Avoiding Configuration Madness The Easy Way

Rich Edelman
Siemens EDA, Fremont, CA

***Abstract*-The SystemVerilog [1] UVM [2] configuration system is widely used and poorly understood. A simpler more transparent system is proposed. It has desirable properties like being easy to read and easy to debug, with a simple implementation using a common base class for all settings. This new system supports setting values by name with a regular expression "scope" match, but eliminates the type-based matching. It is also significantly faster.**

I. INTRODUCTION

A. What is the UVM Configuration Database?

The UVM Configuration database is a way to set values – configuration values or settings or knobs for use in a UVM test. The UVM Configuration database has many features and functionalities, which are not a part of this paper. This is not an exhaustive tour through the UVM Configuration API and use model.

The basic idea of the configuration database is to "set" a value, and later "get" the value in the place where it is needed.

The UVM Configuration setting takes the form

```
uvm_config#(T)::set(scope, instance_path_name, field_name, value);
```

Where value is of type ‘T’.

The getting takes the form

```
uvm_config#(T)::get(scope, instance_path_name, field_name, value);
```

There are other ways to set and get values, but this is the preferred method

B. What is the configuration database used for?

In a UVM test, the configuration database is used to setup values, and to pass things from the "module/instance" world to the dynamic "class" world. For example, from the module "top" to the UVM testbench data structures. Specifically, the configuration database is often used to pass the virtual interfaces that are used to connect the UVM testbench to the actual device-under-test.

There are hooks in the UVM Configuration database to automatically set the values of class member variables which are defined using the field automation macros. See the Appendix for a short tour through that code with a discussion and commentary. [XVI]. There are many other parts of the UVM configuration database which appear to be "helper functions" to help improve automation and productivity.

C. Problems with the UVM Configuration?

The code is big. The UVM Configuration database is implemented mostly in 3 files, containing a total of about 2600 lines of code. That’s a lot of code for something that conceptually is a collection of global variables, looked up by name.

The code is tricky. Small typos or mistakes with setting values or getting values can be very hard to debug. The UVM Configuration database is implemented as a parameterized class, making type matching important when it really should not be. To "get" a value, the scope, full name, property name and the type must match exactly. An 'int' is not a match with a 'bit[4095:0]'.

The issue arose quickly with the built-in ‘recording_detail’. In one implementation it was an ‘int’, in another it was ‘bit [4095:0]’. To fix this potential problem with type mismatch, the UVM looks up the setting for a

‘recording_detail’ value using two different types. Each component in a UVM testbench calls this if-else when it is constructed:

```
// Do local configuration settings
if (!uvm_config_db #(uvm_bitstream_t)::get(this, "", "recording_detail", recording_detail))
  void'(uvm_config_db #(int)::get(this, "", "recording_detail", recording_detail));
```

This is harmless but points out the issue. The types must match EXACTLY.

The code is slow. At least it gets slow for large numbers of sets and gets. Using wildcard matching is prohibitively expensive. Some users have 1000’s of calls to set() using names with wildcards. Those configuration set() calls can take 20 or 30 minutes to finish the ‘set’ and ‘get’ phase.

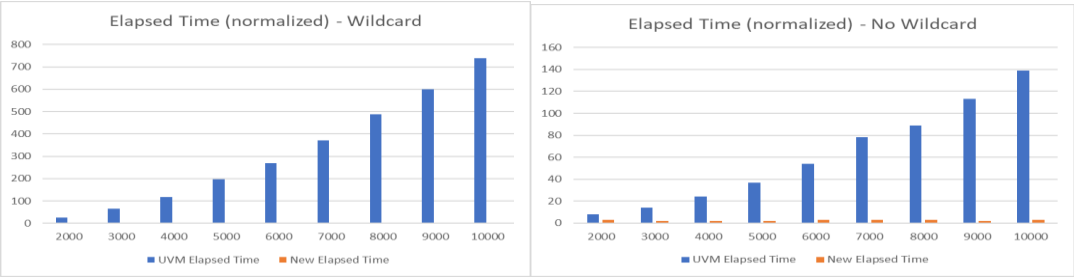

Figure 1 - Elapsed time for set()/get() with and without a wildcard

The two graphs above show UVM Configuration perfomance in blue and an alternative approach presented later in orange. The New solution runs very fast – almost instantly. The Y-axis in the graphs can be interpreted in seconds – the experiment with 5000 sets and 5000 gets takes about 200 seconds using the UVM Configuration. With the alternate approach it takes 2-3 seconds. For 10,000 sets and gets, the UVM Configuration takes more than 6 minutes and the alternative takes 2-3 seconds. For a 5 minute test, that performance issue can matter. For small numbers of sets and gets, it does not matter. The lesson learned is to call ‘set()’ as few times as possible and don’t use wildcards very often.

The left chart is with wildcards. The right chart is with no wildcards. For 5000 sets and gets, the no wildcard is about 40 seconds. For 5000 sets and gets with wildcards, the elapsed time is 200 seconds. The alternate approach is 2-3 seconds in either case.

II. EXPLANATIONS

The UVM Configuration system was developed many years ago, and the design goals are not available to the author. It’s hard to know why certain things are the way they are. But some things appear clear from user usage.

Requirements/Goals

Set values with instance path names and a property name. This represents the following pseudo-assignment.

```
top.a.b.c.d.monitor1.SPEED = 40
or
set("top.a.b.c.d.monitor1", "SPEED", 40)
```

Get values from inside the UVM testbench by using the instance name or other path name and property name. This is the monitor for example, asking what the current SPEED should be.

```
speed = get("top.a.b.c.d.monitor1", "SPEED")
or
get("top.a.b.c.d.monitor1", "SPEED", speed)
```

Use wildcards to mean "hierarchically match paths below here". For example, the set below means "all get() requests that match the regex ‘top.a.b.’ will get the value ‘60’"*

```
set("top.a.b.*", "SPEED", 60)
```

Be able to debug problems easily. For example, where was the set() called? Where was the get() called? Print all the settings. Print a specific setting.

III. SOLUTIONS

It is possible to be successful with the UVM Configuration database. Below are ideas to ensure success.

A. Don't use a UVM Configuration database at all, instead, use a global variable

Really, the simple way to avoid any issues with UVM Configuration is to not use them. Just use a simple global variable. It's not very portable, but it is simple.

```
int global_i;

class C;
  int i;

  function new();
    i = global_i;
  endfunction
endclass

module top();
  initial begin
    global_i = 5;
  end
endmodule
```

Being a little more creative, the global variable could be a container class, with a global varable – a handle to a global configuration class.

```
class configuration;
  int i;
endclass

configuration config_h;

class C;
  int i;

  function new();
    i = config_h.i;
  endfunction
endclass

module top();
  int i;

  initial begin
    config_h = new();
    config_h.i = 5;
  end
endmodule
```

Using a container class is highly recommended. Create a container and populate it will all the knobs and settings. Configuration classes can be hierarchical – a container class could contain other "lower-level" configurations. With a container class, many configuration settings can be passed over as one.

B. Use UVM Configuration – but just once

Do use the UVM Configuration class, but call set() just one time, and pass a large container across. This has the benefit of being "regular" – it uses the UVM Configuration class, but, it avoids the issues, since it is just used once. It uses the normal UVM Configuration system but it calls set() just once and it calls get() just once. The configuration is a hierarchical container of configurations.

C. *Use UVM Configuration – but just a few times*

An extension to the recommendation above is to only call set() a few times. The fewer, the better. Call set() and get() a handful of times. This will not expose the performance issues. Don't use wildcards if possible.

D. *Using a configuration tree*

Using a hierarchical configuration is easy – each configuration contains lower level configurations. It is the responsibility of the higher level testbench component to set the lower level testbench component configuration.

Create a top-level configuration class. It is made up of settings and contains "lower level" configurations. The lower-level configurations are made up of settings and still lower level configurations. At each level in the testbench, a configuration is assigned from the parent

```
lower_level_component.config = config_for_element_ABC;
```

Then the lower level continues, assigns lower levels, repeating down the hierarchy. The UVM Configuration is used once to register the top of the configuration tree.

E. *Use something different*

The "solutions" above - suggestions and recommendations above are not new. They have been discussed over the years many times. They are well documented. They should be used where possible.

The remainder of this paper covers a new configuration database system. It shares the same set() and get() API as the UVM Configuration database. All configuration settings are classes derived from ‘config_item’. It removes the parameterization of the configurations. It removes precedence and removes the lookup algorithm change, depending on whether this is early or late in the testbench. It tries to be simple and transparent. It is blazingly fast – even using regex. It is implemented in about 300 lines of code.

IV. A NEW CONFIGURATION ITEM

A configuration is always a derived class from ‘config_item’. All configurations or values or properties are defined in the derived class. In the configuration setting below the ‘int value’ is the property being set.

Setting individual integers isn’t a good idea in the UVM Configuration database and is not a good idea here. Group settings together and put them into a "configuration item" of your own design. This my_special_config_item might contain 20 or 30 configuration settings (or knobs).

```
class my_special_config_item extends config_item;

  function new(string name = "my_special_config_item");
    super.new(name);
  endfunction

  int value;

  virtual function string convert2string();
    return $sformatf("%s - value=%0d <%s>", get_name(), value, super.convert2string());
  endfunction
endclass
```

Each configuration defined should define the variables – the settings and a pretty-print function for debug. The specific configuration inherits from the common library base class – config_item.

V. THE LIBRARY CONFIGURATION ITEM

In the new config library, the ‘config_item’ is defined simply. It has a name attribute. This is the name of the configuration. A name like "setting" or "ID" or "SPEED". It is the name that will be looked up, along with the instance name.

The config_item has a class member called the ‘accessor_q’. It is a queue of accesses. Any set() or get() call pushes an "accessor" structure onto the queue. All accesses are recorded. The accessor structure is simple, file name, line number and the "kind" (GET or SET). The accessor structure. More on accessors in a section below.

```
typedef struct {
  string kind; // GET or SET
  string file_name;
  int line_number;
} accessor_t;
```

The config_item has a small api as well. The get_name() function returns the name. Convert2string pretty prints the value. The built-in convert2string prints the accessor queue automatically.

```
class config_item

  accessor_t accessor_q[$];

  string name;

  virtual function string get_name();
     return name;
  endfunction

  function new(string name = "config_item");
    this.name = name;
  endfunction

  virtual function string convert2string();
    return $sformatf("%p", accessor_q);
  endfunction
endclass
```

That’s a configuration object.

VI. THE CONFIGURATION DATABASE

The configuration database is quite simple. It is a two-dimensional associative array of queues of configuration items. It is an associative array, whose indices are the strings instance_name and field_name. Finding a "instance_name.field_name" is trivial. Just use the associative array.

It is an associative array because lookup and creation is fast and simple. It is two dimensional because it has two indices – ‘instance_name’ and ‘field_name’.

Each item in the array is a queue of configuration items because each instance_name, field pair could have multiple settings. This is likely an error – or at least the source of a possible error but is supported in the new library since the UVM Configuration database also supported it.

Defining the 'type' of the configuration database is simple.

```
typedef config_item table_of_fields_t[string][string][$];
```

The config_db class itself is also simple. It declares the 'table_of_fields'.

```
class config_db extends uvm_object;
  `uvm_object_utils(config_db)

  function new(string name = "config_db");
    super.new(name);
  endfunction

  table_of_fields_t table_of_fields;
```

A config_db has a set() routine. See the set() implementation below. The set() routine takes a scope which is ignored in this implementation for simplicity. In the UVM Configuration database, the scope and the instance_name work together to form an instance name. Instance name is the key. The UVM Configuration database looks up an instance_name.field_name. The new config_db does the same. The following code implements this magic in the UVM. It uses the get_full_name() or it uses the context or it uses the top or a combination.

```
if(cntxt == null)
  cntxt = top;
if(inst_name == "")
  inst_name = cntxt.get_full_name();
else if(cntxt.get_full_name() != "")
  inst_name = {cntxt.get_full_name(), ".", inst_name};
```

Implementing this magic in the new config_db is an exercise left for the reader.

A simple set() routine was implemented as below.

```
function void set(uvm_component scope, string instance_name,
                                        string field_name, config_item c,
                                        string file_name, int line_number);
   container container_h;
   table_of_fields[field_name][instance_name].push_front(c);
   container_h = root_container.make_container(instance_name);
   container_h.configs_for_a_field[field_name].push_front(c);
  add_accessor("SET", c, file_name, line_number);
endfunction
```

The 'scope' argument is ignored. The field_name and the instance_name are used to index the table_of_fields, which is a q. The configuration item is pushed onto the queue.

Then a 'container' gets made. This is a replica hierarchical tree used to speed wildcard matching. More on it below. A container has a queue indexed by field_name. This is an associative array of queue – one queue per field name.

The get() implementation is simple. Just get the first item in the queue. There is some checking for undefined entries, but it is really that simple.

```
      c = table_of_fields[field_name][instance_name][0];


function config_item get(uvm_component scope, string instance_name,
                                              string field_name,
                                              input string file_name, int line_number);
  if ((table_of_fields.exists(field_name) == 0) ||
      (table_of_fields[field_name].exists(instance_name) == 0)) begin
    c = null;
  end
  else if (table_of_fields[field_name][instance_name].size() > 0)  begin
      c = table_of_fields[field_name][instance_name][0];
  end

  ...
endfunction
endclass
```

That's it. A very simple database with a simple set() and get() interface with "tracing" built-in.

VII. ACCESSOR – TRACEABILITY

Answering the question "who set this"? Or "who called get()"? Can be answered with the new implementation easily. A File name and line number are available in the set() and get() function calls. Those can be supplied by the user, or a macro can be used to automatically insert those last two arguments.

A user could call set() and get() and fill in the file name and line number using the Verilog defines `__FILE__ and `__LINE__.

```
set(null, "top.a.b.*", "SPEED", my_speed_config, `__FILE__, `__LINE__)
get(null, "top.a.b.c.d.monitor1", "SPEED", speedconfig, `__FILE__, `__LINE__)
```

Using a macro makes it simple to use the FILE and LINE macros.

```
`define CONFIG_SET(scope, instance_name, field_name, value) \
                set(scope, instance_name, field_name, value, `__FILE__, `__LINE__)
`define CONFIG_GET(scope, instance_name, field_name, value) \
                get(scope, instance_name, field_name, value, `__FILE__, `__LINE__)

my_speed_config_t my_speed_config;

config_db_h.`CONFIG_SET(null, "top.a.b.*", "SPEED", my_speed_config)
```

Later

```
my_speed_config_t speedconfig;

$cast(speedconfig, config_db_h.`CONFIG_GET(null, "top.a.b.c.d.monitor1", "SPEED"))
```

Other macros could be imagined, included using static functions – but the code above is simple and easy to use.

Understanding who called set() and get() is as simple as printing the "accessor queue". The accessor queue can be printed or queried during debug – it's a simple data structure.

The accessor queue is a SystemVerilog queue of structs. Each struct indicated "GET" or "SET", the filename and line number. Each config item has such a queue.

```
typedef struct {
  string kind; // GET or SET
  string file_name;
  int line_number;
} accessor_t;
```

Finding out who accessed a given configuration item is as simple as

```
$display("Item Accessors: %p", my_config_item.accessor_q);
```

Printing out the entire config_db is easy as well. Each field_name, instance_name pair is iterated. The value of the config is printed with convert2string(). Then the queue of config_items is gone through one-by-one, and in each config_item the accessors are all printed.

```
function void print();
  foreach (table_of_fields[field_name,instance_name]) begin
    config_item_q_t config_item_q;
    config_item_q = table_of_fields[field_name][instance_name];
    $write("FIELD TABLE INFO: %s.%s is ", instance_name, field_name);

    for (int i = 0; i < config_item_q.size(); i++) begin
      config_item ci;
      ci = config_item_q[i];
      if (i == 0)
        $write("%s", ci.convert2string());

      for (int j = 0; j < ci.accessor_q.size(); j++) begin
        accessor_t accessor;
        accessor = ci.accessor_q[j];
        $write("<%s,line %0d> %s, ",
                    accessor.file_name, accessor.line_number, accessor.kind);
      end
    end
    $display("");
   end
 endfunction
```

VIII. More New Configuration Database

The implementation above does not call regular expression matching and does not work for wildcards. It works for regular path names – no wildcards.

But wildcards must be supported. In the UVM Configuration database, there is essentially a list of all the set() commands – instance and field names. When a get() is called, that list is iterated, trying to match the instance name and field in the get(). All the various 'set()' instance names are iterated. That is one reason it is so slow. It mush check each item in the list.

The set() establishes or builds the list and the get() iterates the list, trying to match. It tries to match the entire list. It iterates the entire list. If a wildcard is used it builds a regex and does regular expression matching. This is very expensive.

The new configuration database does all lookup with associative arrays which are blazing fast to setup and use for search.

But wildcards are not supported. Since we must support wildcards, and additional lookup mechanism is needed – containers.

IX. Small "Component" containers

An instance name represents a hierarchy – like the UVM components. For example, 'top.a.b.c.d.*_0', but the last item is not a name – it is a wildcard. An additional kind of search is built – a small component tree of containers. This tree is small and only exists for get() calls that need to match a wildcard. When set() is called, a container tree is built. When get() is called, the container tree is referenced.

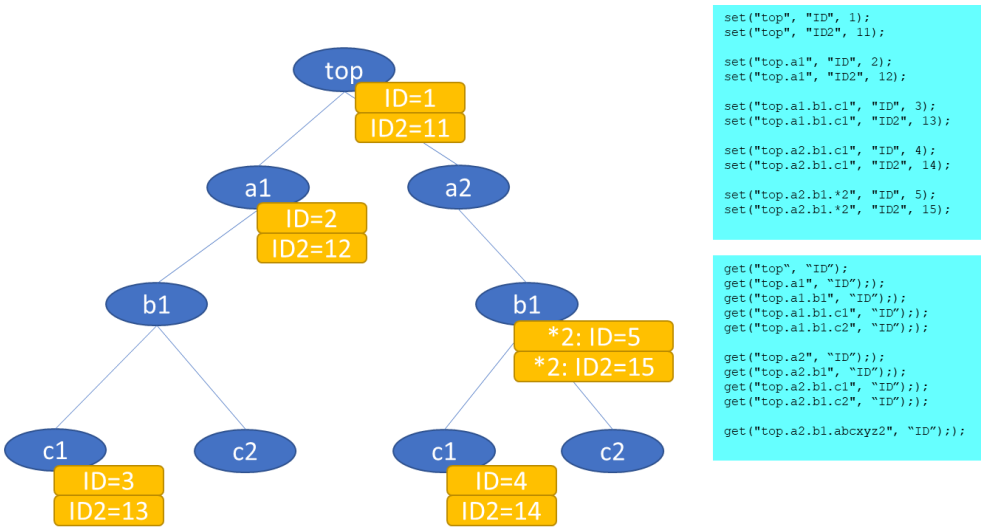

Figure 2 - Container Tree

If the initial, simple associative array lookup fails, then the container tree is traversed, matching the parts of the instance path. By trying to match each element in the hierarchical path, the search list is shortened.

First find_container() tries to find a container match in the tree. Down at the bottom. It uses regular expression matching, and parses the instance name into elements. It continues down the tree until there is no match or until the entire instance name has been matched. If there is no match, it simply means this get() will fail – it will not find a setting, since no set() created this instance name.

If a container is found, then it checks to see if a field (property) exists. If it does not exist, then it climbs up the tree. This climbing the tree represents inheritance. If a lower level instance name does not have a field set, then climb up the tree to find a setting. The climbing the tree allows finding the higher level settings or higher wildcards. The settings that control an entire sub-tree.

```
if (c == null) begin
  // Wasn't found in the easy lookup. Use the name, to look in the container database
  container container_h;
  container_h = root_container.find_container(instance_name);

  if (container_h) begin
    // There's a container. See if there is a property set.
    c = container_h.configs_for_a_field[field_name][0];
    while (c == null) begin
      // Climb the tree, since the config_item is null.
      container_h = container_h.parent;
      if (container_h == null) break; // We reached the top. Nothing.
      c = container_h.configs_for_a_field[field_name][0];
    end
  end
end
```

X. FIND_CONTAINER

The container class is left for inspection in the appendix. It is a simple class that builds a hierarchy tree given an instance name path.

But the container class find_container() function is of interest here. Find_container() tries to walk down the container tree and find a matching container, given an instance path that might be a container.

First, the find_container() code starts at the top of the containers and tries to match the instance path name one hierarchy at a time.

If the matching continues, the tree is traversed, and the proper container is returned.

If the matching fails, that means a hierarchy name in the instance name doesn't exist in the container tree. That is either because there is no such name, or the container tree name is really a wildcard, and we should do a regex match.

The children of the container are iterated and matched with a regex call. If one of the container children matches with the instance hierarchy name, then we can continue below.

If no container child matches this hierarchy in the instance name, try one last thing. Rebuild the instance hierarchy name from 'here' down. This means the remaining, unmatched hierarchy is turned back into a string. The regex then is checked against the container children with this full string.

```
typedef string string_q[$]; // A string path name gets turned into a queue of names

function  container find_container(string full_path_instance_name);
  bit done;
  string_q q;
  container new_c, container_h;

  q = path_name_to_q(full_path_instance_name);
  new_c = null;
  if (list_of_tops.exists(q[0])) begin
    for (int i = 1; i < q.size(); i++) begin
      done = 0;
      if (i==1) container_h = list_of_tops[q[0]];
      else      container_h = new_c;

      // 1. Lookup the name.
      new_c = container_h.get(q[i]);

      // 2. Try regex match ANY of the containers here
      if (new_c == null) begin
        // The the NAME isn't found. Do any of the containers children regex match?
        foreach (container_h.list_of_children[s]) begin
          number_of_regex_tries++;
          if (uvm_re_match(uvm_glob_to_re(s), q[i]) == 0) begin // {
            // Matched. Need to keep looping. We matched a child. Traverse it. Not done yet.
            new_c = container_h.list_of_children[s];
            // Does this new container itself have children? If not there's no hope...
```

```
            if (new_c.list_of_children.size() > 0) // If > 0, then there are children
              break;
            else
              new_c = null; // There are no children. Keep looping here.
          end
        end
      end

      // 3. Try the FULL string
      if (new_c == null) begin
        string fstring;
        // OK. We went through the for loop, but no child matched. Get ready to give up.
        // Can we match the whole string against any of the "children"?
        fstring = "";
        for (int j = i; j < q.size(); j++)
          if (fstring == "")
            fstring = q[j];
          else
            fstring = { fstring, ".", q[j]};
        foreach (container_h.list_of_children[s]) begin
          number_of_regex_tries++;
          if (uvm_re_match(uvm_glob_to_re(s), foo) == 0) begin
            // Matched. Done.
            new_c = container_h.list_of_children[s]; // This is the one.
            done = 1;
            break;
          end
        end
      end
      if (done) break;
    end
  end
  return new_c;
endfunction
```

The clever reader may have realized that the original "lookup in an associative array" is supplanted by this container lookup. Both are not needed. The first lookup does a lookup on the full instance name in an associative array. The container lookup instead looks up each level of the hierarchy in a smaller associative array in each hierarchy level. Both implementations are included. Hopefully, sometime in the future wildcards will not be used or supported and lookup can be simple and fast.

XI. TEST PROGRAM

A simple program to test the old API and the new API can be used. Below, the code creates a configuration object and sets the ‘value’. Then ‘set()’ is called using the macro. ‘get()’ is called with a macro, and the values are compared.

Object setup

```
int rid, wid;
my_special_config_item read_c;

c = new($sformatf("my_special_config_item%0d", gid));
c.value = -99999999;
```

New implementation set(), get() and compare

```
// 1. Set
name = "uvm_test_top.t_1.*";
config_db_h.`CONFIG_SET(null, name, "XXX", c);

// 2. Get
name = "uvm_test_top.t_1.abc.xyz.2.3.4";
$cast(read_c, config_db_h.`CONFIG_GET(null, name, "XXX"));

// 3. Compare
```

```
wid = c.value;
rid = xc.value;
if (rid != wid) $display("Error: %0d != %0d (%s)", rid, wid, name);
else            $display("Match: %0d == %0d (%s)", rid, wid, name);
```

Using the UVM Configuration Database – set(), get() and compare

```
// 1. Set
name = "uvm_test_top.t_1.*";
uvm_config_db#(my_special_config_item)::set(null, name, "ID", c);

// 2. Get
name = "uvm_test_top.t_1.abc.xyz.2.3.4";
uvm_config_db#(my_special_config_item)::get(null, name, "ID", read_c);

// 3. Compare
wid = c.value;
rid = read_c.value;
if (rid != wid) $display("Error: %0d != %0d (%s)", rid, wid, name);
else            $display("Match: %0d == %0d (%s)", rid, wid, name);
```

XII. CONCLUSION

The UVM Configuration database serves a useful purpose – to share data between the module/instance world and the dynamic class-based world in a UVM testbench.

But it should be used sparingly.

If it cannot be used sparingly, then consider something easier to understand and simpler and faster. The package outlined here is about 300 lines and easy to understand. The UVM Configuration database files contain about 2,600 lines of code and is not easy to understand.

The real hope for the reader is a realization that this is all "just code" and some things are better modeled other ways. Using bad habits with the UVM Configuration database will eventually and naturally lead to changes. Either a simple usage or a different implementation. For large numbers of 'set()' calls, the UVM Configuration database consumes a significant amount of time at least compared to other architectures – like the one introduced here.

XIII. REFERENCES

[1] SystemVerilog LRM, "1800-2017 - IEEE Standard for SystemVerilog--Unified Hardware Design, Specification, and Verification Language", https://ieeexplore.ieee.org/document/8299595

[2] UVM 1.1d - https://www.accellera.org/downloads/standards/uvm

[3] UVM Tutorial for Candy Lovers – 13. Configuration Database, http://cluelogic.com/2012/11/uvm-tutorial-for-candy-lovers-configuration-database/, Keisuke Simizu

[4] "Configuration in UVM: The Missing Manual", Mark Glasser, https://dvcon-proceedings.org/wp-content/uploads/configuration-in-uvm-the-missing-manual.pdf

[5] "Demystifying the UVM Configuration Database", Vanessa Cooper and Paul Marriott, https://www.verilab.com/files/configdb_dvcon2014_1.pdf

XIV. APPENDIX – THE CONFIG PACKAGE – THE CONFIG ITEM CLASS

```
package config_pkg;

  import uvm_pkg::*;
  `include "uvm_macros.svh"

  typedef struct {
    string kind; // GET or SET
    string file_name;
    int line_number;
  } accessor_t;


  function int is_path_separator(byte c);
    if (c == "/") return 1;
    else if (c == ".") return 1;
    return 0;
  endfunction
```

```
function bit is_wildcard(byte c);
  if (c == "*") return 1;
  else return 0;
endfunction

function bit name_contains_wildcard(string s);
  for (int i = 0; i < s.len(); i++)
    if (s[i] == "*") return 1;
  return 0;
endfunction

bit verbose = 0;

typedef class container;
container list_of_tops[string];
container root_container = new("/");

int number_of_regex_tries;
int number_of_easy_tries;

class config_item;

  accessor_t accessor_q[$];

  string name;

  function string get_name();
     return name;
  endfunction

  function new(string name = "config_item");
      this.name = name;
  endfunction

  virtual function string convert2string();
    return $sformatf("%p", accessor_q);
  endfunction
endclass

function void add_accessor(string kind, config_item c, string file_name, int line_number);
  accessor_t accessor;
  accessor.kind = kind;
  accessor.file_name = file_name;
  accessor.line_number = line_number;
  c.accessor_q.push_front(accessor);
endfunction

typedef config_item               config_item_q_t[$];                // A queue of config items. A list of them.
typedef config_item_q_t           table_of_instance_names_t[string]; // An associative array of lists, index by instance name
typedef table_of_instance_names_t table_of_fields_t[string];         // An associative array of tables of instance_names,
                                                                     //   indexed by field name

typedef config_item_q_t table_of_field_configs_t[string];

class config_db extends uvm_object;
  `uvm_object_utils(config_db)

  function new(string name = "config_db");
    super.new(name);
  endfunction

  table_of_fields_t table_of_fields;

  function void set(uvm_component scope, string instance_name, string field_name,
                    config_item c, string file_name, int line_number);
     container container_h;
     table_of_fields[field_name][instance_name].push_front(c);
     container_h = root_container.make_container(instance_name);
     container_h.configs_for_a_field[field_name].push_front(c);
    add_accessor("SET", c, file_name, line_number);
  endfunction

  function config_item get(uvm_component scope, string instance_name, string field_name,
                           output config_item c, input string file_name, int line_number);
    // Figure out how to set a default value of empty q for this table.
    if ((table_of_fields.exists(field_name) == 0) || (table_of_fields[field_name].exists(instance_name) == 0)) begin
      c = null;
    end
    else if (table_of_fields[field_name][instance_name].size() > 0)  begin
        c = table_of_fields[field_name][instance_name][0];
        number_of_easy_tries++;
    end

    if (c == null) begin
      // Wasn't found in the easy lookup
      // Use the name, to look in the container database
      container container_h;
      container_h = root_container.find_container(instance_name);
      if (container_h) begin
        // There's a container. See if there is a property set.
        if (verbose) $display("DEBUG: INFO: Checking for field '%s' config set", field_name);
        c = container_h.configs_for_a_field[field_name][0];
        while (c == null) begin
          // Climb the tree, since the config_item is null.
```

```
          container_h = container_h.parent;
          if (container_h == null) break; // We reached the top. Nothing.
          c = container_h.configs_for_a_field[field_name][0];
        end
        if (c != null)
          if (verbose) $display("DEBUG: INFO: Found field '%s' config set in '%s'", field_name, container_h.rprint());
      end
    end

    if ( c != null )
      add_accessor("GET", c, file_name, line_number);
    return c;
  endfunction

  function void print();
    foreach (table_of_fields[field_name,instance_name]) begin
       begin // Show all the configs, so we can see the accessors and values
         config_item_q_t config_item_q;
         config_item_q = table_of_fields[field_name][instance_name];
       end
    end
  endfunction
endclass
```

XV. APPENDIX – THE CONFIG PACKAGE – THE CONTAINER CLASS

```
class container;
  string name;
  container list_of_children[string] = '{default:null};
  container parent;

  table_of_field_configs_t configs_for_a_field; // Field settings here

  function new(string name);
    this.name = name;
  endfunction

  function void print(int indent = 0);
    for (int i = 0; i < indent; i++)
      $write(" ");
    $write("%s\n", name);
    foreach (list_of_children[s])
      list_of_children[s].print(indent+2);
  endfunction

  function string rprint();
    if ((parent == null) || (parent == root_container))
      return name;
    else
      return {parent.rprint(), ".", name};
  endfunction

  function container get(string name);
    container c;
    c = list_of_children[name];
    return c;
  endfunction

  function container add(string name);
    container c;
    c = list_of_children[name];
    if (c == null) begin // The child 'name' doesn't exist here
      c = new(name);
      c.parent = this; // Point up.
      list_of_children[name] = c;
    end
    return c; // Return a new container, or the existing one
  endfunction

  typedef string string_q[$]; // A string path name gets turned into a queue of names

  // Utility. Turn a string path into a Queue.
  function string_q path_name_to_q(string full_path_name);
    string_q q;
    container c;
    string hname;
    int start_i;

    start_i = 0;
    for (int i = 0; i < full_path_name.len()+1; i++) begin // Process the string AND the NULL
      if (is_path_separator(full_path_name[i]) || (full_path_name[i] == 0)) begin
        // Found a path separator
        hname = full_path_name.substr(start_i, i-1); // Grab the name BEFORE the separator
        q.push_back(hname);                          // Put it in the q.
        start_i = i+1;                               // Skip the path separator
      end
    end
    return q;
  endfunction

  function  container find_container(string full_path_instance_name);
    bit matched;
    string_q q;
    container new_c, container_h, bottom_c;
    q = path_name_to_q(full_path_instance_name);
    bottom_c = null;
    if (list_of_tops.exists(q[0])) begin
      container_h = list_of_tops[q[0]];
      for (int i = 1; i < q.size(); i++) begin
        new_c = container_h.get(q[i]);
        if (new_c == null) begin
          // Try regex match
          matched = 0;
          foreach (container_h.list_of_children[s]) begin
            number_of_regex_tries++;
            if (uvm_re_match(uvm_glob_to_re(s), q[i]) == 0) begin
              // Match.
              container_h = container_h.list_of_children[s];
              matched = 1;
              break;
            end
            else begin
              // This child doesn't match. Try the next child
            end
          end
          if (!matched)
```

```
            container_h = null;
          end
          else begin
            // container_h is NOT null. Keep going.
            container_h = new_c;
          end // }
        end
      end
      return container_h;
    endfunction

    function  container make_container(string full_path_name);
      string_q q;
      container c;
      q = path_name_to_q(full_path_name);
      if (list_of_tops.exists(q[0])) begin // This top already exists.
        c = list_of_tops[q[0]];
      end
      else begin // This top doesn't exist.
        c = add(q[0]);
        list_of_tops[q[0]] = c;
      end
      for (int i = 1; i < q.size(); i++)
        c = c.add(q[i]);
      // Return the BOTTOM container - where we'll stuff stuff.
      return c;
    endfunction
  endclass
endpackage
```

XVI. APPENDIX – APPLY_CONFIG_SETTINGS

The UVM contains some very clever, and powerful automation. It's quite sophisticated but summarized simply it works like this. If a property (class member variable) is defined using the field automation macros, and if the build_phase() is used in a certain way, or if apply_config_settings() is used, then that property will be looked up by name and automatically set. The property must be assigned using uvm_config#(T)::set(…). That set() call registers the value of a named property. The sophisticated UVM automation will look it up and then assign that registered value to the class member property.

Implementing the apply_config_settings() using the config_db proposed in this paper would be easy. An apply_config_setting() implementation would lookup the named configurations and apply the settings. Much as it does currently, but it would be fast. That implementation is left to the reader or a future paper.

A. Discussion on UVM apply_config_settings()

When using apply_config_settings and the field automation macros, the user avoids needing to write the following kind of code for each property – the lookup.

```
uvm_config#(T)::get(…"full-path-name", "property-name"), property-name);
```

for example

```
uvm_config#(int)::get(null, "top.a.b.c.d", "bus_width", bus_width);
```

Skipping a one-line implementation for the complexity introduced seems like a poor trade-off.

The field automation macros already have a bad rap. They generate complex, hard to debug code. It's OK to sometimes write code. Automation is good too, but not at the expense of complexity.

Additionally, in the UVM (1.1d and IEEE) the apply_config_settings have a dire warning:

```
// Note: the following is VERY expensive. Needs refactoring. Should
// get config only for the specific field names in 'field_array'.
// That's because the resource pool is organized first by field name.
// Can further optimize by encoding the value for each 'field_array'
// entry to indicate string, uvm_bitstream_t, or object. That way,
// we call 'get' for specific fields of specific types rather than
// the search-and-cast approach here.
```

Furthermore, this entire field_automation and auto-setting is geared to integers, strings or other integral values. Having thousands of integer settings set automatically seems to be solving the wrong problem. The real problem is the thousands of integers. It would be much better to wrap them in a configuration object and do one set() and one get().

B. UVM 1.1d apply_config_settings() from uvm_component.svh:

```
  // Function: apply_config_settings
  //
  // Searches for all config settings matching this component's instance path.
  // For each match, the appropriate set_*_local method is called using the
  // matching config setting's field_name and value. Provided the set_*_local
  // method is implemented, the component property associated with the
  // field_name is assigned the given value.
  //
  // This function is called by <uvm_component::build_phase>.
  //
  // The apply_config_settings method determines all the configuration
  // settings targeting this component and calls the appropriate set_*_local
  // method to set each one. To work, you must override one or more set_*_local
  // methods to accommodate setting of your component's specific properties.
  // Any properties registered with the optional `uvm_*_field macros do not
  // require special handling by the set_*_local methods; the macros provide
  // the set_*_local functionality for you.
  //
  // If you do not want apply_config_settings to be called for a component,
  // then the build_phase() method should be overloaded and you should not call
  // super.build_phase(phase). Likewise, apply_config_settings can be overloaded to
  // customize automated configuration.
  //
  // When the ~verbose~ bit is set, all overrides are printed as they are
  // applied. If the component's <print_config_matches> property is set, then
  // apply_config_settings is automatically called with ~verbose~ = 1.

// apply_config_settings
// ---------------------
```

```
function void uvm_component::apply_config_settings (bit verbose=0);

  uvm_resource_pool rp = uvm_resource_pool::get();
  uvm_queue#(uvm_resource_base) rq;
  uvm_resource_base r;
  string name;
  string search_name;
  int unsigned i;
  int unsigned j;

  // populate an internal 'field_array' with list of
  // fields declared with `uvm_field macros (checking
  // that there aren't any duplicates along the way)
  __m_uvm_field_automation (null, UVM_CHECK_FIELDS, "");

  // if no declared fields, nothing to do.
  if (__m_uvm_status_container.field_array.size() == 0)
    return;

  if(verbose)
    uvm_report_info("CFGAPL","applying configuration settings", UVM_NONE);

  // Note: the following is VERY expensive. Needs refactoring. Should
  // get config only for the specific field names in 'field_array'.
  // That's because the resource pool is organized first by field name.
  // Can further optimize by encoding the value for each 'field_array'
  // entry to indicate string, uvm_bitstream_t, or object. That way,
  // we call 'get' for specific fields of specific types rather than
  // the search-and-cast approach here.
  rq = rp.lookup_scope(get_full_name());
  rp.sort_by_precedence(rq);

  // rq is in precedence order now, so we have to go through in reverse
  // order to do the settings.
  for(int i=rq.size()-1; i>=0; --i) begin

    r = rq.get(i);
    name = r.get_name();

    // does name have brackets [] in it?
    for(j = 0; j < name.len(); j++)
      if(name[j] == "[" || name[j] == ".")
        break;

    // If it does have brackets then we'll use the name
    // up to the brackets to search __m_uvm_status_container.field_array
    if(j < name.len())
      search_name = name.substr(0, j-1);
    else
      search_name = name;

    if(!uvm_resource_pool::m_has_wildcard_names &&
       !__m_uvm_status_container.field_array.exists(search_name) &&
       search_name != "recording_detail")
      continue;

    if(verbose)
      uvm_report_info("CFGAPL",$sformatf("applying configuration to field %s", name),UVM_NONE);

    begin
    uvm_resource#(uvm_bitstream_t) rbs;
    if($cast(rbs, r))
      set_int_local(name, rbs.read(this));
    else begin
      uvm_resource#(int) ri;
      if($cast(ri, r))
        set_int_local(name, ri.read(this));
      else begin
        uvm_resource#(int unsigned) riu;
        if($cast(riu, r))
          set_int_local(name, riu.read(this));
        else begin
          uvm_resource#(string) rs;
          if($cast(rs, r))
            set_string_local(name, rs.read(this));
          else begin
             uvm_resource#(uvm_config_object_wrapper) rcow;
             if ($cast(rcow, r)) begin
                uvm_config_object_wrapper cow = rcow.read();
                set_object_local(name, cow.obj, cow.clone);
             end
             else begin
                uvm_resource#(uvm_object) ro;
                if($cast(ro, r)) begin
                  set_object_local(name, ro.read(this), 0);
                end
                else if (verbose) begin
                  uvm_report_info("CFGAPL", $sformatf("field %s has an unsupported type", name), UVM_NONE);
                end
             end
          end
        end
      end
    end
```

```
    end

  end

  __m_uvm_status_container.field_array.delete();

endfunction
```

C. UVM IEEE

Perhaps the UVM IEEE solved the speed and complexity issues. How does it handle the apply_config_setting()? In the UVM IEEE 1800 version, it looks like apply_config_settings() got "fixed" or was made shorter. Less code is better code, if it is just as powerful. But let's see how the story ends.

```
// apply_config_settings
// ---------------------

function void uvm_component::apply_config_settings (bit verbose=0);

  uvm_resource_pool rp = uvm_resource_pool::get();
  uvm_queue#(uvm_resource_base) rq;
  uvm_resource_base r;

  // The following is VERY expensive. Needs refactoring. Should
  // get config only for the specific field names in 'field_array'.
  // That's because the resource pool is organized first by field name.
  // Can further optimize by encoding the value for each 'field_array'
  // entry to indicate string, uvm_bitstream_t, or object. That way,
  // we call 'get' for specific fields of specific types rather than
  // the search-and-cast approach here.
  rq = rp.lookup_scope(get_full_name());
  rp.sort_by_precedence(rq);

  // rq is in precedence order now, so we have to go through in reverse
  // order to do the settings.
  for(int i=rq.size()-1; i>=0; --i) begin
    r = rq.get(i);

    if(verbose)
      uvm_report_info("CFGAPL",$sformatf("applying configuration to field %s", r.get_name()),UVM_NONE);

    set_local(r);

  end

endfunction
```

It's really simple code. (perhaps – we still see the dire warning). Do some lookups and loop through and call set_local(). Nice.

Set_local() also has a simple implementation – and is designed to be overridden if desired. But it call a macro. But only for recording_detail. The real trick is the macros below.

```
// set_local (override)
// --------

function void uvm_component::set_local(uvm_resource_base rsrc) ;

  bit success;

  //set the local properties
  if((rsrc != null) && (rsrc.get_name() == "recording_detail")) begin
    `uvm_resource_builtin_int_read(success,
                                   rsrc,
                                   recording_detail,
                                   this)
  end

  if (!success)
    super.set_local(rsrc);

endfunction
```

Taking a peek in UVM IEEE at the field automation macro 'uvm_field_int' in uvm_object_defines.svh is exciting. That code is so dense and so precise.

The field automation macros call the `uvm_resource_builtin_int_read(). Ah ha

```
`define uvm_field_int(ARG,FLAG=UVM_DEFAULT) \
  `m_uvm_field_begin(ARG,FLAG) \
    `m_uvm_field_op_begin(COPY,FLAG) \
      ARG = local_rhs__.ARG; \
    `m_uvm_field_op_end(COPY) \
    `m_uvm_field_op_begin(COMPARE,FLAG) \
```

```
    `uvm_compare_int(ARG, local_rhs__.ARG, `m_uvm_field_radix(FLAG), __local_comparer__) \
  `m_uvm_field_op_end(COMPARE) \
  `m_uvm_field_op_begin(PACK,FLAG) \
    `uvm_pack_int(ARG, __local_packer__) \
  `m_uvm_field_op_end(PACK) \
  `m_uvm_field_op_begin(UNPACK,FLAG) \
    `uvm_unpack_int(ARG, __local_packer__) \
  `m_uvm_field_op_end(UNPACK) \
  `m_uvm_field_op_begin(RECORD,FLAG) \
    `uvm_record_int(`"ARG`", \
                    ARG, \
                    $bits(ARG), \
                    `m_uvm_field_radix(FLAG), \
                    __local_recorder__) \
  `m_uvm_field_op_end(RECORD) \
  `m_uvm_field_op_begin(PRINT,FLAG) \
    `uvm_print_int(ARG, $bits(ARG), `m_uvm_field_radix(FLAG),,__local_printer__) \
  `m_uvm_field_op_end(PRINT) \
  `m_uvm_field_op_begin(SET,FLAG) \
    if(local_rsrc_name__ == `"ARG`") begin \
      `uvm_resource_builtin_int_read(local_success__, \
                                     local_rsrc__, \
                                     ARG, \
                                     this) \
      /* TODO if(local_success__ && printing matches) */ \
    end \
  `m_uvm_field_op_end(SET) \
`m_uvm_field_end(ARG)
```

But OK – what's that magic macro do? `uvm_resource_builtin_int_read()? It calls other macros.

```
// uvm_resource_builtin_int_read
// -----------------------------
// Attempts to read an integral typed resource ~RSRC~
// into VAL.
//
// ~SUCCESS~ shall be set to true if the resource was succesfully
// read, false otherwise.
//
// Supported Types:
// - uvm_integral_t
// - uvm_bitstream_t
// - int
// - int unsigned
//

`define uvm_resource_builtin_int_read(SUCCESS, RSRC, VAL, OBJ=null) \
begin                                                               \
  `uvm_resource_read(SUCCESS, RSRC, uvm_integral_t, VAL, OBJ)               \
  if (!SUCCESS)                                                     \
    `uvm_resource_read(SUCCESS, RSRC, uvm_bitstream_t, VAL, OBJ)            \
  if (!SUCCESS)                                                     \
    `uvm_resource_read(SUCCESS, RSRC, int, VAL, OBJ)                        \
  if (!SUCCESS)                                                     \
    `uvm_resource_read(SUCCESS, RSRC, int unsigned, VAL, OBJ)               \
end
```

Uh oh – that macro repeats the "are-you-my-mother" type checking ladder. And calls a macro - `uvm_resource_read(). Disappointingly the kiss from the princess didn't turn the frog into a handsome prince. The coding is gorgeous. But the same structure and algorithm and performance exists. Apply_config_settings did NOT get cleaned up – effectively is got factored out of the UVM and into the field automation code of each attribute – making it much harder to understand, debug and use, in the opinion of this author.

`uvm_resource_read does the cast, check and assign.

```
// m_uvm_resource_read
// -------------------
// Casts ~RSRC~ to uvm_resource#(TYPE) and if successful
// reads the resource into ~VAL~ with accessor ~OBJ~.
//
// ~SUCCESS~ shall be set to true if the resource was succesfully
// read, false otherwise.

`define uvm_resource_read(SUCCESS, RSRC, TYPE, VAL, OBJ=null)   \
begin                                                           \
  uvm_resource#(TYPE) __tmp_rsrc__;                             \
  SUCCESS = $cast(__tmp_rsrc__, RSRC);                          \
  if (SUCCESS) begin                                            \
    VAL = __tmp_rsrc__.read(OBJ);                               \
  end                                                           \
end
```

For example, in the user generated code – the expanded macro for the field automation for "SET" in UVM IEEE looks like the code below. The factored out apply_config_settings code has been found.

```
        UVM_SET:
```

```
if ( (((UVM_DEFAULT)&UVM_SET)) && (!((UVM_DEFAULT)&UVM_NOSET))    ) begin
  if(local_rsrc_name__ == "foo") begin
  begin
    begin
      uvm_resource#(uvm_integral_t) __tmp_rsrc__;
      local_success__ = $cast(__tmp_rsrc__, local_rsrc__);
      if (local_success__) begin
        foo = __tmp_rsrc__.read(this);
      end
    end
    if (!local_success__) begin
      uvm_resource#(uvm_bitstream_t) __tmp_rsrc__;
      local_success__ = $cast(__tmp_rsrc__, local_rsrc__);
      if (local_success__) begin
        foo = __tmp_rsrc__.read(this);
      end
    end
    if (!local_success__) begin
      uvm_resource#(int) __tmp_rsrc__;
      local_success__ = $cast(__tmp_rsrc__, local_rsrc__);
      if (local_success__) begin
        foo = __tmp_rsrc__.read(this);
      end
    end
    if (!local_success__) begin
      uvm_resource#(int unsigned) __tmp_rsrc__;
      local_success__ = $cast(__tmp_rsrc__, local_rsrc__);
      if (local_success__) begin
        foo = __tmp_rsrc__.read(this);
      end
    end
  end
end
```

XVII. WHAT ABOUT THE UVM_RESOURCE_DB? AND OTHER UVM VERSIONS?

A reviewer asked about uvm_resource_db. Could uvm_resource_db be used as a configuration container? Without doing too much investigation, the answer must be 'yes', since the uvm_config_db IS-A uvm_resource_db. Using uvm_resource_db instead of uvm_config_db would lose the "extra" code in uvm_config_db. Does that code matters – is it needed? There are many discussions about the relationships between uvm_config_db and uvm_resource_db available. [3] [4] [5]

But remember, this paper is not about being better than uvm_config_db, but rather it is about thinking about what is being modeled. Certainly, the simplest model of a configuration is a global variable which is a hierarchical collection of configurations. A tree of settings.

To use uvm_resource_db as a "better" solution – the code in uvm_config_db is gone. Uvm_resource_db has a get() routine. Zeroing in on the implementation – it is the same implementation that uvm_config_db would have called – the get() call eventually calls the lookup_name() routine in uvm_resource_pool. That code has been the subject of much analysis. It's code that this author would prefer to avoid.

Given a small set of configurations with limited use of regular expressions, the uvm_config_db is a fine performer. Same for uvm_resource_db. In the "bad places", uvm_resource_db appears to suffer similarly to uvm_config_db, but final details are left to the reader for a conclusive experiment.

A. uvm-1.2

What about uvm-1.2? In terms of config and resources, the uvm-1.2 is mostly the same as uvm-1.1d. It fixes typos and adds better debug visibility and thread safety. The function lookup_regex_name() appears "improved". Uvm_config_db.svh has an integration to core_services – but that's a whole other paper left for some opportunistic explorer. Out of about 2700 lines of code (uvm_resource.svh, uvm_resource_db.svh and uvm_config_db.svh), the 'diff' report between uvm-1.1d and uvm-1.2 is ~400 lines. Very similar code.

B. uvm-ieee

The 'diff' report between uvm-1.1d and uvm-ieee is larger. The code changes are hard to study at a glance, but appear to not address any of the "structural" issues, instead moving code, improving debug again, and cleaning up. It appears there may be some improvements designed-in for performance, but that is left as an exercise for the reader. The uvm_config_db::get() routine coding is identical between uvm-1.1d and uvm-ieee, aside from an additional lookup to the core_services in uvm-ieee, implying the same performance (at least for get()).