Configuration Object Encapsulation & uvm_config_db Usage

Jason Sprott
Agenda

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

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

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

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

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

automatic configuration is done by uvm_component::build_phase() not the database
ENCAPSULATION OF CONFIGURATION DATA
UVC/Environment Configuration

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

- Focused on UVC functionality not register fields
- Usability: amount of work to configure, synchronize and manage
- Extensibility: make it easy to add new functionality
- Maintainability: modify without breaking user code
A Poor Config API Example

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

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

Nice if someone had done that here in the first place

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

PLEASE NO ... 😞
A Better Config API Example

```cpp
if (field.write(val)) {
    rx = extracted_from(val);
    cfg.set_active_rx_chans(rx);
}
```

- Takes care of lower level details and limits exposure to implementation (config variables)
- Check using DUT param
  ```cpp
  if (rx > this.max_chans) {
      uvm_fatal(...)
  }
  // distribute the data
  prx_cfg.set_active_chans(rx);
  stxrx_cfg.set_active_rx_chans(rx);
  // Optional: notify change
  ```
Organizing Configuration

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

- Structural
  - DUT params
  - TB features
  - Devices attached
  - operational constants
  - max_chans
  - addr_width

- Modal
  - protocol
  - control
  - UVC specific
  - system-wide
  - if (rx_chans > max_chans)
  - 2**AW-1

- Shared
  - event pools
  - memory models
  - registers
Events and Pool Example

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

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

```
// events set by
```
```
cfg.m_hard_reset_ev;
```
```
// Env2 from event
```
```
cfg.m_soft_reset_ev;
```
```
// event pool
```
```
cfg.m_frame_end_ev;
```
```
// event pool
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
Using Discrete (Non-object) Entries

Rules of Thumb

• When there is an existing standard
  – e.g. UVC is_active (active/passive) flag
• Simple standalone functional mode or feature
  – e.g. UVC's physical interface or protocol type
  – e.g. UVC performance mode (fast with no frills)
  – Affects other dependent configurations
• Quick wildcard setup required
  – Not as easy with objects, as entries part of a collection
• When run-time modification unlikely
  – Avoid need for run-time ::get() uvm_config_db calls
AUTOMATIC CONFIGURATION OBJECTS AND Enums REVISITED
Some Important Config DB Points

• Look-ups come down to a string match

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

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

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

<table>
<thead>
<tr>
<th>set() using base type required for auto-config</th>
<th>set() using base type both auto and get() work</th>
<th>set() using derived type get() must use derived</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>::set() type</td>
<td>::get() type</td>
<td>auto config</td>
</tr>
<tr>
<td>uvm_object</td>
<td>uvm_object</td>
<td>✓</td>
</tr>
<tr>
<td>uvm_object</td>
<td>CONCRETE</td>
<td>✓</td>
</tr>
<tr>
<td>CONCRETE</td>
<td>CONCRETE</td>
<td></td>
</tr>
<tr>
<td>CONCRETE</td>
<td>uvm_object</td>
<td></td>
</tr>
</tbody>
</table>

**Recommend using uvm_config_object typedef**

© Verilab & Accellera 2015
Use Objects Like This

my_config m_cfg;
...

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

set() and get() use uvm_object type

In the target component

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

uvm_object tmp;

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

Explicit get() needs a cast to concrete type
Parameterized Classes Work Too

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

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

In the target component

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

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

**Explicit get() needs a cast to concrete type**
Wildcard Matching By Name+Type

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

cfg_a m_cfg_a;
cfg_b m_cfg_b;

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

**Concrete types**

cfg_a m_cfg

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

cfg_b m_cfg

```
```cpp
```cpp
```

Possible but **not recommended**: breaks auto-config and a bit risky
### Setting and Getting Enums

**typedef uvm_config_db#(uvm_bitstream_t) uvm_config_int**

<table>
<thead>
<tr>
<th></th>
<th>::set()</th>
<th>::get()</th>
<th>auto</th>
<th>explicit</th>
</tr>
</thead>
<tbody>
<tr>
<td>config_int</td>
<td>config_int</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>config_int</td>
<td>USER</td>
<td>✓</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>USER</td>
<td>USER</td>
<td>X</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>USER</td>
<td>config_int</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

- **set() using integral type**
  - Both **auto** and **get()** work

- **Cast to USER enum type**
  - Required for explicit **get()**

- **Only auto-config works**

- **Only get() works**

- **Wrong**

**Recommend using uvm_config_int typedef**

© Verilab & Accellera
Use Enums Like This

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

set() and get() use integral type

In the target component

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

uvm_bitstream_t tmp;

uvm_config_int::get(..., "m_bus_sz", tmp);

m_bus_sz = my_enum_t(tmp); // back to original type

Explicit get() needs a cast to enum type

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

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

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

We can use a sequencer as the context for `uvm_config_db` lookup

A sequence can lookup entry using handle to the sequencer

Sequencer itself doesn't need to have the target variable

```c
uvm_config_db<T>::get(p_sequencer,"","err_cfg", m_err_cfg)
```
Globally Visible Tag

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

Auto config not possible

Creates a pseudo namespace anyone in hierarchy can use for lookup

Lookup using this tag

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

Limited wildcard options due to uvm_config_db issues
Link Visibility to Self

- Dynamically created instance – no parent available

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

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

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

Fetch value inside the instance

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

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

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

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

**build_phase**
- Highest in hierarchy wins
- Auto configuration
- Done once

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

---

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

---

Nothing reset specific in uvm_config_db
CARE WITH CONFIG OBJECTS
REFERENCES AND CONTENTS
Care with Handles – Local Copies

First set() in `build_phase()`

`env_cfg1`  
@ref  
`uvm_config_db`  
@ref  
`m_env_cfg`  
@ref  
`m_cfg`

Later set() in `run_phase()`

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

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

`m_cfg` members variables used

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

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

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

We can replace m_agent_cfg reference for Agent2 but where?

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

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

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