GO FIGURE – UVM CONFIGURE THE GOOD, THE BAD, THE DEBUG

RICH EDELMAN AND DIRK HANSEN, MENTOR GRAPHICS

WHITEPAPER

Mentor®
A Siemens Business

FUNCTIONAL VERIFICATION

www.mentor.com

I. Introduction

The UVM Configuration database - the uvm_config_db - is used for setting parameters and controls. This database has a hierarchical aspect, is typed, has precedence or priority and has rules about first-wins or last-wins. In short, the UVM Configuration database is a complex machine used by every SystemVerilog UVM testbench. It can also be hard to debug and complex to understand. This paper will try to shed some light on the complexity and provide some useful code extensions for debug.

We will explore the UVM 1.1d Config Database implementation and theory with an effort to explaining the desired functionality to the new user, as well as clarifying how to best do debug when things go wrong. Additionally, UVM 1.1d to UVM 1.2 changes will be discussed, and a proposal for a simple, easier to understand and easier to debug implementation will be proposed.

The output from the instrumented code is demonstrated below. The UVM 1.1d output is listed and the Enhanced UVM 1.1d output is listed. For UVM 1.1d, this output is generated by using +UVM_CONFIG_DB_TRACE +UVM_RESOURCE_DB_TRACE during simulation.

A. *Messages from UVM 1.1d*

In the UVM 1.1d example, the output looks like

```
./uvm-1.1d/src/base/uvm_resource_db.svh(121) @ 101: reporter Configuration 'uvm_test_top.e2.a2.b2.c2.XXX_3'
(type string) read by uvm_test_top.e2.a2.b2.c2 = (string) 2env.a1.<STAR>*::XXX_3
./uvm-1.1d/src/base/uvm_resource_db.svh(121) @ 101: reporter Configuration 'uvm_test_top.e2.a2.b2.c2.XXX_4'
(type string) read by uvm_test_top.e2.a2.b2.c2 = null (failed lookup)
```

Two lines of debug are produced with the final results. Each line has a common prefix referring to the line where the message is printed in the UVM. This is not a helpful location.

```
./uvm-1.1d/src/base/uvm_resource_db.svh(121)
```

For a failure, a “Cannot find” message is printed. Also not a helpful message.

```
reporter Configuration 'uvm_test_top.e2.a2.b2.c2.XXX_4' (type string) read by
uvm_test_top.e2.a2.b2.c2 = null (failed lookup)
```

B. *Messages from UVM 1.1d Enhanced*

In the UVM 1.1d Enhanced library, the complete output looks like

```
t.sv(101) @ 101: uvm_test_top.e2.a2.b2.c2 Looking for 'XXX_3' in 'uvm_test_top.e2.a2.b2.c2'
t.sv(101) @ 101: uvm_test_top.e2.a2.b2.c2 Item # 0: P:1000 Type handle matches ((string) 2env.a1.<STAR>*::XXX_3), Scope
(uvm_test_top.e2.a2.b2.c2) doesn't match (/^uvm_test_top\.e1\.a1\..*$/)
t.sv(101) @ 101: uvm_test_top.e2.a2.b2.c2 Item # 1: P:1000 Type handle matches ((string) 2env.a1.<STAR>*::XXX_3), Scope
(uvm_test_top.e2.a2.b2.c2) doesn't match (/^uvm_test_top\.e1\.a2\..*$/)
t.sv(101) @ 101: uvm_test_top.e2.a2.b2.c2 Item # 2: P:1000 Type handle matches ((string) 2env.a1.<STAR>*::XXX_3), Scope
(uvm_test_top.e2.a2.b2.c2) doesn't match (/^uvm_test_top\.e2\.a1\..*$/)
t.sv(101) @ 101: uvm_test_top.e2.a2.b2.c2 Item # 3: P:1000 Type handle matches ((string) 2env.a1.<STAR>*::XXX_3), Scope
(uvm_test_top.e2.a2.b2.c2) matches (/^uvm_test_top\.e2\.a2\..*$/)
t.sv(101) @ 101: uvm_test_top.e2.a2.b2.c2 Found 1   match for 'XXX_3' in 'uvm_test_top.e2.a2.b2.c2'
t.sv(101) @ 101: uvm_test_top.e2.a2.b2.c2 Looking for highest precedence
t.sv(101) @ 101: uvm_test_top.e2.a2.b2.c2 Found properly typed ((string) 2env.a1.<STAR>*::XXX_3) resource in the queue. (Last
one)
t.sv(101) @ 101: uvm_test_top.e2.a2.b2.c2 'uvm_test_top.e2.a2.b2.c2.XXX_3' (type string) read by uvm_test_top.e2.a2.b2.c2 =
(string) 2env.a1.<STAR>*::XXX_3
t.sv(101) @ 101: uvm_test_top.e2.a2.b2.c2 SUCCESS: Found value "2env.a1.<STAR>*::XXX_3"


t.sv(104) @ 101: uvm_test_top.e2.a2.b2.c2 Looking for 'XXX_4' in 'uvm_test_top.e2.a2.b2.c2'
t.sv(104) @ 101: uvm_test_top.e2.a2.b2.c2 Item # 0: P: 997 Type handle matches ((string) env.a1::XXX_4), Scope
(uvm_test_top.e2.a2.b2.c2) doesn't match (/^uvm_test_top\.e2\.a1$/)
t.sv(104) @ 101: uvm_test_top.e2.a2.b2.c2 Item # 1: P: 997 Type handle matches ((string) env.a1::XXX_4), Scope
(uvm_test_top.e2.a2.b2.c2) doesn't match (/^uvm_test_top\.e1\.a1$/)
t.sv(104) @ 101: uvm_test_top.e2.a2.b2.c2 Found NO matches for 'XXX_4' in 'uvm_test_top.e2.a2.b2.c2'
t.sv(104) @ 101: uvm_test_top.e2.a2.b2.c2 'uvm_test_top.e2.a2.b2.c2.XXX_4' (type string) read by uvm_test_top.e2.a2.b2.c2 =
null (failed lookup)
t.sv(104) @ 101: uvm_test_top.e2.a2.b2.c2 [C] Can't find XXX_4
```

Multiple lines are produced for each call to get(), outlining the internal decisions taken for both failure and success. In the enhanced library, each line is prefixed with the actual line where the get() call is located. This is a helpful location.

```
 t.sv(101)
```

For a failure, a complete history of the algorithm and failure is listed – with which items were checked from the resource queue:

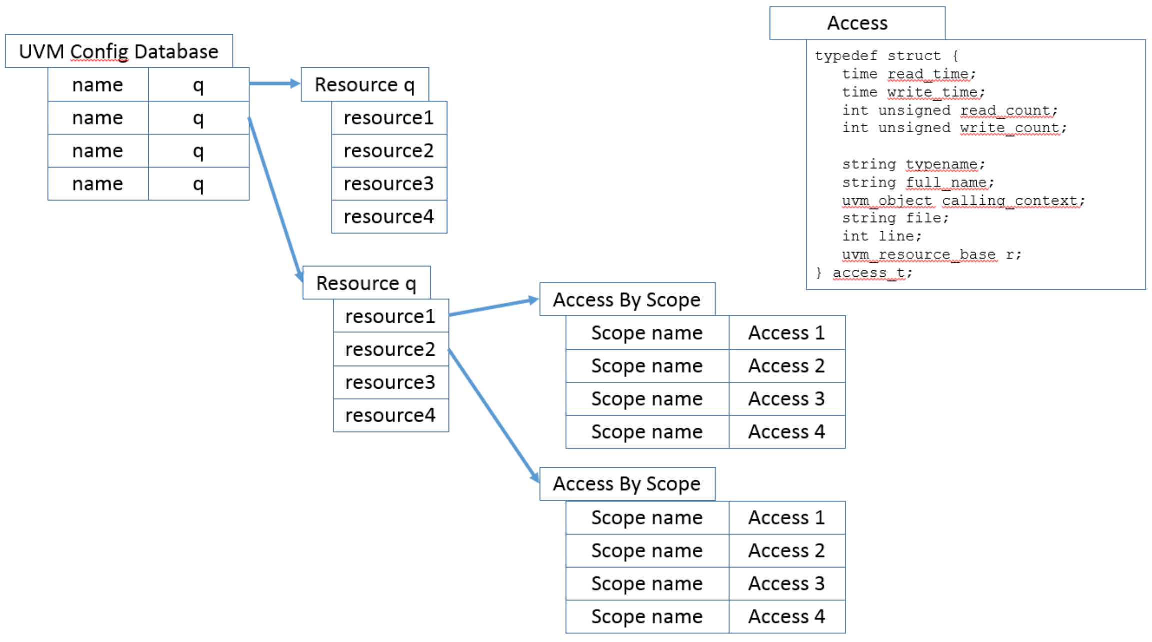
```
Looking for 'XXX_4' in 'uvm_test_top.e2.a2.b2.c2'
        Item # 0: P: 997 Type handle matches ((string) env.a1::XXX_4), Scope
        (uvm_test_top.e2.a2.b2.c2) doesn't match (/^uvm_test_top\.e2\.a1$/)
        Item # 1: P: 997 Type handle matches ((string) env.a1::XXX_4), Scope
        (uvm_test_top.e2.a2.b2.c2) doesn't match (/^uvm_test_top\.e1\.a1$/)
Found NO matches for 'XXX_4' in 'uvm_test_top.e2.a2.b2.c2'
'uvm_test_top.e2.a2.b2.c2.XXX_4' (type string) read by uvm_test_top.e2.a2.b2.c2 = null (failed
lookup)
```

To the uninitiated these messages are all just noise, but for a person trying to figure out why XXX_4 could not be found, they are clues. We can see right away that the XXX_4 property is attached with a regular expression for two levels up. We are asking for the setting too low, or we set it too high. Failure Debugged!

II. Background

The UVM Config database began life as a reimagined, re-implemented configuration database implemented in the OVM. The old version had limitations and was hard to use. The new UVM based implementation was imagined as a global database with a simple (name:value) lookup. As the integration into the UVM code stream began, forces worked to change the code, to layer it, and adapt it for certain poorly defined requirements. The changes made during this integration effort made the implementation more complex, and harder to understand.

The configuration database is best thought of as a global table of (name:value) pairs. You register a configuration name and value, like NUMBER_OF_TRANSACTIONS:1000 so that later other UVM testbench components can lookup NUMBER_OF_TRANSACTIONS and use the value returned accordingly. There is one global table which holds (name:value) pairs.



```
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;
```

III. Sample Usage – What's happens when?

A simple example below illustrates setting and getting from the uvm_config database for the virtual interfaces that connect the testbench to the DUT. This is the traditional usage to set values.

```
module top();
  ...
  intfA vifA1(clk);
  intfA vifA2(clk);
  intfB vifB1(clk);
  intfB vifB2(clk);

  initial begin
    uvm_config_db#(virtual intfA)::set(null, "*e1", "vifA", vifA1);
    uvm_config_db#(virtual intfA)::set(null, "*e2", "vifA", vifA2);
    uvm_config_db#(virtual intfB)::set(null, "*e1", "vifB", vifB1);
    uvm_config_db#(virtual intfB)::set(null, "*e2", "vifB", vifB2);
    run_test();
  end
  ...
endmodule
```

The code below tests the value retrieval for the virtual interfaces – one set of gets() in the build_phase() and the other in the run_phase(). This is the traditional usage to get values.

Original call to get():

```
    if (!uvm_config_db#(virtual intfA)::get(this, "", "vifA", vifA))
      `uvm_warning(get_type_name(), "Can't find vifA");
```

If we add the calling context, the file and line number to the call, then each call to get() or to set() can be tracked back to the original call site. This tracking back to the actual place where the setting is created or retrieved is exactly the kind of debug needed.

Updated, better debug, call to get():

```
  if (!uvm_config_db#(virtual intfA)::get(this, "", "vifA", vifA, this, `__FILE__, `__LINE__))
    `uvm_warning(get_type_name(), "Can't find vifA");
```

IV. Quick Recommendations

The best recommendation to avoid debugging the uvm_config database is to not use it. It is possible to not use it, but most common testbenches do use it. In fact, most common testbenches overuse it. If it is going to be used, then use it once. Or twice. A few times.

A. Spanning module and testbench space

The best case for a configuration database, or just a simple global table is to span the module / testbench space. On the module side, an interface is created, and a virtual interface handle must be passed to the testbench. The testbench does NOT exist until 'run 0'. So the virtual interface handle must be "cached" or saved somewhere. A crude version of this follows in this code snippet. The virtual interface handle is stored in the global variable named 'if_cache'.

```
virtual my_interface if_cache;

class env extends uvm_component;
  `uvm_component_utils(env)

  virtual my_interface if0;

  function void build_phase(uvm_phase phase);
    if0 = if_cache;
    `uvm_info("INTERFACE", if0.get_full_name(), UVM_MEDIUM)
  endfunction
endclass

module top();
  reg clk;
  my_interface if0(clk);
  initial begin
    if_cache = if0;
    run_test();
  end
endmodule
```

A more robust scalable solution, would at least create a simple lookup-by-name structure – such as an associative array indexed by name. Without the need to span module and testbench space, there is limited need for a configuration database, since a test could create the configuration directly either with direct set or randomize calls. Each different configuration would simply be a different test from the test library. The configuration simply becomes the test.

B. A container of configurations

Build a structure or a class container which holds all the controls and settings. This container can be hierarchical – the container can have handles to other containers. The one, first container is the only thing in the configuration database – it is named “settings”, and it is set by the test. The testbench parcels out configuration settings to all the lower levels.

```
class C_configuration;
  int a, b, c;
endclass

class B_configuration;
  C_configuration C1_config;
  C_configuration C2_config;
endclass

class A_configuration;
  B_configuration B1_config;
  B_configuration B2_config;

 endclass

 class env_configuration;
   A_configuration A1_config;
   A_configuration A2_config;
 endclass
```

The test creates the environment configuration (how it wants the environment to be structured and how it should run). Then the test stores the environment configuration in the config database for the environment to lookup. The environment then knows how to pass the "children" configurations, etc. In the simplest approach, the configuration database would not be needed, since each "parent" component could simply set the child configurations directly:

```
class A ….;
  A_configuration config;
  B B1, B2;
  function void build_phase(uvm_phase phase);
    B1.config = config.B1_configuration;
    B2.config = config.B2_configuration;
  endfunction
```

V. THE CONFIG AND RESOURCE DATABASE

There are 5 files that together make up the UVM Configuration system. They are:

Number of Lines	File Name
367	uvm-1.1d/src/base/uvm_config_db.svh
399	uvm-1.1d/src/base/uvm_resource_db.svh
170	uvm-1.1d/src/base/uvm_resource_specializations.svh
1713	uvm-1.1d/src/base/uvm_resource.svh
194	uvm-1.1d/src/base/uvm_spell_chkr.svh
347	uvm-1.1d/src/base/uvm_pool.svh

There are a total of 3190 lines of code. This implementation has many layers and interactions. The sections following will describe some of the interactions and concepts.

A. The uvm_config_db::get() algorithm

The get() function call is used to retrieve the value that matches the context, instance name and field name specified. It is the target of our debug effort – we need better visibility into the get() algorithm. There are many failure modes; each mode means some part of the lookup was incorrectly specified, or the original set() was in error.

```
  static function bit get(uvm_component cntxt,
                          string inst_name,
                          string field_name,
                          inout T value);

   Generate An Instance Name.

   Queue of Results <= Lookup The Instance Name, Field Name and Type.
     Is the Field Name in the Associative Array?
     If yes, then
     foreach (element in the Queue of Resource Settings) {
       if the Queue of Resource Setting[i] type matches the Type, and the scope matches, add
this element to the Queue of Results
     }

   Find the highest precedence in the Queue of Results.
     foreach (element in the Queue of Results) {

        if the Queue of Results[i] type matches the Type, find the highest precedence in the
 queue after this matching element, that is also of the correct Type.
      }
   endfunction
```

The list of failure modes is long:

1. Wrong hierarchy (too high)
2. Wrong hierarchy (too low)
3. Typo in set() call
4. Typo in get() call
5. Wildcard in name incorrect (UVM 1.1d)
6. Database organized by property name. For a large testbench where every component has this property, a performance problem may occur.
7. Setting a property value from two different places, with conflicting values.
8. Setting a property value from two different initial blocks. Process ordering may affect which occurs first.
9. Config database lookup is expensive. Putting a lookup in the inner loop, controlled by a clock. Very slow.

B. *The uvm_config_db::set() algorithm*

```
static function void set(uvm_component cntxt,
                         string inst_name,
                         string field_name,
                         T value);

 Generate An Instance Name.

 Get or create the Resource in the Pool.

 Calculate the Precedence based on the current phase

 Set the Resource Value
 Put the Resource in the Resource Pool Queue of Resources

endfunction
```

VI. Decision Making – lookup_name()

One of the routines used to find a configuration setting is uvm_resource::lookup_name(). The UVM 1.1d version is shown below. It has three sections. The first section checks to see if the name is empty. If the name is empty, then an empty queue is returned. The second section checks to see if the 'name' exists in the associative array. If the name does not exist, then an empty queue is returned. The third section is most interesting. The third section loops through all the elements in the queue associated with the 'name'. Each element in the queue is a 'resource'. If the type name is null and the scope matches, or the type handles match and the scope matches, then the resource is pushed into a queue. That queue is returned. There are more checks for precedence and priority in other parts of the code.

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

  uvm_resource_types::rsrc_q_t rq;
  uvm_resource_types::rsrc_q_t q = new();
  uvm_resource_base rsrc;
  uvm_resource_base r;

  // resources with empty names are anonymous and do not exist in the name map
  if(name == "")
    return q;
    // Does an entry in the name map exist with the specified name?
    // If not, then we're done
    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
```

For the UVM 1.1d Enhanced library the code above has been instrumented to provide detailed information about what is going on. Failures (or mismatches) will be shown, but also, when a match happens, the details of the match will be shown. Matching incorrectly is just as bad as mismatching.

A call to the new macro uvm_info_context_with_file_line will be made. This will allow call site information to be passed on to the user. It is really just a fancy uvm_info.

Each decision point in the code below has been instrumented. The for loop has been instrumented, so that matching and failing queue elements are all printed – if it is processed, then it is printed. This way the behavior of the code can be followed.

```
  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
                                                    );
    uvm_resource_types::rsrc_q_t rq;
    uvm_resource_types::rsrc_q_t q = new();
    uvm_resource_base rsrc;
    uvm_resource_base r;

    // resources with empty names are anonymous and do not exist in the name map
    if(name == "")
      return q;

    // Does an entry in the name map exist with the specified name?
    // If not, then we're done
    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

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

```
// begin/end to create a useful message from:
// if(((type_handle == null) || (r.get_type_handle() == type_handle)) && r.match_scope(scope))

     begin
       string msg;
       msg = $sformatf("  Item #%2d: P:%4d ", i, r.precedence);

       // Type handle message
       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() };
       // Scope message
       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)
     end

     // Do the actual compare/check.
     // 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
```

VII. CODE CHANGES FOR BETTER DEBUG

The routines below have had CONTEXT, FILE and LINENUMBER arguments added. This enables MUCH better debug messages tracking the file and line number and context of the caller – not the file, line number and context of the internal UVM files and structures. And the output printed from the Fancy Dump routine is much easier to read for humans. Additionally, each new resource or resource update is tracked in a queue. It is then printed in order, so the exact ordering of all gets or sets can be seen.

```
 function void record_read_access(uvm_object accessor = null,
                                                  input uvm_object CALLING_CONTEXT = null,
                                                  input string FILE = "",
                                                  input int    LINE = 0);

 function void record_write_access(uvm_object accessor = null,
                                                  input uvm_object CALLING_CONTEXT = null,
                                                  input string FILE = "",
                                                  input int    LINE = 0);

 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 CONTEXT = null,
                                                  input string FILE = "",
                                                  input int    LINE = 0  );

 function uvm_resource_base get_highest_precedence(ref uvm_resource_types::rsrc_q_t q,
                                                  input string FILE = "",
                                                  input int    LINE = 0  );
```

```
function uvm_resource_types::rsrc_q_t lookup_regex_names(string scope,
                                                         string name,
                                                        uvm_resource_base type_handle = null,
                                                   input uvm_object CONTEXT = null,
                                                   input string FILE = "",
                                                   input int    LINE = 0  );

function T read(uvm_object accessor = null,
                                                  input uvm_object CALLING_CONTEXT = null,
                                                  input string FILE = "",
                                                  input int    LINE = 0  );
```

VIII. BETTER "CONFIG.PRINT()"

The UVM can print a summary of the configuration interaction – it dumps the access database. We've augmented the access database with the fields in RED below, adding typename, fullname, calling context, file and line number, as well as a handle back to the resource itself. These fields allow for a more complete message.

```
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;
```

The data structure was sufficient for this paper. It may need to be changed to capture all the semantics of multiple accesses created from different scopes at different times. That is left as an exercise.

The first part of the printed summary output is similar to the original configuration debug messages, but with the details organized in an easier to read way:

A. Fancy Dump Output:

```
Found resource for name 'A', with 1 elements.
    Item # 0: P:1000 Type handle ((string) A-value), Scope (/^uvm_test_top$/)
        Access uvm_test_top reads: 0 @0, writes: 1 @100 (uvm_test_top) [t.sv:167]
Found resource for name 'B', with 1 elements.
    Item # 0: P:1000 Type handle ((string) B-value), Scope (/^uvm_test_top\..*$/)
        Access uvm_test_top reads: 0 @0, writes: 1 @100 (uvm_test_top) [t.sv:168]
Found resource for name 'C', with 1 elements.
    Item # 0: P:1000 Type handle ((string) C-value), Scope ()
        Access  reads: 0 @0, writes: 1 @100 (uvm_test_top) [t.sv:169]
Found resource for name 'D', with 1 elements.
    Item # 0: P:1000 Type handle ((string) D-value), Scope (/^.*$/)
        Access  reads: 0 @0, writes: 1 @100 (uvm_test_top) [t.sv:170]
Found resource for name 'recording_detail', with 2 elements.
    Item # 0: P:1000 Type handle ((reg signed[4095:0]) 1), Scope ()
        Access  reads: 0 @0, writes: 1 @0 (EMPTY) [:0]
        Access uvm_test_top reads: 1 @0, writes: 0 @0 (uvm_test_top)
        Access uvm_test_top.e1 reads: 1 @0, writes: 0 @0 (uvm_test_top.e1)
        Access uvm_test_top.e1.a1 reads: 1 @0, writes: 0 @0 (uvm_test_top.e1.a1)
        Access uvm_test_top.e1.a1.b1 reads: 1 @0, writes: 0 @0 (uvm_test_top.e1.a1.b1)
        Access uvm_test_top.e1.a1.b1.c1 reads: 1 @0, writes: 0 @0 (uvm_test_top.e1.a1.b1.c1)
        Access uvm_test_top.e1.a1.b1.c2 reads: 1 @0, writes: 0 @0 (uvm_test_top.e1.a1.b1.c2)
        Access uvm_test_top.e1.a1.b2 reads: 1 @0, writes: 0 @0 (uvm_test_top.e1.a1.b2)
        Access uvm_test_top.e1.a1.b2.c1 reads: 1 @0, writes: 0 @0 (uvm_test_top.e1.a1.b2.c1)
        Access uvm_test_top.e1.a1.b2.c2 reads: 1 @0, writes: 0 @0 (uvm_test_top.e1.a1.b2.c2)
        Access uvm_test_top.e1.a2 reads: 1 @0, writes: 0 @0 (uvm_test_top.e1.a2)
        Access uvm_test_top.e1.a2.b1 reads: 1 @0, writes: 0 @0 (uvm_test_top.e1.a2.b1)
```

The second part of the printed summary output is a list of ALL access requests, in order. This is the order that reads and writes occurred during simulation.

B. *Fancy Dump (in order), with 'recording_detail' removed for simplicity*

```
1:  Access WRITE  reads: 0 @0, writes: 1 @0 (EMPTY) [t.sv:196]
     Scope (/^.*e1$/), Type handle ((virtual intfA) /top/vifA1) Value (vifA)
3:  Access WRITE  reads: 0 @0, writes: 1 @0 (EMPTY) [t.sv:197]
     Scope (/^.*e2$/), Type handle ((virtual intfA) /top/vifA2) Value (vifA)
4:  Access WRITE  reads: 0 @0, writes: 1 @0 (EMPTY) [t.sv:198]
     Scope (/^.*e1$/), Type handle ((virtual intfB) /top/vifB1) Value (vifB)
5:  Access WRITE  reads: 0 @0, writes: 1 @0 (EMPTY) [t.sv:199]
     Scope (/^.*e2$/), Type handle ((virtual intfB) /top/vifB2) Value (vifB)
7:  Access WRITE uvm_test_top reads: 0 @0, writes: 1 @0 (uvm_test_top) [t.sv:157]
     Scope (/^uvm_test_top$/), Type handle ((string) testA::blue-run) Value (sky)
10:  Access WRITE uvm_test_top.e1 reads: 0 @0, writes: 1 @0 (uvm_test_top.e1) [t.sv:119]
     Scope (/^uvm_test_top\.e1$/), Type handle ((string) env::blue-run) Value (sky)
11:  Access READ uvm_test_top.e1 reads: 1 @0, writes: 0 @0 (uvm_test_top.e1) [t.sv:125]
     Scope (/^.*e1$/), Type handle ((virtual intfA) /top/vifA1) Value (vifA)
12:  Access READ uvm_test_top.e1 reads: 1 @0, writes: 0 @0 (uvm_test_top.e1) [t.sv:127]
     Scope (/^.*e1$/), Type handle ((virtual intfB) /top/vifB1) Value (vifB)
15:  Access WRITE uvm_test_top.e1.a1 reads: 0 @0, writes: 1 @0 (uvm_test_top.e1.a1) [t.sv:90]
     Scope (/^uvm_test_top\.e1\.a1$/), Type handle ((string) A::blue-run) Value (sky)
18:  Access WRITE uvm_test_top.e1.a1.b1 reads: 0 @0, writes: 1 @0 (uvm_test_top.e1.a1.b1) [t.sv:65]
     Scope (/^uvm_test_top\.e1\.a1\.b1$/), Type handle ((string) B::blue-run) Value (sky)
21:  Access WRITE uvm_test_top.e1.a1.b1.c1 reads: 0 @0, writes: 1 @0 (uvm_test_top.e1.a1.b1.c1) [t.sv:30]
     Scope (/^uvm_test_top\.e1\.a1\.b1\.c1$/), Type handle ((string) C::blue-build) Value (sky)
22:  Access READ uvm_test_top.e1.a1.b1.c1 reads: 1 @0, writes: 1 @0 (uvm_test_top.e1.a1.b1.c1) [t.sv:31]
     Scope (/^uvm_test_top\.e1\.a1\.b1\.c1$/), Type handle ((string) C::blue-build) Value (sky)
23:  Access WRITE uvm_test_top.e1.a1.b1.c2 reads: 0 @0, writes: 1 @0 (uvm_test_top.e1.a1.b1.c2) [t.sv:30]
     Scope (/^uvm_test_top\.e1\.a1\.b1\.c2$/), Type handle ((string) C::blue-build) Value (sky)
24:  Access READ uvm_test_top.e1.a1.b1.c2 reads: 1 @0, writes: 1 @0 (uvm_test_top.e1.a1.b1.c2) [t.sv:31]
     Scope (/^uvm_test_top\.e1\.a1\.b1\.c2$/), Type handle ((string) C::blue-build) Value (sky)
25:  Access WRITE uvm_test_top.e1.a1.b2 reads: 0 @0, writes: 1 @0 (uvm_test_top.e1.a1.b2) [t.sv:65]
     Scope (/^uvm_test_top\.e1\.a1\.b2$/), Type handle ((string) B::blue-run) Value (sky)
28:  Access WRITE uvm_test_top.e1.a1.b2.c1 reads: 0 @0, writes: 1 @0 (uvm_test_top.e1.a1.b2.c1) [t.sv:30]
     Scope (/^uvm_test_top\.e1\.a1\.b2\.c1$/), Type handle ((string) C::blue-build) Value (sky)
29:  Access READ uvm_test_top.e1.a1.b2.c1 reads: 1 @0, writes: 1 @0 (uvm_test_top.e1.a1.b2.c1) [t.sv:31]
     Scope (/^uvm_test_top\.e1\.a1\.b2\.c1$/), Type handle ((string) C::blue-build) Value (sky)
30:  Access WRITE uvm_test_top.e1.a1.b2.c2 reads: 0 @0, writes: 1 @0 (uvm_test_top.e1.a1.b2.c2) [t.sv:30]
     Scope (/^uvm_test_top\.e1\.a1\.b2\.c2$/), Type handle ((string) C::blue-build) Value (sky)
31:  Access READ uvm_test_top.e1.a1.b2.c2 reads: 1 @0, writes: 1 @0 (uvm_test_top.e1.a1.b2.c2) [t.sv:31]
     Scope (/^uvm_test_top\.e1\.a1\.b2\.c2$/), Type handle ((string) C::blue-build) Value (sky)
```

IX. The UVM 1.2 Changes to Config

The UVM 1.2 changed the uvm_config_db and uvm_resource in small ways. Hooks were added to use the coreservices for get() and set(). Coreservices is beyond the scope of this paper. $display() statements were replaced with calls to `uvm_info() for print() functionality. The biggest change is that wildcards are no longer special in the "name" field. No wildcard matching is done. This is a good change, since with many configuration settings, using a wildcard in the name might make the simulation run very slowly. This change may make your simulations stop working – since your configuration may be in the wrong place. Additionally some code was changed to try to achieve better random stability.

X. UVM Configuration

A. *Historical Simply Better Config*

Adding debug, and cleaning up the existing code is a noble effort, but one which may not yield a good enough improvement to clarity and ease of use. We suggest something much, much simpler for 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.

This suggested replacement will cause some changes – no more are the semantics of lookup changed between build and run phases. No more are wildcards causing slow behavior. No more are there precedence rules needed about which conflicting setting wins (in fact conflicting or repeated settings should be an error). We suggest that these "limitations" are actually advantages for ease of use, predictability of results and debug-ability.

B. Simple Config – Too Simple?

There was a phone call many years ago, during the summer. "How about a new configuration database? We'll call it a resource database, and it will be simple and fast. What do you think?" There was worry that this database would be used as a substitute for connectivity. Imagine getting a components connections by call config::get(). Each get() could return a class handle of another UVM object. Then that object handle can be used to call functions. Handle.Function1(), Handle.Function2(). Each of these calls is communication between objects. There was worry and a few sleepless nights. Luckily that nightmare scenario never occurred. What did occur was a layering of uvm_config_db on top to the resource data structure, due to some requirements that may be now out of favor or at least avoided.

By investing a few minutes, we believe that you the enthusiastic reader can create a simpler configuration system that meets whatever requirements you might have. For example, something as simple as a typed, global associative array:

```
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
```

XI. Conclusion

Configuration is unavoidable. Without better debugging and tracing in the UVM, it is best to limit the use of UVM Config. There are more than 3000 lines of code involved in the configuration database. That code is quite complex, and has many interdependencies and special behaviors. Our hope is that a new configuration mechanism can be created which is much simpler and more transparent. In the meantime, some simple rules will help. 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. Following these simple guidelines will make your configurations faster and easier to debug. Happy Debugging! Source code changes for UVM 1.1d are available from the authors. Changes to UVM 1.2 should be trivially the same.

References

[1] Hannes Nurminen, Satya Durga Ravi, "Hierarchical Testbench Configuration Using uvm_config_db", June 2014, https://www.synopsys.com/Services/Documents/hierarchical-testbench-configuration-using-uvm.pdf

[2] Vanessa R. Cooper, Paul Marriott, "Demystifying the UVM Configuration Database", DVCON 2014, http://www.verilab.com/files/configdb_dvcon2014.pdf

[3] John Aynsley,"Easier UVM", March 2011, https://www.doulos.com/knowhow/sysverilog/uvm/easier_uvm/configuration/ and https://www.doulos.com/knowhow/video_gallery/#anchor44

[4] Keisuke Shimizu, "UVM Tutorial for Candy Lovers – 13. Configuration Database", July 24, 2016, http://cluelogic.com/2012/11/uvm-tutorial-for-candy-lovers-configuration-database/

This paper was originally presented at DVCon Europe 2016, Munich, Germany.