# Advanced UVM in the real world

## - Tutorial -

Mark Litterick
Jason Sprott
Jonathan Bromley
(Vanessa Cooper)

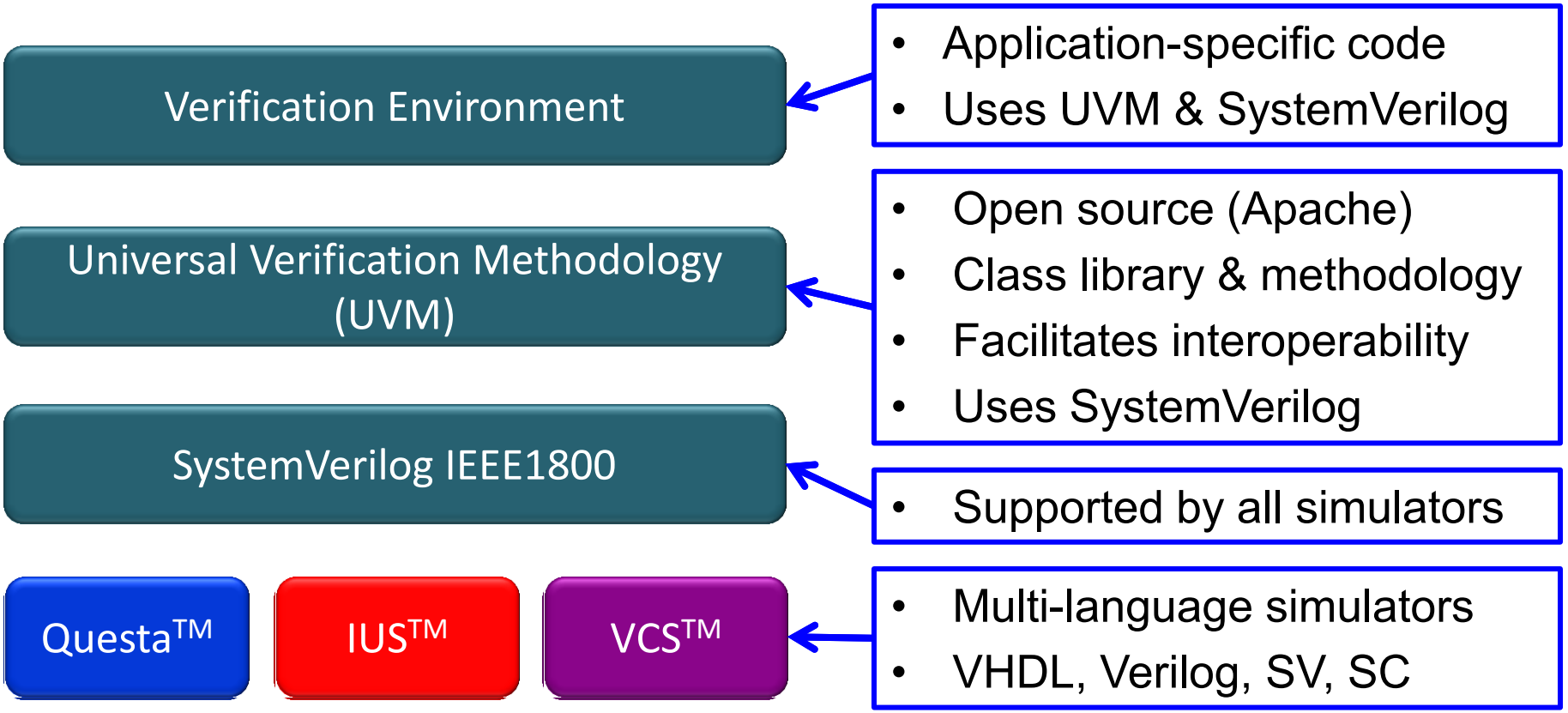
accellera SYSTEMS INITIATIVE

verilab

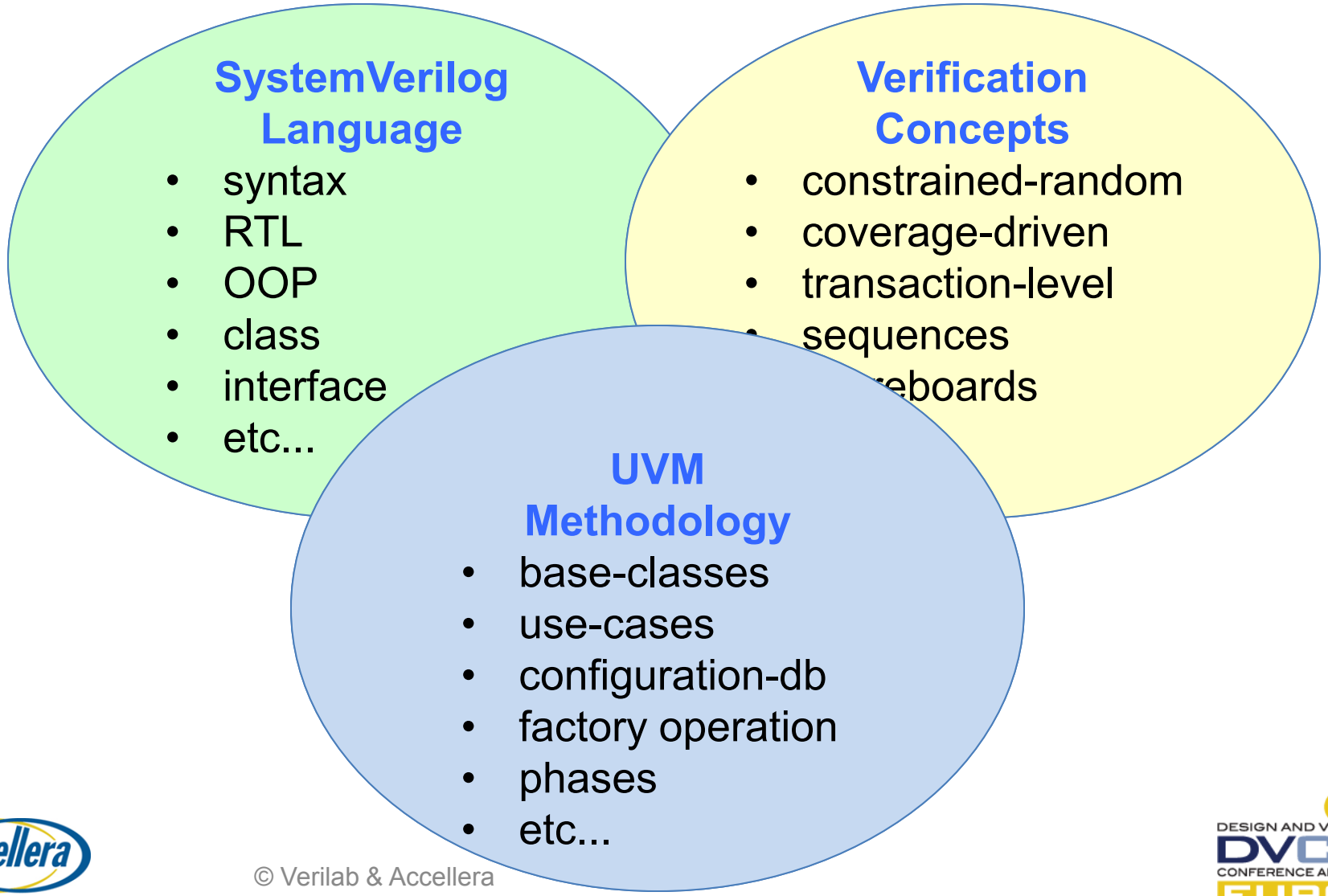
2014 DESIGN AND VERIFICATION DVCON CONFERENCE AND EXHIBITION EUROPE

# INTRODUCTION

# What is UVM?


Verification Environment
Universal Verification Methodology (UVM)
SystemVerilog IEEE1800
Questa™
IUS™
VCS™
• Application-specific code
• Uses UVM & SystemVerilog
• Open source (Apache)
• Class library & methodology
• Facilitates interoperability
• Uses SystemVerilog
• Supported by all simulators
• Multi-language simulators
• VHDL, Verilog, SV, SC

# Key Elements of UVM

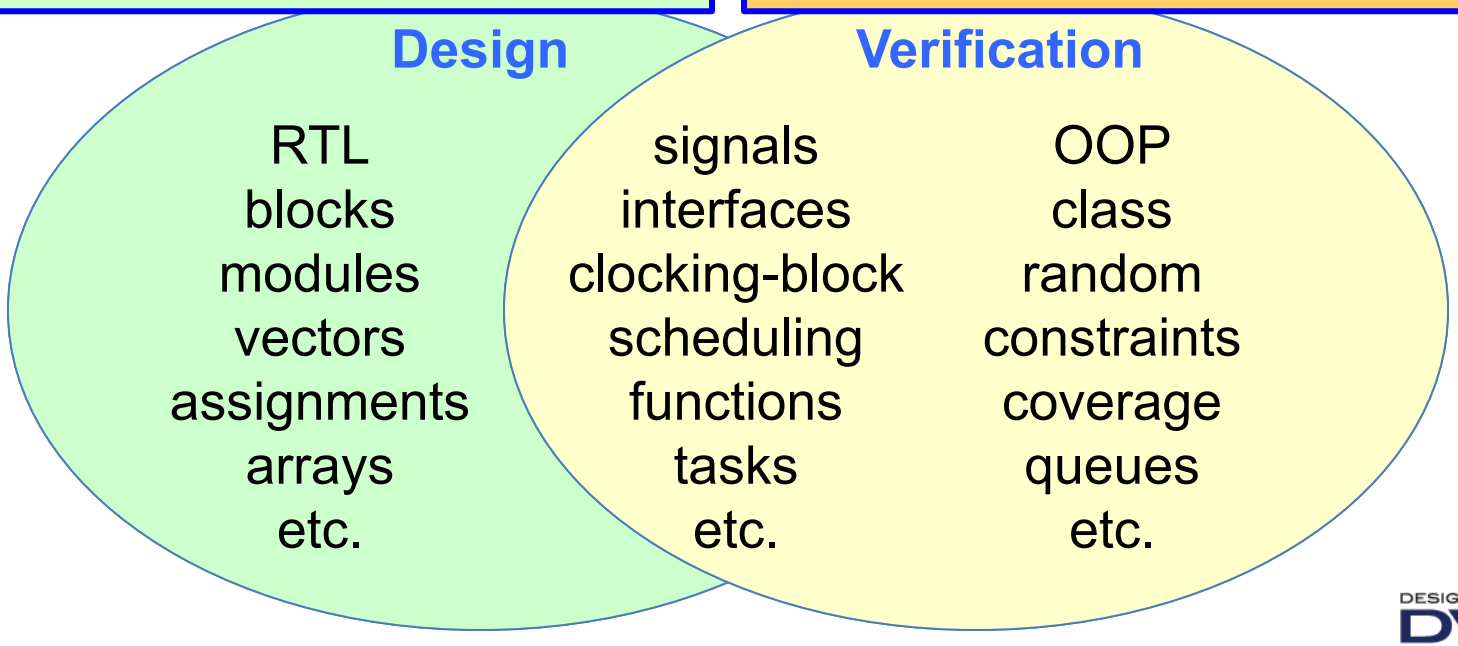


# SystemVerilog

- Language syntax & semantics are pre-requisite
  - detailed understanding is not unique to UVM...

all **SystemVerilog experience** is directly **relevant** for **UVM** (design/RTL, AVM, VMM, etc.)

...but be aware the **verification** part of language is much **bigger** than that used for **design**!

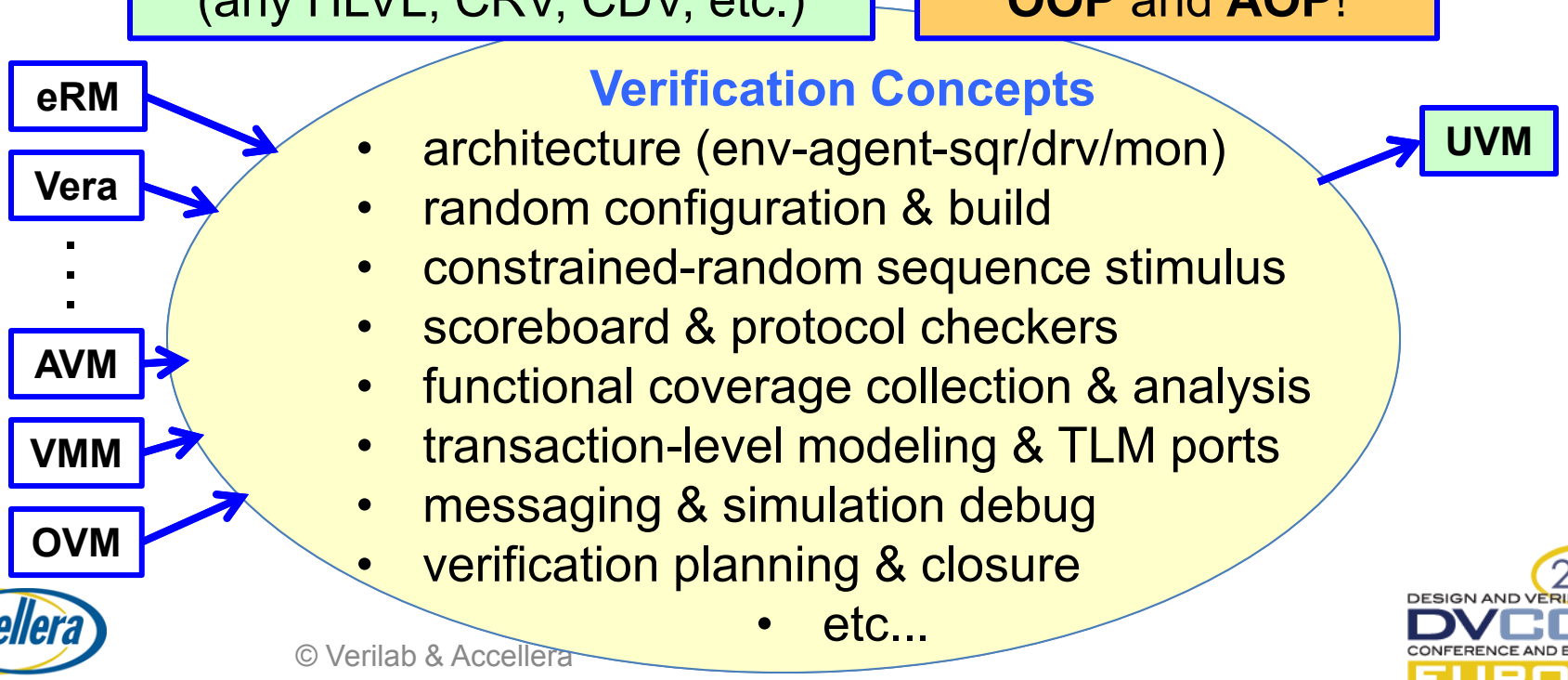


# Verification Concepts

- Generic language-independent concepts apply
  - detailed understanding is not unique to UVM...

all **verification experience** is directly **transferrable** to **UVM** (any HLVL, CRV, CDV, etc.)

...but be aware of the **difference** between **OOP** and **AOP**!



# UVM Methodology

- **Base-class library**
  - generic building blocks
  - solutions to software patterns
  - save time & effort
- **Way of doing things**
  - consistent approach
  - facilitates interoperability
  - enables workforce flexibility

**UVM** specific **stuff** has to be **learned**

...but with **SystemVerilog** and **verification** knowledge it is ***not*** a **huge effort!**

- reg-model
- factory
- config-db
- callbacks
- parameterizing
- sequences
- seq-items
- transactions
- phases
- transaction-recording
- event-pool
- field-macros
- TLM-ports
- virtual-interfaces
- messaging
- components
- objects

**hard**

**easy**

# Tutorial Topics

- Selected based on:
  - **experience** on many projects at different clients
  - relatively **complex** implementation or confusing for user
  - benefit from deeper **understanding** of background code
  - require more **description** than standard documentation
  - **time** available for the tutorial!
- Demystifying the UVM **Configuration Database**
- Behind the Scenes of the **UVM Factory**
- Effective **Stimulus** & **Sequence Hierarchies**
- Advanced UVM **Register Modeling** & Performance

# Demystifying the UVM Configuration Database

Jason Sprott, Verilab, Ltd.

Vanessa Cooper, Verilab, Inc.

accellera SYSTEMS INITIATIVE

verilab

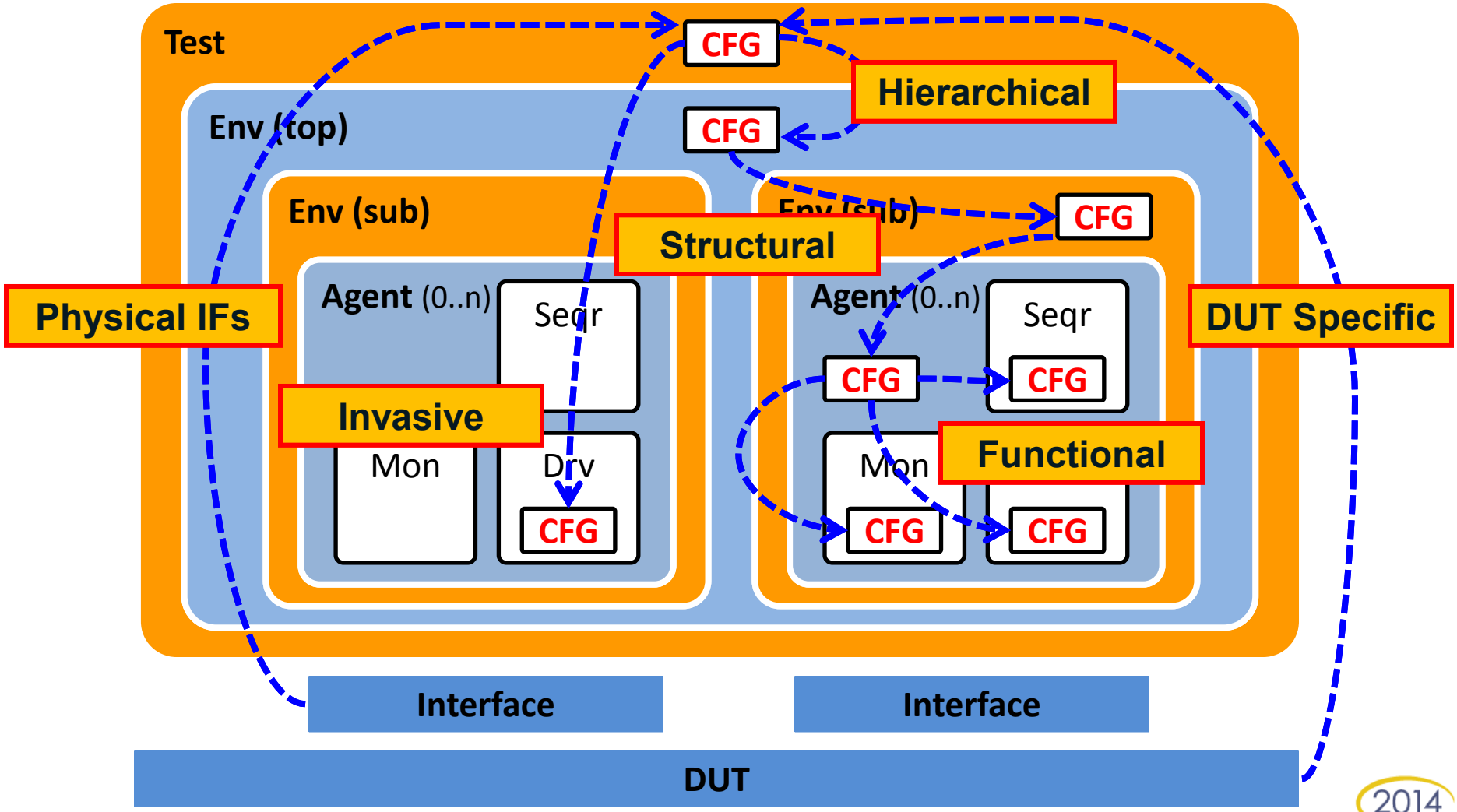
2014 DESIGN AND VERIFICATION DVCON CONFERENCE AND EXHIBITION EUROPE

# Agenda

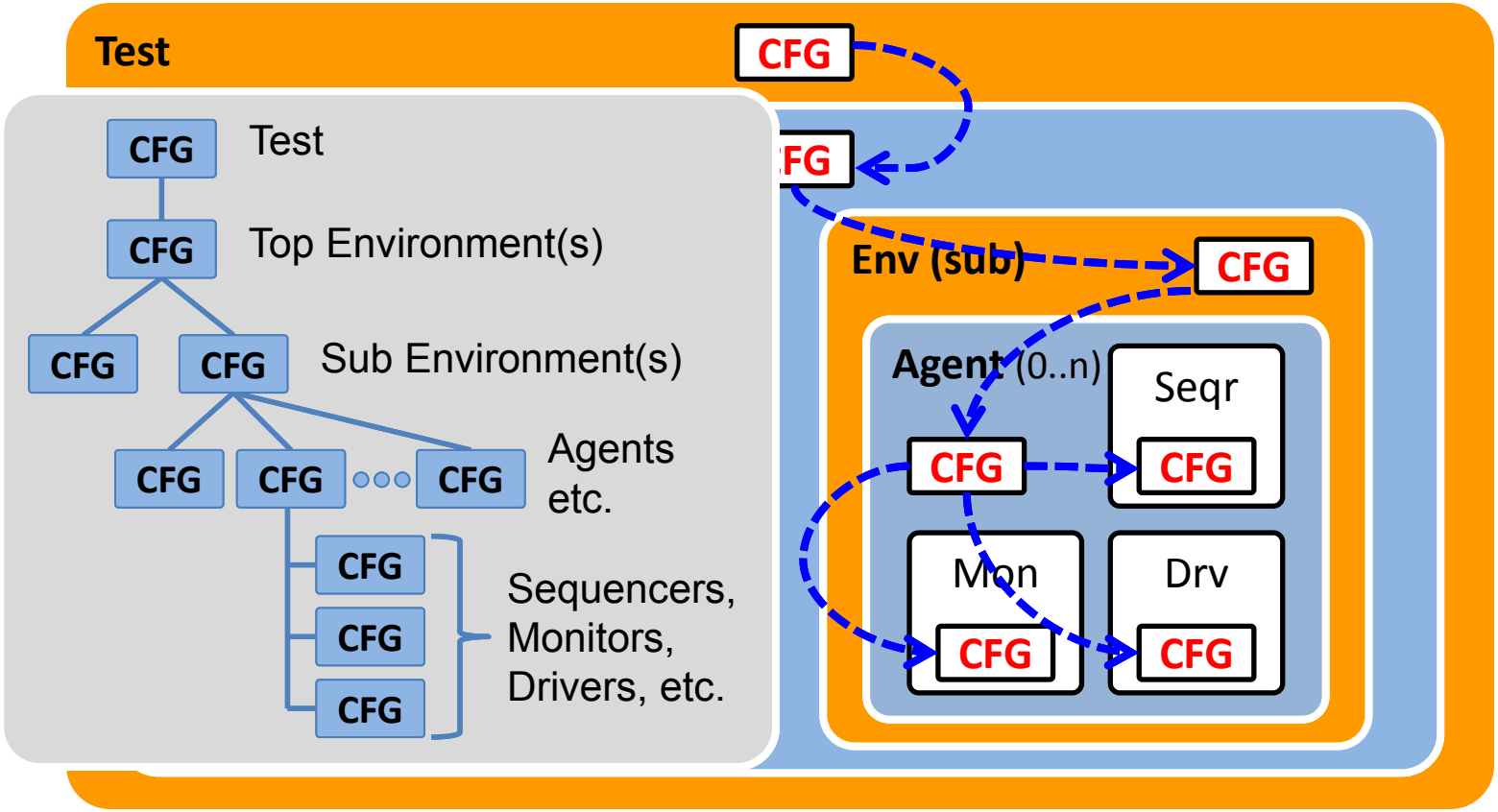
- Overview of the problem
- Summary of relevant UVM 1.2 changes
- Basic syntax and usage
- Automatic configuration
- Hierarchical access discussion and examples
- Using configuration objects
- Debugging
- Gotchas
- Conclusions

# OVERVIEW

# Where is configuration used?



# Config Mirrors Testbench Topology




- We need a consistent way of storing and accessing data
- Really flexible access from anywhere
- Understanding of hierarchy and control of scope

# UVM 1.2 Changes

- Mantis 3472: UVM 1.1 set/get_config* methods deprecated

```
set_config_int(...) => uvm_config_db#(uvm_bitstream_t)::set(cntxt,...)
set_config_string(...) => uvm_config_db#(string)::set(cntxt,...)
set_config_object(...) => uvm_config_db#(uvm_object)::set(cntxt,...)
```

- In UVM 1.2 we use uvm_config_db methods directly

```
uvm_config_db#(T)::set(cntxt,"inst","field",value);
```

```
uvm_config_db#(T)::get(cntxt,"inst","field",value);
```

- Mantis 4666: bug fix for process problem in set()
- Mantis 3693: bug fix for command line enums
- Mantis 4920: bug fix for random stability when config database queried
- There is a uvm11-to-uvm12.pl conversion script

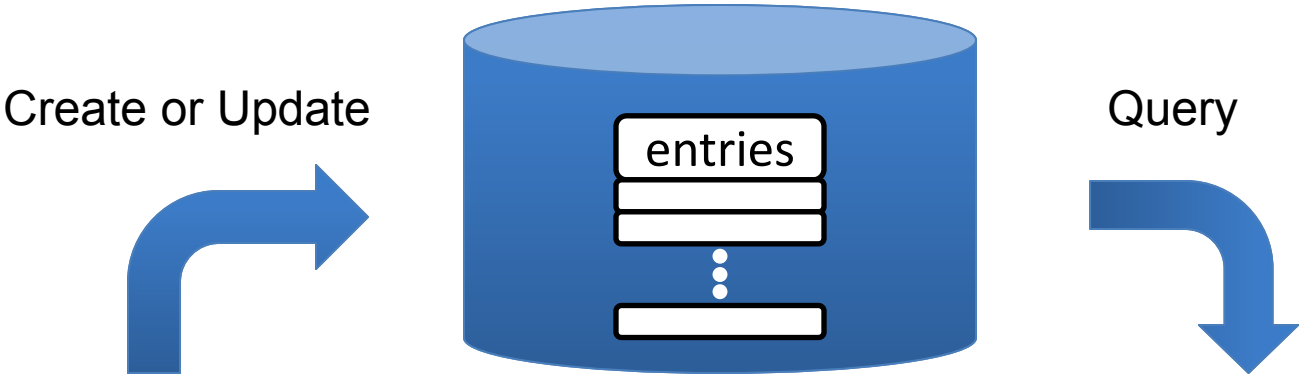
# BASIC SYNTAX AND USAGE

# Configuration Information Database

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

- Built on top of existing **uvm_resource_db**
- Small number of static methods(**::** notation)
- String based keys to store and retrieve entries
- Supports hierarchy and controlled visibility
- Supports built-in and custom types (objects)
- Can be used all levels of a testbench
- Simplifies access, automates processes

# Configuration Information Database




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

```
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 entries
target component variables are modified by **get()**

# Convenience Types

From **uvm_config_db.svh**

```
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;
```

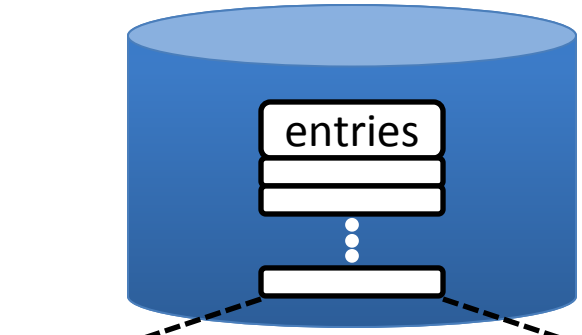
For **objects** use this style (or long-hand version above)

```
set_config_object::set(this,"env.agent1","config", m_config)
```

For **enums** use this style (or long-hand version above)

```
set_config_int::set(this,"env.agent1","bus_sz", SZ_128 )
```

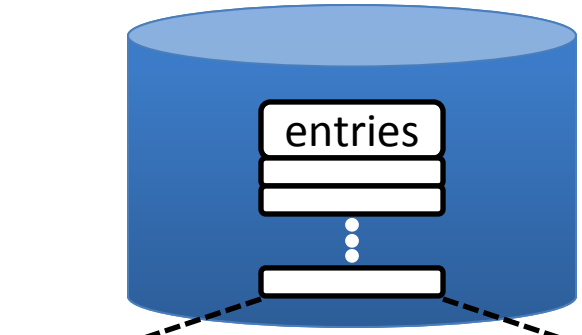
# Creating & Modifying Entries




- Creates entry if none exists
- Updates value if exists
- Does **not** modify target component variables


```
uvm_config_db#(T)::set(cntxt,"inst","field",value);
```

# A bit about context

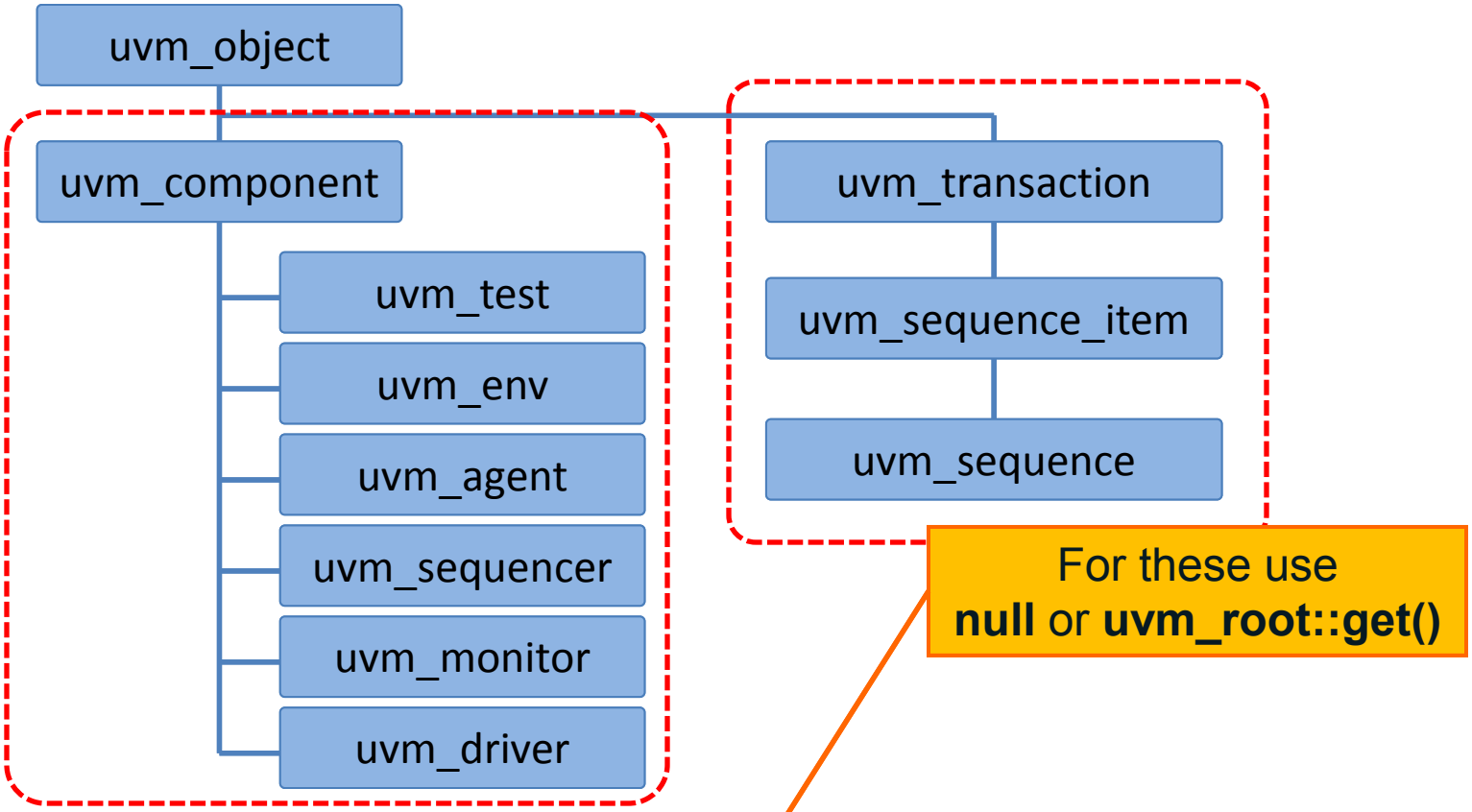



- Creates entry if none exists
- Updates value if exists
- Does **not** modify target component variables

```
uvm_config_db#(uvm_object)::set(this,"env.agent1","config", m_config)
```

- The entry is visible to components matching the **full context**:
  (e.g. if current instance is test1) `test1.env.agent1`
- Is identified using the key `"config"` (not related to value variable name)
- And takes the type specific **value** of `m_config` supplied in argument

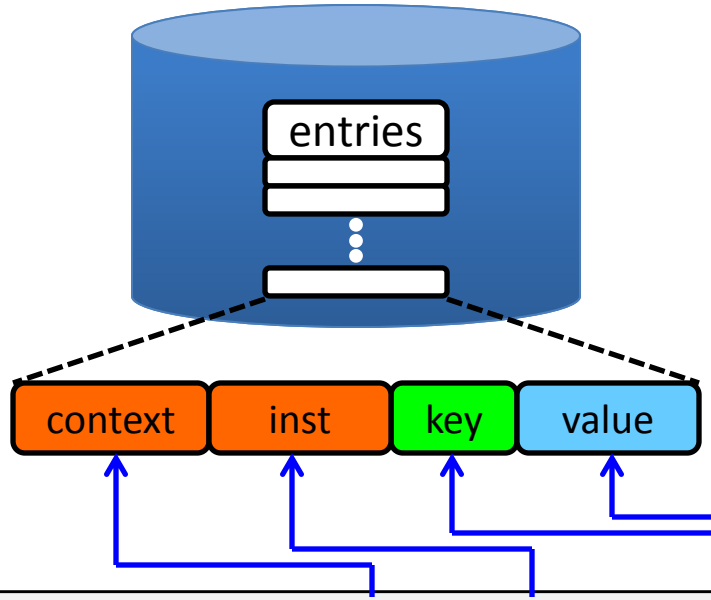
# A bit more about context




```
uvm_config_db#(uvm_object)::set(cntxt,"inst","field",value)
```

**cntxt** must be of type **uvm_component**, **null** or **uvm_root::get()**

# Creating & Modifying Entries Examples



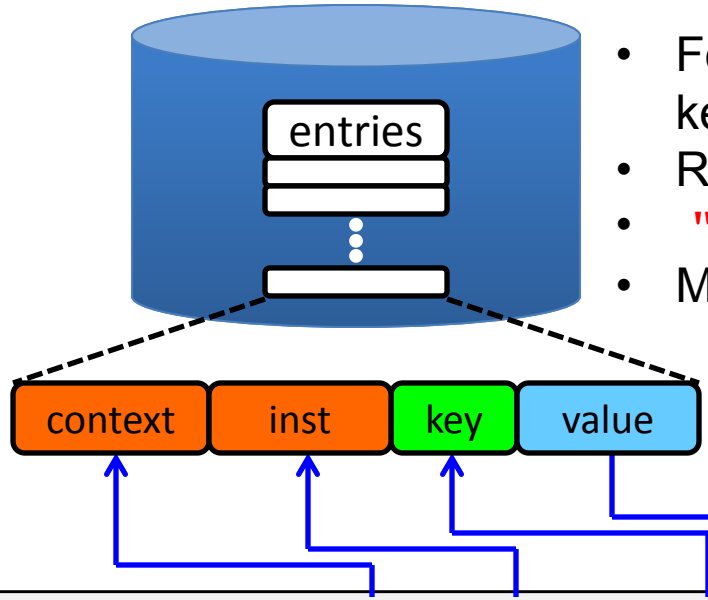

```
uvm_config_db#(T)::set(cntxt,"inst","field",value);
```

```
uvm_config_db#(T)::set(uvm_root::get(),"test1.env.agent1",
                       "config", m_config)
```

```
uvm_config_db#(T)::set(this,"env.agent*","config", m_config)
```

```
uvm_config_db#(T)::set(null,"*","global_cfg", m_global_cfg)
```

# Fetching Entries

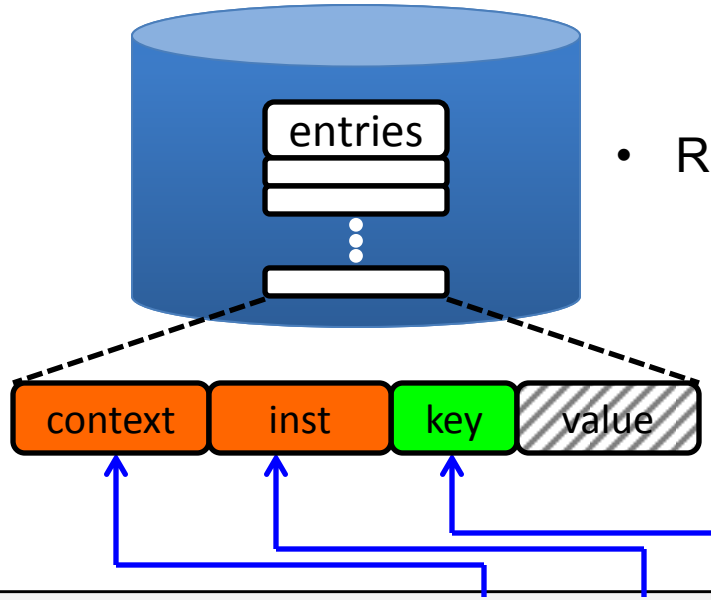



- Fetches entry matching the key string at the full context
- Returns 0 on a fail
- "" = current instance
- Modifies target variable

```
uvm_config_db#(T)::get(cntxt,"inst","field",value);
```

```
if(!uvm_config_db#(uvm_object)::get(this, "","config",m_config))
begin
    `uvm_fatal(...)
end
```

# Checking Entry Exists

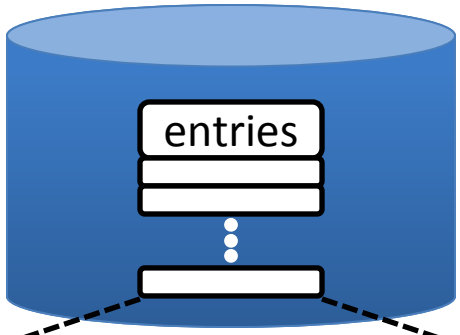



- Returns 1 if entry exists

```
uvm_config_db#(T)::exists(cntxt,"inst","field");
```

```
if(!uvm_config_db#(int)::exists(this,"","low_power_mode")) begin
   // do something interesting
end
```

# Flow Control




- Blocking task
- Waits on a set() to unblock

Only suitable for simple synchronization

```
uvm_config_db#(T)::wait_modified(cntxt,"inst","field");
```

```
// wait until someone changes value of entry
uvm_config_db#(int)::wait_modified(this, "", "sb_enable");
// We know the variable has been modified but we still
// need to do a get()to fetch new value
```

Not sensitive to an object's **contents** changing

# AUTOMATIC CONFIGURATION

# Automatic Field Configuration

- Configures all variables registered using field macros

```
function void uvm_component::build_phase(...);
  apply_config_settings(...); // find fields, do get(), $cast
endfunction
```

Only called **once** at build time

- build phase for derived comps should call **super.build**

```
class my_comp extends
  `uvm_component_utils_begin(my_comp)
    `uvm_field_int(my_field,UVM_DEFAULT)
    `uvm_field_object(my_special,(UVM_DEFAULT|UVM_READONLY))
    `uvm_field_object(my_config,UVM_DEFAULT)

  ...
  function void build_phase(...);
    super.build_phase(...);
    ...
  endfunction
```

missing **field-macro** results in **no auto-config**

UVM_READONLY results in **no auto-config**

missing **super.build** results in **no auto-config**

# Automatic Configuration & Objects

| ::set() type | ::get() type | auto config | explicit ::get() | |
|---|---|---|---|---|
| uvm_object | uvm_object | ✓ | ✓ | $cast to concrete type required for explicit get() |
| uvm_object | my_config | ✓ | ✗ | Wrong |
| my_config | my_config | ✗ | ✓ | Breaks auto-config ☹ |
| my_config | uvm_object | ✗ | ✗ | Wrong |

Recommend using **uvm_config_object** typedef

# Explicit get() needs a cast

For **objects**

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

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

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

For **enums**

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

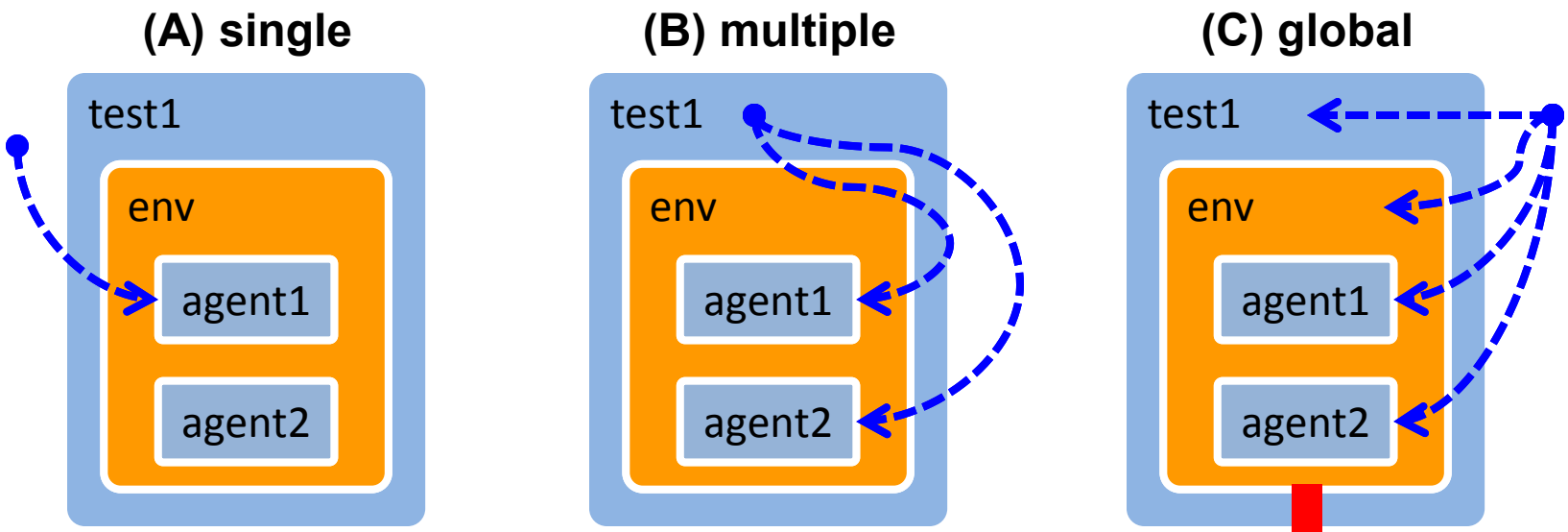
```
int tmp;
uvm_config_int::get(... ,"m_bus_sz", tmp)
m_bus_sz = my_enum_t'(tmp); // back to original type
```

enums can use **int**, **uvm_bitstream_t** or **uvm_integral_t**

using convenience types is typically less hassle

# HIERARCHICAL ACCESS DISCUSSION & EXAMPLES

# Hierarchical Access Examples




```
uvm_config_db#(T)::set(uvm_root::get(), "test1.env.agent1",
                       "config", m_config)                    (A)

uvm_config_db#(T)::set(this,"env.agent*","config", m_config)  (B)

uvm_config_db#(T)::set(null,"*","global_cfg", m_global_cfg)   (C)
```

Dangerous unless you can guarantee no name clashes

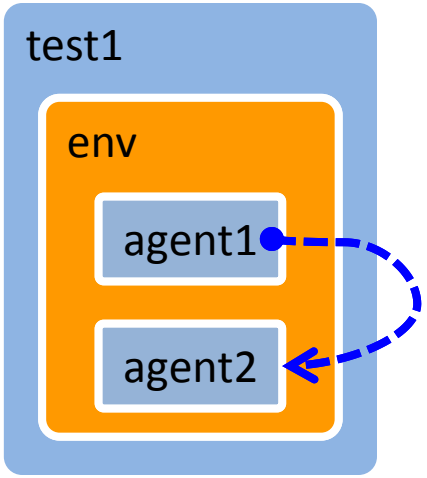
# Hierarchical Access Examples

- Normally we only fetch what we are supposed to see

```
uvm_config_db#(T)::get(this,"","config", m_config)
```

- We can actually access anything

Not advisable unless components are tightly coupled




```
uvm_config_db#(T)::get(uvm_root::get(),"test1.env.agent2",
                       "config", m_config)
```

# CONFIGURATION OBJECTS

# Using objects for configuration

- Do it, but you don't have to for everything
  - There will still be some discrete variables
- Group related data
- Pass by reference is useful and efficient
  - Object handles rarely change after build()
  - Changes to object contents can be seen immediately
- We can use any type of variable inside a class
- We have the option of adding a custom functionality
- Option to randomize
- Good for reuse – also recommend using the factory

```
m_srio_config = uvc_srio_config::type_id::create("m_srio_config");
```

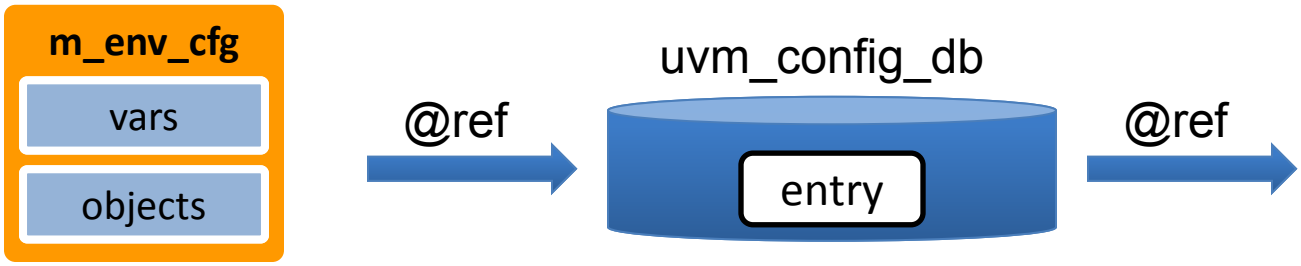
# Using Config Objects

**before or during build phase**

```
::set(...,"config",m_env_cfg);
```

**in or after build phase**

```
::get(...,"config",m_config);
```




object reference stored, not object contents

We will see changes to the object **contents** without a get()

```
if (m_config.var1 == ...)
```

```
if (m_config.obj1.var2 == ...)
```

```
m_copy_of_var = m_config.var1
```

copies can go out-of-date if contents change

# Comparing Styles

```
set(this,"env","config",m_cfg)
```

## Using uvm_config_db

- Target **pulls** (if needed)
- set() only makes data available to target(s)
- Target doesn't have to exist
- Automatic configuration

Recommended

```
env.m_foo_config = m_cfg
```

## Traditional

- Enclosing **pushes**
- Target must exist
- Potential ordering issues
- "Mother knows best!"
  - Sometimes she does
  - e.g. legacy VIPs

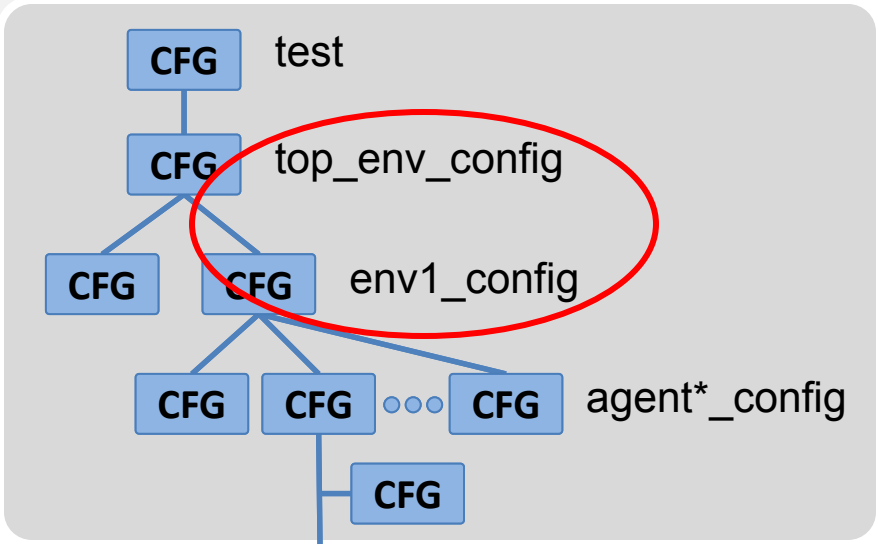
You may need a bit of both and that's OK

# Hierarchy of Config Objects

```
class top_env_config extends uvm_object;
   env1_config m_env1_config;
   env2_config m_env2_config;
   ...
endclass
```

handle to an instance of another config object

```
class env1_config extends uvm_object;
   int some_var;
   int some_other_var;
   agent1_config m_agent1_config;
   agent2_config m_agent2_config;
   ...
endclass
```



# Auto Config and Setup Next

```
class top_env extends uvm_env;
   env1 m_env1;
   env2 m_env2;
   top_env_config m_config;
   ...
   `uvm_component_utils_begin(top_env)
      `uvm_field_object(m_config, UVM_DEFAULT)
   `uvm_component_utils_end

   function build_phase();
      super.build_phase();
      set_config_object(this, "env1","m_config",
                        m_config.m_env1_config);
      set_config_object(this, "env2","m_config",
                        m_config.m_env2_config);
      m_env1 = env1::type_id::create("m_env1");
      m_env2 = env2::type_id::create("m_env2");
   endfunction
endclass
```

populated by auto config from set() done up the hierarchy

makes embedded env1 config **visible** to env1 (doesn't set any variables)

done even before env1 created

# Same Again Next Level Down

```
class env1 extends uvm_env;
   env1_config m_config;
   ...
   'uvm_component_utils_begin(top_env)
      'uvm_field_object(m_config, UVM_DEFAULT)
   'uvm_component_utils_end

   function void build_phase();
      super.build_phase();
      set_config_object(this, "m_agent1","m_config",
                        m_config.m_agent1_config);
      set_config_object(this, "m_agent2","m_config",
                        m_config.m_agent2_config);
      m_agent1 = env1::type_id::create("m_agent1");
      m_agent2 = env2::type_id::create("m_agent2");
   endfunction
endclass
```

populated by auto config from set() done up the hierarchy

makes embedded config **visible** to agent1 (doesn't set any variables)

# DEBUGGING

# Enabling Debug Trace

```
sim_cmd +UVM_TESTNAME=my_test +UVM_CONFIG_DB_TRACE
```

```
UVM_INFO reporter [CFGDB/SET] Configuration "*_agent.*_in_intf"
(type virtual interface dut_if) set by = (virtual interface
dut_if)

UVM_INFO report [CFGDB/GET] Configuration
"uvm_test_top.env.agent.driver.in_intf" (type virtual interface
dut_if) read by uvm_test_top.env.agent.driver = (virtual
interface dut_if) ?
```

Automatic configuration not as rigorous as your own checks

# Debug: check get() return value

- Defensive programming and informative messages

```
if (!uvm_config_db #(uvm_object)::get(this,"","m_config",m_config)
    || m_config == null)
begin
   print_config_with_audit(); // optional - context sensitive
   `uvm_fatal(get_type_name(),
   "Fetch of m_config failed. Needs to be setup by
    enclosing environment")
```

```
# UVM_INFO @ 0: uvm_test_top.env [CFGPRT] visible resources:
# ...
#   config [uvm_test_top.env] : (class uvm_pkg::uvm_object)
{top_env_config}
# UVM_INFO ... uvm_test_top reads: 0 @ 0  writes: 1 @ 0
#
# UVM_FATAL top_env.sv(44) @ 0: uvm_test_top.env [top_env] Fetch
  of m_config failed. Needs to be setup by enclosing environment
```

- print_config_with_audit() also shows variable values

# GOTCHAS

# Common Gotchas

- **Missing super.build_phase():** no automatic configuration
- **Missing field macro:** no automatic configuration
- **UVM_READONLY** on field: no automatic configuration
  - occasionally intentional to highlight explicit get() requirement
- **Missing get()** when **set()** called after build_phase()
  - Explicit get() required, as set() does not call *apply_config_settings*()
- **uvm_config_object::set()** writes a **null** object
  - automated configuration doesn't check for this
- **Wrong or mismatched types:** on enum or object set()
  - causes auto configuration issues
- **Missing cast:** for object or enum **explicit get()**
- **Typo** in string for inst or field names
- **Wrong context** as a starting point for access visibility
- **Wildcard** in ::set() path too wild creating too much visibility

# CONCLUSION AND REFERENCES

# Conclusion

- The **uvm_config_db** provides a consistent and flexible mechanism of storing configuration data
- Fits into hierarchical configuration paradigm
- Automatic configuration can simplify things
  - but you need to understand how it works
- Recommend: encapsulating configuration in objects
  - especially data that might change after build phase
- Recommend: **uvm_config_object** for objects and **uvm_config_int** for enums
  - helps avoid specifying wrong type causing issues with auto configuration
- There are some easy to spot gotchas
- It's not an "all or nothing" approach

# Additional Reading & References

- Accellera
  - http://www.accellera.org
- Getting Started with UVM: A Beginner's Guide, Vanessa Cooper, Verilab Publishing 2013
- Doulos UVM Guidelines:
  - http://www.doulos.com/knowhow/sysverilog/uvm/easier_uvm_guidelines
- DVCON2014: Advanced UVM Register Modelling:
  - http://www.verilab.com/files/litterick_register_final_1.pdf
- DVCON2014: Demystifying the UVM Configuration Database
  - http://www.verilab.com/files/configdb_dvcon2014.pdf
- Hierarchical Testbench Configuration Using uvm_config_db:
  - http://www.synopsys.com/Services/Documents/hierarchical-testbench-configuration-using-uvm.pdf

# Behind the Scenes of the UVM Factory

Mark Litterick, Verilab GmbH.



# Introduction

- **Factory pattern** in OOP
  - standard software paradigm
- **Implementation** in UVM
  - base-class implementation and operation
- **Usage** of factory and build configuration
  - understanding detailed usage model
- **Debugging** factory problems & gotchas
  - things the watch out for and common mistakes
- **Conclusion**
  - additional reading and references

# FACTORY PATTERN

# Software Patterns

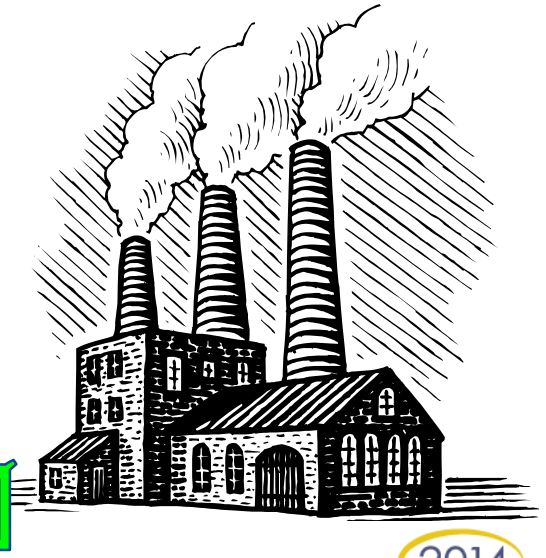
In ***software engineering***, a **design pattern** is a general **reusable solution** to a commonly occurring problem within a given context.

- ***SystemVerilog*** is an Object-Oriented Programming language
- **UVM** makes extensive use of **standard OOP patterns**
  - **Factory** - creation of objects without specifying exact type
  - **Object Pool** - sharing set of initialized objects
  - **Singleton** - ensure only one instance with global access
  - **Proxy** - provides surrogate or placeholder for another object
  - **Publisher/Subscriber** - object distribution to 0 or more targets
  - **Strategy/Policy** - implement behavioural parameter sets
  - etc...

# The Factory Pattern

The **factory method pattern** is an object-oriented creational design pattern to implement the concept of factories and deals with the problem of **creating objects without specifying the exact class** of object that will be created.

- UVM implements a version of the *factory method* pattern
- Factory method pattern overview:
  - define a seperate method for creating objects
  - subclasses override method to specify derived type
  - client receives handle to derived class
- Factory pattern enables:
  - users override class types and operation without modifying environment code
  - just *add* derived class & override line
  - original code operates on derived class without being aware of substitution



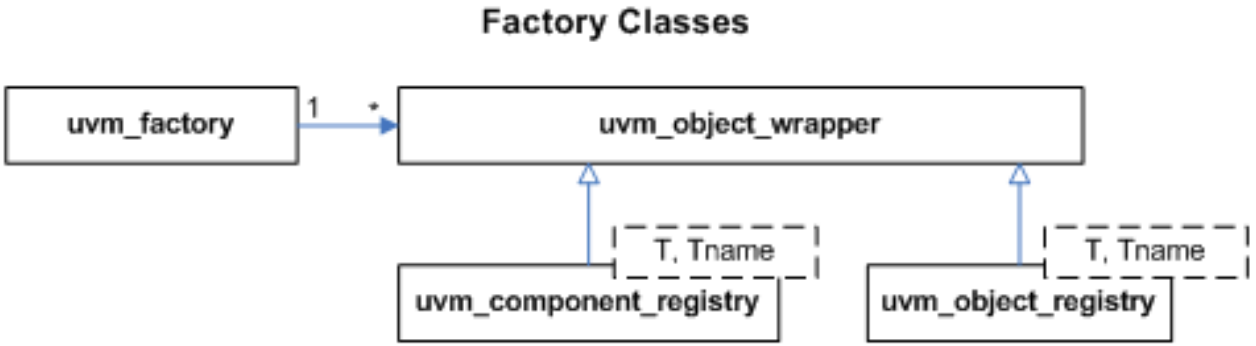
**substitute** *any* **component** or **object** in the verification environment **without modifying** a single line of **code**

# Factory Usage in UVM

- Factory is an **essential** part of **UVM**
  - ***required*** for **test** registration and operation
  - ***recommended*** for all **components**
    (env, agent, sequencer, driver, monitor, scoreboard, etc.)
  - **recommended** for all **objects**
    (config, transaction, seq_item, etc.)
  - **not appropriate** for **static interconnect**
    (TLM port, TLM FIFO, cover group, interface, etc.)
- Operates in conjunction with configuration
  - both affect **topology** and **behavior** of environment
  - **factory** responsible for inst and type overrides and **construction**
  - **configuration** responsible for **build** and **functional behavior**

# FACTORY IMPLEMENTATION

# UVM Factory Implementation




- The main **UVM files** are:
  - ***uvm_object_defines.svh***
  - ***uvm_registry.svh***
  - ***uvm_factory.svh***
- Overview:
  - user object and component **types** are **registered** via typedef
  - factory generates and stores **proxies**: **_registry#(T,Tname)*
  - **proxy** *only* knows how to **construct** the object it represents
  - factory determines **what type** to create based on configuration, then **asks** that type's **proxy** to **construct instance** for the user

a great **benefit** of **UVM** is that all **source-code** is **open-source**

# User API

- **Register** components and objects with the factory

```
`uvm_component_utils(component_type)
```

```
`uvm_object_utils(object_type)
```

do not use deprecated sequence*_utils

- Construct components and objects using **create** not *new*
  - components should be created during build phase of parent

```
component_type::type_id::create("name",this);
```

```
object_type::type_id::create("name",this);
```

- Use type-based **override** mechanisms

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

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

do not use name-based API

# `uvm_component_utils - Macro

```
`define uvm_component_utils(T) \
  ...T) \
```

```
class my_comp extends uvm_component;
  `uvm_component_utils(my_comp)
endclass
```

```
class my_comp extends uvm_component;
  typedef uvm_component_registry #(my_comp,"my_comp") type_id;
  static function type_id get_type();
    return type_id::get();
  endfunction
  ...t_wrapper get_object_type();
  endfunction
  const static string type_name = "my_comp";
  virtual function string get_type_name ();
    return type_name;
  endfunction
endclass
```

declared a typedef specialization of **uvm_component_registry** class

explains what **my_comp::type_id** is

but what about **register** and **::create ???**

# uvm_component_registry - Register

```
class uvm_component_registry
      #(type T, string Tname) extends uvm_
  typedef uvm_component_registry #(T,Tname) this_type;
  local static this_type me = get();
  static function this_type get();
    if (me == null) begin
      uvm_factory f = uvm_factory::get();
      me = new;
      f.register(me);
    end
    return me;
  endfunction
  virtu
  stati
  stati
  stati
endclas
```

**proxy** type
lightweight substitute for real object

local **static proxy** variable calls **get()**

**construct** instance of **proxy**, not actual class

**register** proxy with factory

registration is via **static initialization**
=> happens at **simulation load time**

```
function void uvm_factory::register (uvm_object_wrapper obj);
  // add to associative arrays
  m_type_names[obj.get_type_name()] = obj;
  m_types[obj] = 1;
  ...
endfunction
```

to **register** a class type, you only need a **typedef** specialization of its proxy class, using **`uvm_*_utils**

# uvm_component_registry - Create

```
comp = my_comp::type_id::create("comp",this);
```

static **create** function

create is called during **build_phase**
=> happens at **simulation run time**

```
class uvm_component_registry #(T, Tna
  ...
  static function T create(name,parent,contxt="");
    uvm_object obj;
    uvm_factory f = uvm_factory::get();
    obj = f.create_component_by_type(get(),contxt,name,parent);
    if(...t_fatal(...);
  endfu
  virtual function uvm_component create_component (name,parent);
    T obj;
    obj = new(na
    re
  endf
  ...
endcla
```

request factory **create** based on existing type overrides (if any)

return handle to **actual class** instance

**construct** instance of **actual** class

search queues for **overrides**

```
function uvm_component uvm_factory::create_component_by_type
  (type,contxt,name,parent);
  requested_type = find_override_by_type(requested_type, path);
  return requested_type.create_component(name, parent);
```

call **create_component** for **proxy** of **override** type (or original if no override)

to **enable factory** for a class you only need to register it and call **type_id::create** instead of **new**

accellera SYSTEMS INITIATIVE

2014 DESIGN AND VERIFICATION DVCON CONFERENCE AND EXHIBITION EUROPE

# Factory Overrides

*not shown*: use static ***_type::get_type()** in all cases

- Users can override original types with derived types:
  - using registry wrapper methods

```
original_type::type_id::set_type_override(override_type);
```

```
original_type::type_id::set_inst_override(override_type,...);
```

  - using component factory methods

```
set_type_override_by_type(original_type,override_type);
```

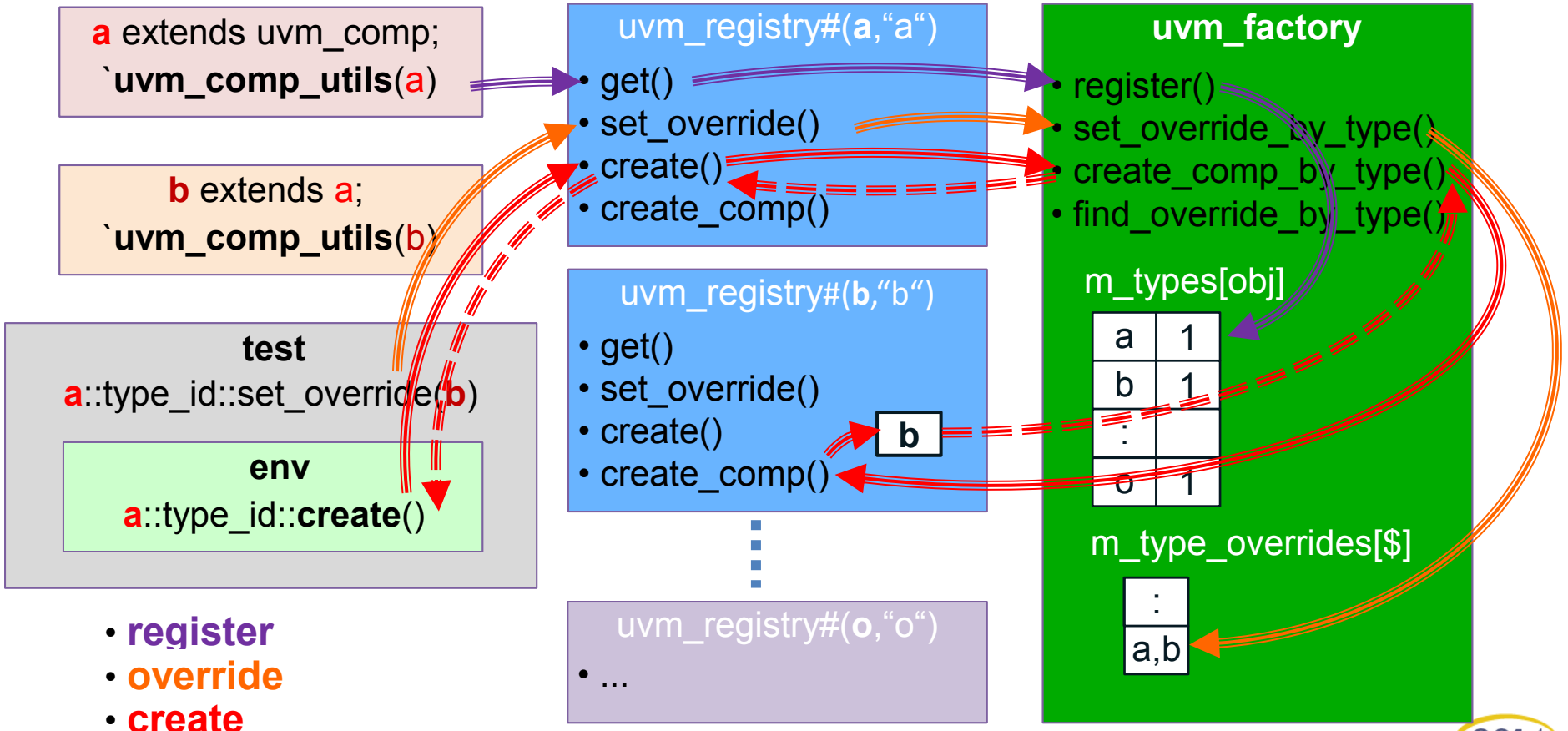
```
set_inst_override_by_type(...,original_type,override_type);
```

- Factory constructs override *descriptor* and adds to a queue:

```
function void uvm_factory::set_type_override_by_type (...);
  override = new(...);
  m_type_overrides.push_back(override);
endfunction
```

this is the queue searched by **uvm_factory::find_override_by_type**

# Factory Operation



# INTERACTION OF FACTORY & CONFIGURATION

# UVM Configuration

- ***config_db::set***, e.g. using convenience type for *uvm_object*

```
uvm_config_object::set(this,"","field",value)
```

- **build** phase for **component base-class** automatically configures all fields *registered* using *field macros*

```
function void uvm_component::build_phase(...);
  apply_config_settings(..); // search for fields & configure
endfunction
```

- **build** phase for **derived** comps must call **super.build**

```
class my_comp extends
  `uvm_component_util
    `uvm_field_int(my_field,UVM_DEFAULT)
  ...
  function void buil
    super.build_phase(..);
    // class-specific build operations like create
```

missing **field-macro** results in **no auto-config**

missing **super.build** results in **no auto-config**

# Example Environment

```
class my_comp extends uvm_component;
  `uvm_component_utils(my_comp)
endclass

class my_obj extends uvm_object;
  `uvm_object_utils(my_obj)
endclass

class my_env extends uvm_env;
  my_comp comp;
  my_obj  obj;
  `uvm_component_utils_begin(my_env)
    `uvm_field_object(obj,UVM_DEFAULT)
  `uvm_component_utils_end
  function new(..);
  function void build_phase(..);
    super.build_phase(..);
    if (obj==null) `uvm_fatal(..)
    comp = my_comp::type_id::create("comp",this);
  endfunction
endclass
```

**register** class type with **factory**

**register** field for **automation**

allow **auto-config** using **apply_config_settings**()

(example) requires ***obj*** to be in **config_db**
(there is no create/new inside this env)

use **create()** instead of **new()** for children

# Example Configure and Override

```
class test_comp extends my_comp;
  `uvm_component_utils(test_comp)
  // modify behavior
endclass

class my_test extends uvm_t
  my_env env;
  my_obj obj;
  `uvm_component_utils(my_test)
  function new(..);
  function void build_
    super.build_phase(..);
    env = my_env::type_id::create("env",this);
    obj = my_obj::type_id::create("obj",this);
    set_type_overide_by_type(
      my_comp::get_type(),
      test_comp::get_type());
    uvm_config_object::set(this,"env","obj",obj);
  endfunction
endclass
```

must be **derived** in order to **substitute**

"**class** test_comp **extends** uvm_component;" **does not work**, must be derived from my_comp

**create** using **factory** (results *only* in **new, build** comes later)

**override type** in factory prior to env::build

**configure *obj*** in db prior to env::build

**build** phase is **top-down**
lower-level ***child::build*** comes **after** ***parent::build*** completed

# Override Order

override env and comp before my_env::type_id::**create** is always **OK**

remember after **create** only **new()** has occurred, no **build yet**

```
function void my_test::build_phase(..);
  ...
  set_type_overide_by_type(my_comp,test_comp); // Good
  set_type_overide_by_type(my_env,test_env);   // Good
  env = my_env::type_id::create("env",this);
  set_type_overide_by_type(my_comp,test_comp); // Good
  set_type_overide_by_type(my_env,test_env);   // Bad
  ...
endfunction
```

override comp after my_env::type_id::**create** is **OK**
since my_comp is **not yet created**
(it is created later in my_env::build_phase)

override env after my_env::type_id::**create** is **BAD**
since my_env is **already created**
(hence override is simply ignored)

# Configure Order

**config::set** using a **null** value is an **error**
(*obj* is not yet constructed)

**config::set after obj** is created and **before env** is created is **OK**
(env create does not use the value anyway)

```
function void my_t
  ...
  uvm_config_object::set(this,"env","obj",obj); // Bad
  obj = my_obj::type_id::create("obj",this);
  uvm_config_object::set(this,"env","obj",obj); // Good
  env = my_env::type_id::create("env",this);
  uvm_config_object::set(this,"env","obj",obj); // Good
  ...
endfunction
```

**config::set after** both **obj** and **env** are created is also **OK**
(*obj* setting in *config_db* is not used until env::build phase)

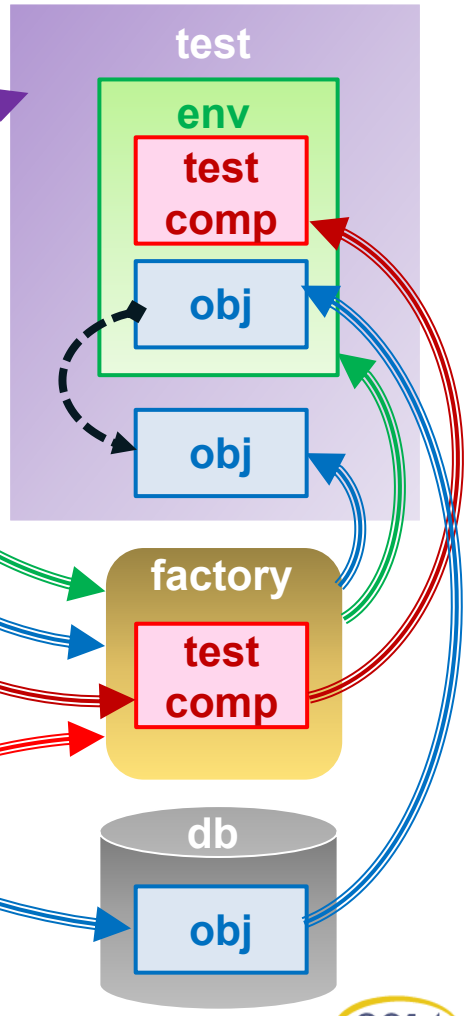
so **config*::set** can come **before or after** the **create** for corresponding component

**do not confuse create** (which tells the factory to ***new*** original or override type)
**with build** phase (which is **top-down** dynamic building of environment)

# Interaction of Factory, Config & Build

```
class my_test extends uvm_test;
  function new(..);
  function void build_phase(..);
    env = my_env::type_id::create("env",this);
    obj = my_obj::type_id::create("obj",this);
    set_type_overide_by_type(my_comp,test_comp);
    uvm_config_object::set(this,"env","obj",obj);
  endfunction
endclass
```

```
class my_env extends uvm_env;
  `uvm_field_object(obj,UVM_DEFAULT)
  function void build_phase(..);
    super.build_phase(..);
    if (obj==null) `uvm_fatal(..)
    comp = my_comp::type_id::create("comp",this);
  endfunction
endclass
```



# FACTORY PROBLEMS

# Problem Detection

- Factory and configuration problems are especially **frustrating**
  - often the code compiles and runs, because it is *legal code*
  - but ignores the user overrides and specialization
- Different kinds of problems may be **detected**:
  - at compile time (if you are lucky or careless!)
  - at run-time (usually during initial phases)
  - never...
  - ...by inspection only!

- Worse still, accuracy of report is **tool dependant**
  - although some bugs are reported by UVM base-classes

**factory** and **configuration** problems are a **special** category of bugs

# Common Factory Problems

- using **new** instead of **::type_id::create**
  - typically deep in hierarchy somewhere, and not exposed
- **deriving override** class **from** same **base** as original class
  - override class *must* derive from original class for substitution
- performing **::type_id::create** on override instead of original
  - this will limit flexibility and was probably not intended
- factory **override after** an instance of original class **created**
  - this order problem is hard to see and reports no errors
- **confusing** class **inheritance** with build **composition**
  - super has nothing to do with parent/child relationship
  - it is only related to super-class and sub-class inheritance
- **bad string** matching and **typos** when using name-based API
  - name-based factory API is not recommended, use type-based

# Debugging Factory Usage

- call ***factory.print()*** in **base-test** end_of_elaboration phase
  - prints all classes registered with factory and current overrides

```
if (uvm_report_enabled(UVM_FULL)) factory.print();
```

- call ***this.print()*** in **base-test** end_of_elaboration phase
  - prints the entire test environment topology that was actually built

```
if (uvm_report_enabled(UVM_FULL)) this.print();
```

- **temporarily** call **this.print() anywhere** during build
  - e.g. at the end of relevant suspicious new and build* functions
- use ***+UVM_CONFIG_DB_TRACE*** to debug configuration
- pay attention to the ***handle identifiers*** in tool windows
  - e.g. *component@123* or *object@456*
  - they should be identical for all references to the same thing

# CONCLUSION & REFERENCES

# Conclusion

- UVM Factory is **easy to use**
  - **simple user API** and guidelines
  - **complicated** behind the scenes
  - can be **difficult to debug**
- **Standard OOP pattern** - not invented for OVM/UVM
  - but implemented by the base class library
- Used in conjunction with configuration to control testbench
  - topology, class types, content and behavior
  - without modifying source code of environment
- You *do not need to understand* detailed internal operation
  - but **open-source UVM** code means we can see implementation ...
  - ... learn **cool stuff** that keeps us **interested** and **informed**!

# Additional Reading & References

- *UVM base-class code*
- *UVM class reference* documentation
- *"The OVM/UVM Factory & Factory Overrides: How They Work - Why They Are Important"*
  - SNUG 2012, Cliff Cummings, www.sunburst-design.com
- *"Improve Your SystemVerilog OOP Skills: By Learning Principles and Patterns"*
  - SVUG 2008, Jason Sprott, www.verilab.com
- *"Understanding the Factory and Configuration"*
  - Verification Academy, Mentor, www.mentor.com

# Questions

# UVM Stimulus and Sequences

Jonathan Bromley, Verilab Ltd

Mark Litterick, Verilab GmbH



# Introduction

- **You already know about sequencers and sequences**

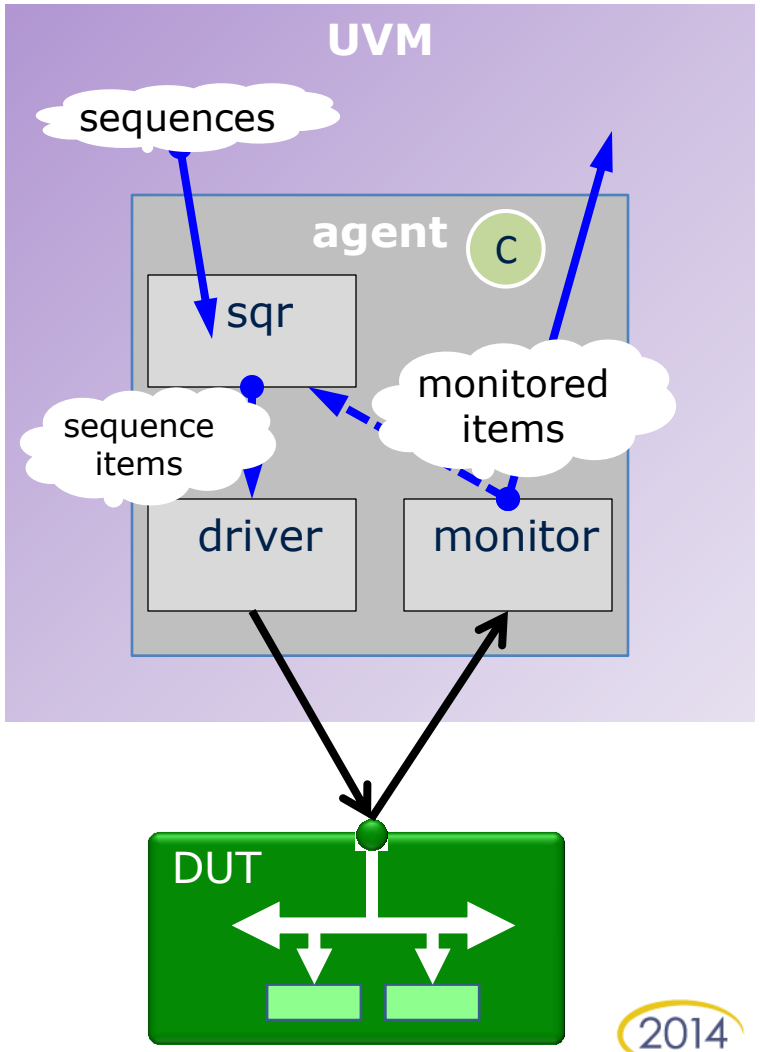
**In this session:**

- **Review of some fundamentals**
- **Structuring your environment and sequences for…**
  - **… localization of responsibilities**
  - **… flexibility for environment developers and test writers**
- **Integrating sequences with other UVM features …**
  - **… configuration, messaging, objections**

# GETTING THE BASICS RIGHT

# UVM stimulus architecture review

- Monitor+driver+sequencer = *active agent* implementing a protocol
- Stimulus driven into DUT by a *driver*
- Stimulus data sent to driver from a *sequencer*
- Run *sequences* on sequencer to create interesting activity

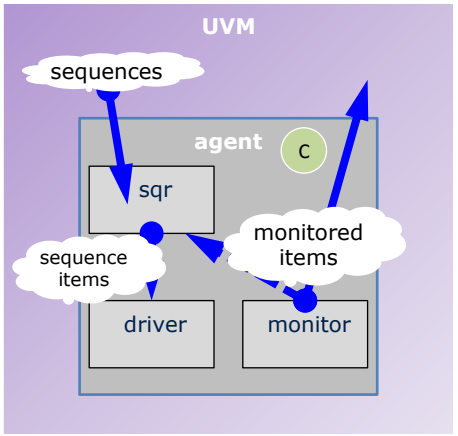


# Stimulus transaction class (item)

- **Item base class** should contain ONLY transaction data

```
class vbus_item extends uvm_sequence_item;
  rand logic [15:0] addr;
  ...
  `uvm_object_utils_begin(vbus_item)
    `uvm_field_int(addr, UVM_DEFAULT)
    ...
```

**Used by monitor**




- **Stimulus item** needs additional constraints and control knobs

```
class vbus_seq_item extends vbus_item;
  rand bit only_IO_space;
  constraint c_restrict_IO {
    only_IO_space -> (addr >= 'hFC00);
  }
...
```

**Bus protocol controls *only*!**
**Class is part of UVC**
- ***NO*** **distribution constraints**
- ***NO*** **DUT-specific strategy**

# Low-level sequences

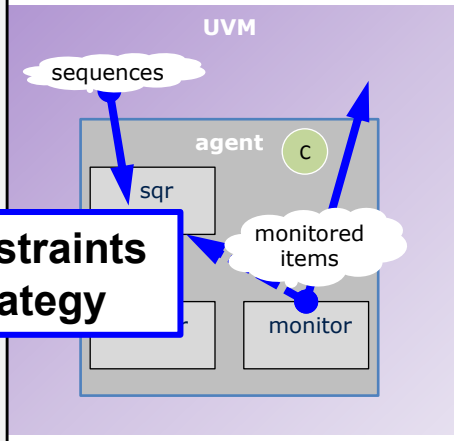
- Simple, general-purpose stream of transactions with some coordination

**Not DUT-specific! Supplied with the UVC**

```
class vbus_seq_block_wr extends vbus_sequence;
  rand bit [15:0] block_size;
  rand bit [15:0] base_addr;
  constraint c_block_align {
    block_size inside {1,2,4,8,16};
    base_addr % block_size == 0;
  }
  vbus_seq_item item;
  task body();
    for (int beat=0; beat<block_size; beat++) begin
      `uvm_do_with( item,
        {addr==base_addr+beat; dir==WR;} )
    end
  endtask
  ...
```

**Control knobs available for users**

- **NO** distribution constraints
- **NO** DUT-specific strategy




**Legal and meaningful even without any external constraint**

# UVC-provided sequence library

- Just a collection of useful sequences
- In a single sequence-library file
  - exception to usual one-class-per-file guideline

- ***NOT*** **DUT-specific!**
- **Minimal user API**
- **Run on agent sequencer**

```
/// Sequence library for vbus UVC (1): Master sequences
typedef class vbus_seq_block_wr;  ///< Write a block of locations
typedef class vbus_seq_block_rd;  ///< Read a bl
typedef class vbus_seq_rmw;       ///< Read-modi
...
class vbus_seq_lib_base extends uvm_sequence;
...
// Sequence implementations
class vbus_seq_block_wr extends vbus_seq_lib_base;
  `uvm_object_utils(vbus_seq_block_wr)
  ...
```

**Forward typedefs:**
- **provide a manifest**
- **avoid code order issues**

**Extended from seq_lib base class**

**Mainly for use by environment writers, not test writers**

**Legal and meaningful even without any external constraint**

# Naming of control knobs

- In a constraint, names resolve into **the object being randomized** - *not* into the local context!
- Creates a problem of choice of name:

```
rand bit [15:0] base_addr;
vbus_seq_item item;
...
  `uvm_do_with( item, {addr == base_addr + 3;} )
```

**only because there is no `base_addr` in `item`**

**`item.addr`**

- Use the **`local::`** qualifier

```
rand bit [15:0] addr;
...
  `uvm_do_with( item, {addr == local::addr + 3;} )
```

**SV-2009 feature - OK in all major tools**

**`item.addr`**

**See also *restricted constraint block* (has very poor tool support)**

# The story so far

**UVC should provide a built-in sequence library that...**

- provides a flexible base for customization
- does not restrict the UVC's applicability
- is already interesting for reactive slave sequences
  - predominantly random
- may be useful for simple bring-up tests
- needs a layer above to provide useful test writer API

# LAUNCHING SEQUENCES

# Launching a sequence: `uvm_do

- On same sequencer, from another sequence's body
  - good for simple sequence composition

```
class vbus_seq_block_wr ...
  rand bit [15:0] block_size;
  rand bit [15:0] base_addr;
```

```
class vbus_seq_bwr2 extends vbus_seq_lib_base;
  `uvm_object_utils(vbus_seq_bwr2)
  vbus_seq_block_wr bwr_seq;
  rand bit [15:0] first_addr;
  task body();
    bit [15:0] follow_addr;
    `uvm_do_with(bwr_seq, {base_addr == local::first_addr;})
    follow_addr = bwr_seq.base_addr + bwr_seq.block_size;
    `uvm_do_with( bwr_seq, {base_addr == local::follow_addr;} )
    ...
```

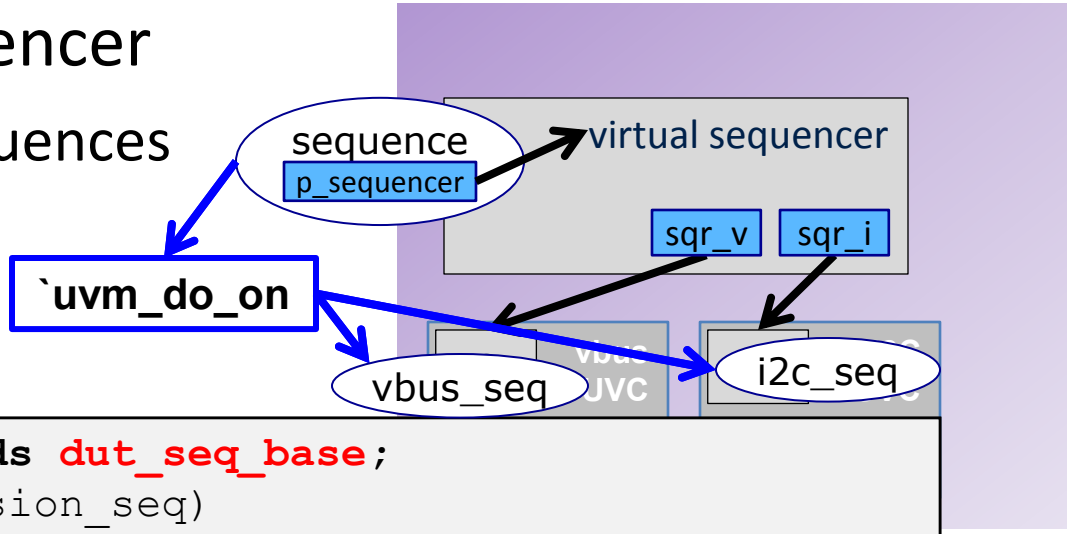
**user-API control knob**

**lower sequence runs on same sequencer**

**constraint using values picked from previous sequence's randomization**

# Launching a sequence: `uvm_do_on

- On a different sequencer
  - good for virtual sequences




```
class collision_seq extends dut_seq_base;
  `uvm_object_utils(collision_seq)
  `uvm_declare_p_sequencer(dut_sequencer)
  vbus_write_seq vbus_seq;
  i2c_write_seq  i2c_seq;
  task body();
    fork
      `uvm_do_on_with(vbus_seq, p_sequencer.sqr_v, {...})
      `uvm_do_on_with(i2c_seq, p_sequencer.sqr_i, {...})
    join
    ...
```

datatype of virtual sequencer

properties of the virtual sequencer

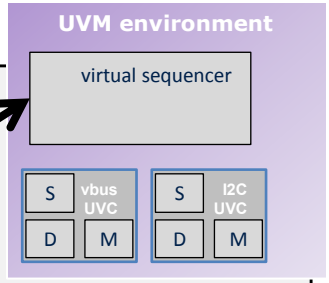
# Launching a sequence: `start`

- Can be called from any code
- Always used for top-level test sequence

```
class collision_test extends dut_test_base;
  `uvm_component_utils(collision_test)
  collision_seq test_seq;
  base_dut_env env;
  ...
  task run_phase(uvm_phase phase);
    ...
    test_seq = collision_seq::type_id::create("collision_test");
    ...
    test_seq.start(env.top_sequencer);
    ...
```

**env is built by test**

**configure/randomize the test seq**



# When does randomization occur?

- `uvm_do` macros randomize as late as possible
  - Allows randomization to be influenced by environment

**use `uvm_do` macros for any sequence that must react to DUT or TB state**

- `seq.start()` doesn't allow late randomization

**use `seq.start` only for top level sequences**

- *alternative*: explicitly call sub-methods of `start()`

```
seq.pre_start()
seq.pre_body()
parent_seq.pre_do(is_item)
seq.randomize() with...
parent_seq.mid_do(sub_seq)
  seq.body()
parent_seq.post_do(seq)
seq.post_body()
seq.post_start()
```

**not invoked by `uvm_do` macros**

# Review of UVM1.2 changes

- Default sequence of sequencer is deprecated
  - don't configure or use the `count` variable
  - don't expect a test sequence to start automatically
  - no *random* or *simple* sequences
  - no `uvm_update_sequence_lib_and_item` macro

# IMPLEMENTATION HINTS

# Exploiting the sequencer

- m_sequencer
  - reference to the sequencer we're running on
  - datatype is uvm_sequence, too generic for most uses
- p_sequencer
  - exists only if you use `uvm_declare_p_sequencer
  - has the correct data type for the sequence's chosen sequencer class **must run on a sequencer of that type**
  - allows access to members of the sequencer
    - persistent data across the life of many sequences
    - storage of configuration information, sub-sequencer references, ...

# Readback from a sequence item

- For read items, driver can populate data ...

```
class vbus_seq_item extends ...
  class vbus_item extends ...
    rand logic [15:0] addr;
    rand logic [15:0] data;
    rand bit   writeNotRead;
```

```
class vbus_driver ...
  ...
  task read(vbus_item rd_item);
    ...
    rd_item.data = vif.DATABUS;
    ...
```

- ... then sequence user can collect the data:

```
class vbus_readback_seq extends vbus_seq_base;
  vbus_seq_item item;
  logic [15:0] readback_data;
  ...
  `uvm_do_with( item, {!writeNotRead;} );
  readback_data = item.data;
```

# Readback from a sequence

- Sequence has no obvious place to store the data
- Specific provision is needed in each sequence layer

```
class vbus_block_readback_seq extends vbus_seq_base;
  vbus_readback_seq rb;
  rand int unsigned block_size;
  logic [15:0] readback_block[$];
  task body();
    for (int i=0; i<block_size; i++) begin
      `uvm_do_with( rb, {...;} );
      readback_block.push_back(rb.readback_data);
```

sequence provides non-rand storage for result

collect result data from lower-level sequence

```
class vbus_readback_seq ...
  ...
  `uvm_do_with( item, {!writeNotRead;} );
  readback_data = item.data;
```

```
class vbus_seq_item ...
  class vbus_item ...
    rand logic [15:0] addr;
    rand logic [15:0] data;
    rand bit   writeNotRead;
```

# Other readback techniques

- Collect data from the monitor
  - Requires an analysis export
  - Timing can be non-obvious
- Use sequence response item instead of request item
  - Response can be same type as request, or different
  - Harder to code and manage than using the request item
  - Easy to get into trouble with response queue
- Use UVM1.2 response handler hook
  - Automated user-specified handling of every response item
  - Custom support for out-of-order responses etc.

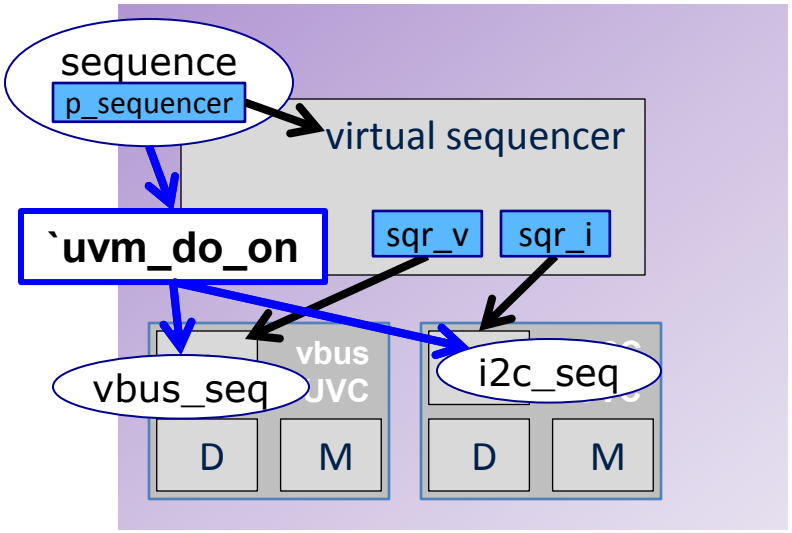
# LAYERING

# Virtual sequences and sequencers

- No sequence item type

```
class env_sqr extends uvm_sequencer;
 vbus_sqr sqr_v;
 i2c_sqr  sqr_i;
```

**set by env's connect_phase**



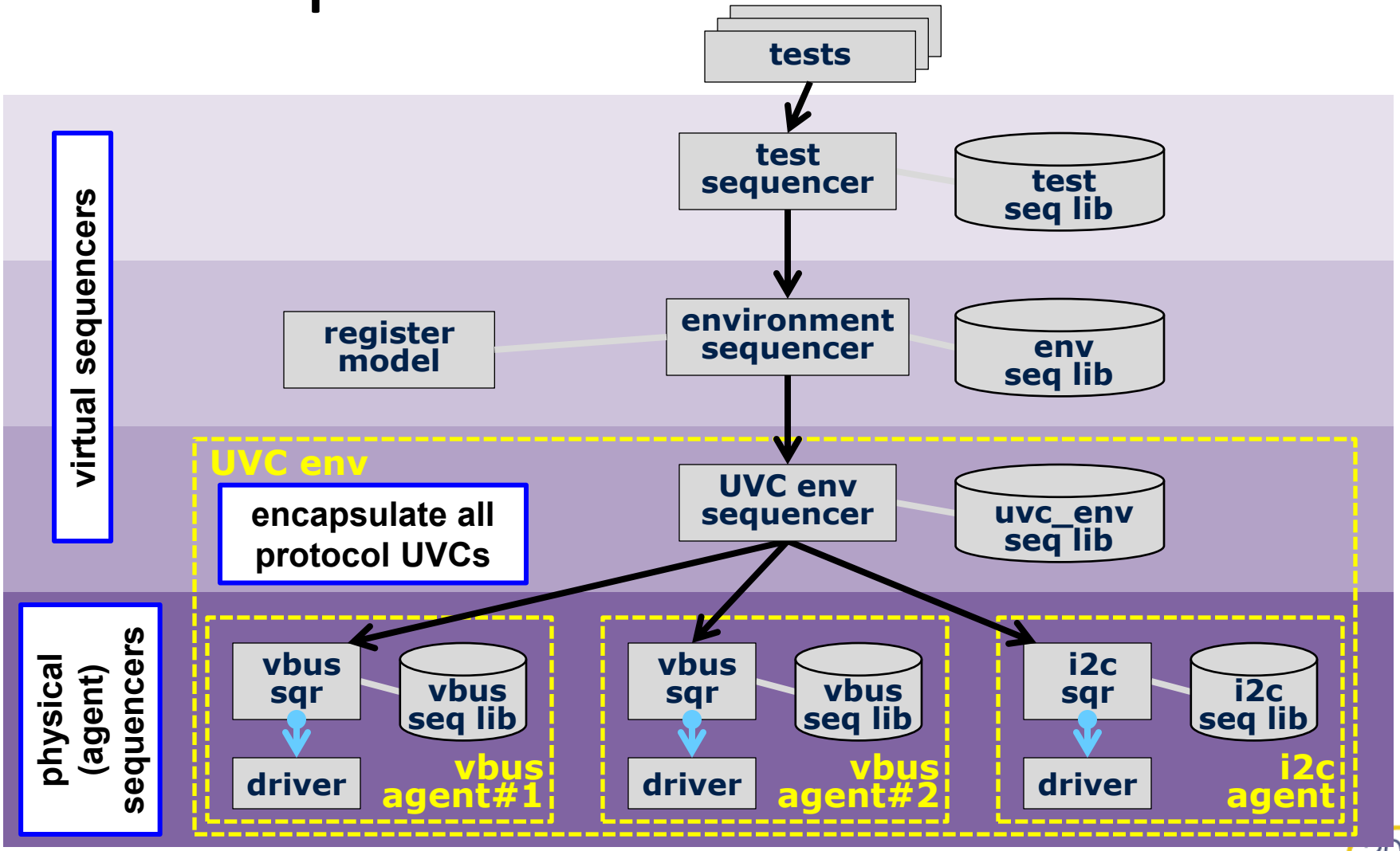

- Coordinate the work of multiple sequence(r)s

```
class vbusN_then_i2c_seq extends env_seq_base;
  rand int unsigned vbus_count;
  task body();
    repeat (vbus_count) begin
      `uvm_do_on_with(vbus_seq, p_sequencer.sqr_v, {...})
    end
    `uvm_do_on_with(i2c_seq,  p_sequencer.sqr_i, {...})
    ...
```

**properties of the virtual sequencer**

**control knobs**

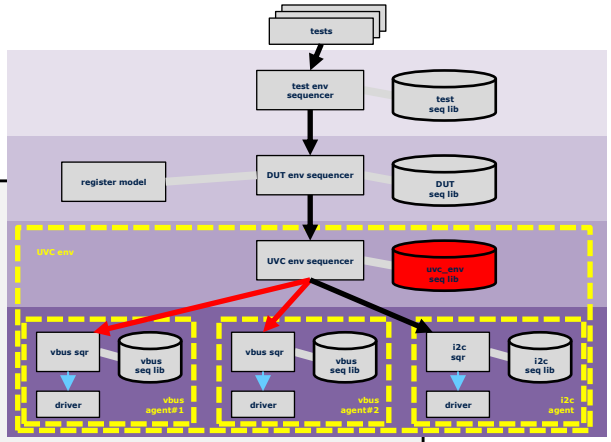
# Sequences at various levels



# UVC-environment sequences

- Coordinate actions across multiple agents
  - as required by protocol
- Example: request on one port, response on another

**Likely to be useful in higher level sequences**




```
class req1_rsp2_seq ...
  rand bit [15:0] req_adrs;
       bit [15:0] rsp_data;
  vbus_seq_item vbus_item;
  ...
    `uvm_do_on_with( vbus_item, p_sequencer.vbus1_sqr,
      {adrs==REQUEST_ADRS; data==req_adrs; writeNotRead;} )
    `uvm_do_on_with( vbus_item, p_sequencer.vbus2_sqr,
      {adrs==RESPONSE_ADRS; !writeNotRead;} )
    rsp_data = vbus_item.data;
    ...
```

**control knob**

**readback result**

# DUT-level virtual sequences

- Provide API for writer of test-level sequences
  - Setup, normal traffic, scenario building blocks
- Have detailed control knobs to customize operation
  - but must make sense if run unconstrained

```
class dut_setup_seq extends env_seq_base;
```

```
class dut_stop_seq extends env_seq_base;
```

```
class dut_training_seq extends env_seq_base;
  rand int unsigned preamble_length;
  rand bit          early_abort_error;
  rand bit          sync_loss_error;
  ...
```

# Test-level virtual sequences

- Provide primary API for test writer
  - Complete setup and traffic scenarios
  - Background irritators to run in parallel with other tests
- Access to non-protocol blocks: clock UVC, interrupts...
- Directed tests mandated by spec. or verification plan

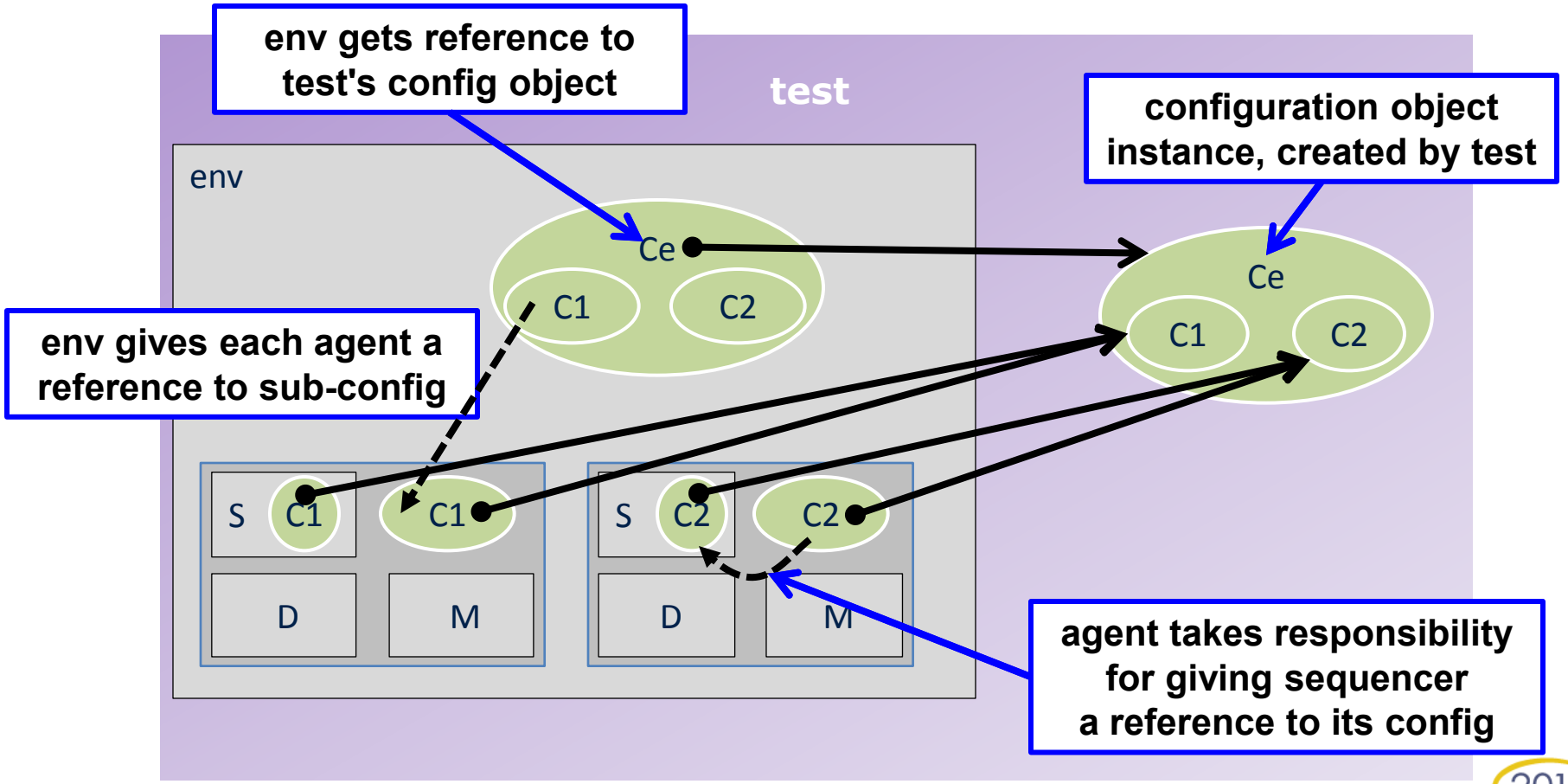
```
class clock_off_on_seq extends test_seq_base;
  rand int unsigned clock_off_cycles;
  rand bit reset_while_clock_off;
  rand bit clock_active_on_reset_release;
  ...
```

```
class dut_init_over_i2c_seq extends test_seq_base;
  rand bit reset_before_init;
  ...
```

# WORKING WITH OTHER UVM FEATURES

# Sequences and configuration

- Avoid pulling data directly from the configuration DB



# Using objections in sequences

- roughly, *don't*
- but there are some exceptions:
  - top-level test sequence
  - directed-test functionality that must complete
- automatic per-sequence objections are deprecated
  - don't use
- if possible, raise/drop *outside* the sequence
  - preserves sequence's re-usability

# Sequences and messaging

- Messaging from sequences or sequence items automatically uses their sequencer's reporter

```
class test_seq extends uvm_sequence;
  ...
  task body();
    `uvm_info("BODY", "test_seq runs", UVM_LOW)
    ...
```

don't add your own hierarchy information

```
...
test_seq ts = new();
ts.start(test_sqr);
...
```

```
UVM_INFO ../src/test_seq_reporting.sv(13) @ 0:
   uvm_test_top.test_env.test_agent.test_sqr@@ [BODY] test_seq runs
```

# Questions

# Advanced UVM Register Modeling & Performance

Mark Litterick, Verilab GmbH.



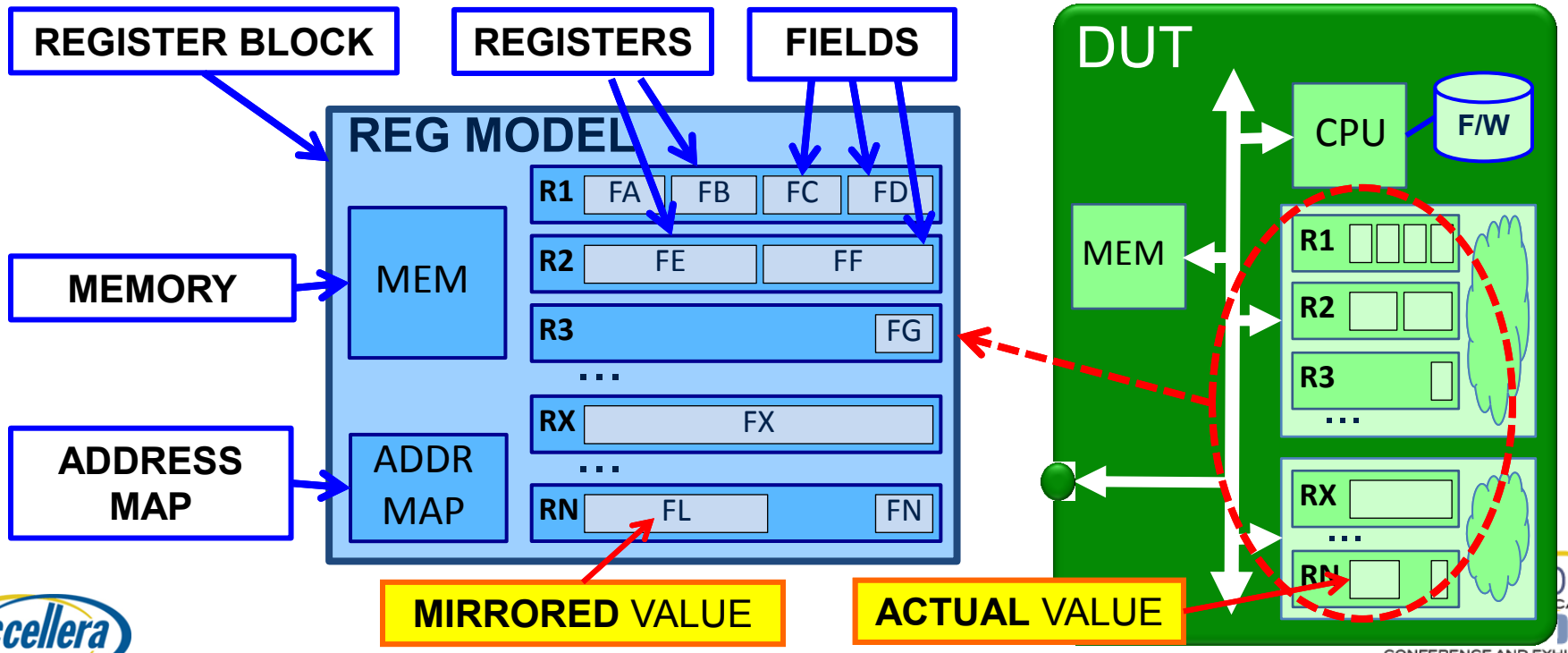
# Introduction

- UVM register model **overview**
  - structure, integration, concepts & operation
  - field modeling, access policies & interaction
  - behavior modification using hooks & callbacks
- Modeling **examples**
  - worked examples with multiple solutions illustrated
  - field access policies, field interaction, model interaction
- Register model **performance**
  - impact of factory on large register model environments

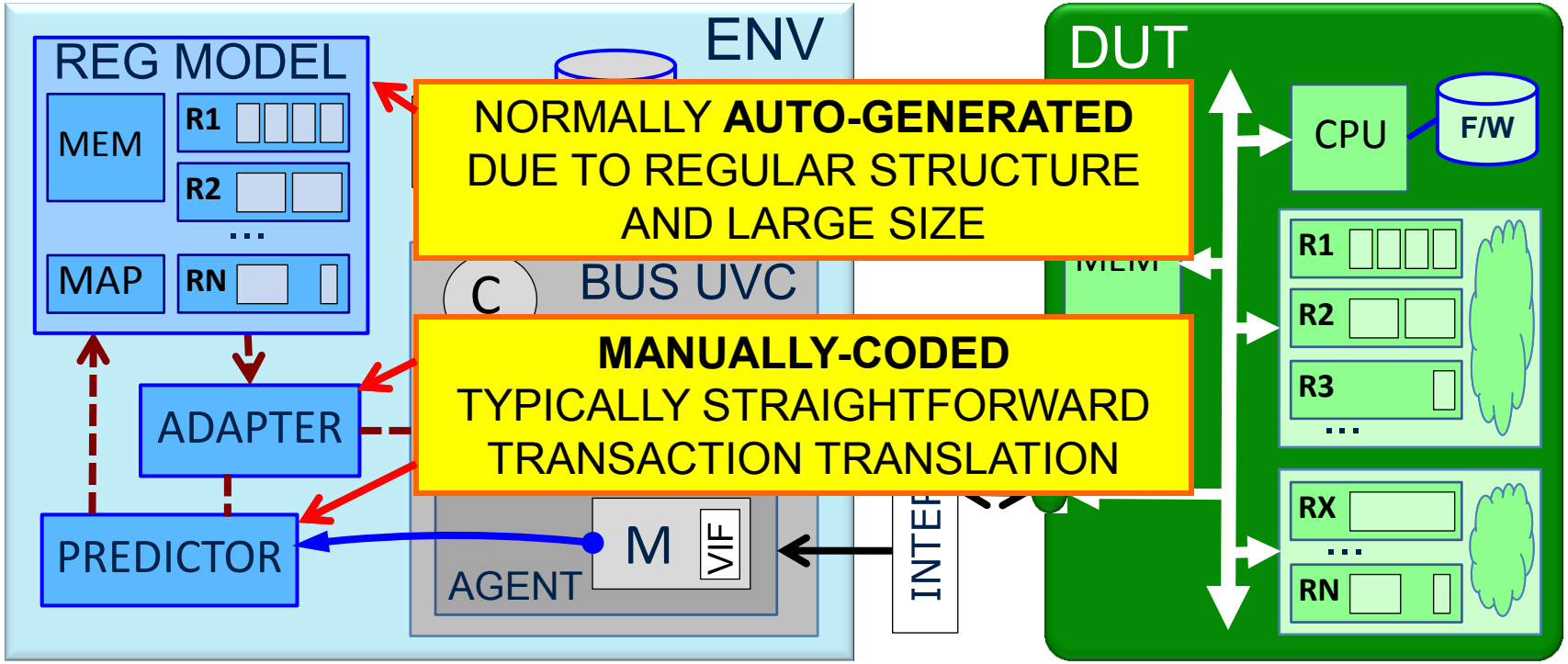
# REGISTER MODEL OVERVIEW

# Register Model Structure

- Register model (or *register abstraction layer*)
  - **models** memory-mapped **behavior of registers** in **DUT**
  - topology, organization, packing, mapping, operation, ...
  - facilitates **stimulus** generation, **checks** & **coverage**

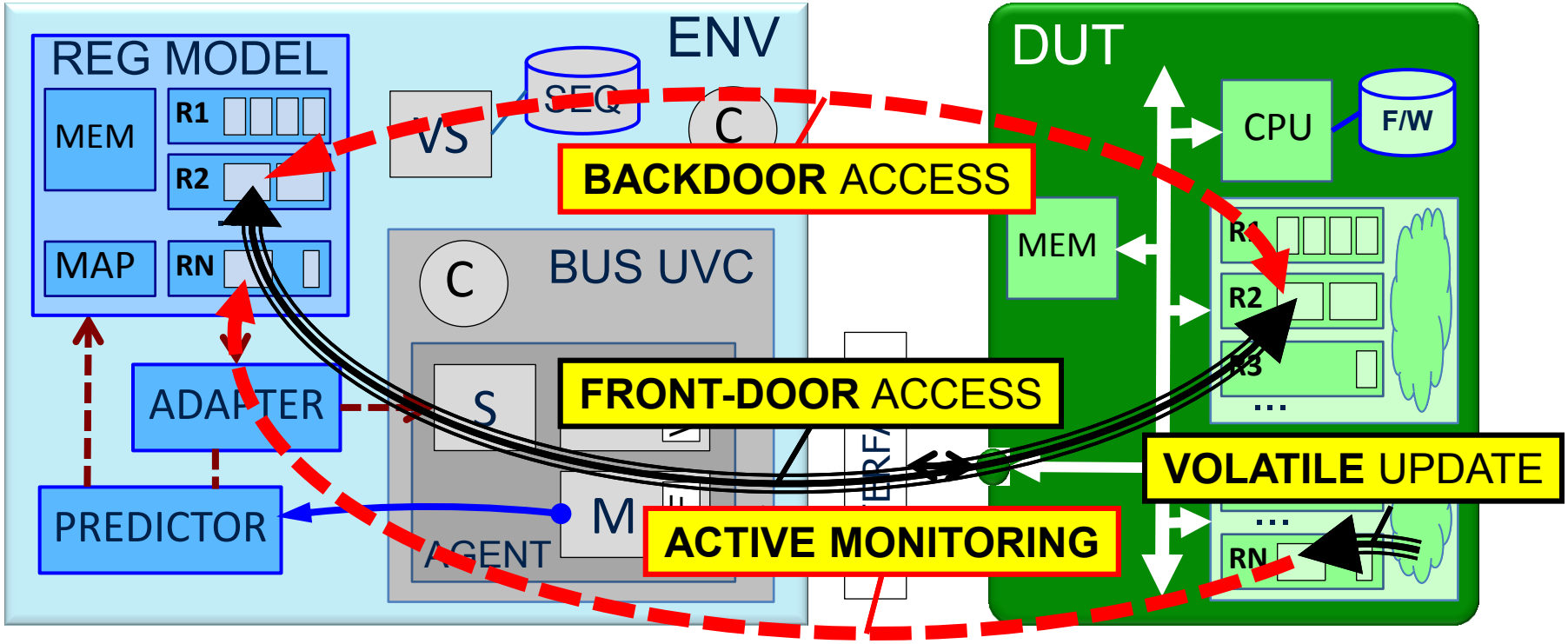


# Register Model Integration



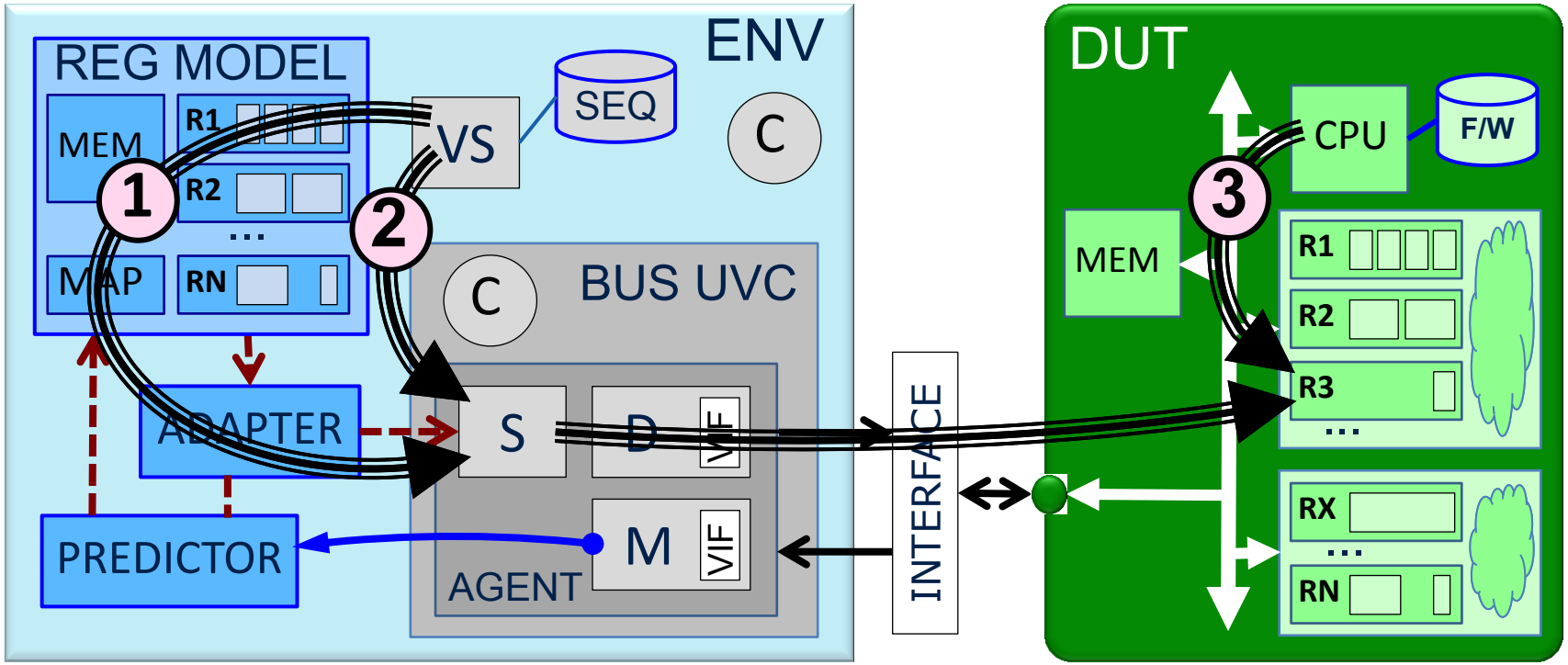

- Set of **DUT-specific** files that extend ***uvm_reg**** base
- Instantiated in *env* alongside bus interface UVCs
  - ***adapter*** converts generic *read*/*write* to bus transactions
  - ***predictor*** updates model based on observed transactions

# Register Model Concepts



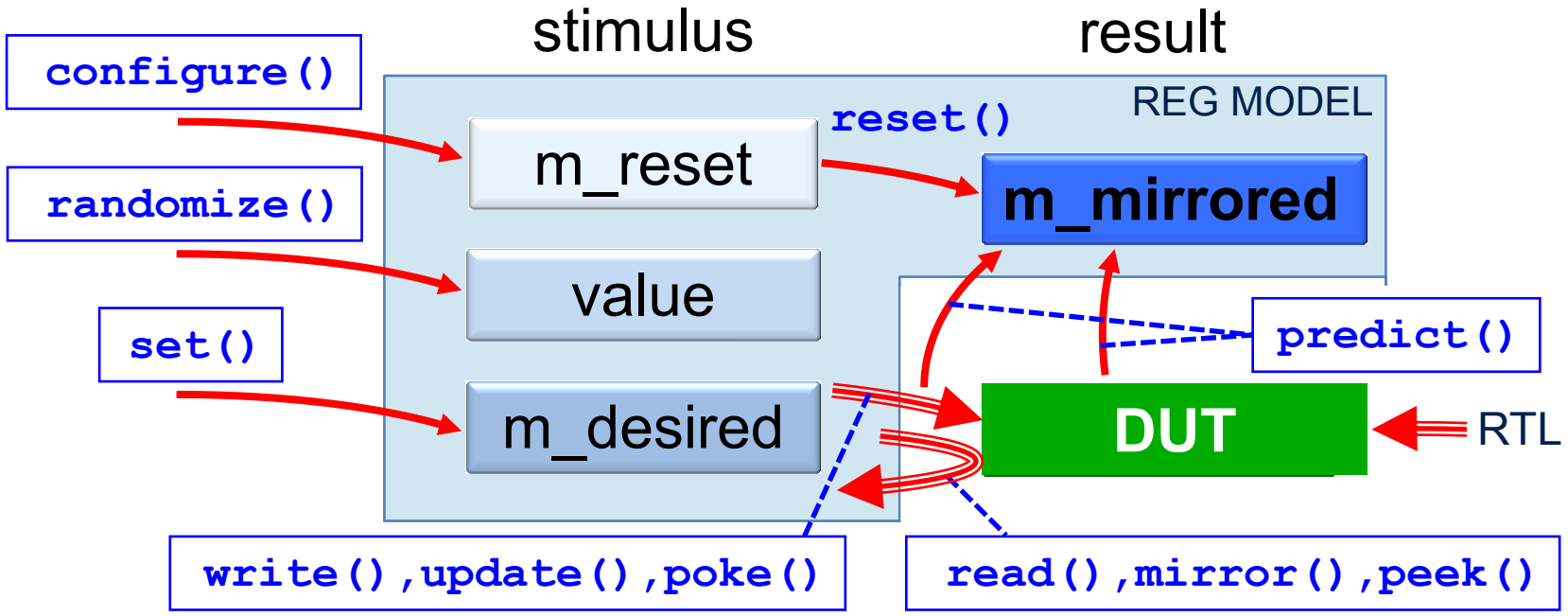

- Normal **front-door** access **via bus transaction** & **I/F**
  - sneaky **backdoor** access via ***hdl_path*** - no bus transaction
- **Volatile** fields modified by non-bus RTL functionality
  - model updated using **active monitoring** via ***hdl_path***

# Active & Passive Operation




- Model must tolerate active & passive operations:
  1. **active** model read/write generates items via adapter
  2. **passive** behavior when a sequence does not use model
  3. **passive** behavior when embedded CPU updates register

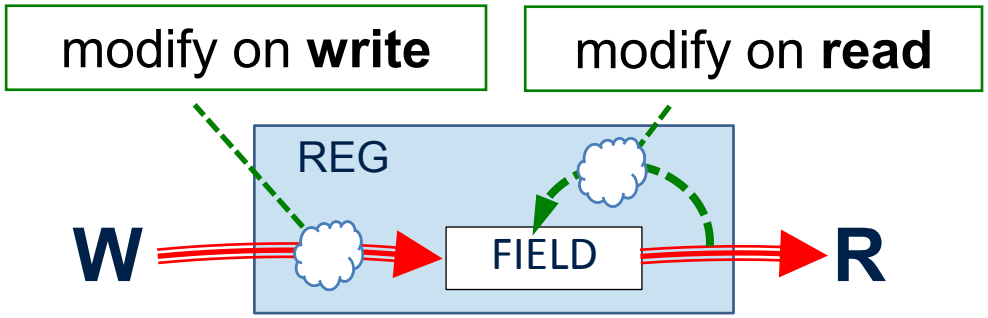
# Register Access API




- Use-case can be register- or field-centric
  - **constrained random** stimulus typically **register-centric**
    e.g. reg.randomize(); reg.update();
  - **directed** or higher-level **scenarios** typically **field-centric**
    e.g. var.randomize() with {...}; field.write(var.value);

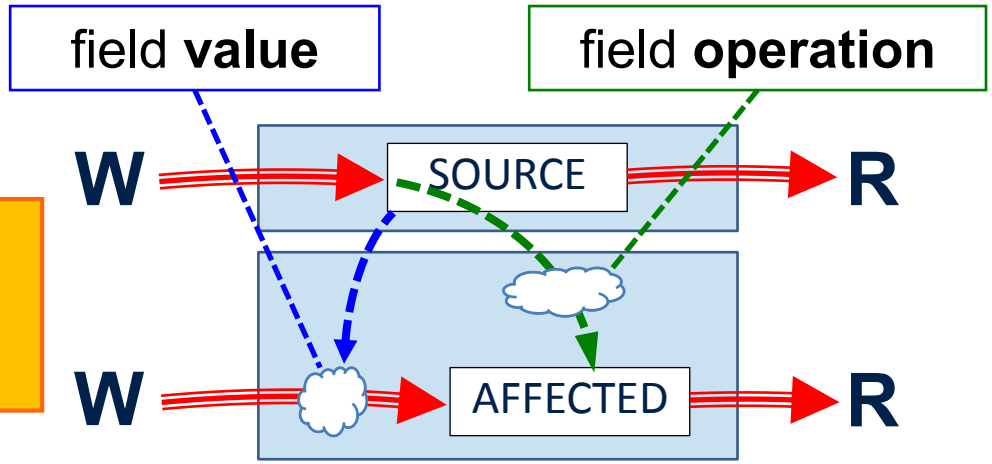
# Register Field Modeling

- Field **access policy**
  - self-contained operations on this register field




- Field **interaction**

Most **complex** modeling related to field **interaction** not field access policies




- Register **access rights** in associated memory map
- Model **behavior of DUT** to check **volatile** fields

**Difficult problem** but outside scope of register model

# Field Access Policies

- Comprehensive **pre-defined field access policies**

| | NO WRITE | WRITE VALUE | WRITE CLEAR | WRITE SET | WRITE TOGGLE | WRITE ONCE |
|---|---|---|---|---|---|---|
| NO READ | NOACCESS | WO | WOC | WOS | - | WO1 |
| READ VALUE | RO | RW | WC<br>W1C<br>W0C | WS<br>W1S<br>W0S | W1T<br>W0T | W1 |
| READ CLEAR | RC | WRC | - | WSRC<br>W1SRC<br>W0SRC | | |
| READ SET | RS | WRS | WCRS<br>W1CRS<br>W0CRS | - | | |

Just **defining** access policy is ***not enough***!

Must also **implement special behavior**!

- **User-defined field access policies** can be added

```
local static bit m = uvm_reg_field::define_access("UDAP");
```

```
if(!uvm_reg_field::define_access("UDAP")) `uvm_error(...)
```

# Hooks & Callbacks

- **Field** base class has empty **virtual method hooks**
  - **implement** in derived field to specialize behavior

```
class my_reg_field extends uvm_reg_field;
  virtual task post_write(item rw);
    //
  endta
```

***pre/post_write*** and ***pre/post_read*** are all **active** operations on model

*pre_write*
*post_write*
*pre_read*
*post_read*

- **Callback** base class has empty **virtual methods**
  - **implement** in derived callback & ***register*** it with field

```
class my_field_cb extends uvm_reg_cbs;
  function new(string name, ...);
  virtual task post_
    // specific impl
  endtask
```

most **important** callback for **passive** operation is ***post_predict***

```
my_field_cb my_cb = new("my_cb", ...);
uvm_reg_field_cb::add(regX.fieldY, my_cb);
```

*pre_write*
*post_write*
*pre_read*
*post_read*
***post_predict***
*encode*
*decode*

# Hook & Callback Execution

- Field method **hooks** are **always** executed
- **Callback** methods are **only** executed **if registered**

```
task uvm_reg_field::do_write(item rw);
  ...
  rw.local_map.do_write(rw);
  ...
  post_write(rw);
  for (uvm_reg_cbs cb=cbs.first();
       cb!=null;
       cb=cbs.next())
    cb.post_write(rw);
  ...
endtask
```

ACTUAL **WRITE**

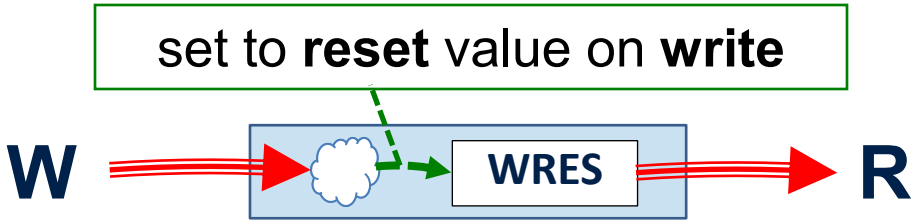
**HOOK** METHOD

**CALLBACK** METHOD
FOR ALL **REGISTERED CBS**

- **callbacks registered** with field using ***add***
- **multiple callbacks** can be **registered** with **field**
- callback **methods executed** in ***cbs*** **queue order**

# MODELING EXAMPLES

# Write-to-Reset Example




- Example **user-defined field access policy**
  - pre-defined access policies for Write-to-Clear/Set (WC,WS)
  - user-defined policy required for Write-to-Reset (WRES)

```
uvm_reg_field::define_access("WRES")
```

- Demonstrate three possible solutions:
  - ***post_write hook*** implementation in **derived field**
  - ***post_write*** implementation in **callback**
  - ***post_predict*** implementation in **callback**

# WRES Using *post_write* Hook

```
class wres_field_t extends uvm_reg_field;
 ...
 virtual task post_write(uvm_reg_item rw);
  if (!predict(rw.get_reset())) `u
```

DERIVED **FIELD**

**IMPLEMENT *post_write*** TO SET **MIRROR** TO **RESET** VALUE

**NOT PASSIVE**

```
class wres_reg_t extends uvm_reg;
 rand wres_field_t wres_field;
 ...
 function void build();
  // wres_field create()/configure(..“WRES”..)
```

USE **DERIVED** FIELD

FIELD CREATED IN **REG::BUILD**

```
class my_reg_block extends uvm_reg_block;
 rand wres_reg_t wres_reg;
 ...
 function void build();
  // wres_reg create()/configure()/build()/add_map()
```

REGISTER CREATED IN **BLOCK::BUILD**

reg/block **build()** is **not** a UVM component **build_phase()**

# WRES Using *post_write* Callback

```
class wres_field_cb extends uvm_reg_cbs;
 ...
 virtual task post_write(uvm_reg_item rw);
  if (!predict(rw.get_reset())) `u
```

DERIVED **CALLBACK**

**IMPLEMENT *post_write*** TO SET **MIRROR** TO **RESET** VALUE

**NOT PASSIVE**

```
class wres_reg_t extends uvm_reg;
 rand uvm_reg_field wres_field;
 ...
 function void build();
  // wres_field create()/configure(..“WRES”..)
```

USE **BASE** FIELD

```
class my_reg_block extends uvm_reg_blo
 rand wres_reg_t wres_reg;
 ...
 function void build();
  // wres_reg create()/configure()/build()/add_map()
  wres_field_cb wres_cb = new("wres_cb");
  uvm_reg_field_cb::add(wres_reg.wres_field, wres_cb);
```

**CONSTRUCT CALLBACK**

**REGISTER CALLBACK** WITH REQUIRED **FIELD**

# WRES Using *post_predict* Callback

```
class wres_field_cb extends
 ...
 virtual function void post_predict(..,fld,value,..);
  if(kind==UVM_PREDICT_WRITE) value = fld.get_reset();
```

**IMPLEMENT *post_predict* TO SET MIRROR VALUE TO RESET STATE**

**PASSIVE OPERATION**

```
class wres_reg_t exte
 rand uvm_reg_field w
 ...
 function void build(
  // wres_field create
```

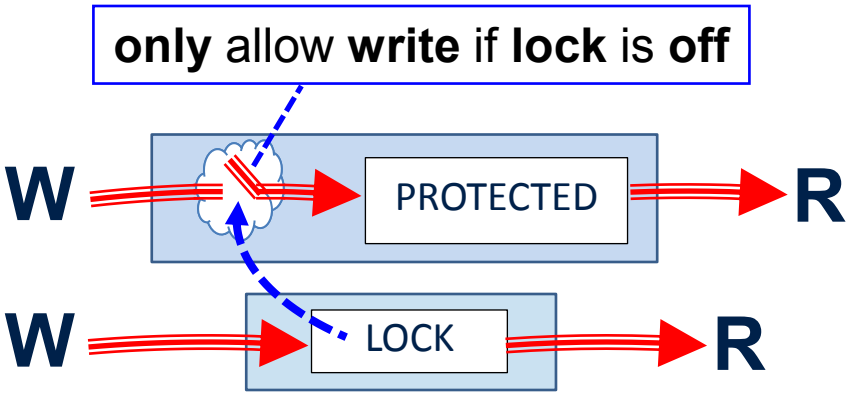
```
virtual function void post_predict(
  input  uvm_reg_field    fld,
  input  uvm_reg_data_t   previous,
  inout  uvm_reg_data_t   value,
  input  uvm_predict_e    kind,
  input  uvm_path_e       path,
  input  uvm_reg_map      map
```

```
class my_reg_bloc
 rand wres_reg_t
 ...
 function void bu
  // wres_reg create()/configure()/build()/add_map()
  wres_field_cb wres_cb = new("wres_cb");
  uvm_reg_field_cb::add(wres_reg.wres_field, wres_cb);
```

***post_predict* is only available for *fields* *not* registers**

if we **use** this **callback with** a **register** we get **silent non-operation**!

# Lock/Protect Example




- Example **register field interaction**
  - **protected** field behavior based on **state** of **lock** field, or
  - **lock** field **operation** modifies behavior of **protected** field
- Demonstrate two possible solutions:
  - ***post_predict*** implementation in **callback**
  - **dynamic field access policy** controlled by **callback**
  - (not **bad** *pre_write* implementation from *UVM UG*)

# Lock Using *post_predict* Callback

```
class prot_field_cb extends uvm_reg_cbs;
 local uvm_reg_field lock_field;
 function new (string name, uvm_reg_field lock);
  super.new (name);
  this.lock_field = lock;
 endfunction
 virtual function void post_predict(..previous,value);
  if (kind == UVM_PREDICT_WRITE)
   if (lock_field.get())
    value = previous;
 endfunction
```

**HANDLE** TO **LOCK** FIELD

**ADD** TO **NEW()** SIGNATURE

**REVERT** TO **PREVIOUS VALUE** IF **LOCK ACTIVE**

**CONNECT LOCK** FIELD

```
class my_reg_block extends uvm_reg_block;
 prot_field_cb prot_cb = new("prot_cb", lock_field);
 uvm_reg_field_cb::add(prot_field, prot_cb);
```

**REGISTER CALLBACK** WITH **PROTECTED** FIELD

# Lock Using Dynamic Access Policy

```
class lock_field_cb extends uvm_reg_cbs;
 local uvm_reg_field prot_field;
 function new (string name, uvm_reg_field prot);
  super.new (name);
  this.prot_field = prot;
 endfunction
 virtual function void post_predict(
  if (kind == UVM_PREDICT_WRITE)
   if (value)
    void'(prot_field.set_access("RO"));
   else
    void'(prot_field.set_access("RW"
 en
```

**HANDLE** TO **PROTECTED** FIELD

SET **ACCESS POLICY** FOR **PROTECTED** FIELD BASED ON **LOCK OPERATION**

***prot_field.get_access()*** RETURNS CURRENT POLICY

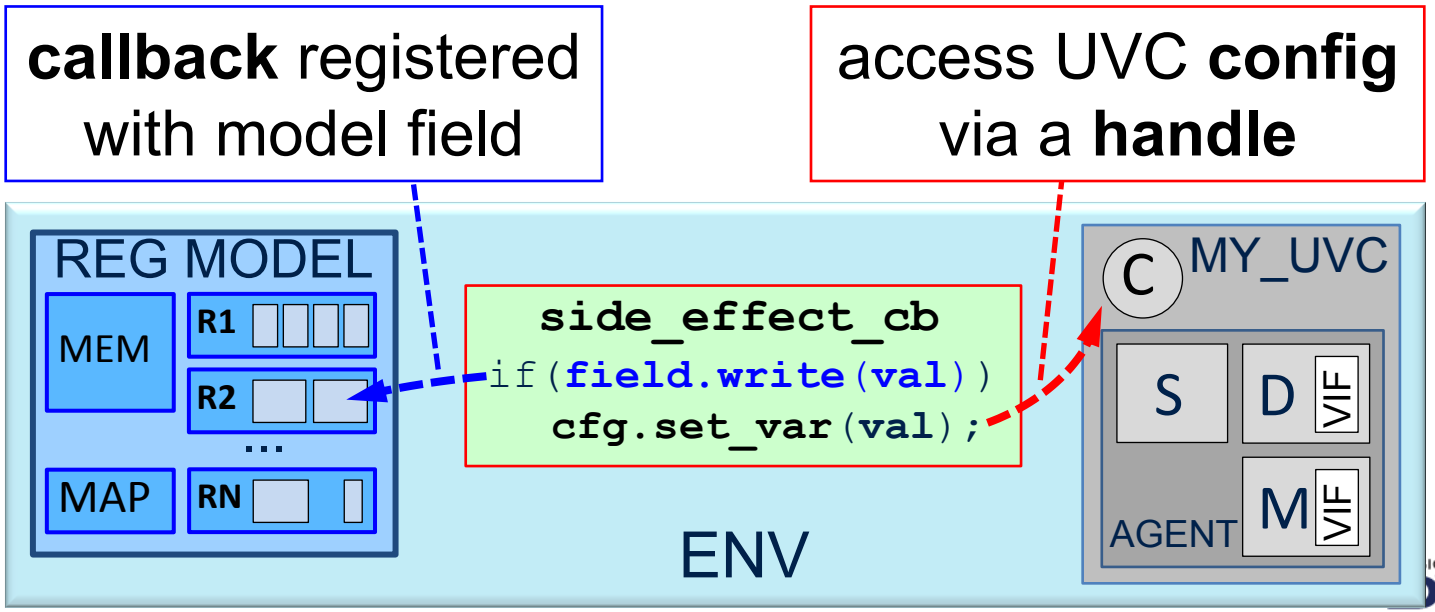
**REGISTER CALLBACK** WITH **LOCK** FIELD

**CONNECT PROTECTED** FIELD

```
class my_reg_block extends uvm_reg_block;
 lock_field_cb lock_cb = new("lock_cb", prot_field);
 uvm_reg_field_cb::add(lock_field, lock_cb);
```

# Register Side-Effects Example

- Randomize or modify registers & reconfigure DUT
  - what about **UVC configuration?**
    - update from **register sequences** — not passive
    - **snoop** on DUT bus transactions — not backdoor
    - implement ***post_predict*** **callback** — passive & backdoor



# Config Update Using Callback

```
class reg_cfg_cb extends uvm_reg_cbs;
 my_config cfg;

 function new (string name, my_config cfg);
  super.new (name);
  this.cfg = cfg;
 endfunction

 virtual function void post_predict(.
  if (kind == UVM_PREDICT_WRITE)
   cfg.set_var(my_enum_t'(value));
 endfunction
```

HANDLE TO **CONFIG** OBJECT

SET **CONFIG** ON **WRITE** TO REGISTER FIELD (TRANSLATE IF REQUIRED)

```
class my_env extends uvm_env;
 ...
 uvc = my_uvc::type_id::create(...);
 reg_model = my_reg_block::type_id::create(...);
 ...
 reg_cfg_cb cfg_cb = new("cfg_cb", uvc.cfg);
 uvm_reg_field_cb::add(reg_model.reg.field, cfg_cb);
```

ENVIRONMENT HAS **UVC** & **REG_MODEL**

**CONNECT CONFIG**

**REGISTER CALLBACK**

# REGISTER MODEL PERFORMANCE

# Performance

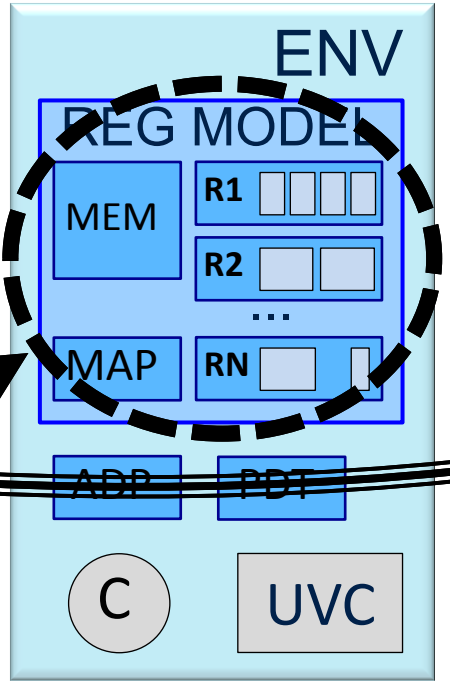
- Big register models have **performance impact**
  - full SoC can have >10k fields
- Register model & RTL typically auto-**generated**
  - **made-to-measure** for each derivative

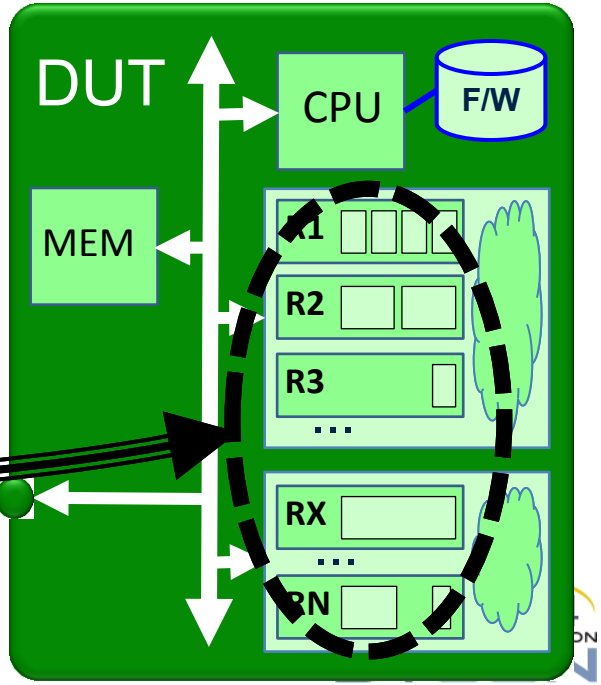
**MANY REGISTER CLASSES**
(MORE THAN REST OF ENV)

**DIFFERENT USE-CASE THAN FACTORY**

**REGISTER DESCRIPTION**
(TEXT, **XML**, YAML, etc.)

**GENERATOR** TOOL/SCRIPTS



# Life Without The Factory

- Example **SoC** with **14k+ fields** in **7k registers**
  - many **register classes** (most fields are base type)
  - **not using factory** overrides – **generated** on demand

| MODE | FACTORY TYPES | COMPILE TIME | LOAD TIME | BUILD TIME | DISK USAGE |
|---|---|---|---|---|---|
| NO REGISTER MODEL | 598 | 23 | 9 | 1 | 280M |
| +REGISTERS USING FACTORY | 8563 | 141 | 95 | 13 | 702M |
| +REGISTERS NO FACTORY | 784 | 71 | 17 | 1 | 398M |

**COMPILE TIME x2**
**+1 min infrequently**

**LOAD + BUILD TIME x5**
**+1.5 min for every sim**

- Register model still **works without** the **factory**
  - **do not use *uvm_object_utils*** macro for fields & registers
  - **construct** registers using ***new*** instead of ***type_id::create***

# `uvm_object_utils

```
`define uvm_object_utils(T) \
  `m_uvm_object_registry_internal(T,T) \
```

```
class my_reg extends uvm_reg;
  `uvm_object_utils(my_reg)
endclass
```

```
`define m_uvm_object_registry_internal(T,S) \
```

```
class my_reg extends uvm_reg;
  typedef uvm_object_registry #(my_reg,"my_reg") type_id;
  static function type_id get_type();
    return type_id::get();
  endfunction
  t_wrapper get_object_type();
  endfunction
  function uvm_object create (strin
  const static string type_name = "my_reg";
  virtual function string get_type_name ();
    return type_name;
```

declare a **typedef** specialization of **uvm_object_registry** class

explains what **my_reg::type_id** is

but what about **factory registration** and **type_id::create ???**

declare some **methods** for factory API

# Load Time Penalty

```
class uvm_object_registry
      #(type T, string Tname) extends uvm_
  typedef uvm_object_registry #(T,Tname) this_type;
  local static this_type me = get();
  static function this_type get();
    if (me == null) begin
      uvm_factory f = uvm_factory::get();
      me = new;
      f.register(me);
    end
    return me;
  endfunction
  virtu
  stati
  stat
  stat
endcla
```

**proxy** type
lightweight substitute for real object

local **static proxy** variable calls **get()**

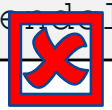
**construct** instance of proxy, not real class

**register** proxy with factory

registration is via **static initialization**
=> happens at **simulation load time**

```
function void uvm_factory::register (uvm_object_wrapper obj);
  ...
  // add to associative arrays
```

- **thousands of registers** means **thousands of proxy classes** are **constructed** and **added** to factory when files **loaded**
- **do not need** these classes for **register generator** use-case!



# Build Time Penalty

```
reg= my_reg::type_id::create("reg",,get_full_name());
```

```
class uvm_object_registry #(T, Tname) extends uvm_object_wrapper;
  ...
  static function T create(name,parent,contxt="");
    uvm_object obj;
    uvm_factory f = uvm_factory::get();
    obj = f.create_object_by_type(get(),contxt,name,parent);
    if (!$cast(create, obj)) uvm_report_fatal(...);
  endfunc
  virtual function uvm_object create_object (name,parent);
    T obj;
    obj = new(name, parent);
    re
  end
  ...
endc
```

```
uvm_factory::create_object_by_type
(type,contxt,name,parent);
```

request factory **create** based on existing type overrides (if any)

return **handle** to object

search queues for **overrides**

constructs **actual** object

- **create** and factory **search** takes time for **thousands of registers** during the **build_phase** for the environment (**build time**)
- **no need** to search for overrides for **register generator** use-case!

# Conclusions

- Register models are **complicated**!
  - consider: passive operation, backdoor access, use-cases,...
  - this problem is *not* unique to *uvm_reg*
- Multiple possible **modeling solutions**...
  - ... but some are better than others!
  - effort for developers & generators (but easy for users)
- Full-chip SoC register model **performance impact**
  - for generated models we can avoid using the factory
- All solutions evaluated in **UVM-1.2** & **OVM-2.1.2**
  - updated *uvm_reg_pkg* that includes UVM bug fixes (available from www.verilab.com )

# Additional Reading & References

- *UVM base-class code*
- *UVM class reference* documentation
- *"Advanced UVM Register Modeling : There's More Than One Way To Skin A Reg"*
  - DVCon 2014, Litterick & Harnisch, www.verilab.com (includes additional examples like triggered writes)

# Questions