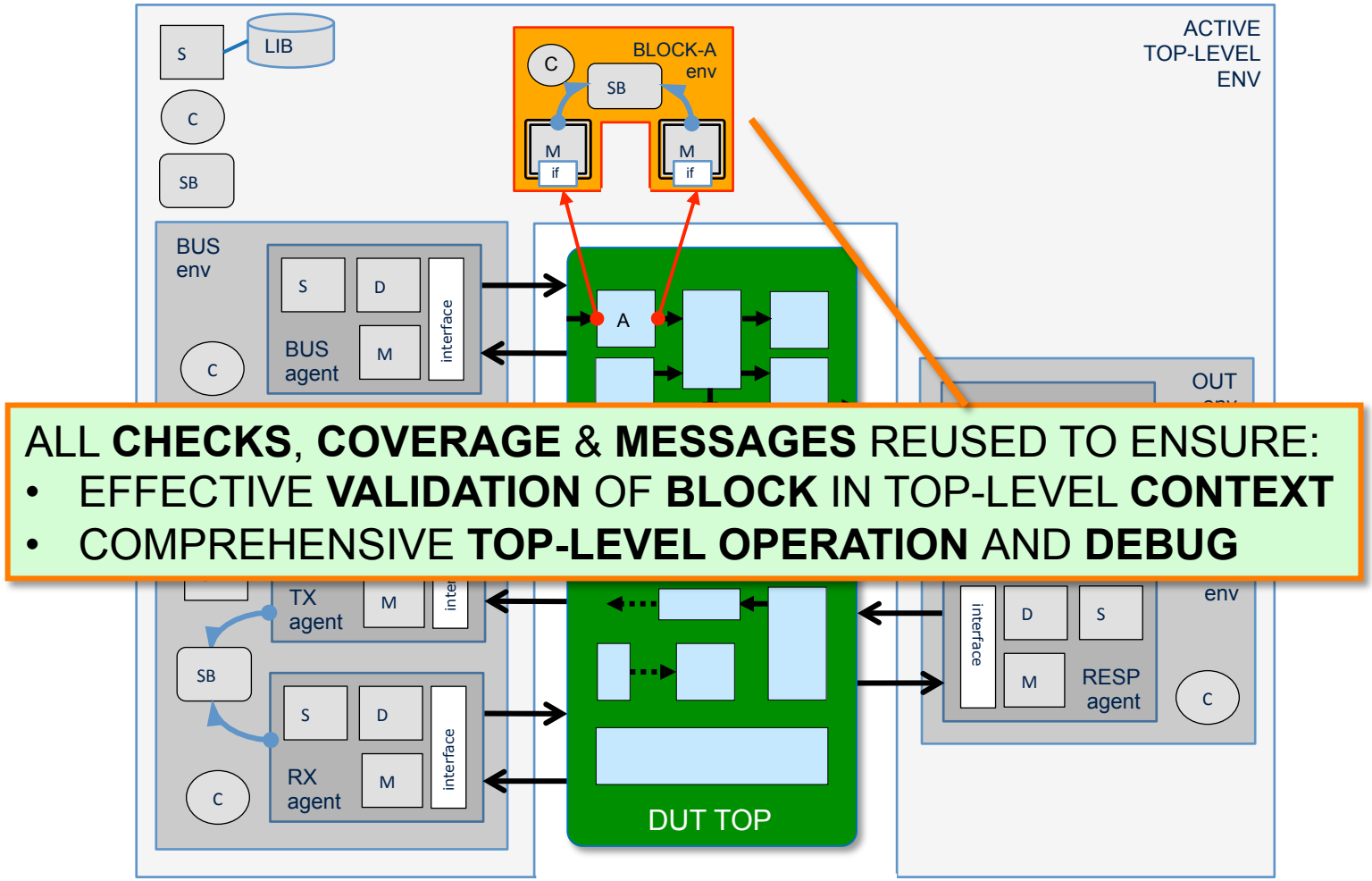
- **Stimulus** is provided by sequencers and drivers
  - **proactive master** generates request stimulus
  - **reactive slave** generates response stimulus
- **Checks** performed by interfaces, monitors & scoreboards
- **Coverage** is collected by monitors & scoreboards
- **Messages** are generated by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



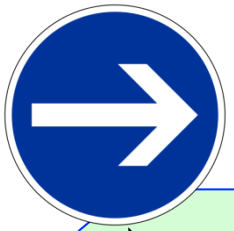
ALL **CHECKS**, **COVERAGE** & **MESSAGES** REUSED TO ENSURE:

- 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



(**OBVIOUSLY**) **ACTIVE** COMPONENT IS **NOT PRESENT** IN **PASSIVE** MODE

(***NOT*** SO **OBVIOUSLY) ACTIVE STIMULUS** IS ALSO ***NOT*** **PRESENT**

- **complete env** must consider **active/passive**
- do ***not*** connect **scoreboards** to **active** components
- perform functional **checks** in **passive** comps
- collect functional **coverage** in **p**
- generate **important messages**
- **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**

UPDATE PSEUDO-STATIC **CONFIG:**
- ***NOT*** **SEQUENCE**, BUT **MONITOR**
- ***NOT*** **REG WRITE**, BUT **PREDICT**

INTERNAL BUS MAY STILL BE **ACTIVE** WHEN **TOP SCENARIO COMPLETES**

TOP-LEVEL **SCENARIO** DECIDES WHEN TO **END THE TEST**

Problems of Scale

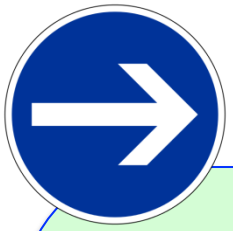
- **Top-level** environment is already **complex**
- Integrating ***many*** **block-level** environments introduces additional requirements:
  - build and integration **encapsulation**
  - **configuration** containment and **consistency**
  - **namespace** isolation and **cohabitation**

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**
- ...

Scale Guidelines



TEST IS **NOT REUSED** BUT ENVIRONMENT IS

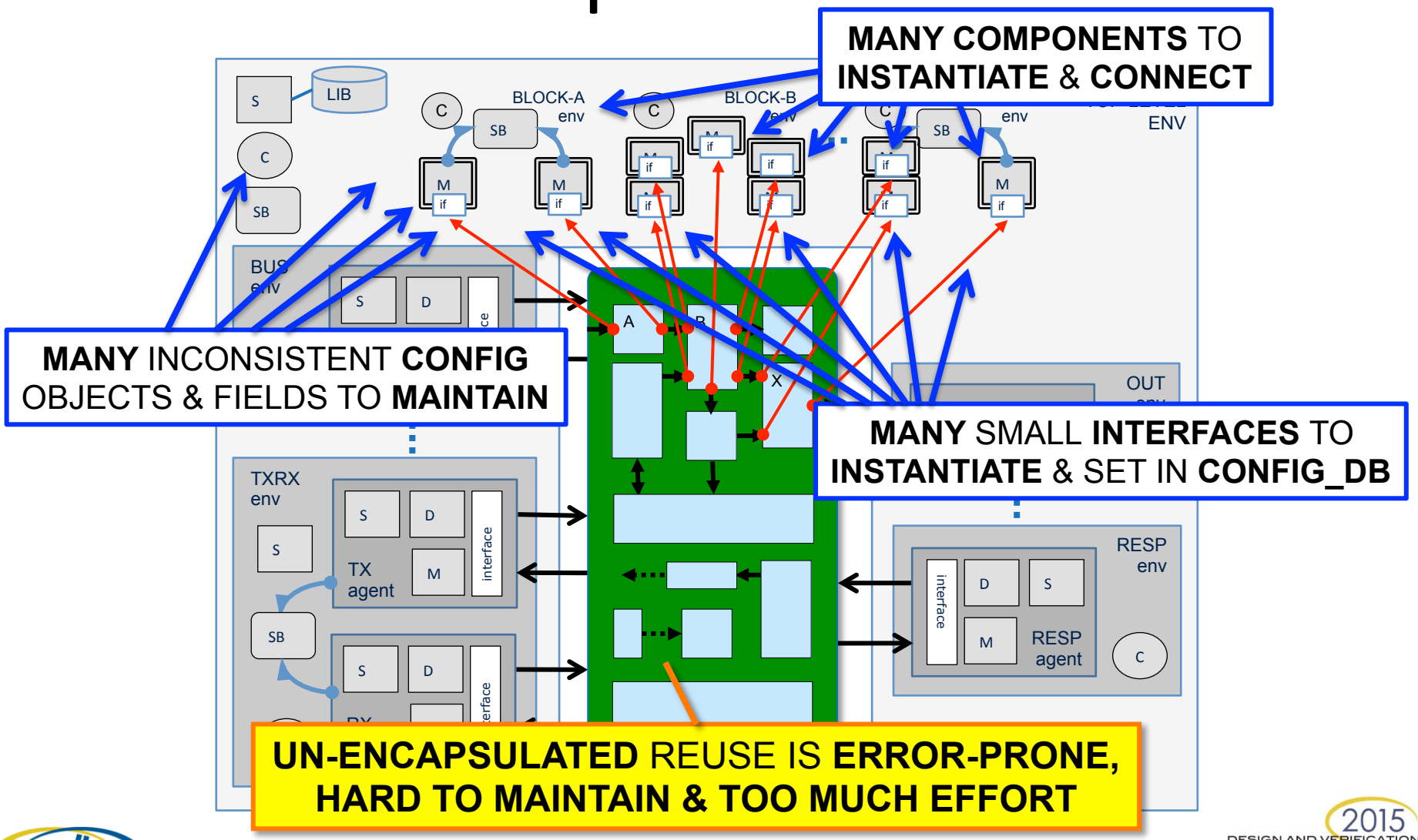
USER ONLY SETS **TOP-CONFIG** WHICH **PROPAGATES** DOWN

- **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
- ...

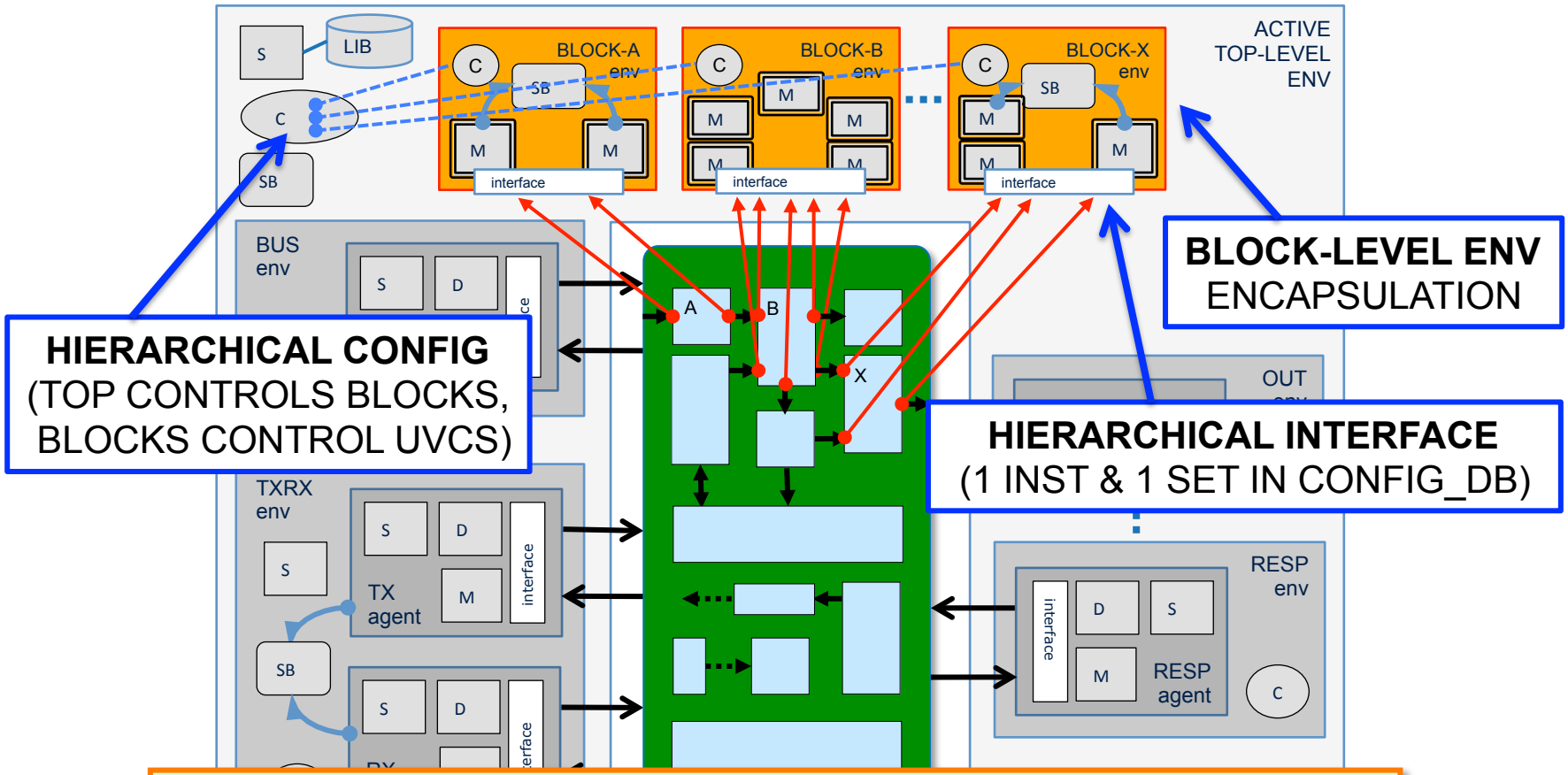
MESSAGE **CLUTTER UNACCEPTABLE** WITH A **LOT** OF **REUSED** BLOCK ENVS

USER ONLY HAS **ONE INTERFACE** TO **INSTANTIATE** & ADD TO **CONFIG_DB**

Un-Encapsulated Reuse



Encapsulated Reuse



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**

REUSE DOES **NOT** COME FOR **FREE – YOU MUST DO SOMETHING !**

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

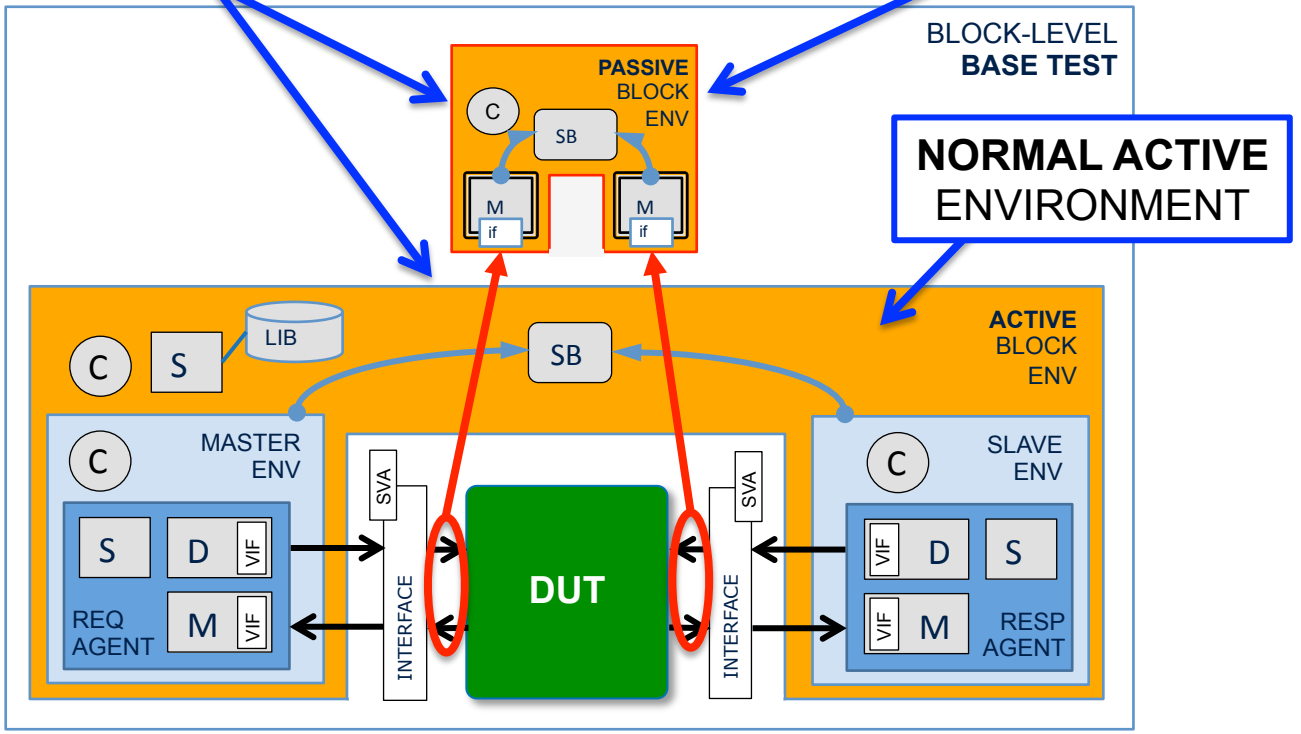
- First, determine **what** you have
  - probably supplied documentation will not be good enough
  - print & review the topology, config and factory settings
- Next, determine **scope** of rework
  - active/passive build control or major scoreboard reconnect?
- Implement **changes** in block-level environment
  - 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



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***"
  M.Litterick & M.Harnisch, DVCon 2014, www.verilab.com/resources
- "***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
- Coverage can mislead!
- SV covergroups are not easy to configure
- SV covergroups are not easy to **re**configure

SELF TUNING in response to configuration, parameters etc

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

More detail and recommendations later in this tutorial

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

specific purpose, often verbose

not extensible, no factory

- **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

environment structure

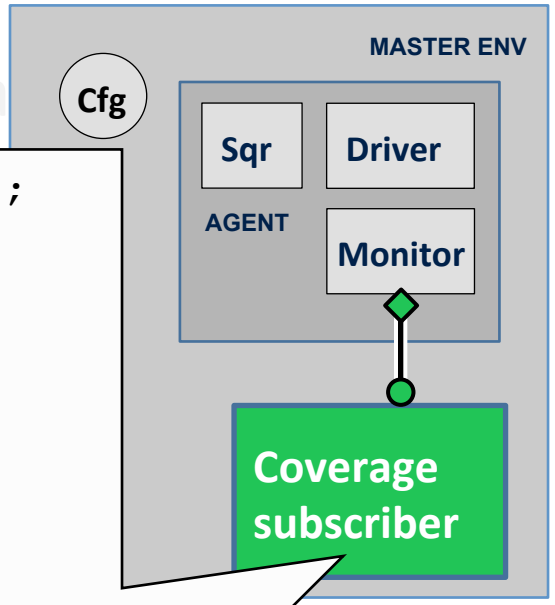
```
class ... extends uvm_subscriber#(dvcon_txn);

  dvcon_txn cov_txn;

  covergroup cg;
    ...
    coverpoint cov_txn.size {...}
    ...
  endgroup

  function void write(dvcon_txn t);
    cov_txn = t;
    cg.sample();
  endfunction
```

no need for deep copy



Implementation Options in SV

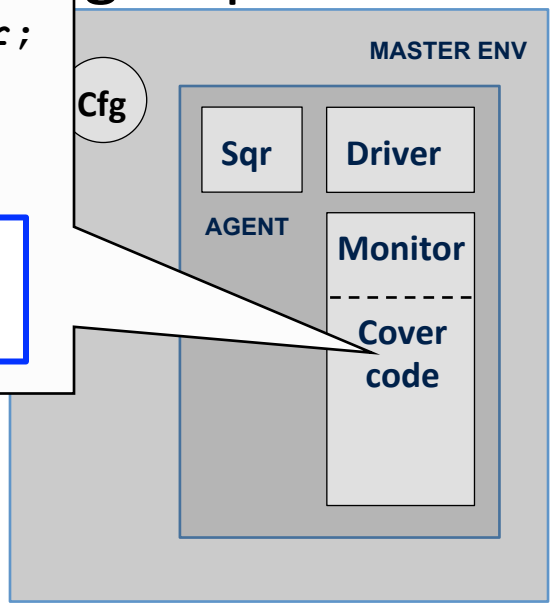
- Coverage in a subscriber class
- Extend a component to add coverage groups

```
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*

inheritance




```
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!

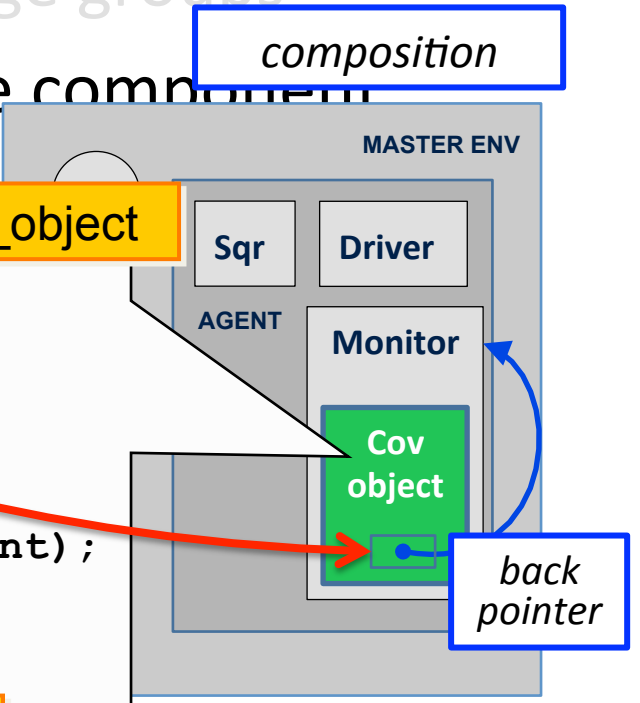
Implementation Options in SV

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

```
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
```

or uvm_object

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 ad
  - flexible, but limits TB struct
  - 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

environment structure

inheritance

cannot factory-replace coverage alone

composition

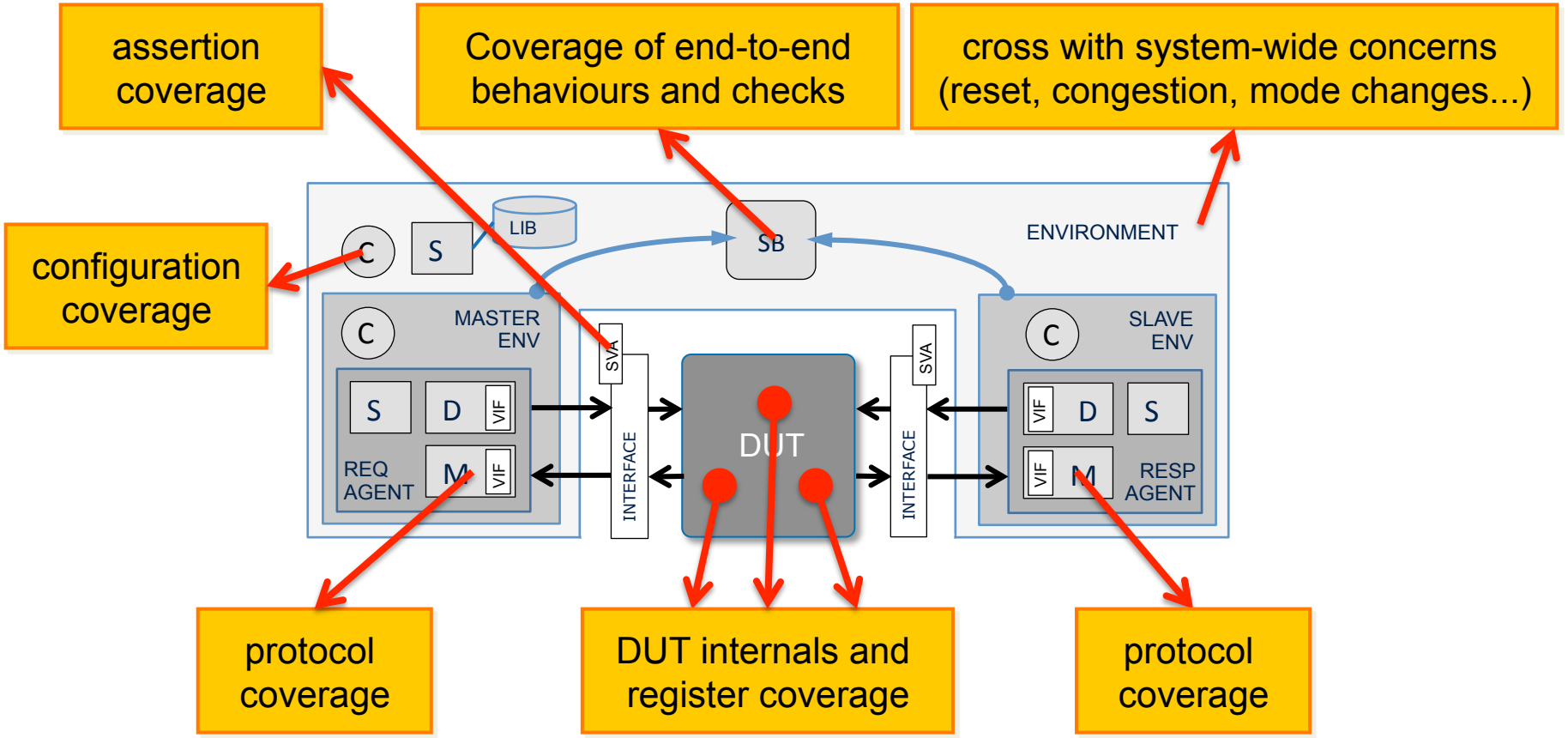
Best choice depends on application!

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?

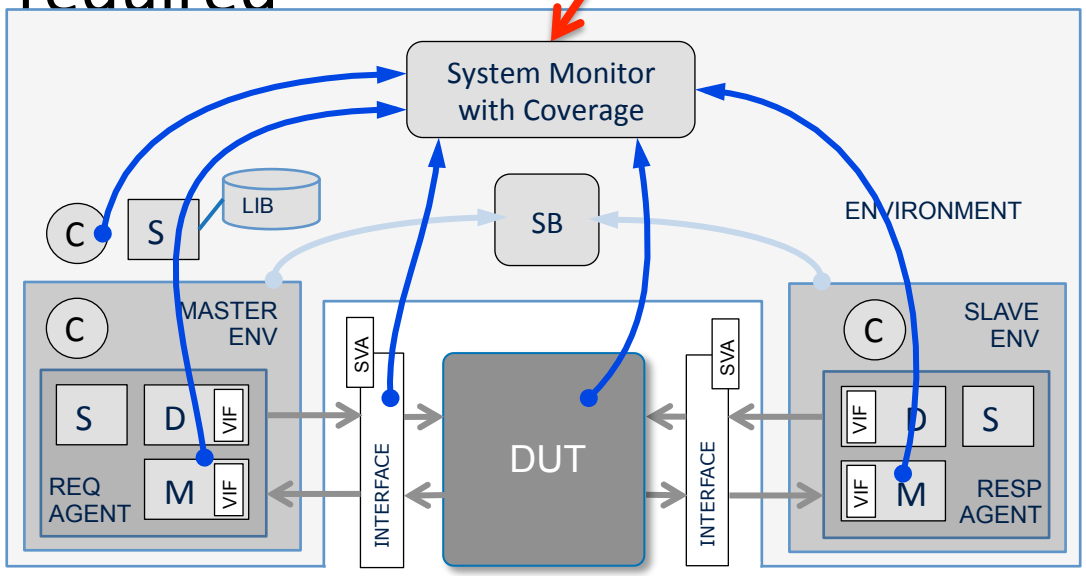
Coverage Is Distributed

- Coverage from *many places* contributes to verif plan



Coverage Is Distributed

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



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
  - ***e*** 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
  - —
  - —
  - —
  - —
  - —
- SV
  - —
  - —

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

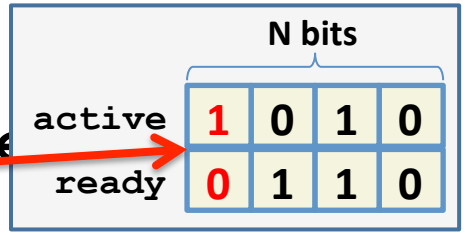
```
bit this_bit;
int bit_num;
covergroup cg;
  coverpoint this_bit;
  coverpoint bit_num;
  cross this_bit, bit_num;
endgroup
function sample (bit [31:0] x);
  for (bit_num = 0; bit_num < 32; bit_num++) begin
    this_bit = x[bit_num];
    cg.sample();
  end
endfunction
```

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 if [illegible]
  - Illegal if **`active`** has any bits set that are not set in **`ready`**



- **`binsof`** expressions: messy, inflexible

```
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
```

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:

```
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(...);
```

configuration not yet available

- Config information is available much later

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

get automatic config

illegal! must be in constructor

- We need a workaround...

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 — *config info may not be available*
- Solution 2: Encapsulate CG in a non-UVM class
  - pass config in constructor arguments
  - factory cannot replace a non-UVM class

good for dedicated coverage components

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");
   ... populate cfg object
   uvm_config_db#(dvcon_cov_cfg)::set(
     null, cov_name, "cfg", cfg);
   cov_wrap = dvcon_cov::type_id::create(cov_name);
  endfunctio
  ...
```

component needs configurable coverage

coverage object can be factory-overridden

late config and creation

```
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

```
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

```
class dvcon_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   ]};
    }
  endgroup
  function new(...); ...; endfunction
  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

SUMMARY

- 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

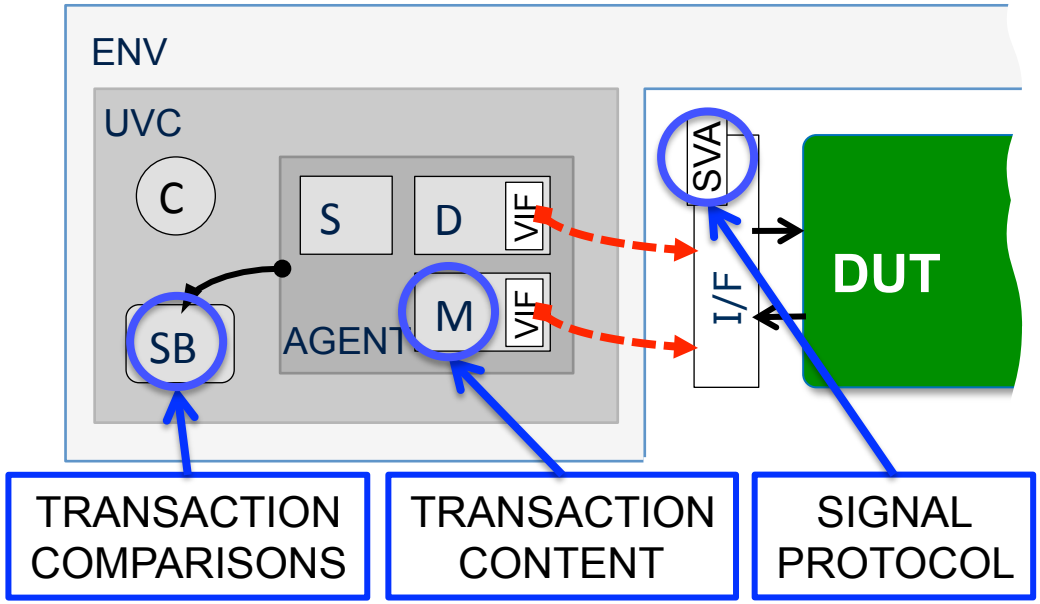
Mark Litterick

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

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

```
class my_monitor ...
 trans.check_parity(
  cfg.odd_even
 );
```

```
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);
```

SVA Configuration & Control

```
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");
```

CONTROL KNOB

CONFIG FIELD

range operators must be **constants**

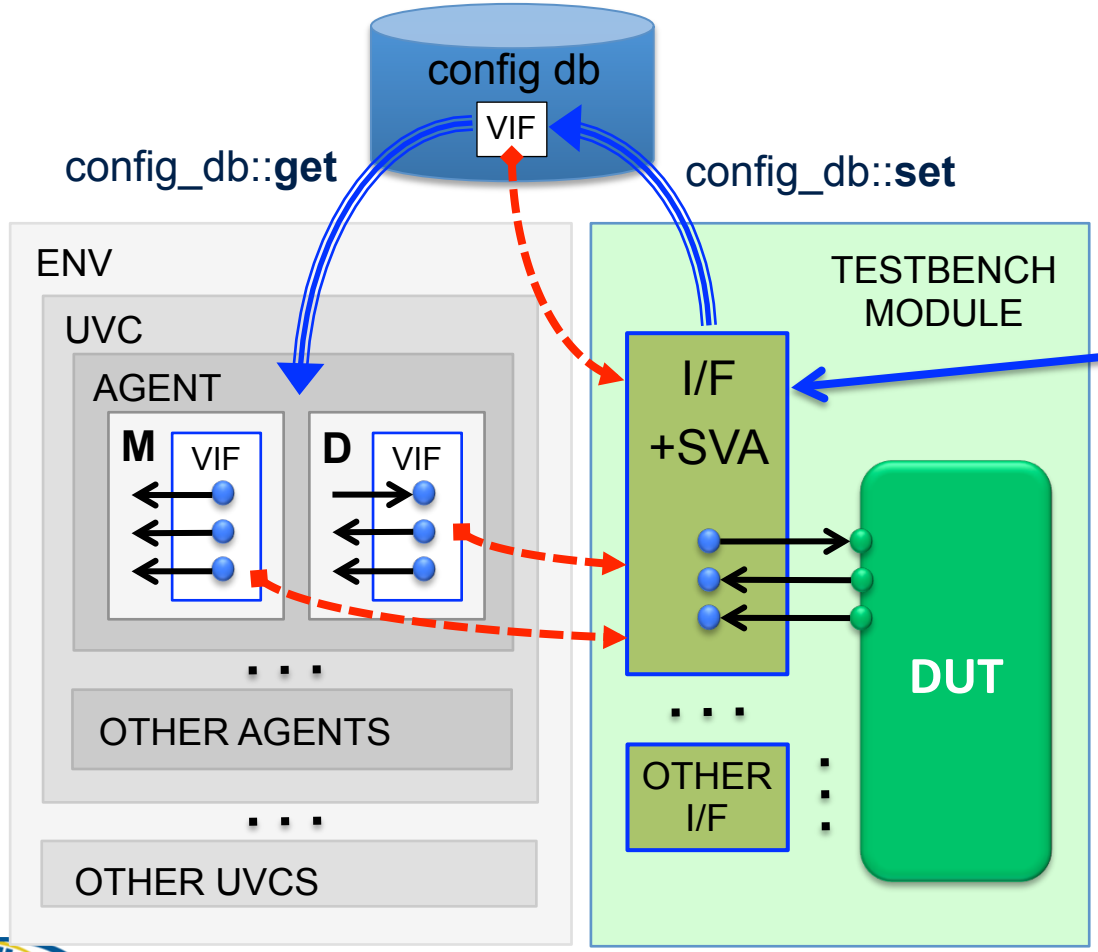
no class variables in concurrent **assertions**

cfg.speed_mode is **not** allowed

local variables used for **SVA**

cfg copied to local variables

SVA Interface



this **interface** is **required** for UVC-DUT signals

... but what about **SVA**?

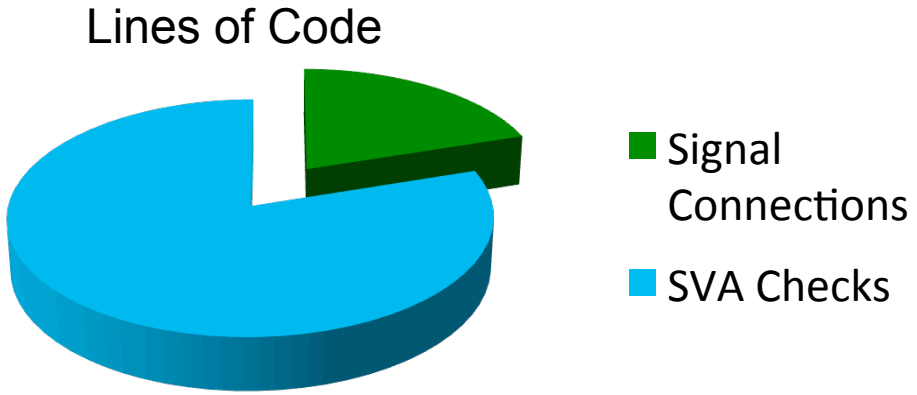
SVA Encapsulation

```
interface my_interface;

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

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

endinterface
```



SVA checks create **clutter** in **interface** construct

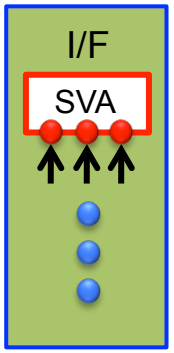
SVA code is verbose, complex & not related to signal communication

=> isolate and **encapsulate SVA**

Interface in Interface

```
interface my_interface;
  // 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
```

implicit instantiation

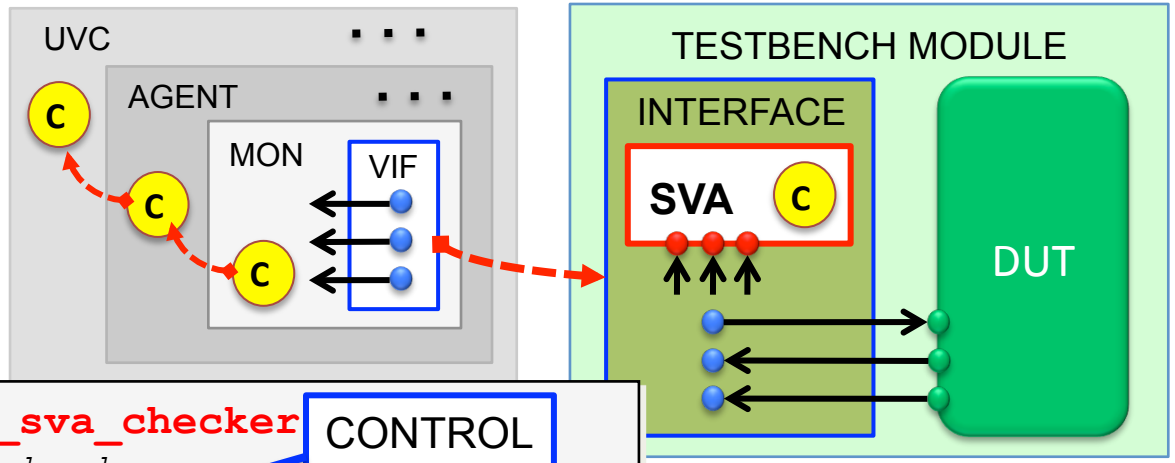
ports have same name as interface signals (but can be a subset)

required signal subset **well encapsulated**

module not allowed inside interface

=> use **interface**

SVA Configuration




```
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
```

CONTROL KNOBS

CONFIG FIELDS

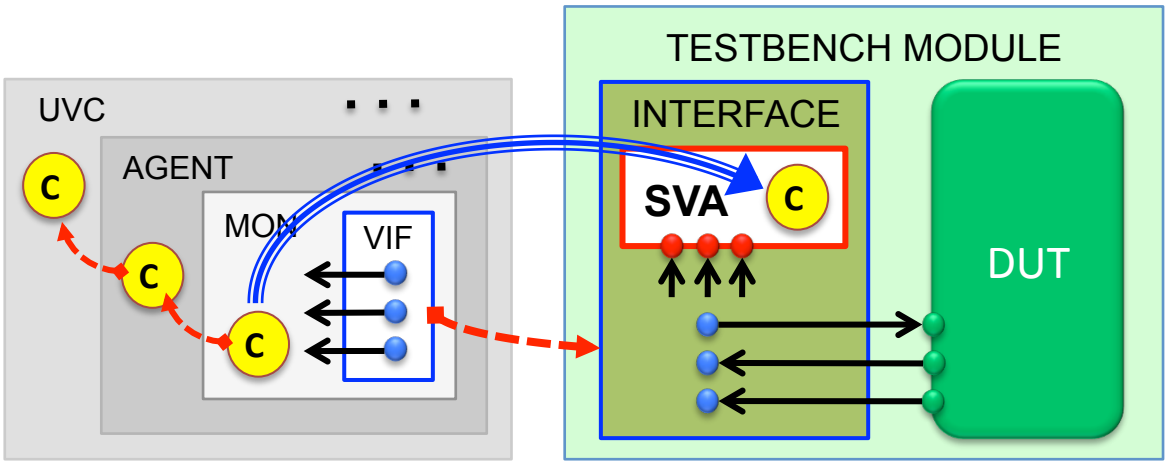
No **class variables** inside concurrent assertions

class variables are allowed in support code

Interface is **not** a **phased** component

when is **config class** built?

Method API



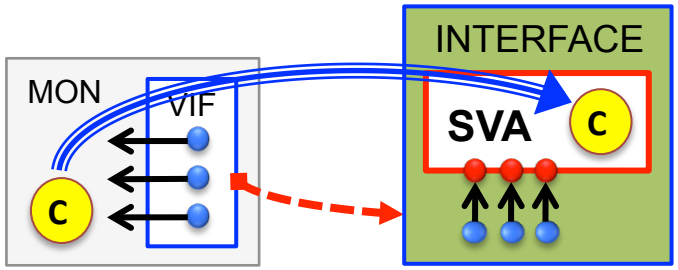
push CFG from class environment **to SVA** interface

manual operation and only partially **encapsulated** by **interfaces**

Interface Methods

```
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
```



required fields well encapsulated

internal SVA details well hidden

```
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

API – Initial Configuration

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

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

```
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

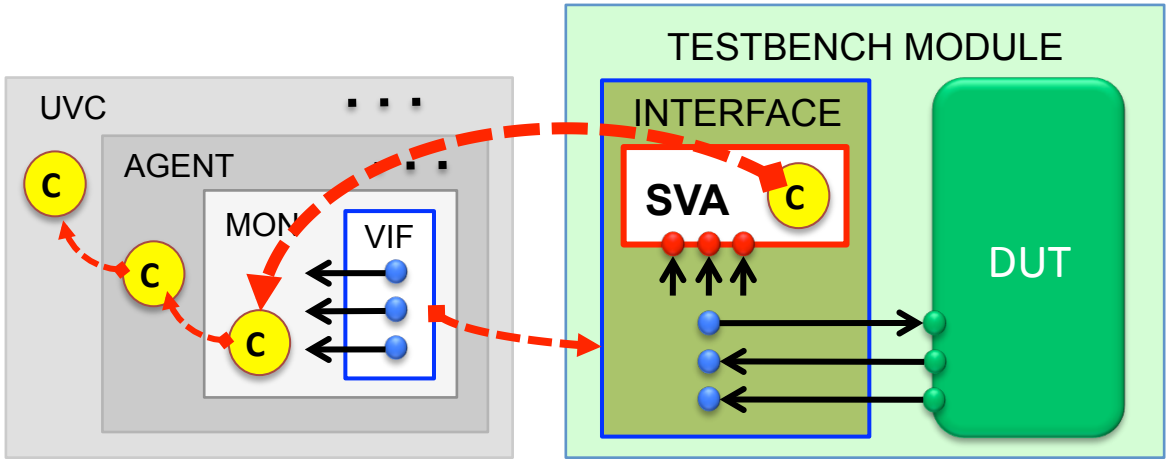
must be **PASSIVE** component

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

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

additional work for monitor
hard to debug if done wrong

Phase-Aware Interface



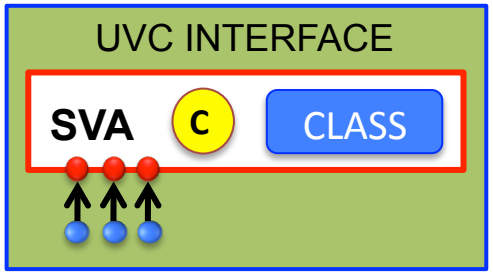
reference CFG from class environment inside **SVA** interface

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**




```
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

  // construc
  // (at the top-level under uvm_top)
  checker_phaser m_phase = new($psprintf("%m.m_phase"));
endinterface
```

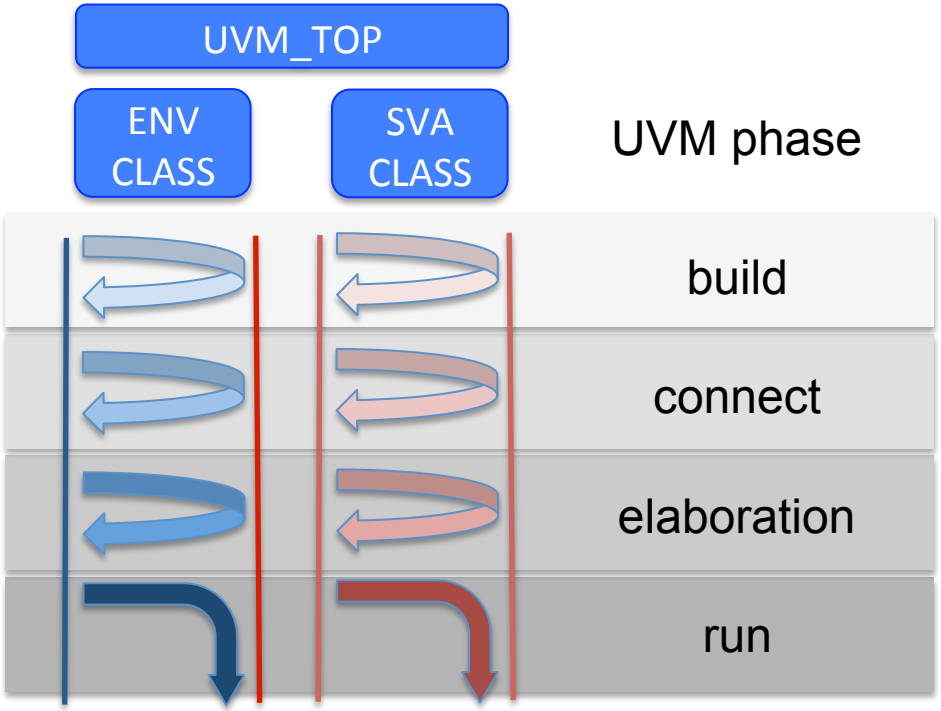
no component_utils if **multiple** instances required (can't register same type multiple times with factory)

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

SVA Class – Initial Configuration

use **end_of_elaboration_phase** to avoid race condition

```
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
```

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.da
  end
endinterface
```

detect change in **cfg** object

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

```
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",
    end
  endtask
endclass
```

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)
```

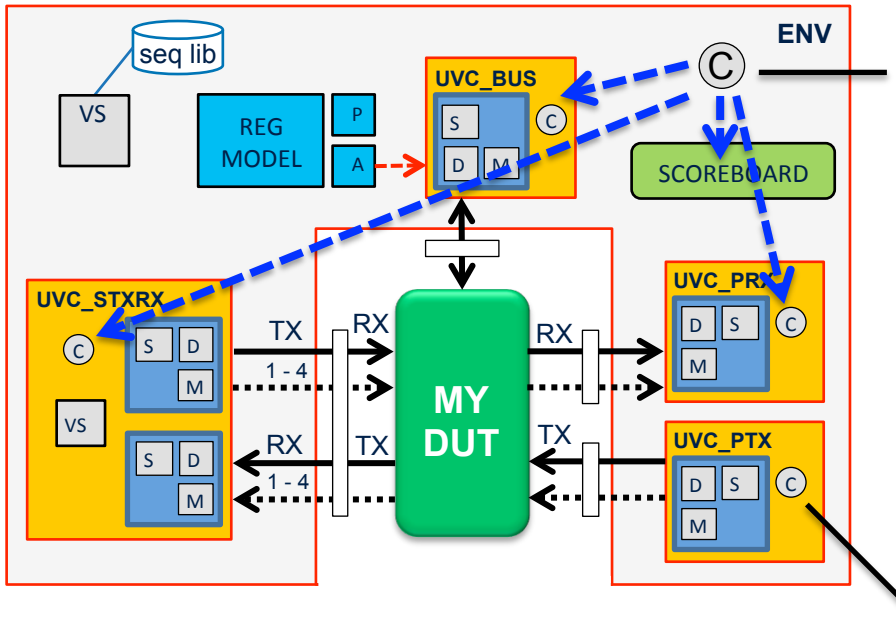
```
uvm_config_db#(T)::set(...)
uvm_config_db#(T)::get(...)
uvm_config_db#(T)::exists(...)
uvm_config_db#(T)::wait_modified(...)
```

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

automatic configuration is done by **uvm_component::build_phase()** not the database

ENCAPSULATION OF CONFIGURATION DATA

UVC/Environment Configuration



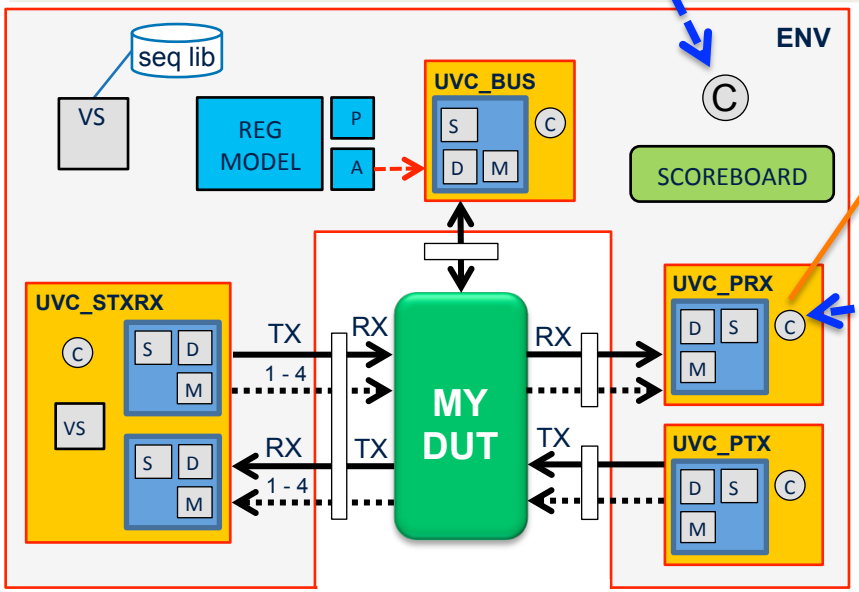
- Easy to configure from a test or other part of environment (e.g. callbacks)
- Manage diverse configurations
- Manage functional/structural changes
- Manage cross cutting configurations

- 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

A Poor Config API Example

```
class prx_cfg extends ...
   // encapsulate UVC vars
   // partition nicely
   // maybe add some methods
   ...
enclass
```

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

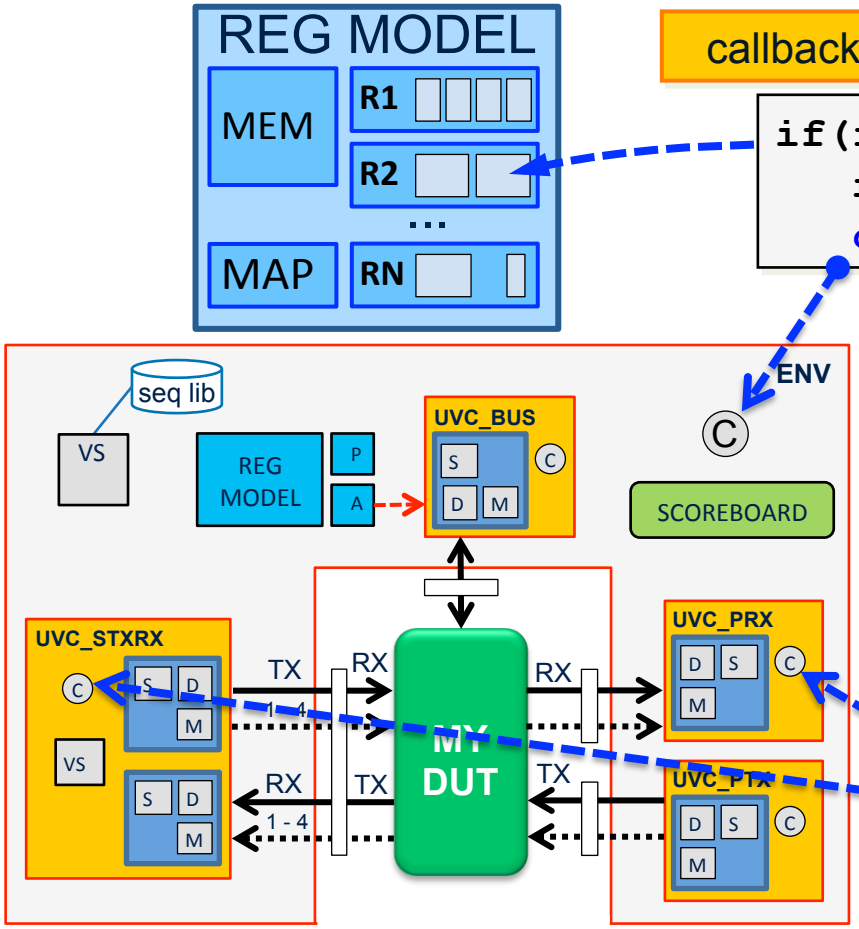


Nice if someone had done
that here in the first place

```
`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



callback uses config API

```
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)

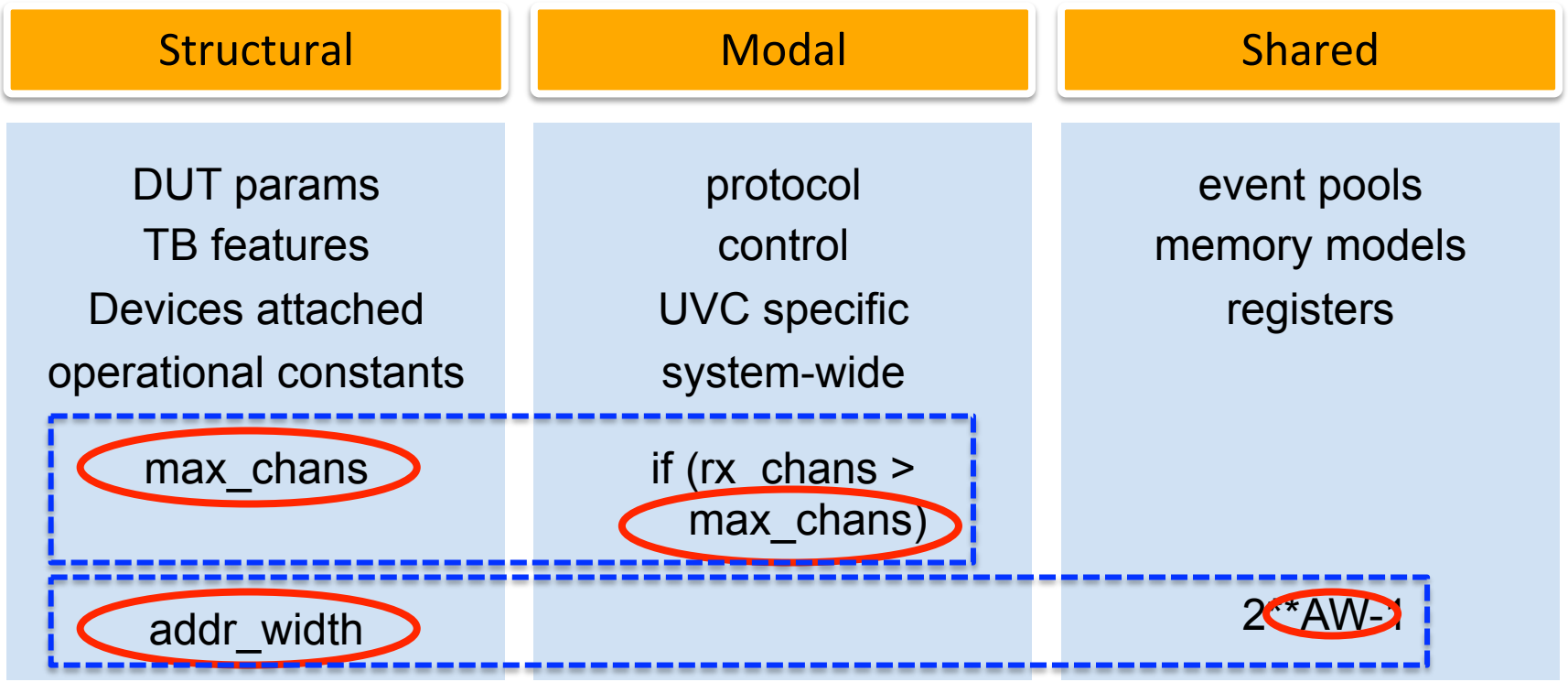
```
// do some checking
if (rx > this.max_chans)
    `uvm_fatal(...)

// distrubite the data
prx_cfg.set_active_chans(rx);
stxrx_cfg.set_active_rx_chans(rx);

// Optional: notify change
```

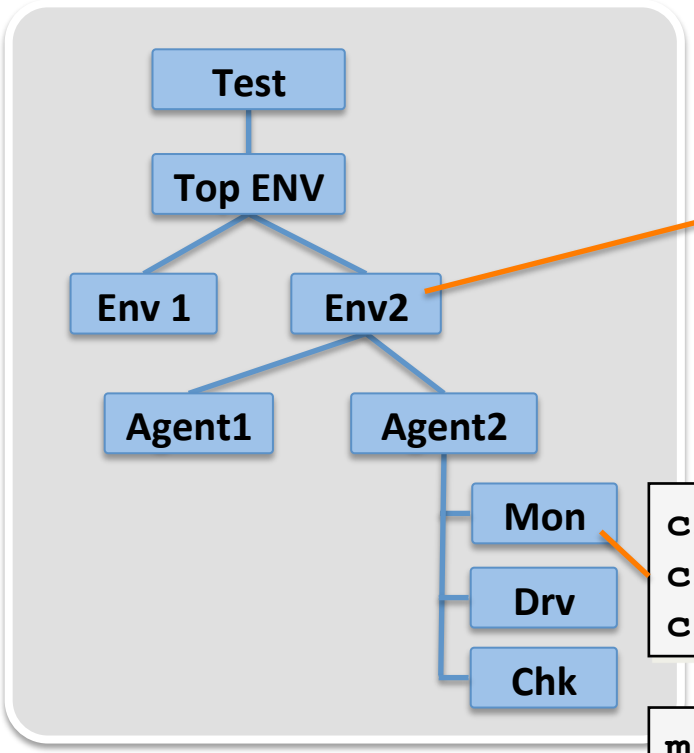
Check using DUT param

Organizing Configuration



Configuration objects at the same level often have common configuration data
e.g. DUT parameter derived values

Events and Pool Example



uvm_event and uvm_event_pool can be shared via uvm_config_db

```
uvm_event_pool m_env2_ev_pool
```

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

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

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

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

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

set() using **base** type required for **auto-config**

set() using **base** type both **auto** and **get()** work

::set() type	::get() type	auto config	explicit ::get()	
uvm_object	uvm_object	✓	✓	$cast to concrete type required for explicit get()
uvm_object	CONCRETE	✓	✗	Only auto-config works
CONCRETE	CONCRETE	✗	✓	Only get() works
CONCRETE	uvm_object	✗	✗	

set() using **derived** type **get()** must use **derived**

Recommend using **uvm_config_object** typedef

Use Objects Like This

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

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

In the target component

```
`uvm_component_utils_begin(top_env)
      `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

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(top_env)
      '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

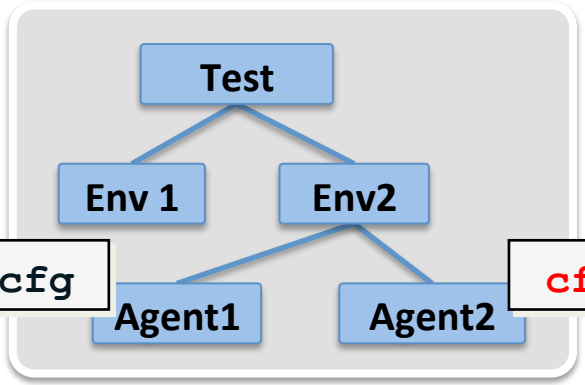
Wildcard Matching By Name+Type

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

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

Wildcard matching name+type

Concrete types



`cfg_a m_cfg`

`cfg_b m_cfg`

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

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

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

Setting and Getting Enums

```
typedef uvm_config_db#(uvm_bitstream_t) uvm_config_int
```

::set() type	::get() type	auto config	explicit ::get()	
config_int	config_int	✓	✓	Cast to USER enum type required for explicit get()
config_int	USER	✓	✗	Only auto-config works
USER	USER	✗	✓	Only get() works
USER	config_int	✗	✗	Wrong

set() using integral type both **auto** and **get()** work

Recommend using **uvm_config_int** typedef

Use Enums Like This

```
uvm_config_int::set(..., "m_bus_sz", SZ16);
```

set() and **get()** use **integral** type

In the target component

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

```
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

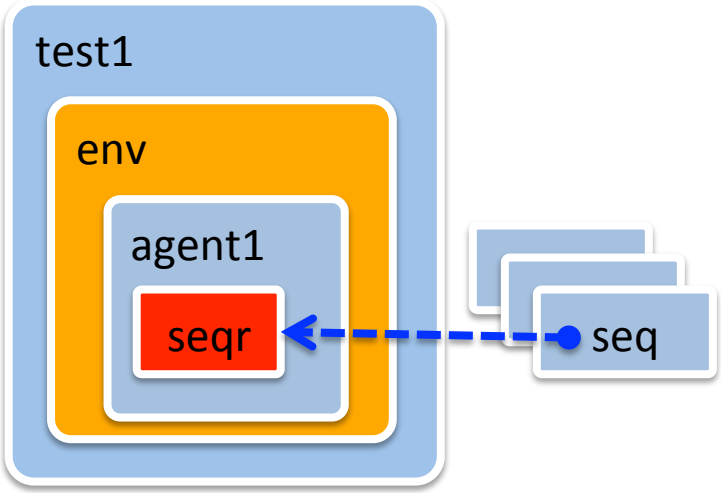
using convenience types is typically less hassle

ACCESSING CONFIG ENTRIES (MORE EXAMPLES)

Link Visibility to a Component

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

```
uvm_config_db#(T)::set(this,"env.agent1.seqr", "err_cfg", m_err_cfg)
```



We can use a sequencer as the context for uvm_config_db lookup

A sequence can lookup entry using handle to the sequencer

Sequencer itself doesn't need to have the target variable

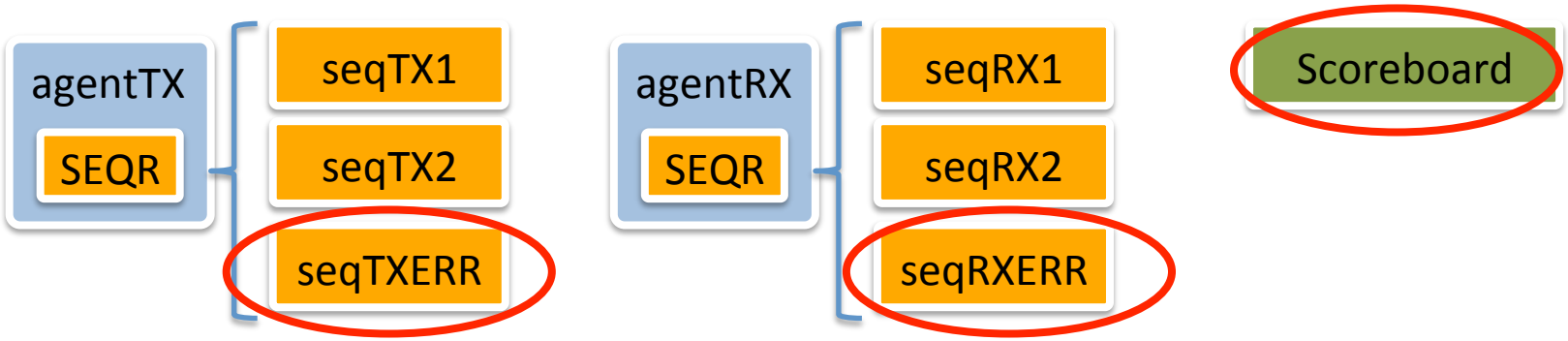
```
uvm_config_db#(T)::get(p_sequencer,"","err_cfg", m_err_cfg)
```

Globally Visible Tag

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

Auto config not possible

Creates a pseudo namespace anyone in hierarchy can use for lookup



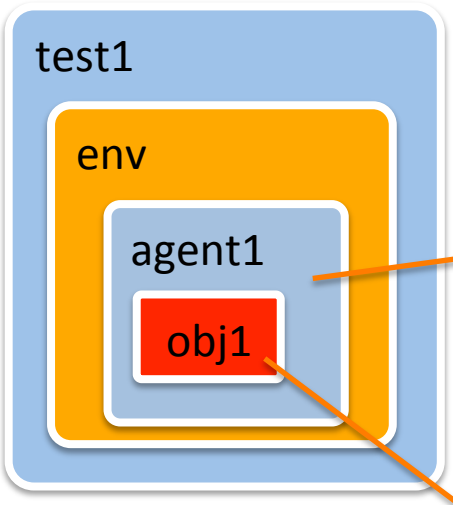
Lookup using this tag

```
uvm_config_db#(T)::get(null,"ERRINJ::","err_cfg", tmp)
$cast(m_err_config, tmp); // back to original type
```

Limited wildcard options due to uvm_config_db issues

Link Visibility to Self

- Dynamically created instance – no parent available



Build full path from enclosing component, plus instance name – **yes the get() will be fragile**

```
obj1 = my_class::type_id::create(
  {this.get_full_name(),".obj1"}, this);
```

Fetch value inside the instance

```
uvm_config_db#(T)::get(null,this.get_name(),"m_val", m_val)
```

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

```
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

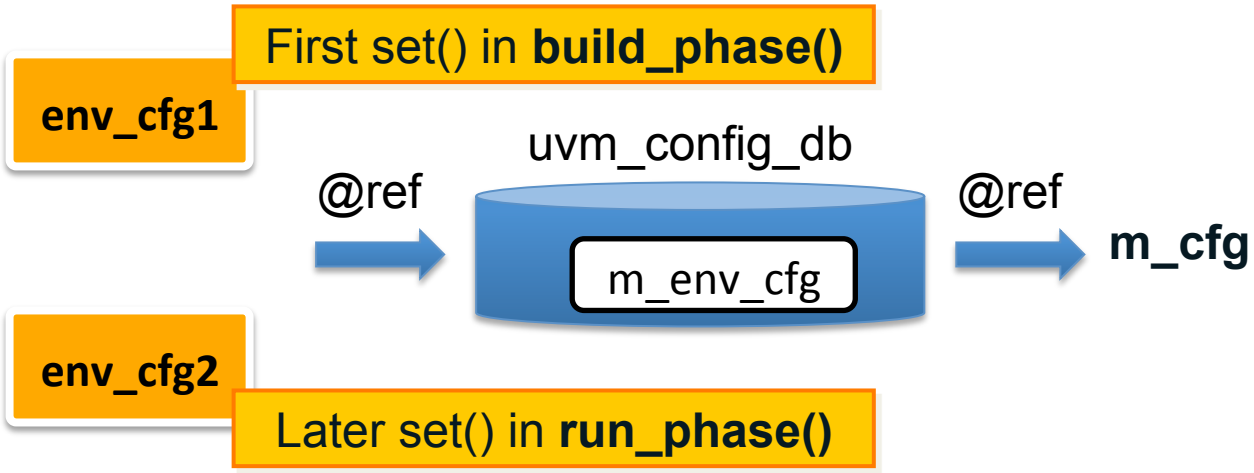
build_phase	after build_phase
• **Highest in hierarchy wins** • Auto configuration • **Done once**	• **Last written wins** • This applies in run_phase() • Run-time phases may repeat • Config changes possible • User must police changes • Using objects is easier

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

CARE WITH CONFIG OBJECTS REFERENCES AND CONTENTS

Care with Handles – Local Copies



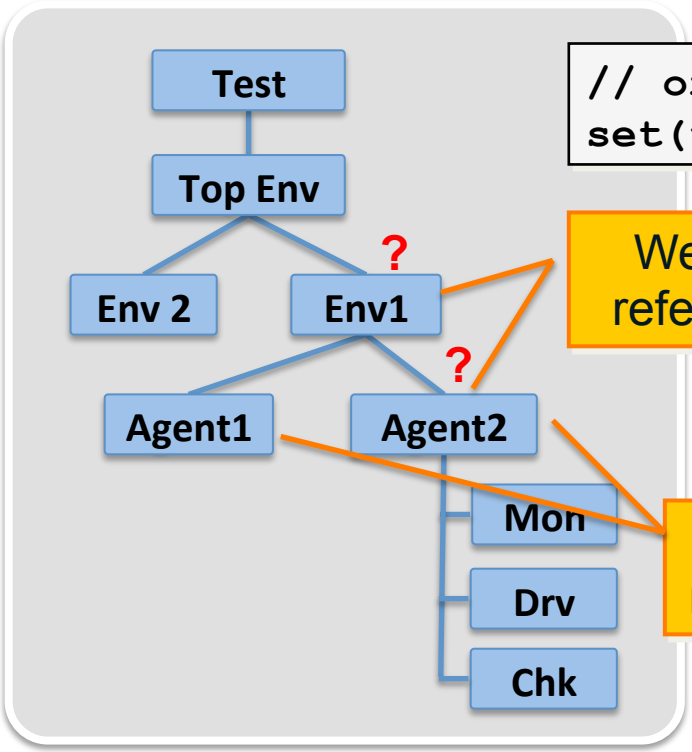
m_cfg points to **env_cfg1** after first get() in build_phase()

```
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.

CONCLUSION AND REFERENCES

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

- Getting Started with UVM: A Beginner's Guide, Vanessa Cooper, Verilab Publishing 2013
- Accellera
  - http://www.accellera.org
- DVCON2014: Advanced UVM Register Modelling:
  - http://www.verilab.com/files/litterick_register_final_1.pdf
- DVCON Europe 2014: Advanced UVM Tutorial
  - http://www.verilab.com/files/verilab_dvcon_tutorial_a.pdf
- Hierarchical Testbench Configuration Using uvm_config_db:
  - http://www.synopsys.com/Services/Documents/hierarchical-testbench-configuration-using-uvm.pdf
- Configuration in UVM: The Missing Manual (DVCON India 2014), Mark Glasser
  - http://dvcon-india.org/wp-content/uploads/2014/proceedings/1130-1300/D2M2-2-3-DV_Configuration_in_UVM_Paper.pdf
- Verification Academy:
  - https://verificationacademy.com/forums/uvm/uvmconfigdb-usage-big-confusion
  - 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...

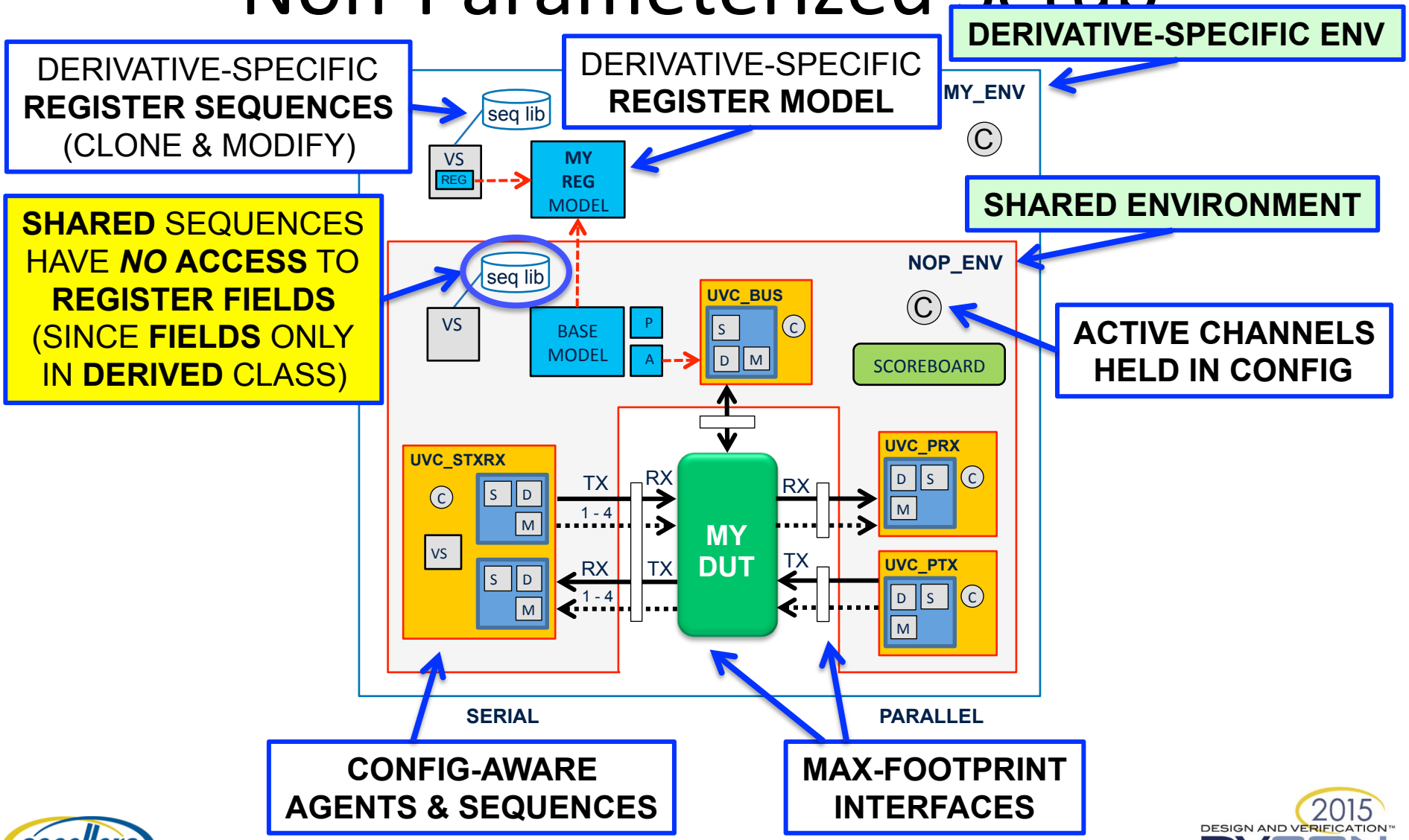
PARAMETERIZED **CLASSES** | PARAMETERIZED **INTERFACES** | PARAMETERIZED **REGISTERS**

PARAMETERIZED CLASSES

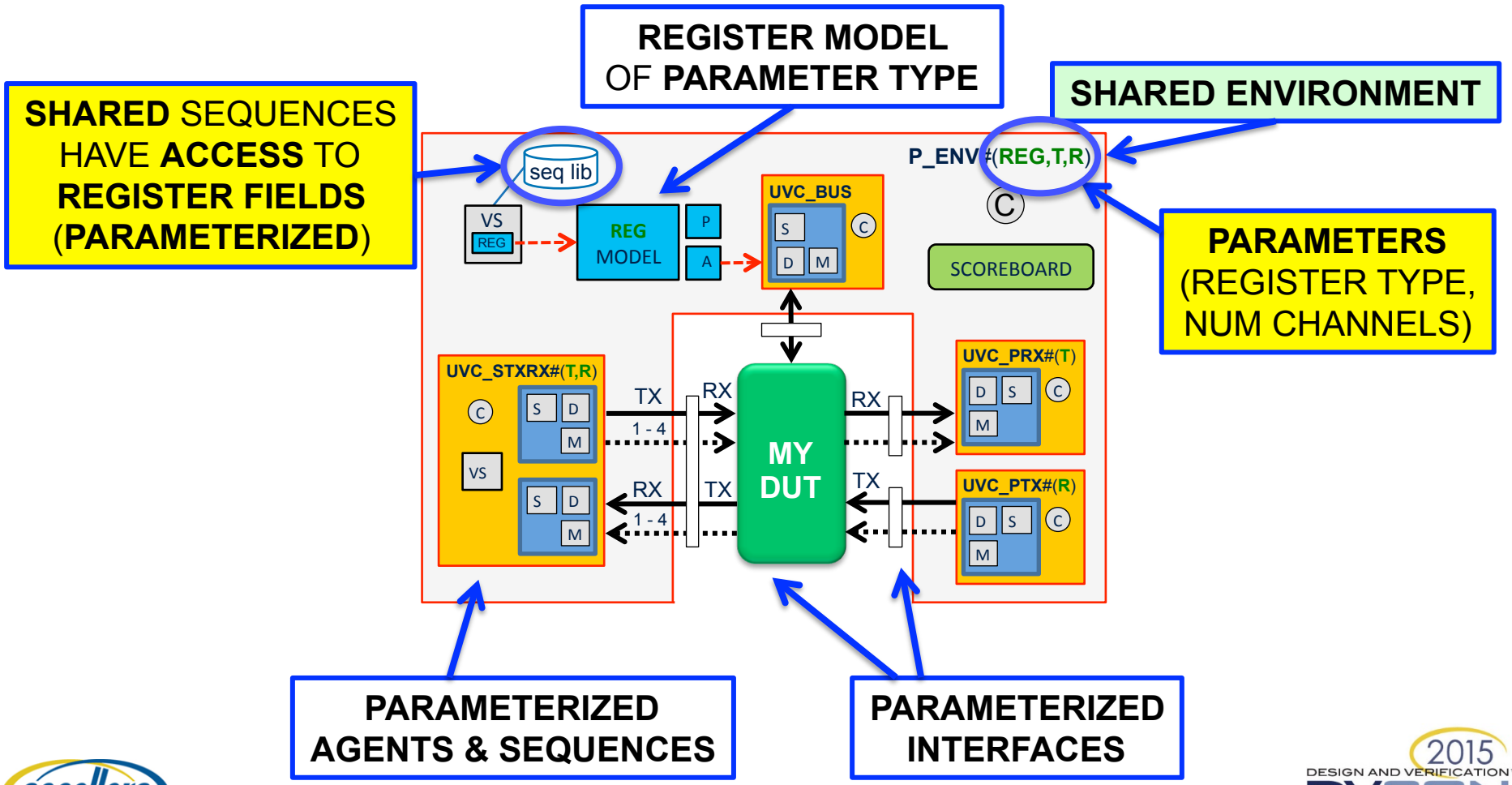
Parameterized Classes

- 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!

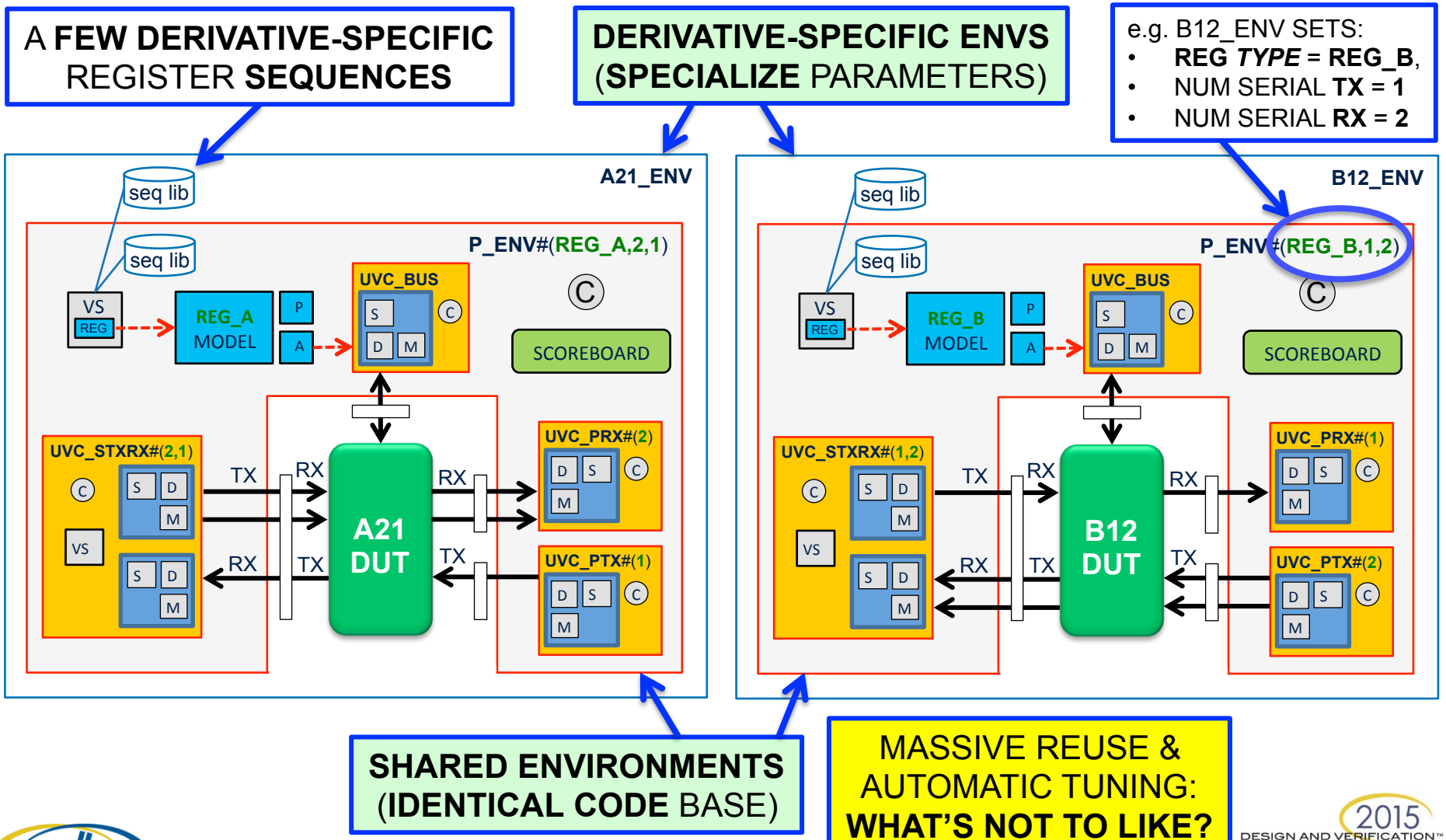
Non-Parameterized Setup



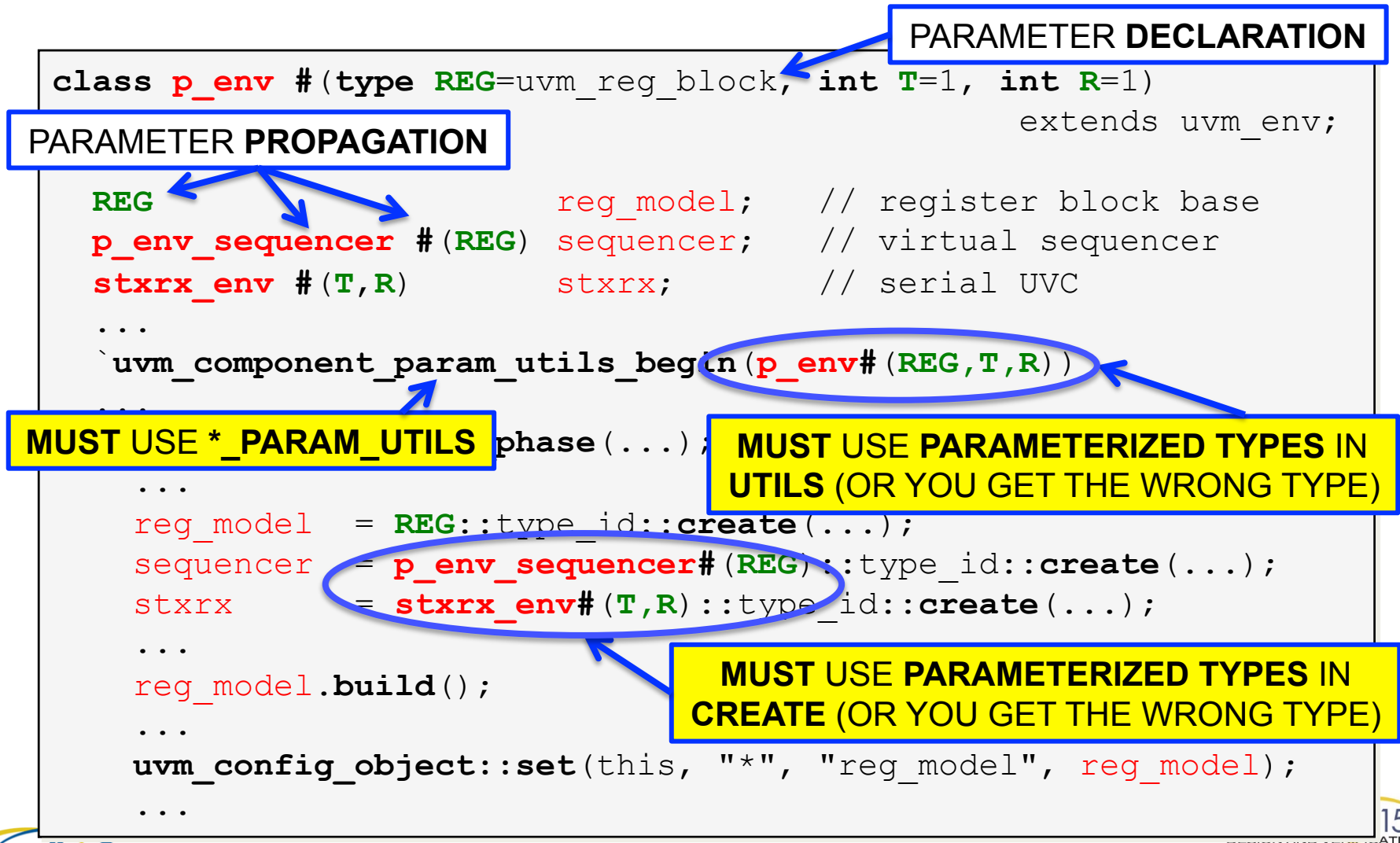
Parameterized Environment



Reuse of Parameterized Env



Generic *p_env* Environment




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

  REG                  reg_model;    // register block base
  p_env_sequencer #(REG) sequencer;  // virtual sequencer
  stxrx_env #(T,R)     stxrx;        // serial UVC
  ...
  `uvm_component_param_utils_begin(p_env#(REG,T,R))
  ...
  function void build_phase(...);
    ...
    reg_model  = REG::type_id::create(...);
    sequencer  = p_env_sequencer#(REG)::type_id::create(...);
    stxrx      = stxrx_env#(T,R)::type_id::create(...);
    ...
    reg_model.build();
    ...
    uvm_config_object::set(this, "*", "reg_model", reg_model);
    ...
```

uvc_stxrx – Env/Agent/Comps

```
class stxrx_env #(int NT=1, int NR=1) extends uvm_env;
  stx_agent #(NT) tx_agent;
  srx_agent #(NR) rx_agent;
  ...
```

PROPAGATE PARAMETERS THROUGHOUT **HIERARCHY**

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

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

```
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.)

p_env – Virtual Sequencer

PARAMETERIZE SEQUENCER TO GIVE **SEQUENCES ACCESS** TO **REG_MODEL** OF **CORRECT TYPE**

```
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

P_SEQUENCER MUST BE **CORRECT TYPE**
OR YOU GET **RUN-TIME CAST FAIL** ERROR
"...SEQ CANNOT RUN ON SEQUENCER TYPE..."

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

  `uvm_declare_p_sequencer(p_env_sequencer #(REG))
  `uvm_object_param_utils(p_env_base_seq #(REG))
  ...
```

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

```
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

  `uvm_object_param_utils_begin(p_env_init_seq #(REG))
  ...
```

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);
    ...
```

DON'T FORGET THIS EITHER ☺

... OR **REPLACE** ALL OF THAT **WITH CONVENIENCE TYPE** DEFINITION

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

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;
```

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

USE CONVENIENCE TYPE

```
typedef p_env_init_seq   #(reg_a,2,1) a21_init_seq;
typedef p_env_config_seq #(reg_a,2,1) a21_config_seq;
...
```

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

DUT-Specific Tests

```
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);
```

SEQUENCES ARE ALREADY **SPECIALIZED** BY **TYPEDEFS**

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

SEQUENCES ALL **RUN** ON **ENVIRONMENT SEQUENCER**

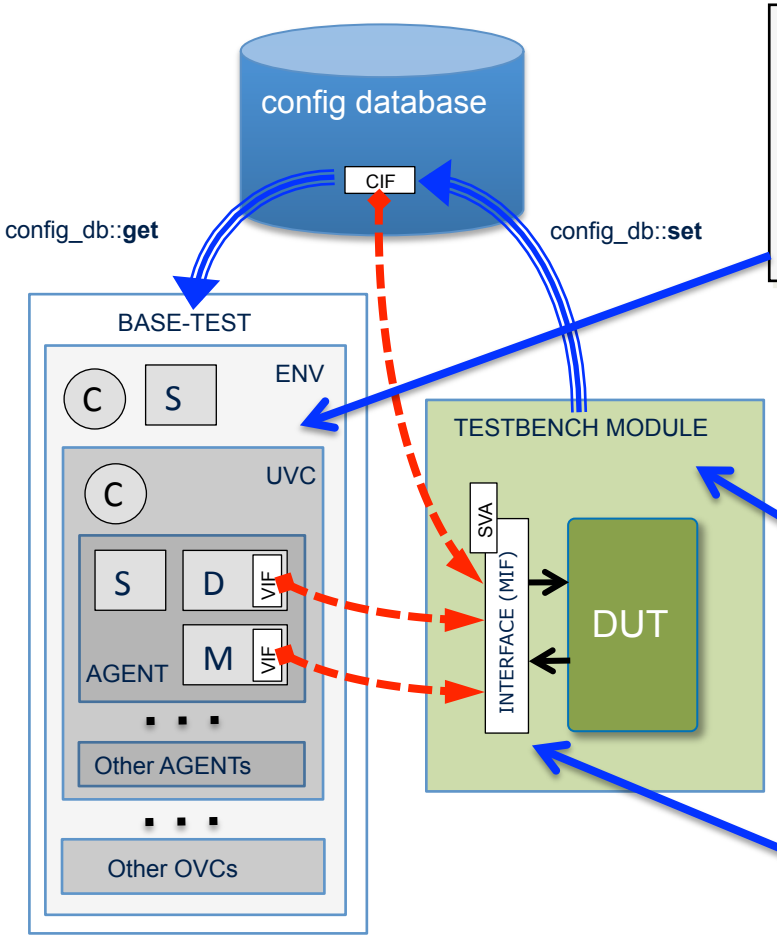
```
class test_example extends a21_base_test;
  test_example_seq example_seq = new();
  virtual task run_phase(...);
    example_seq.start(tb.env.sequencer);
    tb.env.reg_model.RX1_OFFSET_FIELD.write(status, 0);
    ...
```

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 ...

```
interface my_intf #(int WIDTH=1)();
 logic [WIDTH-1:0] addr;
 ...
```

SIMPLE CHANGE TO **ADD PARAMETER** TO INTERFACE

```
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 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)
 `uvm_component_param_utils(monitor#(W))
 uvm_analysis_port #(trans#(W))
 virtual my_intf#(W) vif;
```

```
class agent#(int W=1) extends uv
 `uvm_component_param_utils(agen
 monitor#(W) mon;
```

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

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

PARAMETERIZED INTERFACE

PARAMETER PROLIFERATION

```
class trans#(int W=1)
                    uvm_sequence_item;
 rand bit[W-1:0] addr;
 `uvm_object_param_utils(trans#(W))
```

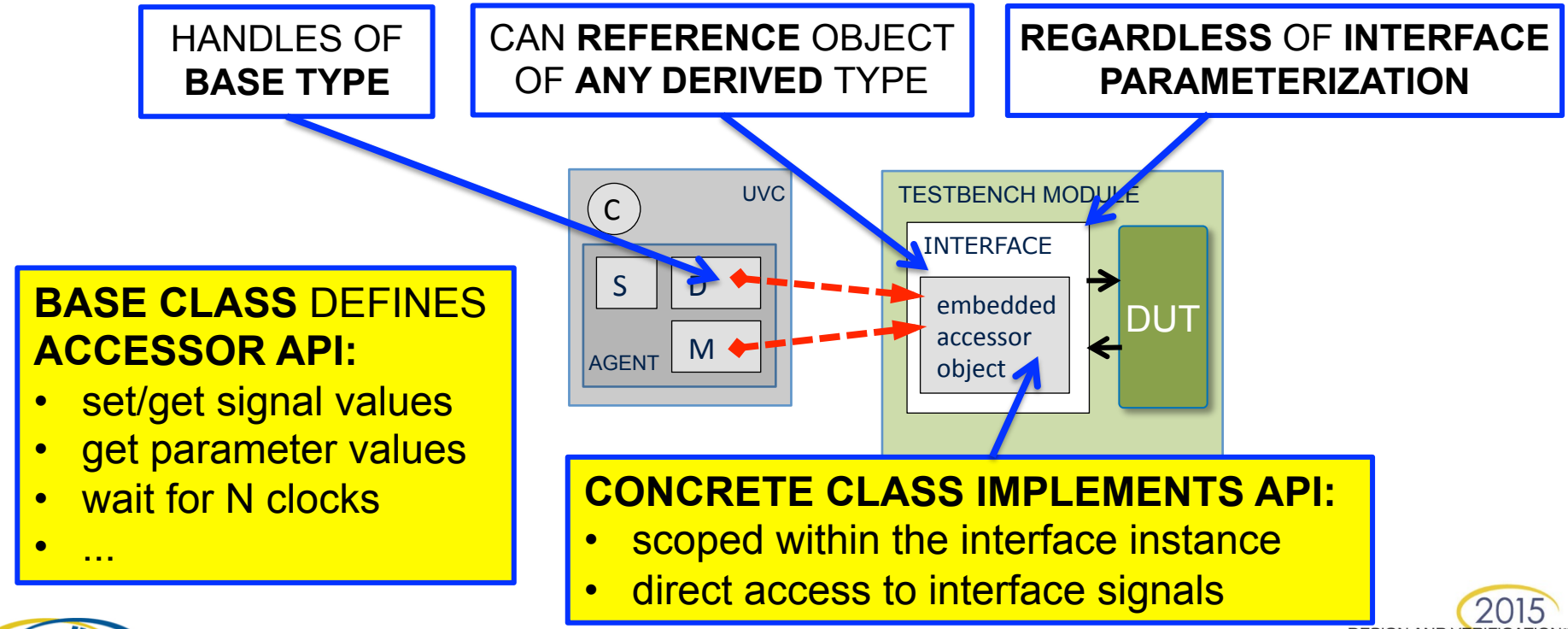
```
class seq#(int W=1)extends
        uvm_sequence#(trans#(W))
 `uvm_object_param_utils(seq#(W))
```

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!



Accessor Class Code

```
`define MAX_WIDTH 256
...
typedef bit [MAX_WIDTH-1:0] t_uvc_data;
```

MAX-FOOTPRINT PARAMETER

ABSTRACT **BASE CLASS** DEFINES **ACCESSOR API**

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

DEFINES **FULL API,** IS A **LOT OF CODE, RESTRICTED** TO API

GET HANDLE FROM **CONFIG_DB** (**LIKE VIF** BUT A **CLASS** HANDLE)

```
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();
      ...
```

MONITOR & **DRIVER** USE VARIABLE OF THE **ABSTRACT TYPE** TO DO **LOW-LEVEL ACCESSES**

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
```

PARAMETERIZED INTERFACE

CONTAINS ACCESSOR CLASS

EXTEND ABSTRACT **BASE-CLASS & IMPLEMENT METHODS**

DIRECT ACCESS TO INTERFACE **VARIABLES** & **PARAMETERS**

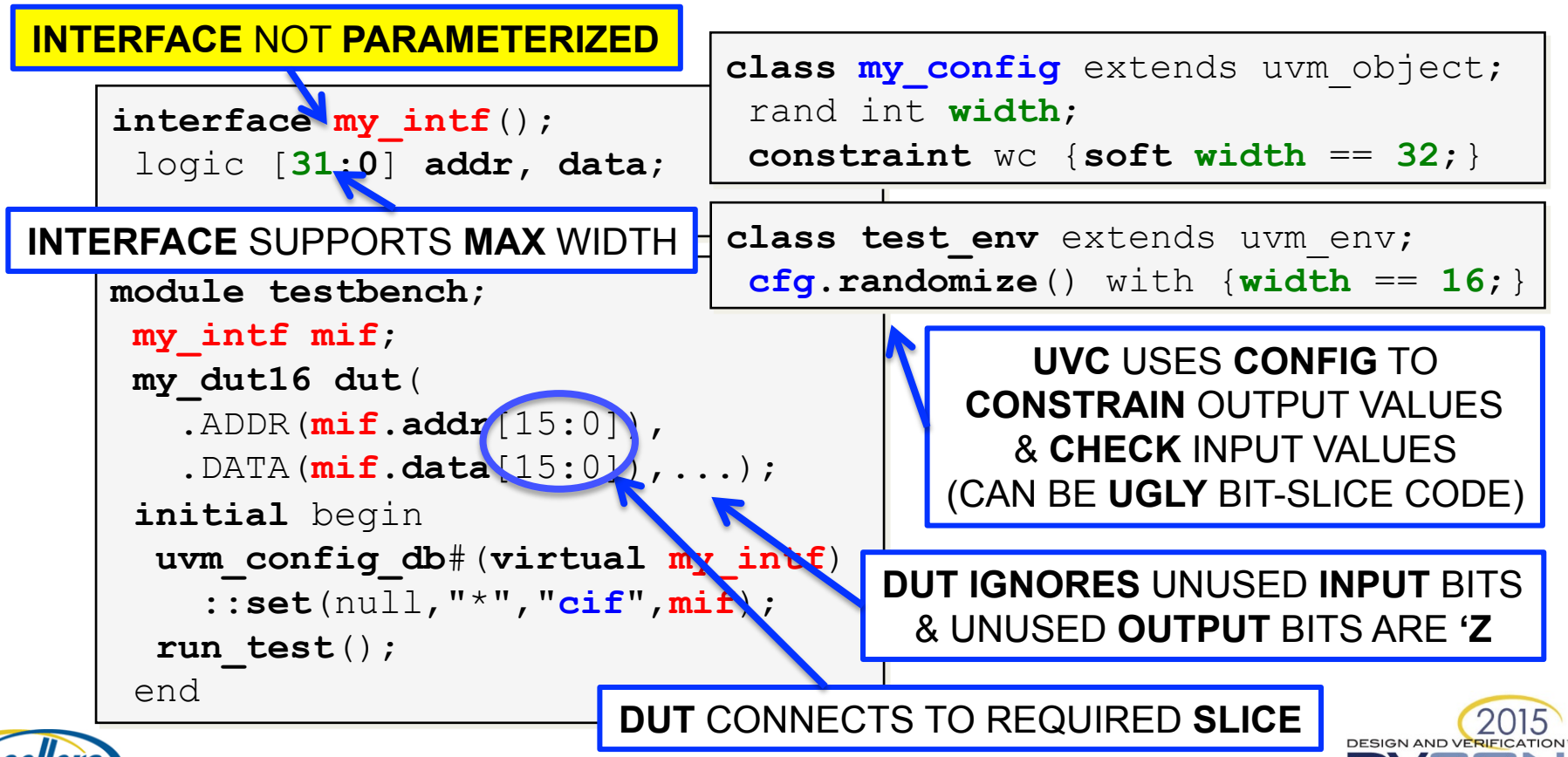
CREATE (WITH UNIQUE NAME) AT **INITIALIZATION** TO **AVOID RACE** WITH **UVM TEST START**

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



Interface Wrapper

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

```
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; //
 assign addr = mif.addr[WIDTH-1:0]; //
 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();
```

PARAMETERIZED WRAPPER INTERFACE

NON-PARAMETERIZED MAXIMUM FOOTPRINT INTERFACE FOR UVC

ENCAPSULATE PARTIAL SIGNAL **ASSIGNMENTS**

ENCAPSULATE CONFIG FOR **VIRTUAL INTERFACE**

EASY WRAPPER USE IN TESTBENCH MODULE

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

MOST **COMMON** APPROACH

APPROPRIATE FOR **RTL IP** & **UVC** COMBO PACKAGES

PARAMETERIZED REGISTERS

CONFIGURABLE REGISTERS

Normal Registers

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

```
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);
```

CREATE & CONFIGURE EACH **FIELD**

FIELD **WIDTH** USES A **GENERATED NUMBER**

```
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** & **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);
```

FIELD **WIDTH** USES **PARAMETER**

```
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();
```

PARAMETER PROLIFERATION (REG, BLOCK,...)

ALL **GENERATED** CODE

Using Parameterized Regis

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

- 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
  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));
  `uvm_declare_p_sequencer(reg_sequencer#(W))
```

```
class top_vsequencer extends uvm_sequencer;
  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...)

ALL **USER** CODE

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();
```

NOTE : REG_MODEL IS **CONSTRUCTED** BUT **EMPTY** AFTER **CREATE**

CREATE THE **REG_BLOCK**

```
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();
```

BUILD THE **REG_BLOCK**

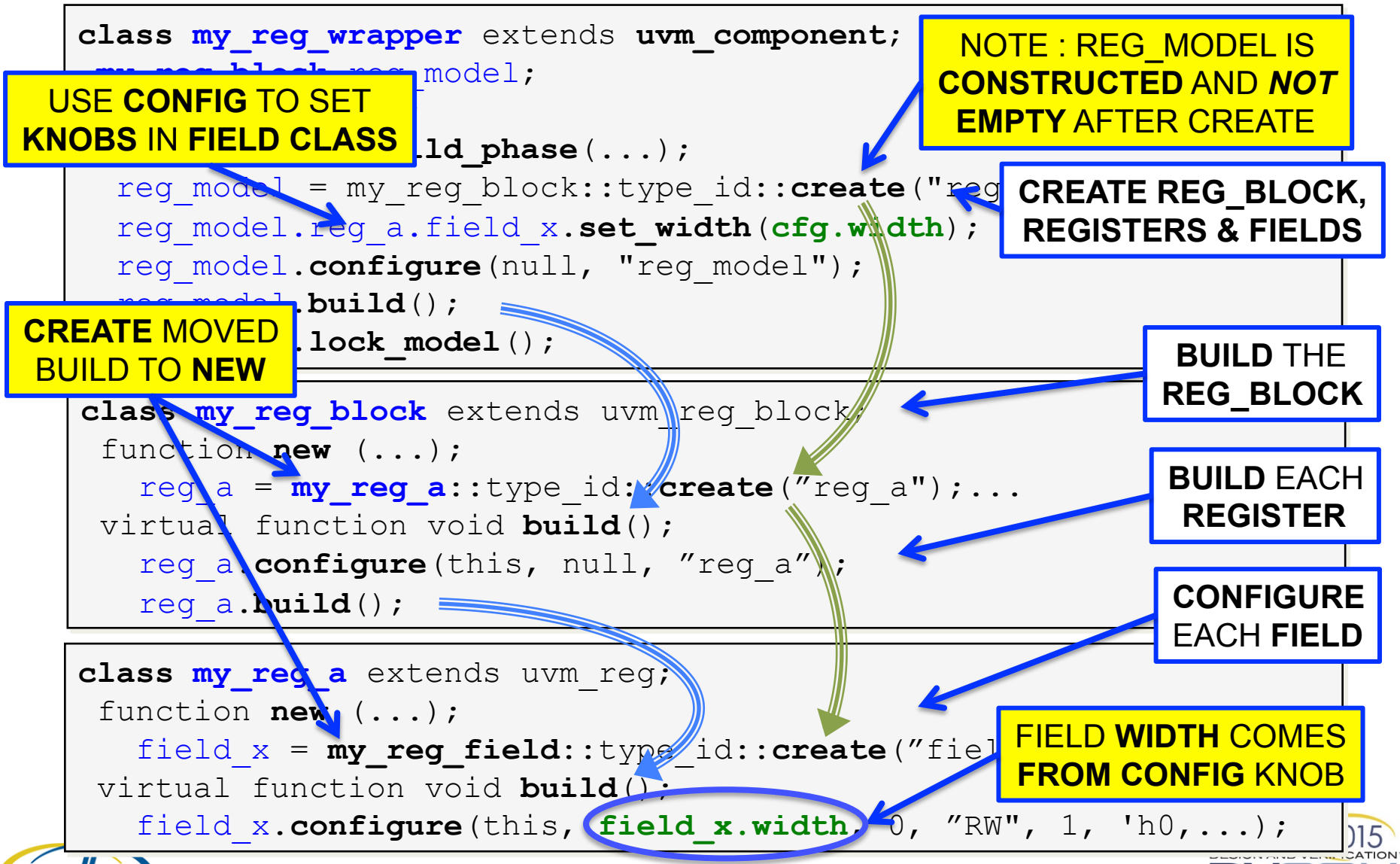
CREATE & BUILD EACH **REGISTER**

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

CREATE & CONFIGURE EACH **FIELD**

FIELD **WIDTH** USES A **GENERATED NUMBER**

Configurable Register Build



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);
    ...
```

CONFIGURATION-AWARE REGISTERS
USING **WRAPPER** COMPONENT
IS MORE **AD-HOC** AND **LIMITED**
BUT BETTER **ENCAPSULATED**

INSTANTIATE WRAPPER

CREATE WRAPPER

PASS DOWN CONFIG

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***"
  D.Rich, J.Bromley, DVCon 2008, www.doulos.com/downloads
- "***Parameterized Interfaces and Reusable VIP***"
  A.Pratt, Blog, https://blogs.synopsys.com/vip-central
- "***On SystemVerilog Interface Polymorphism and Extendability***"
  T.Timi, Blog, http://blog.verificationgentleman.com
- "***Pragmatic Verification Reuse in a Vertical World***"
  M.Litterick, DVCon 2013, www.verilab.com/resources

Questions