Slicing Through the UVM's Red Tape
A Frustrated User's Survival Guide

Jonathan Bromley
UVM != Straitjacket

- Doesn't cover everything
- Some key common requirements unsatisfied
- User must make intelligent choices
- Basic standard approaches not always optimal
Example 1: Multi-Subscriber

How to implement?
Multi-Subscriber: BAD IDEAS

How to implement?

- `uvm_analysis_fifo`
  - poor fit with typical problems
  - "collector" process is hard to design well

- `uvm_analysis_imp_decl`
  - doesn't scale to N similar sources
  - unbelievably nasty

- subscriber instances
  - too much customization work
Consider the Essentials

- Configurable (dynamic) array of analysis_export
- `write(txn,index)` for custom functionality

```plaintext
class uvm_indexed_subscriber #(type T)
  extends uvm_component;

  int n_exports;
  uvm_analysis_export #(T) analysis_export[];

  virtual function void write(T t, int index);

  case (index) ...
```

As easy to use as uvm_subscriber
Implement!

- Base class contains embedded subscriber class
- Base class constructs arrays

```plaintext
class uvm_indexed_subscriber #(type T)
    extends uvm_component;

    int n_exports;
    uvm_analysis_export #(T) analysis_export[];
    idx_sub #(T) subscriber[];

    function void write(T t);
        parent.write(t, index);
    endfunction

subscriber[n_exports-1]
    int index;
    indexed_subscriber#(T) parent;
    function void write(T t);
        parent.write(t, index);
    endfunction
```

© Accellera Systems Initiative
2: Deploying Config Objects

- Config DB? *not the only way* to get information into components
- Not even optimal, within a well-defined structure
- Use config DB to get stuff into the agent...

Agent then uses simple copy to distribute it to child components
Config Deployment: BAD IDEAS (1)

• insist on an entry in config_db
  – maybe it was provided by copy!

```plaintext
function void build_phase(...);
  super.build_phase(phase);
  if (!uvm_config_db#(my_cfg_class)::get(...)) begin
    `uvm_error("NO_CFG", ...);
  end
```
Config Deployment: BAD IDEAS (2)

• push a value downwards in connect_phase
  – it's bottom-up!

```plaintext
build_phase
child.cfg = this.cfg;

connect_phase
child.cfg = this.cfg;
```
Config Deployment: BAD IDEAS (3)

• Excessively lax wildcard matching
  – jeopardizes reusability

```
#(my_cfg_class)::set(
    this,
    "*",
    "cfg",
    cfg);
```

Targets *any* descendant component!
Consider the Essentials

• Within a well-known structure, use methods or copy
• Allow for off-standard customization (use configDB)

```cpp
function void build_phase(...);
  super.build_phase(phase);
  if (uvm_config_db#(my_cfg_class)::get(..., cfg)) begin
    `uvm_info("CFG_FROM_DB", ..., )
  end
  if (cfg == null) begin
    `uvm_error("NO_CFG", ...)
  end
```

Config-DB can override copy

Flexible for any component
3: Layered Data-intensive Sequences

class serdes_symbol extends uvm_sequence_item;
rand bit [9:0] symbol_bits;
rand int n_bits;
rand bit timing_error;
constraint symbol_10_bits {
    soft !timing_error;
    if (timing_error) {
        n_bits inside {{1:10}};
    } else {
        n_bits == 10;
    }
}

A B C
D E F
U V W
X Y Z

randomize

A B C D E F W X Y Z

randomize

encA encB ------- encY encZ

© Accellera Systems Initiative
3: Layered Data-intensive Sequences

class serdes_symbol extends uvm_sequence_item;
  rand bit [9:0] symbol_bits;

class frame_sequence extends uvm_sequence;
  rand bit [7:0] fr_data[];
  task body();
    serdes_symbol sym;
    foreach (fr_data[i]) begin
      bit [9:0] val = encode(fr_data[i]);
      `uvm_do_with( sym,
        {symbol_bits == local::val;} )
    end
  endtask
...
endclass
3: Layered Data-intensive Sequences

class serdes_symbol extends uvm_sequence_item;
    rand bit [9:0] symbol_bits;

class frame_sequence extends uvm_sequence;
    rand bit [7:0] fr_data[];

class data24_sequence extends uvm_sequence;
    rand bit [23:0] stream_data[];
    task body();
        frame_sequence fr_seq;
        `uvm_do_with( fr_seq, {
            frame_data.size() == 3*stream_data.size();
            foreach (stream_data[i]) {
                frame_data[3*i]    == stream_data[i][23:16];
                frame_data[3*i+1] == stream_data[i][15:8];
                frame_data[3*i+2] == stream_data[i][7:0];
            }
        })
Data Sequences: BAD IDEAS

• randomize everything
  – inefficient, clumsy

• strictly deterministic code
  – limits flexibility
Consider the Essentials

• Allow for error injection
• Plan for flexibility
• Consider efficiency
  – especially for small low-level sequences

```verilog
rand bit [7:0] fr_data[];
task body();
  serdes_symbol sym;
  foreach (fr_data[i]) begin
    `uvm_create( sym )
    sym.symbol_bits = encode(fr_data[i]);
    `uvm_send( sym )
  end
endtask
```

No flexibility lost
Takeaway

• UVM is a great kick-start
  – not the end of the journey
• It's only code! Be prepared to try out new ideas
  – but have a fallback; ideas sometimes don't work out
• Don't trust “UVM has done all the work for you”
• Share good ideas to limit wheel reinvention
  – forum sites
  – constructive review
  – in-house toolkits / patterns
Thank You!

Any Questions?