Improving UVM Testbench Debug Productivity and Visibility

Alex Grove
European Application Engineer

DVClub Europe – May 2016
UVM Testbench Reality Check?

Tutorial-sized block-level verification environment

- 2 Agents
- Regular Structure
- Simple Hookup
- Trivial Scoreboard
- Days/Weeks of Code
**UVM Testbench Reality**

‘Real World’ block-level verification environment

- Multiple Layers
- Inheritance
- Parameters
- Interactions
- Real Scoreboard
- Vertical Reuse
- YEARS of Code
Complex UVM Testbench Navigation

Even well-designed UVM testbenches can be difficult to navigate!

How much harder still for those that are disorganized or have grown in an organic way.
Navigation Wish List

■ See the component hierarchy
  — in an abstracted form – filterable, searchable
  — in a graphical form - clickable

■ Click around in a graphical representation
  — explore the source code via the schematic window
  — explore relationships among code elements

■ See class hierarchy as well as static hierarchy
  — object inheritance and parameterization
  — members/methods in scope
Visualizer Debug

Visualizer™
Debug Environment

Intuitive, High Performance and Capacity Debug Solution

© 2016 Mentor Graphics Corporation, Inc.
Navigate with Visualizer Debug

- Intuitive
  - Natural layout
  - Easy navigation

- Responsive
  - Fast waveform
  - Fast source windows

- Advanced Schematic View
  - Fast and full featured
  - RTL, Gates and UVM

- All windows synchronized
  - Instant time and space sync
Full Visibility with Visualizer

- RTL and Testbench together
- SV Class Debug – handles, contents, context
- Transactions
Common UVM Debug Challenges

- #1 Finding Config database mismatches
- #2 Keeping Track of Factory Overrides
- #3 Diagnosing TLM Connection Hookup
- #4 Understand Hierarchy of Sequences
- #5 Debugging Fragile Scoreboards
- #6 Lower Block to System Reuse Stress
- #7 Navigate The Base Class Hierarchy
- #8 Parallel / Reentrancy Bad Behavior
- #9 Why Don’t My Objections Work?
- #10 Which Class is Causing Trouble?

All we have time for today!

https://verificationacademy.com/seminars/uvm-forum/improve-uvm-testbench-debug-productivity
#1 Finding Config database mismatches

- The Config Database is a data exchange
  - Any SV data type can be passed
  - Can set or get values anywhere in the testbench
  - Set with an ‘address’ which the receiver can match

- Manual Debug
  - static function uvm_config_db#(T)::turn_on_tracing()
    - displays all Config DB accesses for type T
  - command line plusarg +UVM_CONFIG_DB_TRACE
    - shortcut to turn on all Config DB tracing
  - static function uvm_config_db#(T)::dump()
    - prints full contents of the Config DB for type T
## #1 Finding Config database mismatches

### Visualizer Debug

<table>
<thead>
<tr>
<th>Name</th>
<th>Regex</th>
<th>Value</th>
<th>Write Count</th>
<th>Read Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>config_object</td>
<td>/uvm_test_top\i2_agentA.*$/</td>
<td>(class agentA_pkg::ag...</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>i1_agentA_vif</td>
<td>**$/</td>
<td>(virtual abc_if) /top/inte...</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>i1_agentB_vif</td>
<td>**$/</td>
<td>(virtual abc_if) /top/inte...</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>i2_agentA_vif</td>
<td>**$/</td>
<td>(virtual abc_if) /top/inte...</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>i2_agentB_vif</td>
<td>**$/</td>
<td>(virtual abc_if) /top/inte...</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>my_int</td>
<td>/uvm_test_top\i2_agentA\driver$/</td>
<td>(int) 1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>my_int</td>
<td>/uvm_test_top\i2_agentA*$/</td>
<td>(int) 3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>my_string</td>
<td>/uvm_test_top\i2_agentA*$/</td>
<td>(string) Hi</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>my_test_int</td>
<td>/uvm_test_top\i2_agentA*$/</td>
<td>(int) 519</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>no_matches</td>
<td>/uvm_test_top\i2_agentA*$/</td>
<td>(string) None</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>num_sequence_A...</td>
<td>/uvm_test_top\i2_agentA*$/</td>
<td>(int) 1000</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>recording_detail</td>
<td>**$/</td>
<td>(int) 1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>recording_detail</td>
<td></td>
<td>(reg signed[4095:0])</td>
<td>1</td>
<td>73</td>
</tr>
<tr>
<td>recording_detail</td>
<td></td>
<td>(int) 12</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>recording_detail</td>
<td></td>
<td>(int) 3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>simple_int</td>
<td>/uvm_test_top\i2_agentA\driver$/</td>
<td>(string) barrier_name</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>simple_int</td>
<td>/uvm_test_top\i2_agentA*$/</td>
<td>(string) Only_matches</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>simple_int</td>
<td>/uvm_test_top\i2_agentA*$/</td>
<td>(string) Only overrides</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>simple_int</td>
<td>/uvm_test_top\i2_agentA\driver$/</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>test_override</td>
<td>/uvm_test_top\i2_agentA*$/</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>test_override</td>
<td>/uvm_test_top\i2_agentA\driver$/</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Legend:**
- **Type:** Data field type
- **Hier Path or Regexp:** Hierarchical path or regular expression
- **Value:** Value associated with the config
- **Activity:** Activity count (write or read)
The UVM Factory can override any class type
— When creating code uses create() rather than new()

Each override has three main elements
— Registration
— Creation
— Specifying Override

Overrides can be specified in multiple ways
#2 Keeping Track of Factory Overrides

- **Registration**
- **Creation**
- **Override**
- **Instance Override**

```cpp
// create a new color object
pixel = color::type_id::create("pixel", this);

// override all instances of color with red:
color::type_id::set_type_override(red::get_type(), 1);

// will return a red, rather than a color
pixel = color::type_id::create("pixel", this);

// override one particular instances of color with red:
colour::type_id::set_inst_override(red::get_type(), "top.env.foo.bar");
```
#2 Keeping Track of Factory Overrides

- Visualizer Debug

Class Overrides

Class Factory (including parameterized)
#3 Diagnosing TLM Connection Hookup

- The environment and agent `connect_phase()`
  - hooks up TLM ports and analysis ports
  - connects monitors, scoreboards, coverage objects
  - hooks up sequencer to driver TLM connections

- Multiple possible points of failure
  - Port connections crossing multiple hierarchy levels
  - Instances hooked up successfully to the wrong port
  - Analysis ports that never see traffic, result is a ‘pass’
#3 Diagnosing TLM Connection Hookup

- **Visualizer Debug**
  - UVM Schematic View
    - TLM connections
    - Analysis Ports
    - Virtual Interfaces
  - SV Class Debug
    - Class handles in waves
    - Class members in waves
    - See all dynamic activity
#4 Understand Hierarchy of Sequences

- **Parent/Child sequences**
  - Implement layered protocols
    - frames, channels, bursts
  - Provide high level stimulus
    - high level activity sequences
    - ..made up of ‘worker’ sequences
    - ..calling API level sequences

- **Know what is running**
  - where
  - and why

- **Manual Debug**
  - grep
  - editor
  - code review
#4 Understand Hierarchy of Sequences

- Visualizer Debug
  - Live sequence hierarchy activity by sequencer
#5 Debugging Fragile Scoreboards

- Scoreboards can become brittle the more accurate they are coded
  - Don’t get to the point where you should tape out the scoreboard not the design

- Keeping scoreboards relaxed requires complex data structures
  - extensive use of queues, associative arrays, or arrays of classes containing groups of information including protocol analysis reports
#5 Debugging Fragile Scoreboards

**Visualizer Debug**

- SV Classes and their members are First Class Citizens
  - drag handles/values into wave window from source or object
  - see transactions, analysis report items, scoreboard objects in context
Visualizer Debug

- Built for Speed
  - High Performance and High Capacity

- Built for Productivity
  - Intuitive and easy-to-use
  - Powerful automation - find bugs fast

- Built for Today’s Complex Projects
  - SV Class Debug
  - UVM Debug

- Built for Today’s Platforms
  - Questa Simulation
  - Questa Formal
  - Veloce Emulation
Visualizer is Mentor’s Enterprise Verification Debugger

Stimulus

Management & Analysis

Visualizer™ Debug

Verification Infrastructure

Vista™ Virtual Prototype

Questa® Formal

Questa® Simulation

Veloce® Emulation

FPGA Prototype