Advanced UVM in the real world
- Tutorial -

Mark Litterick
Jason Sprott
Jonathan Bromley
(Vanessa Cooper)
INTRODUCTION
What is UVM?

Verification Environment

Universal Verification Methodology (UVM)

SystemVerilog IEEE1800

• 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

© Verilab & Accellera
Key Elements of UVM

- **SystemVerilog Language**
  - syntax
  - RTL
  - OOP
  - class
  - interface
  - etc...

- **Verification Concepts**
  - constrained-random
  - coverage-driven
  - transaction-level
  - sequences
  - scoreboards

- **UVM Methodology**
  - base-classes
  - use-cases
  - configuration-db
  - factory operation
  - phases
  - etc...
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!

Design

RTL
blocks
modules
vectors
assignments
arrays
etc.

Verification

signals
interfaces
clocking-block
scheduling
functions
tasks
etc.

OOP
class
random
constraints
coverage
queues
etc.
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!

Verification Concepts
• architecture (env-agent-sqr/drv/mon)
• random configuration & build
• constrained-random sequence stimulus
• scoreboard & protocol checkers
• functional coverage collection & analysis
• transaction-level modeling & TLM ports
• messaging & simulation debug
• verification planning & closure
  • etc...
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!
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.
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?

- Physical IFs
- Invasive
- Structural
- Hierarchical
- DUT Specific
- Functional
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

```plaintext
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

```plaintext
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

- 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

```java
class uvm_config_db#(type T=int) extends uvm_resource_db#(T)
```
Configuration Information Database

Create or Update

Query

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`

```plaintext
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)

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

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

```plaintext
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

```c++
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

```ucf
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

```verbatim
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

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

// wait until someone changes value of entry
```c
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

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

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

```plaintext
class my_comp extends uvm_component;
  `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)
  `uvm_component_utils_end

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

- `UVM_READONLY` results in no auto-config
- `missing field-macro` results in no auto-config
- `missing super.build` results in no auto-config

Only called **once** at build time
## Automatic Configuration & Objects

<table>
<thead>
<tr>
<th>::set() type</th>
<th>::get() type</th>
<th>auto config</th>
<th>explicit ::get()</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>uvm_object</td>
<td>uvm_object</td>
<td>✓</td>
<td>✓</td>
<td>$cast to concrete type required for explicit get()</td>
</tr>
<tr>
<td>uvm_object</td>
<td>my_config</td>
<td>✓</td>
<td>✗</td>
<td>Wrong</td>
</tr>
<tr>
<td>my_config</td>
<td>my_config</td>
<td>✗</td>
<td>✓</td>
<td>Breaks auto-config 😞</td>
</tr>
<tr>
<td>my_config</td>
<td>uvm_object</td>
<td>✗</td>
<td>✗</td>
<td>Wrong</td>
</tr>
</tbody>
</table>

Recommend using `uvm_config_object` typedef.
Explicit get() needs a cast

For objects

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

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

(A) single

test1

env

agent1

agent2

(B) multiple

test1

env

agent1

agent2

(C) global

test1

env

agent1

agent2

```
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

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

Not advisable unless components are tightly coupled
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

```cpp
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);

m_env_cfg

vars

objects

uvm_config_db

entry

@ref

@ref

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

© Verilab & Accellera
Comparing Styles

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

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

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

test handle to an instance of another config object
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

makes embedded env1 config visible to env1 (doesn't set any variables)
populated by auto config from set() done up the hierarchy

done even before env1 created
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

© Verilab & Accellera
Debug: check get() return value

• Defensive programming and informative messages

```cpp
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")
end
```

• 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
• Doulos UVM Guidelines:
• DVCON2014: Advanced UVM Register Modelling:
• DVCON2014: Demystifying the UVM Configuration Database
• Hierarchical Testbench Configuration Using uvm_config_db:
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 separate 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**
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
User API

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

  ```
  `uvm_component_utils(component_type)
  `uvm_object_utils(object_type)
  ```

- 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 deprecated** `sequence*_utils`

  **Do not use** name-based API
\`uvm_component_utils - Macro

\`
\define uvm_component_utils(T) \\
\protect \m_{uvm_component_registry\_internal}(T,T) \\
\protect \m_{uvm_get\_type\_name\_func}(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

const static string type\_name = "my\_comp";
virtual function string get\_type\_name();
return type\_name;
endfunction
endclass

explained a typedef specialization of uvm\_component\_registry class

but what about register and ::create ???

© Verilab & Accellera
To register a class type, you only need a `typedef` specialization of its proxy class, using `\`uvm_*_util\`.
uvm_component_registry - Create

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

**static create function**

```
class uvm_component_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_component_by_type(get(), contxt, name, parent);
    if (!$cast(create, obj)) uvm_report_fatal(...);
  endfunction

  virtual function uvm_component create_component(name, parent);
    T obj;
    obj = new(name, parent);
    return obj;
  endfunction

endclass
```

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

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

```
return handle to actual class instance
```

```
construct instance of actual class
```

```
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**
Factory Overrides

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

```c++
original_type::type_id::set_type_override(override_type);
original_type::type_id::set_inst_override(override_type,...);
```

  – using component factory methods

```c++
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:

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

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

this is the queue searched by uvm_factory::find_override_by_type
Factory Operation

- `a` extends `uvm_comp`
  - `uvm_comp_utils(a)`

- `b` extends `uvm_comp`
  - `a` extends `uvm_comp`
  - `b` extends `uvm_comp`
  - `uvm_comp_utils(b)`

- `test`
  - `a::type_id::set_override(b)`

- `env`
  - `a::type_id::create()`

- `uvm_registry#(`
  - `a,"a"`)

- `uvm_registry#(`
  - `b,"b"`)

- `uvm_factory`

- `m_types[objc]`
  - `a 1`
  - `b 1`
  - `o 1`

- `m_type_overrides[$]`
  - `a,b`
UVM Configuration

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

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

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

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

• **build phase for derived comps** must call `super.build`

```cpp
class my_comp extends uvm_component;
    `uvm_component_utils_begin
        `uvm_field_int(my_field,UVM_DEFAULT)
    `uvm_component_utils_end

    function void build_phase(...);
        super.build_phase(..);
        // class-specific build operations like create
endfunction
```

© Verilab & Accellera
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

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

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

- **Build Phase is Top-Down**: Lower-level `child::build` comes after `parent::build` completed.
- **Configure `obj`** in db prior to `env::build`.
- **Override Type** in factory prior to `env::build`.
- **Create Using Factory** (results only in `new`, `build` comes later).
- **Must be Derived** in order to substitute.
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**

```cpp
function void my_test::build_phase(..);
...
set_type_override_by_type(my_comp, test_comp);  // Good
set_type_override_by_type(my_env, test_env);    // Good
env = my_env::type_id::create("env", this);
set_type_override_by_type(my_comp, test_comp);  // Good
set_type_override_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_test::build_phase( .. );

...

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_override_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

```cpp
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

```cpp
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](http://www.sunburst-design.com)

• “Improve Your SystemVerilog OOP Skills: By Learning Principles and Patterns”

• “Understanding the Factory and Configuration”
  – Verification Academy, Mentor, [www.mentor.com](http://www.mentor.com)
Questions
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

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

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

```verilog
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

```verilog
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
...
```

- Not DUT-specific! Supplied with the UVC
- 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

```cpp
// 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 block of locations
typedef class vbus_seq_rmw;       ///< Read-modify-write one location...
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;} )
```

```
item.addr
```

• Use the local:: qualifier

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

```
item.addr
```

only because there is no base_addr in item

SV-2009 feature - OK in all major tools

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
      ...
```
Launching a sequence: \texttt{start}

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

```verilog
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);
    ...
```

\textbf{UVM environment}

\begin{itemize}
    \item virtual sequencer
    \item S
    \item bus
    \item UVC
    \item D
    \item M
    \item I2C
    \item UVC
\end{itemize}

\textbf{configure/randomize the test seq}

\textbf{env is built by test}
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
  – 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 ...

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

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

```verilog
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

```plaintext
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);
        end

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;
```

sequence provides non-rand storage for result
collect result data from lower-level sequence
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, {...})
...
```

- Control knobs

- Properties of the virtual sequencer

- Sequence p_sequencer

- Virtual sequencer

- VBUS UVC S

- SD M

- Vbus

- I2C

- VBUS UVC S

- SD M

- Connect phase

- Coordinate the work of multiple sequence(r)s
Sequences at various levels

virtual sequencers

register model

environment sequencer

test sequencer

tests

UVC env

encapsulate all protocol UVCs

physical (agent) sequencers

UVC env sequencer

uvc_env seq lib

vbus

driver

vbus sqr

vbus seq lib

vbus agent#1

vbus

driver

vbus sqr

vbus seq lib

vbus agent#2

i2c

driver

i2c sqr

i2c seq lib

i2c agent

© Verilab & Accellera

99
UVC-environment sequences

• Coordinate actions across multiple agents
  – as required by protocol

• Example: request on one port, response on another

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;
...```
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

env gets reference to test's config object

env gives each agent a reference to sub-config

configuration object instance, created by test

agent takes responsibility for giving sequencer a reference to its config
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 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**

<table>
<thead>
<tr>
<th></th>
<th>NO WRITE</th>
<th>WRITE VALUE</th>
<th>WRITE CLEAR</th>
<th>WRITE SET</th>
<th>WRITE TOGGLE</th>
<th>WRITE ONCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>NO READ</td>
<td>NOACCESS</td>
<td>WO</td>
<td>WOC</td>
<td>WOS</td>
<td>-</td>
<td>W01</td>
</tr>
<tr>
<td>READ VALUE</td>
<td>RO</td>
<td>RW</td>
<td>WC</td>
<td>WS</td>
<td>W1T</td>
<td>W1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>W1C</td>
<td>W1S</td>
<td>W0T</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>W0C</td>
<td>W0S</td>
<td></td>
<td></td>
</tr>
<tr>
<td>READ CLEAR</td>
<td>RC</td>
<td>WRC</td>
<td>-</td>
<td>WSRC</td>
<td>W1SRC</td>
<td>W0SRC</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>W0SRC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>READ SET</td>
<td>RS</td>
<td>WRS</td>
<td>WCRS</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>W1CRS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>W0CRS</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Just defining access policy is **not enough**!
Must also **implement special behavior**!

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

```plaintext
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);
// specific implementation
endtask
```

- **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_write(item rw);
// specific implementation
endtask

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

- Most important callback for passive operation is **post_predict**
Hook & Callback Execution

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

```verilog
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
```

- 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)

```cpp
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()))
    uvm_error..
endtask

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

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

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()))
    uvm_error..
endtask

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

class my_reg_block extends uvm_reg_block;
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);

[Highlighted areas explaining the code:
- **DERIVED CALLBACK**
- **IMPLEMENT post_write TO SET MIRROR TO RESET VALUE**
- **NOT PASSIVE**
- **USE BASE FIELD**
- **CONSTRUCT CALLBACK**
- **REGISTER CALLBACK WITH REQUIRED FIELD**]
WRES Using `post_predict` Callback

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

class wres_reg_t extends uvm_reg
{
  rand uvm_reg_field wres_field;
  function void build();
    // wres_field create() / configure() / build() / add_map()
}

class my_reg_block
{
  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);
}
```

**PASSIVE OPERATION**

虚方法 `post_predict` 只在字段而非寄存器可用。

如果使用此回调与一个寄存器，我们会得到无声的非操作！

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

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

class wres_reg_t extends uvm_reg
{
  rand uvm_reg_field wres_field;
  function void build();
    // wres_field create() / configure() / build() / add_map()
}
```

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

**POST_PASSIVE_OPERATION**

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

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);

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"));
  endfunction

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);

CONNECT PROTECTED FIELD
SET ACCESS POLICY FOR PROTECTED FIELD BASED ON LOCK OPERATION
HANDLE TO PROTECTED FIELD
REGISTER CALLBACK WITH LOCK FIELD
prot_field.get_access() RETURNS CURRENT POLICY
Register Side-Effects Example

- Randomize or modify registers & reconfigure DUT
  - what about UVC configuration?
    - update from register sequences
    - snoop on DUT bus transactions
    - implement post_predict callback

**callback** registered with model field

access UVC config via a **handle**

```
side_effect_cb
  if(field.write(val))
  cfg.set_var(val);
```
Config Update Using Callback

class regCfgCb 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

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

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

GENERATOR TOOL/SCRIPTS

MANY REGISTER CLASSES
(MORE THAN REST OF ENV)

DIFFERENT USE-CASE
THAN FACTORY

© Verilab & Accellera
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

<table>
<thead>
<tr>
<th>MODE</th>
<th>FACTORY TYPES</th>
<th>COMPIL TIME</th>
<th>LOAD TIME</th>
<th>BUILD TIME</th>
<th>DISK USAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>NO REGISTER MODEL</td>
<td>598</td>
<td>23</td>
<td>9</td>
<td>1</td>
<td>280M</td>
</tr>
<tr>
<td>+REGISTERS USING FACTORY</td>
<td>8563</td>
<td>141</td>
<td>95</td>
<td>13</td>
<td>702M</td>
</tr>
<tr>
<td>+REGISTERS NO FACTORY</td>
<td>784</td>
<td>71</td>
<td>17</td>
<td>1</td>
<td>398M</td>
</tr>
</tbody>
</table>

- 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
  function uvm_object_wrapper get_object_type();
    return type_id::get();
  endfunction
  function uvm_object create (string name="");
    const static string type_name = "my_reg";
    virtual function string get_type_name ();
      return type_name;
  endfunction

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
  virtual function uvm_object_wrapper get_object_type();
    return type_id::get();
  endfunction
  function uvm_object create (string name="");
    const static string type_name = "my_reg";
    virtual function string get_type_name ();
      return type_name;
  endfunction
endclass

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

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

declare some **methods** for factory API

explains what **my_reg::type_id** is
Load Time Penalty

class uvm_object_registry
    #(type T, string Tname) extends uvm_object_wrapper

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

virtual function uvm_object create_object(...);

static function T create(...);

static function void set_type_override(type, replace);

static function void set_inst_override(type, inst, parent);
endclass

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

- 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

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

```cpp
class uvm_object_registry #(T, Tname) extends uvm_object_wrapper;

static function T create(name, parent, ctxt="");
    uvm_object obj;
    uvm_factory f = uvm_factory::get();
    obj = f.create_object_by_type(get(), ctxt, name, parent);
    if (!$cast(create, obj)) uvm_report_fatal(...);
endfunction

virtual function uvm_object create_object (name, parent);
    T obj;
    obj = new(name, parent);
    return obj;
endfunction
```

- **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](http://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](http://www.verilab.com) (includes additional examples like triggered writes)
Questions