2014 DESIGN AND VERIFICATION DVCON CONFERENCE AND EXHIBITION INDIA
Sept 25-26, 2014
Hotel Park Plaza
Bangalore, India

Please! Can someone make UVM easier to use?

Raghu Ardeishar, Verification Technologist, Mentor Graphics, McLean, VA, USA
(*raghu_ardeishar@mentor.com*)

Rich Edelman, Verification Technologist , Mentor Graphics, San Jose, CA, USA
(*rich_edelman@mentor.com*)

***Abstract*—UVM was designed as a means of simplifying and standardizing verification which had been fragmented as a result of many methodologies in use like eRM, VMM, OVM. It started off quite simple. Later on as a result feature creep many of the issues with the older methodologies found its way into UVM. This paper looks at some of those issues and suggests ways of simplifying the verification environment.**

Keywords—SystemVerilog, UVM, Configuration, Sequences

I. INTRODUCTION

Why is the UVM becoming so difficult to use? When the UVM was conceived the idea was to take the plethora of existing verification methodologies and create a single one based on the power of SystemVerilog. Good practices from the different objected oriented languages were adopted to ease the pain of verification. As a result everyone was "forced" to become a software developer. Many of the current users of UVM are RTL designers who have been forced to morph into a role not quite native to them.

The verification engineers are now plagued with similar growing pains as those which confronted the early C++ folks. A classic example is the template library. Templates are widely used by people and the initial roll out issues are a distant memory.

In the deployment of UVM the newly minted software engineers are asked to replace tasks with sequences. Simple variable lookups have been transformed to undecipherable config lookups. Let's not forget a simple display statement has been replaced with the UVM messaging library. All these provide incredible flexibility and power but with a cost. The designers use many of these features whether they need it or not. Many times this is further complicated by needlessly parameterizing sequences and tests. This makes overriding tests and sequences, the hallmark of object oriented programming, too confusing for the developer. Layer on top of this the myriad of macros in existence; you have just created the perfect confusion soup. Due to the fact that there is no good IDE in existence many designers are scratching their heads trying to debug their code.

This paper will show how to cut through the clutter of UVM and write easy to debug code. Pitfalls of parameterization will be addressed in addition to showing where to use it and where not to. In addition techniques will be shown on how to simplify configurations and cut through the clutter. Macro usage will also be touched upon showcasing the appropriate places to use them. Performance of code will be addressed by showing how to avoid writing complicated code.

We will demonstrate where the advanced concepts like parameterizations make sense and where they don't, thus resulting in code which is easier to write, understand, port and maintain.

This demonstration will take the form of some "before" and "after" code snippets.

II. PARAMETERIZED CLASSES

Parameterized classes are very powerful but quite often misunderstood and misused. They can be used to significantly cut down code bloat and simplify the code base. But along with the power comes baggage. The baggage is in terms of performance and use with UVM utilities like the factory and macros. Plus they are not needed in many cases. If you are going to have only one or two "types" of class instances it might not make sense to parameterize them. Overriding parameterized classes takes more care and once you layer on UVM and the

factory macros it becomes significantly more complicated. Once you learn the nuances you are good to go but parameterization should be used sparingly.

A. Parameterize a value

Classes can be parameterized in many ways. We will take a look at a couple of examples. Lets first take a look at parameterization using "values". In this case "V" is a parameter which can be an integer. So when you instantiate it the value can be 3 , 4 or any integer.

```
class classValue #(int V = 3)
  int delay = V;
endclass
classValue #(4) cV4;
classValue #(10) cV10;
```

B. Parameterize a type

You can also parameterize classes using types as shown below. The Default type is "int" but that can be changed during instantiation.

```
class classType #(int T = int)
  T delay;
endclass
classType #(int) cInt;
classType #(integer) cInteger;
```

How does that affect overriding, macros, factories etc? Let's first take a look at basic polymorphism.

C. Basic Polymorphism

Polymorphism is one of the main reasons to use classes. A class handle can be assigned another class handle which is a subclass. Example:

```
class classValue;
endclass

class classValueNew extends classValue
endclass

classValue     cV = new;
classValueNew cVN = new;

cV = cVN;
```

The base class is classValue. classValueNew is an extension of classValue. Therefore classValueNew can be assigned to classValue because they are "type compatible". If you notice the code above it is non-parameterized and fairly simple in concept. But what happens to polymorphism when the class is parameterized?

D. Polymorphism and Parameterization

If the class is parameterized the assignment is no longer simple. Consider the code below:

```
class classValue (int V = 3);
```

```
endclass

classValue #(3) cV3 = new();
classValue #(4) cV4 = new();

cV3 = cV4;  //ERROR
```

classValue is a parameterized class and the 2 instantiations show above (one with parameter V = 3 and the other with V = 4) create 2 separate types. cV3 and cV4 are no longer type compatible. Though the code might compile and load it will not run. So the question you want to ask is "Do you need to parameterize this class?" Let's take another example where parameterization leads to this issue.

```
class classType (type T = int);
endclass

classType #(int)         cInt = new();
classType #(integer) cInteger = new();

cInt = cInteger;  //ERROR
```

classType is a parameterized class and the 2 instantiations show above (one with parameter T = int and the other with T = integer) create 2 separate types. ('int' is a 32 bit integer and 'integer' is a 32 bit 4 state integer). cInt and cInteger are not type compatible. Though the code might compile and load, it will not run.

So how can we recode around these issues? One example is to move the "parameter" inside as a class property. By moving "V" inside as a class property you can change it in the extended class and still retain type compatibility.

```
class classValue;
 int  V = 3;
endclass

class classValueNew extends classValue
 int V = 4;
endclass

classValue     cV = new;
classValueNew cVN = new;

cV = cVN;
```

E. UVM and Parameterization

How does UVM and factories add to the issue of parameterized classes? In UVM classes are typically registered with the factory using `uvm_object_utils, `uvm_component_utils etc. These work predictably with non-parameterized classes as shown below.

```
class packet extends uvm_object ;
 `uvm_object_utils(packet)
Endclass
```

```
class packetD extends packet;
`uvm_object_utils(packetD)
Endclass

packet   p = new();
packetD pD = new();

p = pD;  //Works!!
```

The reason being packet and packetD are type compatible, since packetD is an extension of packet. If you use the uvm_top.print_topology() or factory.print() routines you get what you expect. Try the following code snippet and see how it works.

```
virtual function end_of_elaboration_phase(uvm_phase phase) ;
  uvm_top.print_topology();
  factory.print();
endclass
```

Now let'sparameterize the class and see what happens. At a minimum you will have to use the parameterized equivalent of the macros i.e. `uvm_object_param_utils, `uvm_component_param_utils etc. Unfortunately even these macros will not create the necessary routines to print and override using the factory. You will need to register the class manually by writing this simple piece of code!

```
class driverB #(type T = int) extends uvm_driver #(T);
    //`uvm_component_param_utils(driverB#(T))
    localparam type_name = $sformatf("driverB#(%s)", T::type_name);
    typedef uvm_component_registry #(driverB#(T), type_name) type_id;
    static  function type_id get_type();
      return type_id::get();
    endfunction
    virtual function uvm_object_wrapper get_object_type();
      return type_id::get();
    endfunction
    virtual function string get_type_name();
      return type_name;
    endfunction
endclass

class driverD2 #(type T = uvm_object) extends driverB #(T);
endclass

typedef driverD2#(packet)    driverD2packet;
typedef driverD2#(packetD)   driverD2packetD;
```

`uvm_component_param_utils does register the class with the factory but no unique type name is created. Hence overriding and printing using names becomes hard. What you are doing with the code snippet shown above essentially manually expands the macro. As a result of this factory.print() will show the overrides in the system. But the inherent issues remain, driverD2packet and driverD2packetD are not type compatible. So how do we "fix" this problem. For starters we can "de-parameterize" the class as shown below.

```
class classType #(type T = int);
  T myDelay;
  function calcDelay();
  endfunction
endclass
```

Can we rewritten as:

```
class config env_config extends uvm_object
  rand int delay;
endclass

class classType;
   int myDelay ;
   env_config e;
   uvm_config_db :: get(…"e",e);
   function new ( );
     myDelay  = e.delay;
   endfunction
endclass
```

F. Parameterized tests and sequences

Sequences and tests are parameterized often. But not always needed. It is tempting to parameterize tests and sequences based on bus widths, number of lanes e.g., PCIe. But doing this will create issues while trying to run sequences which have been parameterized using other values. One workaround is to instantiate with the maximim possible bus widths and control the individual dimensions using environment variables. You will need to create a new sequence for each variation of parameters leading to code bloat. Example:

```
class test #(int LANES=2,int pipeByteMax= 1,int numOfFuncs = 1) extends uvm_test;
  typedef pcieSeq #(LANES, pipeByteMax, numOfFuncs) pcieSeqT;
  ….
  task run_phase;
   pcieSeqT pcieSeq = pcieSeqT::type_id…;
   pcieSeq.start(sequencer);
 endtask
endclass
```

Let's simplify the tests using configs. We will create a configuration object "env_config" and add the environment variables as properties. Then we retrieve the object with the desired settings in the test and retrieve the variables.

```
class env_config extends uvm_object
  rand int LANES;
  rand int PIPE_BYTE_MAX;
  rand int NUM_OF_FUNCTIONS;
endclass
```

The above configuration object is added to the config database in the top module.

```
module top;
 initial begin
  env_config eC = new();
  randomize(eC) with …;
  uvm_config_db #(env_config)::set(uvm_root::get(),"*","eC", eC);
 end
endmodule
```

This will be retrieved in the test.

```
class test extends uvm_test;
 int LANES;
 int PIPE_BYTE_MAX;
 int NUM_OF_FUNCTIONS;
 typedef pcieSeq  pcieSeqT;
 ….
  task run_phase;
   pcieSeqT pcieSeq = pcieSeqT::type_id…;
   env_config eC;
   uvm_config_db :: get(…"eC",e);
   LANES = eC.LANES;
   PIPE_BYTE_MAX = eC.PIPE_BYTE_MAX;
   pcieSeq.start(sequencer);
 endtask
endclass
```

III. CONFIGURATION DB

Configuration DB's are very useful but also misused. They are great for lookups but are expensive. They are used to set and get interfaces, UVM objects and even simple variables like integers; and therein lies the problem. By calling set and get on configuration objects multiple times you run the risk of slowing the system down. Let's look at a typical config db set command.

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

You can use "set" to set the value in or outside a class.

- Inside a class to set the value:

```
uvm_config_db #(type)::set(this,"*.pathname", "label",value);
```

- Outside a class to set the value:

```
uvm_config_db #(type)::set(uvm_root::get(),"*.pathname", "label",value);
```

Inside a class to get the value

```
uvm_config_db #(type)::get(this,"","label",value)
```

- Use +UVM_CONFIG_DB_TRACE (simulator command line option) to debug set/get issues.

Use unique names for variables and avoid variables with the same name in different instance paths. If you have two PCIe interfaces, calling both "pcieIntf" and relying on different instance pathnames eg, /u/cpu/pcieIntf and /u/dma/pcieIntf, to distinguish between them would create problems. Do NOT set the variables as shown below.

```
uvm_config_db #(type)::set(uvm_root::get(),"/u/cpu", "pcieIntf",value);
uvm_config_db #(type)::set(uvm_root::get(),"/u/dma", "pcieIntf",value);
```

Use "*" for the instance names avoiding specific paths. It would be preferable to do this:

```
uvm_config_db #(type)::set(uvm_root::get(),"*", "pcieIntfCpu",value);
uvm_config_db #(type)::set(uvm_root::get(),"*", "pcieIntfDMA",value);
```

It's a very big hammer but worthwhile in the long run. It will avoid picking up wrong instances. For example you could pick up the variable "pcieIntf" meant for "/u/cpu" and assign it to "/u/dma". What the other approach using "*" does is put the objects in global space. It is anathema for most programmers but serves engineers who have assumed the role of software designers.

Avoid using macros with the config database. An example is the property "is_active". It is a mistaken assumption that `uvm_component_utils implements the uvm_config_db::get method. You will have to manually implement the "get" routine in the test/env or use the `uvm_field_enum macro. Example:

```
`uvm_field_enum(uvm_active_passive_enum, is_active, UVM_ALL_ON)
```

IV. FINALLY MACROS

To use or not to use? That's the question. [2] Well, that depends on the macro. Many of them will have an impact on performance. Most simulators have optimized the performance issue but the debug issues remain. The code bloat of many macros are not worth the shortcut it buys us. It is worthwhile to learn the way the code is written and "inline" the code instead of using macros. We will not go into full details here. You can refer to an excellent paper written on that topic. For example avoid the `uvm_do macro. You don't need a macro to execute sequences. The macros expand into complicated unneeded code. Instead of using `uvm_do to send a sequence item:

```
task sequence::body;
  myItem item;
  `uvm_do(item) // AVOID
endtask
```

Rewrite as calls to start_item() and finish_item().

```
task sequence::body;
 myItem item = myItem::type_id::create(…);
 start_item(item);
 randomize(item);
 finish_item(item);
endtask
```

and instead of using `uvm_do() to start a sequence

```
task sequence::body;
  mySeq seq;
  `uvm_do(seq) // AVOID
endtask
```

Simply create it yourself and call seq.start().

```
task sequence::body;
 mySeq seq = mySeq::type_id::create(…);
 seq.start(…);
endtask
```

Avoid the `uvm_field macros also. It implements copy, compare, pack, unpack etc but creates very complex and very hard to debug code. Write the above routines manually which are a lot easier to debug.

V. SUMMARY

UVM unified many of the older methodologies and is a culmination of many years of work. But many of the practices of yester years crept in. Using some of those practices can cause needless complication in your testbench. But the complication can be avoided. Parameterization should be sparingly used and manually registered with the factory. Most often tests and sequences can be written without parameterization. Configuration DBs should also be used with caution limiting the number of "sets" and "gets". Macros which cause the most problems should be used very sparingly. Following these simple rules make the testbench easy to manage and debug.

REFERENCES

[1] V.Cooper, Paul Marriott, "Demystifying the UVM Configuration Database"

[2] Adam Erikson, Are OVM and UVM macros Evil? A Cost-Benefit Analysis.