Go Figure – UVM Configure
The Good, The Bad, The Debug

Rich Edelman, Mentor Graphics Corporation
Dirk Hansen, Mentor Graphics Corporation
UVM Config

• Using Config
• Debugging Config
• Improving Config
  – Deleggability
  – Transparency
  – Simplifications?
What happens in the background if you use:
Set into config DB
What happens in the background if you use:
get config DB
Any user of the config DB as a problem

• The UVM Config database is hard to understand
• It has strange data structures
• It has poor debug messaging
The list of failure modes is long:

- Wrong hierarchy (too high)
- Wrong hierarchy (too low)
- Typo in set() call
- Typo in get() call
- Wildcard in name incorrect (UVM 1.1d)
- Database organized by property name. For a large testbench where every component has this property, a performance problem may occur.
- Setting a property value from two different places, with conflicting values.
- Setting a property value from two different initial blocks. Process ordering may affect which occurs first.
Any conclusions from these entry slides???

• The best recommendation to avoid debugging the uvm_config database is to not use it.
  • BUT
• most common testbenches do use it. In fact, most common testbenches overuse it
• => so what needs to be done to make the usage easier??
Basic rules for usage of config db

• Do not use the “auto configuration” mechanism in the field automation macros.

• Do not set simple integers and other small sets of variables. Instead, create a “configuration container” and put that in the configuration database.

• Do not use the configuration database in a monitor or scoreboard, where the configuration database is being searched on every clock edge.

• Do not use wildcards in the name field.
Traditional debug options

• Usage of command line option:
  • `+UVM_CONFIG_DB_TRACE`
  • `+UVM_RESOURCE_DB_TRACE`
    – makes all `set()` and `get()` calls visible in the simulation log
      • Drawback: simulation log becomes cluttered with a lot of messages thus additional filtering becomes necessary

• Print statements for according error messages:
  – `if(!uvm_config_db#(virtual svt_axi_if)::get(this,"","vif", vif))`
    ```
    `uvm_fatal("AXI_AGENT:NOVIF", "The virtual interface get is not successful");
    ```
    • Drawback: message does not point to real reason of the issue
Solutions for better debug

• Extension of code to provide detailed information
• Extension of config.print
• Usage of simple configuration
Example of instrumentation

function uvm_resource_types::rsrc_q_t lookup_name (  
    string scope = "",  
    string name,  
    uvm_resource_base type_handle = null,  
    bit rpterr = 1,

    // NEW STUFF
    input uvm_object CALLINGCONTEXT = null,
    input string FILE = "",
    input int LINE = 0
)
);
Enhanced Debug

• Add three arguments to ::set() and ::get() to enable call site tracking.
  – ‘this’ → some uvm_component who “owns” this call
  – FILE → the file name
  – LINE → the line number

```c
uvm_config_db#(string)::get(context, inst_name, field_name, variable_name, this, `__FILE__, `__LINE__));
```

```c
uvm_config_db#(string)::set(context, inst_name, field_name, value, this, `__FILE__, `__LINE__));
```
Printing
Old Messages vs. New Messages

Source Code example calling ::set and calling ::get()

```plaintext
uvm_config_db #(virtual dut_if)::set(null, "uvm_test_top","virtual_interface", input_pins);
if(!uvm_config_db #(virtual dut_if)::get(null, "uvm_test_top", "no_virtual_interface", v_input_pin_if)) begin
    `uvm_error("build_phase", "interface not found in config db")
```

Error message points to the line of code but no further information what went wrong

```
UVM_ERROR ../sv_src/dut_driver.svh(19) @ 0: uvm_test_top.adder_env0.dut input agent.driver [build_phase] interface not found in config db
```
Old Messages vs. New Messages

Source Code calling ::get()

if(!uvm_config_db #(virtual dut_if)::get(null, "uvm_test_top", "no_virtual_interface", v_input_pin_if, this, `__FILE__, `__LINE__))
begin `uvm_error("build_phase", "interface not found in config db")

Enhanced 1.1d

Details about decision making

# UVM_INFO @ 0: reporter [CFGDB/GET DEBUG] Looking for 'no_virtual_interface' in 'uvm_test_top'
# UVM_INFO @ 0: reporter [CFGDB/GET DEBUG] No resource for name 'no_virtual_interface' exists
# UVM_INFO @ 0: reporter [CFGDB/GET DEBUG] Found NO matches for 'no_virtual_interface' in 'uvm_test_top'
# UVM_ERROR ../sv_src/dut_driver.svh(19) @ 0: uvm_test_top.adder_env0.dut input agent.driver [build_phase] interface not found in config db
Extension of config.print

- The UVM can print a summary of the configuration interaction – it dumps the access database.
- Base code of struct:

```c
typedef struct
{
  time read_time;
  time write_time;
  int unsigned read_count;
  int unsigned write_count;
} access_t
```
typedef struct
{
  time read_time;
  time write_time;
  int unsigned read_count;
  int unsigned write_count;
  string typename;
  string full_name;
  uvm_object calling_context;
  string file;
  int line;
  uvm_resource_base r;
} access_t;
Printing contents of config_db using new print function

No information related to „no_virtual_interface“ found

=> This entry is missing, so this is the reason for the problem
function uvm_resource_types::rsrc_q_t lookup_name(string scope = ",",
    string name,
    uvm_resource_base type_handle = null,
    bit rpterr = 1);

if((rpterr && !spell_check(name)) || (!rpterr && !rtab.exists(name))) begin

  return q;
end

rsrcc = null;
rq = rtab[name];

for(int i=0; i<rq.size(); ++i) begin

  r = rq.get(i);

  // does the type and scope match?
  if(((type_handle == null) || (r.get_type_handle() == type_handle)) &&
     r.match_scope(scope))
    q.push_back(r);
end

return q;
endfunction
What does better debug look like?

function uvm_resource_types::rsrc_q_t lookup_name(string scope = "", string name, uvm_resource_base type_handle = null, bit rpterr = 1, input uvm_object CALLING_CONTEXT = null, input string FILE = "", input int LINE = 0);

if((rpterr && !spell_check(name)) || (!rpterr && !rtab.exists(name))) begin
  `uvm_info_context_with_file_line("CFGDB/GET DEBUG", $sformatf("  No resource for name '%s' exists", name), UVM_MEDIUM, CALLING_CONTEXT, FILE, LINE)
  return q;
end

rsrc = null;
rq = rtab[name];
if (rq.size()<1) begin
  `uvm_info_context_with_file_line("CFGDB/GET DEBUG", $sformatf("  Found resource for name '%s', but q is empty.", name), UVM_MEDIUM, CALLING_CONTEXT, FILE, LINE)
  return q;
end
What does better debug look like?

```plaintext
for(int i=0; i<rq.size(); ++i) begin
  string msg;
  r = rq.get(i);

  msg = $sformatf(" Item #%2d: P:%4d ", i, r.precedence);
  if (type_handle == null)
    msg = {msg, $sformatf("Type handle is NULL")};
  else if (r.get_type_handle() == type_handle)
    msg = {msg, $sformatf("Type handle matches (%s)", r.convert2string())};
  else
    msg = {msg, $sformatf("Type handle does not match (looking for '%s', found '%%s')", type_handle.convert2string(), r.get_type_handle().convert2string())};

  if (r.match_scope(scope))
    msg = {msg, $sformatf("Scope (%s) matches (%s)", scope, r.get_scope())};
  else
    msg = {msg, $sformatf("Scope (%s) doesn't match (%s)", scope, r.get_scope())};

  `uvm_info_context_with_file_line("CFGDB/GET DEBUG", msg, UVM_MEDIUM, CALLING_CONTEXT, FILE, LINE)

  // does the type and scope match?
  if(((type_handle == null) || (r.get_type_handle() == type_handle)) && r.match_scope(scope))
    q.push_back(r);
end
return q;
endfunction
```
What does better debug look like?

```cpp
function uvm_resource_types::rsrc_q_t lookup_name(string scope = "", string name, uvm_resource_base type_handle = null, bit rpterr = 1);
if((rpterr && !spell_check(name)) || (!rpterr && !rtab.exists(name))) begin
    return q;
end
rsrc = null;
 rq = rtab[name];
for(int i=0; i<rq.size(); ++i) begin
    r = rq.get(i);
    // does the type and scope match?
    if(((type_handle == null) || (r.get_type_handle() == type_handle)) && r.match_scope(scope))
        q.push_back(r);
end
return q;
endfunction
```

Code with all extensions (font too small for a projector)
What does better debug look like?

• All if-then-elses are instrumented
• Boolean expressions are taken apart
• Return from functions are instrumented

• Every decision gets an instrumentation point.
• Now we can track every decision.
Usage of simple configuration

- The configuration database is really just a global variable – a global table of names of properties. Each property has a list of scopes.
- The replacement configuration database is a global variable – a global table of names of full path names, including the property name – it is a simple table of name:value pairs. It is an associative array, indexed by the full path names. It will be fast, and it will be used to hold hierarchical configurations, so it will be small.
- Lookup a property (a configuration) by using get_full_name(). Each component could choose to continue in a loop climbing the component instance tree: search to “a.b.c.d.block1”, then “a.b.c.d”, then “a.b.c”, then “a.b”, then “a”, then uvm_root. This is a well-known algorithm, and easy to debug.

A NEW different implementation.
```verilog
class simple_config #(type T = int);
  static T lookup_table[string];

  static function T get(string name);
    T return_value;
    if (lookup_table.exists(name))
      return_value = lookup_table[name];
    return return_value;
  endfunction

  static function set(string name, T set_value);
    lookup_table[name] = set_value;
  endfunction
endclass
```
class B extends uvm_component;
  function void build_phase(uvm_phase phase);
    `uvm_info("MSG", $sformatf("MSG=%s, N=%0d", 
      simple_config#(string)::get("MSG"), 
      simple_config#(int)::get("N")), UVM_MEDIUM)
    `uvm_info("MSG", $sformatf("SPECIAL_MSG=%s, N=%0d", 
      simple_config#(string)::get({get_full_name(), ".MSG"}), 
      simple_config#(int)::get("N")), UVM_MEDIUM)
  endfunction
endclass

module top();
  initial begin
    simple_config#(int)::set("N", 24);
    simple_config#(string)::set("MSG", "Hello World!");
    simple_config#(string)::set("uvm_test_top.a1.b1.MSG", 
      "Hello World!, uvm_test_top.a1.b1!!!");
    run_test();
  end
endmodule
Conclusion

• Avoid overusing existing config db
  – Cumbersome debug
  – Slows down simulation

• Regular debug of uvm_config db is hardly possible
  – Use extension of code to add according debug informations
    • Source code changes are available from the authors

• Apply new configuration mechanism, whenever possible
Questions

Email authors for source code or questions

rich_edelman@mentor.com
dirk_hansen@mentor.com