Advanced UVM in the real world
- Tutorial -

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

- Application-specific code
- Uses UVM building blocks
- Open source (Apache)
- Class library
- Consistent methodology
- Facilitates interoperability
- Supported by all simulators

Verification Environment

Universal Verification Methodology (UVM)

SystemVerilog IEEE1800

Questa™

IUS™

VCS™

- Multi-language simulators
- VHDL, Verilog, SystemVerilog, SystemC
Key Elements

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

Verification Concepts
- constrained-random
- coverage-driven
- transaction-level
- sequences
- scoreboards
- etc...

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...
  – ...but, verification superset much bigger than design!
Verification Concepts

• Generic language-independent concepts apply
  – detailed understanding is not unique to UVM...
  – ...but, implementation details do vary!

  - 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...
Methodology

• **Base-class library**
  – generic building blocks
  – solutions to software patterns
  – save time & effort

• **Way of doing things**
  – consistent approach
  – facilitates interoperability
  – engineering resource flexibility

range of **complexity**, implementation **difficulty**, and **learning** curve
Tutorial Topics

• Selected based on:
  – experiences on many projects at different clients
  – relatively complex implementation or confusing for user
  – benefit from deeper understanding of background code
  – require more description than available documentation

• 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

Vanessa Cooper, Verilab, Inc.
Paul Marriott, Verilab Canada
Introduction

• What is the `uvm_config_db`?
• When is the `uvm_config_db` used?
• How is data stored and retrieved?
• How do I debug when something goes wrong?
• Conclusion
WHAT IS THE CONFIGURATION DATABASE
Configuration Database

The **database** is essentially a lookup table which uses a string as a key and allows you to add and retrieve entries.

- **uvm_resource_db**
  - data sharing mechanism where hierarchy is not important
  - each entry is called a resource
  - when accessing the database, you must specify the resource type as a parameter

```
class uvm_resource_db#(type T=uvm_object)
```
# Configuration Database

<table>
<thead>
<tr>
<th>Methods</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>get_by_type</code></td>
<td>Gets the resource by the type specified by the parameter so the only argument is the scope</td>
</tr>
<tr>
<td><code>get_by_name</code></td>
<td>Gets a resource by using both the scope and name</td>
</tr>
<tr>
<td><code>set</code></td>
<td>Creates a new resource in the database</td>
</tr>
<tr>
<td><code>read_by_name</code></td>
<td>Locates the resource using the scope and name</td>
</tr>
<tr>
<td><code>read_by_type</code></td>
<td>Locates the resource using only the scope</td>
</tr>
<tr>
<td><code>write_by_name</code></td>
<td>Creates the resource by scope and name</td>
</tr>
<tr>
<td><code>write_by_type</code></td>
<td>Creates the resource by scope only</td>
</tr>
</tbody>
</table>

**TABLE 1: uvm_resource_db methods**
Example: A scoreboard has a bit `disable_sb` that turns off checking if the value is 1. How do you change the value of that bit using the `uvm_resource_db`?

```plaintext
static function void set(
    input string scope,
    input string name,
    T val,
    input uvm_object accessor=null)
    uvm_resource_db#(bit) :: set("CHECKS_DISABLE", "disable_scoreboard", 1, this);

static function bit read_by_name(
    input string scope,
    input string name,
    inout T val,
    input uvm_object accessor=null)
    uvm_resource_db#(bit) :: read_by_name("CHECKS_DISABLE", "disable_scoreboard", disable_sb);
```

All functions are static and must use scope resolution operator `::`.
Example: Using the uvm_resource_db with register tests

```cpp
uvm_resource_db#(bit)::set("REG::",
   m_env.m_reg.get_full_name(),
   ".cfg_reg"),
   "NO_REG_TESTS", 1, this);
```
WHEN IS THE CONFIGURATION DATABASE USED
Configuration Database

The database is essentially a lookup table which uses a string as a key and allows you to add and retrieve entries.

- **uvm_config_db**
  - used when hierarchy is important
  - can specify, with great detail, the level of access to a resource
  - almost always used instead of the resource database

```cpp
class uvm_config_db#(type T=int) extends uvm_resource_db#(T)
```
DATA STORAGE AND RETRIEVAL
Configuration Database

**cntxt**: starting point

**inst_name**: instance name which limits accessibility

**field_name**: label used for lookup

**value**: value to be stored

---

static function void set(uvm_component cntxt, 
string inst_name, 
string field_name, 
T value)
**Configuration Database**

- **cntxt**: starting point
- **inst_name**: instance name which limits accessibility
- **field_name**: label used for lookup
- **value**: value to be stored

Example:
```
Example: The Virtual Interface
```
```
uvm_config_db#(virtual tb_intf)
::set(uvm_root::get(),
  "*",
  "dut_intf",
  vif);
```

* should rarely be used
Configuration Database

```
uvm_config_db#(TYPE)::set(uvm_root::get(),"*.path","label",value);
```

```
uvm_config_db#(TYPE)::get(this,"","label",value);
```

| "dut_intf" | "vif"            |
| "retry_count" | "rty_cnt" |
| "my_env_cfg" | "env_cfg" |
| "retry_count" | "rty_cnt" |
DEBUGGING
if(!uvm_config_db#(TYPE)::get(this,"","label",value))
  `uvm_fatal("NOVIF", "Virtual interface GET failed.")
Configuration Database

```
sim_cmd +UVM_TESTNAME=my_test +UVM_RESOURCE_DB_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) ?
CONCLUSION
Configuration Database

• The database is a powerful facility used in testbench construction
• The resource database can be thought of as a pool of variables used without concern for hierarchy
• The configuration database is structured hierarchically and is more suited to data that is related to the structure of the testbench itself.
REFERENCES
Additional Reading & References

- [http://www.accellera.org](http://www.accellera.org)
Questions
Behind the Scenes of the UVM Factory

Mark Litterick, Verilab GmbH.
Introduction

• **Factory pattern** in OOP
  • standard software paradigm

• **Implementation** in OVM/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
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
- **OVM/UVM** make 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.

- OVM/UVM implement 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 OVM/UVM

- Factory is an **essential** part of OVM/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
The main O/UVM files are:
- `o/uvm_object_defines.svh`
- `o/uvm_registry.svh`
- `o/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(...);
  ```
```
define uvm_component_utils(T) \
    m_uvm_component_registry_internal(T,T) \
    m_uvm_get_type_name_func(T) \
enddefine
```

```
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 
    virtual function uvm_object_wrapper get_object_type(); 
        return type_id::get(); 
    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. 

```
explained what `my_comp:::type_id` is 
```

But what about `register` and `::create` ???
class uvm_component_registry
    #(type T, string Tname) extends uvm_object;

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

virtual function uvm_component create_component(name,parent);

static function T create(name, parent, context);

static function void set_type_override(type, replace);

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

local static proxy variable calls get()

proxy type
lightweight substitute for real object

local static proxy variable calls get()

construct instance of proxy, not real class

register proxy with factory

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;
endclass

to register a component type, you only need a typedef
specialization of its proxy class, using `uvm_component_utils

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

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);
endfunction

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

uvm_component_registry #(my_comp,"my_comp")

static create function

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

return handle to real override object

search queues for overrides

call create_component for proxy of override type
Factory Overrides

• 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

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

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

- **b** extends a;
  - `uvm_comp_utils(b)

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

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

- uvm_registry#(a,“a“)
  - get()
  - set_override()
  - create()
  - create_comp()

- uvm_registry#(b,“b“)
  - get()
  - set_override()
  - create()
  - create_comp()

- uvm_factroy
  - register()
  - set_override_by_type()
  - create_comp_by_type()
  - find_override_by_type()

- m_types[obj]
  - a 1
  - b 1
  - o 1

- m_type_overrides[$]
  - a,b

© Verilab & Accellera
Summary of Factory Operation

- **users** only have to call macros to register types with factory
  - resulting typedef is enough for registration to occur
  - static initialization registers proxy classes with factory
- call **type_id::create** instead of **new**
  - allows factory to search for type overrides
  - factory creates instance of required type and returns handle
  - components call create in build phase to allow configuration
- **override** using **type-based** interface
  - factory constructs descriptor of override and adds to queue
  - overrides can be per-instance or for all instances of that type
  - override types must derive from original types
- external files can **modify environment** class structure and behavior without changing code
CONFIGURATION DATABASE
OVM/UVM Configuration Overview

- OVM & UVM provide global and component set/get_config*
  - for int (integral), string and object types, e.g.
    ```
    set_config_object("inst","field",value,0);
    ```
  - if (!this.get_config_object("field",value)) `uvm_fatal(...)

- UVM these are mapped to config_db, e.g. (simplification)
  ```
  function void set_config_object("inst","field",value);
    uvm_config_db#(uvm_object)::set(this,"inst","field",value);
  endfunction
  ```

- In UVM it is recommended to use config_db explicitly
  ```
  uvm_config_db#(my_type)::set(cntxt,"inst","field",value);
  ```

- Implements string-based lookup tables in a database
  - set methods do not configure targeted component fields directly

© Verilab & Accellera
Automatic Field Configuration

- Normally the user has to do an explicit **get** from **db**, e.g.

```plaintext
uvm_config_db#(my_type)::get(this,"","field",value)
```

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

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

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

```plaintext
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
INTERACTION OF FACTORY & CONFIGURATION
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_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());
    set_config_object("env","obj",obj,0);
  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 not build)

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

© Verilab & Accellera 2014

21
Override Order

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

```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
```

remember `create` only results in a `new()` not a `build`

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

**set_config** using a **null** value is an **error**

*(obj)* is not yet constructed

```cpp
function void my_test::build_phase( .. );
...
set_config_object(“env”,“obj”,obj,0); // Bad
obj = my_obj::type_id::create(“obj”,this);
set_config_object(“env”,“obj”,obj,0); // Good
env = my_env::type_id::create(“env”,this);
set_config_object(“env”,“obj”,obj,0); // Good
...
endfunction
```

**set_config after** **obj** is created and **before** **env** is created is **OK**

*(env create does not use the value anyway)*

**set_config after** both **obj** and **env** are created is also **OK**

*(obj) setting in config_db is not used until env::build phase)*

so **set_config** 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)
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);
      set_config_object("env","obj",obj,0);
  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 & CONFIGURATION 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 OVM/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
Common Configuration Problems

• **missing field macro** when using automatic configuration
  – *apply_config_settings()* only works with registered fields

• **missing super.build** when using automatic configuration
  – *apply_config_settings()* is only in uvm_component base

• **missing** *config_bd::get* when *config_db::set* was used
  – *config_db::set* does not interact with *apply_config_settings()*
  – need an explicit *config_db::get* to retrieve settings

• attempting **set_config_object** on a **null** object

• **bad string** matching and **typos** for inst and name settings

• **scope** and **context** problems with string-based config
Debugging Hints

• 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

- OVM/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

- 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 OVM/UVM have **open-source** code
  - so we can see how it is implemented and learn...
  - ...**cool stuff** that keeps us **interested** and **informed**!
Additional Reading & References

• *OVM and UVM base-class code*
• *OVM and 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”
  – SVUG 2008, Jason Spro7, [www.verilab.com](http://www.verilab.com)
• [https://verificationacademy.com/sessions/understanding-factory-and-configuration](https://verificationacademy.com/sessions/understanding-factory-and-configuration)
  – Verification Academy Video, Mentor
  – UVM Tutorial, ClueLogic
Questions
UVM Stimulus and Sequences

Jonathan Bromley, Verilab Ltd
Mark Litterick, Verilab GmbH
Introduction

• tbd
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 make interesting activity
Stimulus transaction class (item)

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

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

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

```vhdl
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

```plaintext
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};
    addr % block_size == 0;
  }
  rand 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

NO distribution constraints
- NO DUT-specific strategy

Behaviour is meaningful even without any external constraint
The story so far...

- provides a flexible base for customization
- does not restrict the uVC's applicability
- already interesting for reactive slave sequences
  - predominantly random
  - more guidance later in this section
- may be useful for simple bring-up tests
LAUNCHING SEQUENCES
Launching a sequence: `uvm_do

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

```
class vbus_seq_bwr2 extends vbus_sequence;
    vbus_seq_block_wr bwr_seq;
    task body();
    bit [15:0] follow_addr;
    `uvm_do( bwr_seq )
    follow_addr = bwr_seq.base_addr + bwr_seq.block_size;
    `uvm_do_with( bwr_seq, {base_addr == follow_addr;} )
...
```

```
rand bit [15:0] block_size;
rand bit [15:0] base_addr;
```

Constraint using values picked from previous sequence's randomization

Lower sequence runs on same sequencer
Launching a sequence: `uvm_do_on

- On a different sequencer
  - good for virtual sequences

```vhdl
class collision_seq extends dut_sequence;
`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: **start**

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

```verilog
class smoke_test extends dut_test;
    `uvm_component_utils(smoke_test)
    smoke_test_seq test_seq;
    base_dut_env env;
    ...
    task run_phase(uvm_phase phase);
    ...
    test_seq = smoke_test_seq::type_id::create("smoke_test");
    ...
    test_seq.start(env.top_sequencer);
    ...
```

**configure/randomize the test seq**
Review of UVM1.2 changes

- Default sequence of sequencer is deprecated
- <describe other consequent changes>
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 expected
    sequencer class
  – allows access to members of that class
    • persistent data across the life of many sequences
    • storage of configuration information, sub-sequencer references, ...
Virtual sequences and sequencers

• Without a sequence item
• Roles and responsibilities
• Methodology details come later in this section
Sequences without sequencers?

• We recommend you do *not* do this
• A hierarchy of sequencers allows clear isolation of concerns
  – each layer of the TB takes responsibility for its own activity
  – make use of facilities provided by lower levels
• Each sequencer can be given references to all the lower-level sequencers it needs to use
Example hierarchical sequences

• TBD: example showing different styles of constraint and different sequence design concerns at each level
  – lowest (UVC) level: completely generic, no strategy, only legality and control knobs to configure legality and basic values
  – UVC sequence library level: a big repertoire of sequences to perform typical operations, with control knobs relevant to those operations
  – DUT level: coordinate sequences across multiple UVCs to establish setup activity, typical operation scenarios, error conditions
Responsibilities of sequences at various levels

• TBD: see previous slide, closer look at concerns on each level
• role of control knobs in seq vs. scenario
• different styles of constraint at different levels
Readback from a sequence

• If a sequence does something that returns a result...
  – read data from a bus
  – transaction has a response value

• ...later parts of the sequence may need the result

• For a sequence *item*: get result directly from the item when the sequence has finished

• For a *sequence*: provide storage in the sequence, populate from lower-level sequence or item
  – needs consistent planning through the sequence hierarchy
Readback from a sequence

• TBD: several slides illustrating preferred techniques
• brief mention of methods based on use of the monitor
• some mention of item_done() responses, generally discouraged - better to use ref to request item
Using a uvm_reg model in sequences (briefly!)

• built-in reg sequences
• reading and writing registers
How sequences should interact with the config DB and user config objects

• Sequences should avoid pulling data directly from the config-DB
  – heavy performance overhead

• As always, populate a config object from the config DB at build time
  – A reference to a centralized config object means that value updates in the central object are automatically visible

• Sequence should look into its sequencer to find config object
  – using p_sequencer
Guidelines for using objections in sequences

• roughly, don't
• but there are some exceptions
• TBD: prescriptive guidance and suggestions, probably "only in top level test sequence" (???)
A couple of fancy examples:

• TBD:
  – interrupt sequence (illustrate use of grab)
  – random choice of sequence in a scenario (using uvm_sequence_library?)
CONCLUSION & REFERENCES
Conclusion

• TBD
Additional Reading & References

- *UVM base-class code*
- *UVM class reference* documentation
- ...
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

© Verilab & Accellera
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
• 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. object.randomize(); field.write(object.var.value);
Register Field Modeling

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

• Field *interaction*  
  – between different register fields

• Register *access rights* in associated memory map

• Model functional *behavior of DUT* for *volatile* fields
### 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>-</td>
<td>WO</td>
<td>WOC</td>
<td>WOS</td>
<td>-</td>
<td>WO1</td>
</tr>
<tr>
<td>READ VALUE</td>
<td>RO</td>
<td>RW</td>
<td>WC W1C W0C</td>
<td>WS W1S W0S</td>
<td>W1T W0T</td>
<td>W1</td>
</tr>
<tr>
<td>READ CLEAR</td>
<td>RC</td>
<td>WRC</td>
<td>-</td>
<td>WSRC W1SRC W0SRC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>READ SET</td>
<td>RS</td>
<td>WRS</td>
<td>WCRS W1CRS W0CRS</td>
<td>-</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **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(...)
```

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

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

```verbatim
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

```verbatim
class my_field_cb extends uvm_reg_cbs;
    function new(string name, ...);
    virtual task post_write();
    // 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*
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()
WRES Using `post_write` Callback

```cpp
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);
```

- **Derived Callback**: `virtual task post_write(uvm_reg_item rw);`
- **Implement `post_write`**: TO SET MIRROR TO RESET VALUE
- **Not Passive**: `if (!predict(rw.get_reset())) `uvm_error(..)`
- **Use Base Field**: `rand uvm_reg_field wres_field;`
- **Construct Callback**: `wres_field_cb wres_cb = new("wres_cb");`
- **Register Callback**: WITH REQUIRED FIELD

© Verilab & Accellera
class my_reg_block
extends uvm_reg_block;

rand wres_reg_t wres_reg;

function void build();

class wres_reg_t extends uvm_reg;
rand uvm_reg_field wres_field;

function void build();

class wres_field_cb extends uvm_reg_cbs;

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);

if (kind==UVM_PREDICT_WRITE) value = fld.get_reset();

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 User Guide)
Lock Using *post_predict* Callback

```cpp
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
```

```cpp
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);
```

- Handle to lock field
- Connect lock field
- Revert to previous value if lock active
- 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"));
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);
Buffered Write Example

- Example register field interaction
  - trigger field operation effects buffered field behavior
- Demonstrate two possible solutions:
  - overlapped register implementation with callback
  - derived buffer field controlled by multiple callbacks
Buffered Write Using 2 Registers

```verbatim
class trig_field_cb extends uvm_reg_cb {
    local uvm_reg_field current, buffer;

    function new (string name, uvm_reg_field current, uvm_reg_field buffer);
    ...
    virtual function void post_predict(...);
    ...
    if (kind == UVM_PREDICT_WRITE) begin
        uvm_reg_data_t val = buffer.get_mirrored_value();
        if (!current.predict(val)) `uvm_error(...);
}

class my_reg_block extends uvm_reg_block {
    ...
    default_map.add_reg(cur_reg, 'h10, "RO");
    default_map.add_reg(buf_reg, 'h10, "WO");
    ...
    trig_field_cb trig_cb = new("trig_cb", cur_reg.cur_field, buf_reg.buf_field);
    uvm_reg_field_cb::add(trig_field, trig_cb);
}
```

- Handles to both current & buffer fields
- Copy from buffer to current on write to trigger
- RO & WO register at same address
  - All writes go to WO buffer
  - All reads come from RO current
- Register callback with trigger field
- Cannot share address again
- Complicated to generate
- Confusing map for user

© Verilab & Accellera
Buffered Write Using Derived Field

class buf_reg_field extends uvm_reg_field;
  uvm_reg_data_t buffer;
  virtual function void reset(string kind);
    super.reset(kind);
    buffer = get_reset(kind);

class buf_field_cb extends uvm_reg_cbs;
  local buf_reg_field buf_field;
  virtual function void post_predict(...);
    // if write
    buf_field.buffer = value;
    value = previous;

class trig_field_cb extends uvm_reg_cbs;
  local buf_reg_field buf_field;
  virtual function void post_predict(...);
    buf_field.predict(buf_field.buffer);

buf_field_cb buf_cb = new("buf_cb", buf_field);
uvm_reg_field_cb:::add(buf_field, buf_cb);
trig_field_cb trig_cb = new("trig_cb", buf_field);
uvm_reg_field_cb:::add(trig_field, trig_cb);

ADD BUFFER TO DERIVED FIELD
RESET BUFFER TO FIELD RESET VALUE
post_predict callback required for passive
SET BUFFER TO VALUE ON WRITE TO FIELD, SET MIRROR TO PREVIOUS (UNCHANGED)
COPY BUFFER TO MIRROR ON WRITE TO TRIGGER
REGISTER CALLBACKS WITH BUFFERED & TRIGGER FIELDS
Register Side-Effects Example

• Randomize or modify registers & reconfigure DUT
  – what about UVC configuration?
    • update from register sequences
    • snoop on DUT bus transactions
    • implement \textit{post\_predict} callback

\begin{verbatim}
callback registered with model field
side_effect_cb
if(field.write(val))
cfg.set_var(val);
\end{verbatim}

access UVC config via a \texttt{handle}

© Verilab & Accellera
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

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);
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 device derivative

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

**GENERATOR** TOOL/SCRIPTS

**REG MODEL**
- MEM
- R1
- R2
- RN
- MAP
- ADP
- PDT
- UVC

**ENV**

**DUT**
- CPU
- F/W

**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>Compile 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>

- **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

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
```

declare a `typedef specialization of `uvm_object_registry class explains what `my_reg:::type_id is

declare some `methods for factory API but what about `factory registration and `type_id::create ???
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

local static proxy variable calls get()

proxy type lightweight substitute for real object

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

m_type_names[obj.get_type_name()] = obj;

return [child] ...
endclass

• 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!
type_id::create

```cpp
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(...);
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 **pre-run phase** for the environment (**build time**)!
- **No need** to search for overrides for **register generator** use-case!
Conclusions

• There’s more than one way to skin a reg...
  – but some are better than others!
  – consider: passive operation, backdoor access, use-cases,...

• Full-chip SoC register model performance impact
  – for generated models we can avoid using the factory

• All solutions evaluated in UVM-1.1d & OVM-2.1.2
  – updated uvm_reg_pkg that includes UVM-1.1d bug fixes
    (available from www.verilab.com)
Questions