Configuring Your Resources the UVM Way!

Parag Goel,
Synopsys, India
91.80.40188000
paragg@synopsys.com

Amit Sharma,
Synopsys, India
+91.80.40188000
amits@synopsys.com

Rajiv Hasija,
LSI, India
+91.80.41977811
rajiv.hasija@lsi.com

ABSTRACT

With increasing complexity of semiconductor designs, and with more and more IP blocks to be integrated in SoC's, there is an increasing need of verifying their numerous configurations. The large numbers of permutations through which different functionalities can be enabled at any given time require the verification environment to mimic the same. The testbench configuration classes with different properties are typically used in verification environments to model these various aspects. The constraints across the various properties ensure that the testbench can pick up valid configurations.

Now, these configuration classes have to be available or accessible across all the testbench components. This ensures that all the components can independently configure themselves based on the testbench topology and the relevant DUT configuration that is currently being verified. Additionally, the testbench components need to be able to change the properties of the configuration classes dynamically based on the DUT responses and also be able to react to similar changes affected by the other components. Efficient resource management is also vital in such complex environments.

The reference implementation of the base classes as specified in the Universal Verification Methodology (UVM) provides a lot of relevant functionality to achieve some of these requirements. To ensure that these capabilities are optimally leveraged, the paper helps the reader achieve a deeper understanding of the mechanism through which this functionality is delivered. The different usage scenarios demonstrated show how the desired benefits can be achieved in the effective configuration and management of testbench resources.

Categories and Subject Descriptors

Testbench Methodology, UVM, Testbench Configuration.

General Terms

Management, Performance, Verification, Design

Keywords

Testbench, Configuration, UVM, SystemVerilog

1. INTRODUCTION

Managing the testbench configuration is a challenge in complex verification environments today. The different power modes, performance optimizations, memory accesses modes and their management, and other interrelated functionality require different components in the verification environment to be configured differently. Each component in a verification environment would typically support different capabilities which might not be required for a specific project or a DUT configuration. Thus, different testbench components could require information to be passed to it from external sources. To encapsulate this information, there is a need for a structured container, with appropriate constraints, and values which are themselves configurable. This container or class needs some knowledge of the circumstance at hand so that it can alter its topology or behavior accordingly.

Thus, dynamic testbench reconfiguration and modeling of configurable scenarios are required to verify real world scenarios for the complex chips. From a horizontal and vertical reuse point of view, it is imperative that that the components are made sufficiently configurable. The specification of the base classes in the Universal Verification Methodology (UVM) provides a convenient interface for achieving some of these requirements. Leveraging the UVM configuration mechanism, we will show how a resource management infrastructure can be created which will make all the components configurable by design. We would also demonstrate the usage of the capabilities available to achieve different verification requirements. Specific guidelines would be highlighted so that optimal usage is achieved.

2. UVM Configuration Mechanism: A Feature Description

From Figure 1, it can be seen that each component ideally needs a ‘configuration’ component to bring in the desired ‘flexibility’ or ‘reusability’. The configuration of each component needs to be changed based on the different requirements. Depending on the configuration changes, the components need to be Also, as the various configurations are changed or applied across the testbench; individual components have to be aware of all such changes.

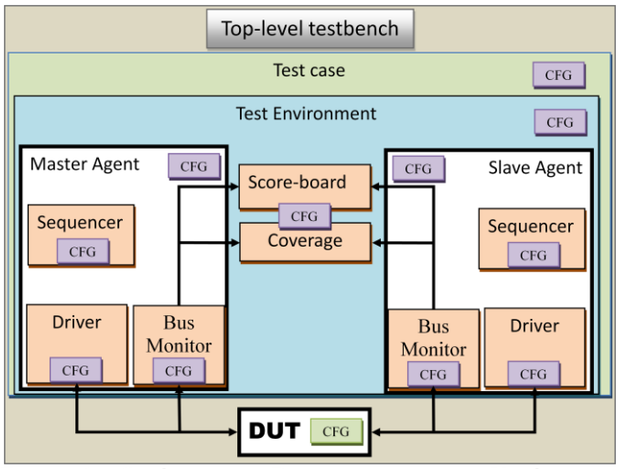

Figure 1: Configuration Requirements in a Verification Environment

For efficient synchronization and integration, a uniform configuration should be propagated across the hierarchy. Hence, as a general recommendation, the attributes that control functional aspect of a verification component must be encapsulated in a class wrapper. This class object can be passed down to different components in the testbench environment. The testbench configuration attributes which decides on the topology and coarse testbench controls can continue to be distributed across discrete elements.

```
class vc_cfg extends uvm_object;          Extended from uvm_object
  rand int number_of_trans;
  rand int mode_of_operation;             Elements of configuration
  constraint valid_mode {
    mode_of_operation inside {MASTER, SLAVE };
  }                                       Set of valid constraints

  `uvm_object_utils_begin(vc_cfg)
    `uvm_field_int(number_of_trans,   UVM_PRINT | UVM_COPY)
    `uvm_field_int(mode_of_operation, UVM_PRINT | UVM_COPY)
  `uvm_object_utils_end
  function new (string name = "vc_cfg");
    super.new(name);                                  Factory registration
  endfunction : new                                    using field macros
endclass : vc_cfg
```

Figure 2: A Typical Configuration Class

A typical example of a Configuration class is shown in Figure 2. The constraints across the various properties ensure that the valid combinations are generated when the class is randomized. The Configuration class should be implemented by extending the *uvm_object* or the *uvm_sequence_item* base class. This object must be factory-enabled.

How do we ensure that this Configuration class can easily be propagated across the testbench environment? In UVM, objects are often instantiated through the factory infrastructure. The factory infrastructure is responsible for invoking the constructors and the user does not have the possibility to modify constructor arguments. Hence, it is not recommended to pass configuration objects in constructor arguments. Thus, we need a capability through which functions can put or retrieve configuration attributes and objects from a central database dynamically.

UVM provides a facility called 'resources' which provides the configuration infrastructure and API. It is comprised of polymorphic resource containers, a database for storing those resource containers and an infrastructure for locating resources in the database. This infrastructure can then be used to propagate configuration information to different components. Each component retrieves a value from the resource pool and uses the retrieved value to control its behavior.

2.1 Understanding UVM Resources

The generic resource database provided by UVM is a low-level database which has its own access classes and methods. The Figure 3 represents the organization of the resource database., The UVM Resources provides multiple APIs which are distilled down to the user to access, retrieve, audit, query, lock and set priority levels while interfacing with the underlying database. For the user, UVM configuration facility provides two separate API interface. These are the *uvm_resource_db* and *uvm_config_db*. These two classes interface between the UVM resource facility and the actual user.

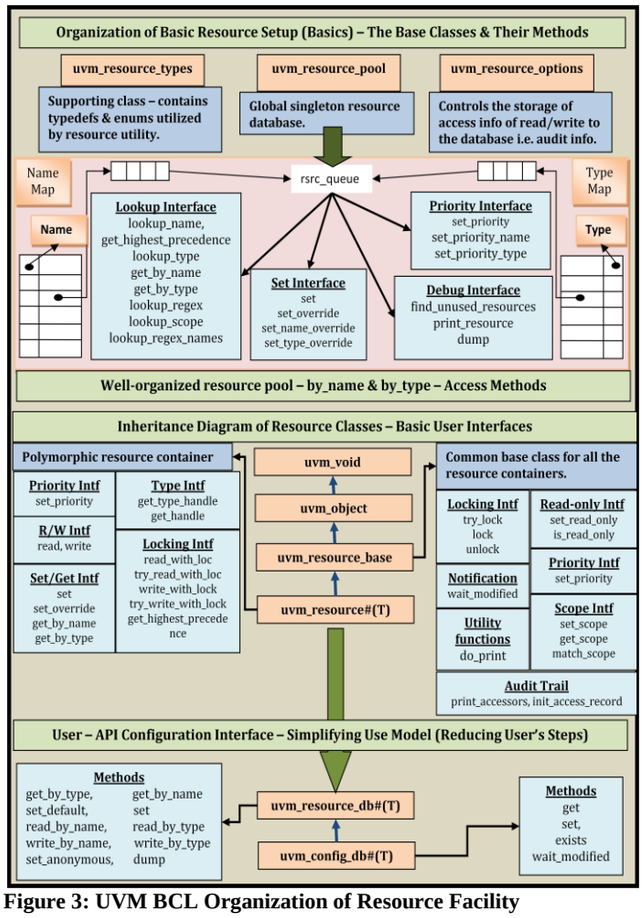

Figure 3: UVM BCL Organization of Resource Facility

The class *uvm_resource_db#(T)* is a convenience layer on top of the low-level database. It can be used on its own and different layers of type-specific specializations can be built on top of it. A collection of static functions operate on the resources and the resource pool. The *uvm_config_db#(T)* class provides a layer on top of the *uvm_resource_db#*(T) to enable the accesses in the hierarchical context. These classes enable the users to retrieve different attributes and objects with the appropriate look-up names. User can "set" a default configuration object in the environment, "get" the configuration object in the test and modify it, then "get" the modified configuration object in the VIP or the block level agents.

Users can retrieve a testbench attribute from the resource database or dump an attribute into the database either by the 'type' or by 'name' of the attribute. Given that both these parameterized classes share the same underlying database, it is possible to write to the database using *uvm_config_db::set()* and retrieve from the database using *uvm_resource_db::read_by_name()*. With respect to usage, *uvm_config_db* is a type-specific UVM configuration mechanism which creates a visibility of a resource over a specified scope defined by the *uvm_component* hierarchy whereas *uvm_resource_db* is mainly for general purpose usage, which is used to set shared resource which can be accessed anywhere in the testbench scope. A resource is set through non-hierarchical context while for *uvm_config_db*, a hierarchical uvm_component context is required. Hence, the APIs provided by both these interfaces differ primarily in ability to provide the hierarchical context as shown in Figure 4.

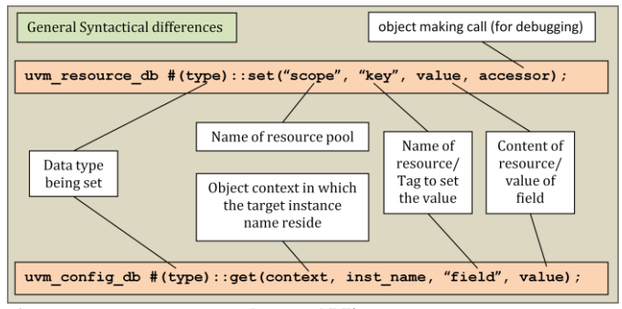

Figure 4: UVM Resource Access API's

Also, there are distinct methods for the retrieval of a configuration object or attribute in case of uvm_resource_db by name or by type. These are:

```
uvm_resource_db#(type)::read_by_name("scope",
key, type_var, accessor);
uvm_resource_db#(type)::read_by_type("scope",
type_field, accessor);
```

The scope can be a regular expression in *'set'* only. In *'get'*, it must be an actual value.

Now, going back to the "Configuration" class, once it is randomized, the test writer can use the *uvm_config_db::set()* mechanism to assign the configuration object to one or more environments within the testbench.

2.2. Basic Use Cases

2.2.1 'Getting' and 'Setting' Configuration Attributes:

Using *uvm_resource_db* **:** The following will create a new 'resource', populate it and insert it into the resource pool.

```
uvm_resource_db#(int)::set("top.env*,"A",",12
34,this);
```

The above code snippet will create a new 'resource' of type *int*, assign a value of "1234" to it, and 'set' it in the database using the name "A". This 'resource' would be made visible

in the scopes identified by "*top.env.**". This assumes there is only one matching argument 'A' in top*, which happens to be top.env.A. The last argument is used for auditing.

The component that retrieves this resource would read the value from the database.

```
if(!uvm_resource_db# (int)::read_by_name(get_
full_name(),"A",value,this))
`uvm_error(get_full_name(),"The resource A
cannot be retrieved. It is not found in the
specified scope ");
```

Here, *read_by_name()* returns a bit that indicates whether or not the lookup succeeded. It is always a good coding practice to check the return value of *uvm_config_db::get()* or *uvm_resource_db::read_by_name()*. An appropriate message can be issued to report whether a retrieval was successful or not. This can help avoid unnecessary debug cycles.

Using *uvm_config_db*:

```
class my_mon extends uvm_monitor;
     bit check_en;
     `uvm_object_utils(my_mon)
     . . .
     function void build_phase(uvm_phase
phase)
       uvm_config_db #(bit)::get(this, "",
"check_en", check_en)
     endfunction
endclass
```

The corresponding code snippet for setting the value of the *check_e*n attribute would be the following :

```
uvm_config_db #(bit)::set(this, ".*mon*",
"check_en", 1);
```

The above code ensures that the 'setting' of the value happens in the monitor instance path only. The *uvm_config_db #(bit)::get*() call can be avoided if the field automation macros are used in the UVM component instance as shown below:

```
`uvm_field_int(check_en, UVM_PRINT|
UVM_COPY);
```

The *uvm_component*, which in this case is the Monitor class should be instantiated in the component hierarchy and that *super.build_phase()* or *apply_config_settings()* should be called in the component's *build_phase()*. In this case, the 'get' call is implicitly invoked. However, the recommendation is not to rely on the field automation macros, and invoke an explicit *uvm_config_db::get*(). This enables better type checking and traceability through the issuing of a warning/error message on an unsuccessful get).

2.2.2 Propagating the Configuration Object Across the Testbench Hierarchy

To set the configuration object across the hierarchy, see Figure 5.

```
class my_drv extends uvm_driver;
     drv_cfg cfg;
     `uvm_object_utils_begin(my_drv)
        `uvm_field_object(cfg, UVM_DEFAULT)
     `uvm_object_utils_end
     function void build_phase(uvm_phase phase);
       super.build_phase(phase);
       uvm_config_db #(drv_cfg)::get(this, "", "cfg", cfg)
     endfunction
endclass

Setting the driver configuration object from test
case/environment:
uvm_config_db #(drv_cfg)::set(this, ".*drv*", "cfg", my_cfg);
```

Figure 5: Setting the Configuration Object Across Components

Here, irrespective of whether the field automation macros are used or not, an explicit *get()* for the class object in the driver class is required. This is because the data type of the configuration class, *drv_cfg*, is strongly typed. A *get()* call could be avoided if the configuration object passed was of type *uvm_object*. In this case, an explicit cast has to be made to the extended configuration class handle.

The precedence rules ensure that when multiple overrides or *'sets* are applied, the one applied at the end succeeds. For a testbench configuration class, it is always intended that configurations *'set'* at the highest-level of testbench hierarchy always override the configurations set from the lower-level components.

The configuration object can also be propagated down to classes which is not necessarily an *uvm_component* extension as shown in Figure 6

```
class my_trans extends uvm_sequence_item;
     sys_cfg cfg;
     `uvm_object_utils_begin(my_trans)
        `uvm_field_object(cfg, UVM_DEFAULT)
     `uvm_object_utils_end
     function void pre_randomize();
       uvm_config_db #(sys_cfg)::get(null, "", "cfg", cfg)
     endfunction
endclass

Setting the driver configuration object from test
case/environment:
uvm_config_db #(sys_cfg)::set(null, "*", "cfg", my_cfg);
```

Figure 6: Configuration Passing Across Component and Object Hierarchy

Here, the important thing to note is that the 'scope' is made 'null'. Not doing the same will result in an error. This is because the 'set' is invoked the *uvm_component* scope and the corresponding 'get' in the *uvm_object* scope.

3. Addressing Verification Requirements Using UVM Resources

Here, we look at the common verification challenges and see how the UVM configuration mechanism help us address some of them.

3.1 Propagating Virtual Interfaces

The Resource Database works with all SystemVerilog data types. One ubiquitous requirement across all verification environments is to have a mechanism which enables passing the virtual interfaces easily across different verification components. It is important to avoid using cross-module references (XMRs) in environments as it makes it impossible to put them in packages and thus compile them separately. It also makes the environment inherently non-reusable.

Thus, a better approach is to "push" the virtual interface into the Configuration Database from the top-level module. That top-level module is specific to the environment used to verify the DUT. It is thus perfectly acceptable to specify environment-specific hierarchical names in that module. Since the module is not an *uvm_component*, "null" is specified as the context argument and the absolute hierarchical name is specified for the agent where the virtual interface is assigned. As the environment is usually instantiated by the test, the absolute hierarchical name will start with *"uvm_test_top."*.

```
module tb_top;
   my_if my_if0(...);
   my_if my_if1(...);
   initial begin
      uvm_config_db#(virtual my_if)::set(null,
                  "uvm_test_top.env.agt0", "vif", my_if0);
      uvm_config_db#(virtual my_if)::set(null,
                  "uvm_test_top.env.agt1", "vif", my_if1);
   end
endmodule
```

Figure 7: Propagating Virtual Interfaces using UVM Resources

Agents should extract their virtual interfaces from the Configuration Database entry named "*vif*" during the *build* phase. A fatal error should be issued if no virtual interface is retrieved. The agents are then responsible for propagating that virtual interface to the drivers and monitors it encapsulates.

.

```
class my_agent extends uvm_agent;
  virtual my_if vif;
  ...
  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    ...
    if (!uvm_config_db#(virtual my_if)::
                        get(this, "", "vif", vif)) begin
       `uvm_fatal("NOVIF", "No virtual interface specified");
    end
    uvm_config_db#(virtual my_if)::set(this, "drv", "vif", vif);
    uvm_config_db#(virtual my_if)::set(this, "mon", "vif", vif);
    ...
  endfunction
  ...
endclass
```

Figure 8: Trickling the Interface to the Sub-Components

3.2 Configurability in Stimulus Generation and Execution

There might be specific requirements when the sequence item's constraints depend on the values in configuration object. The Resource Database can be used in the sequences as well. Though the UVM configuration mechanism is designed around components, when a non-component object needs to access specific attributes, this non-component object must access the configuration field through a component handle. In the case of sequences, *'m_sequencer'* is the handle to the sequencer that is executing the sequence. It is a built-in member of the *uvm_sequence* class. The configuration object can be accessed through the *'m_sequencer'* handle as shown in Figure 9.

```
//Reference configuration field through parent sequencer
class packet_sequence extends uvm_sequence;
  int item_count = 10;
  task body();
   uvm_config_db#(int)::get(m_sequencer, "", "icount", icount);
    repeat(item_count) begin
      `uvm_do(req)
    end
  endtask
endclass
//Set in test
class test_20_items extends test_base;
function void build_phase(uvm_phase phase);
  super.build_phase(phase);
  uvm_config_db#(int)::set(this, "env.agent.seqr", "icount", 20);
 endfunction
endclass
```

Tag to get value

Parent sequencer

Variable to store value unchanged if not set

Tag to set value

Context of search

name path to sequencer

Value

Figure 9: Configurable Sequences

The '*m_sequencer*' handle is no longer required if global configuration is desired. This can then be achieved through *uvm_resource_db.*

```
uvm_resource_db#(int)::read_by_name("SEQ_CNTR
L", "item_count", item_count);
```

The 'item_count' value for the resource then can be set in the testcase as:

```
uvm_resource_db#(int)::set("SEQ_CNTRL",
"item_count", 10);
```

This essentially allows the sequences to reconfigure themselves based on the scenarios at any point in time. For example, a sequence can retrieve the memory allocation patterns from the Register Model, and target the next set of accesses in the un-allocated regions. The sequences themselves become more reusable from a vertical reuse perspective as they reconfigure themselves based on newer constraints applied in the configuration space.

With the new run time phases in the UVM domain, sequences can be made to run in different phases. This gives a lot of granularity in terms of stimulus control and ensuring that a right set of sequences are executed in different phases. Also, this allows the UVM components themselves to be generic in nature which makes them reusable easily across a wider range of requirements.

To execute a sequence in a specific phase in UVM, one needs to override the *'default_sequence'* string for a specific sequencer for that phase. The easiest mechanism is to use the Resource Database to override the 'default_sequence' string from the test case or in the environment.

```
uvm_config_db
#(uvm_object_wrapper)::set(this,
"<path_to_sequencer>.main_phase",
"default_sequence",
sequence_name>::type_id::get());
```

The same or different sequences can be made to run in different phases easily as well.

```
class phase_test extends test_base;
  typedef uvm_config_db #(uvm_object_wrapper) seq_phase;
  .. .
virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    seq_phase::set(this, "env.seqr.reset_phase",
              "default_sequence", simple_seq_RST::get_type());
    seq_phase::set(this, "env.seqr.configure_phase",
              "default_sequence", simple_seq_CFG::get_type());
    seq_phase::set(this, "env.seqr.main_phase",
              "default_sequence", simple_seq_MAIN::get_type());
  endfunction
endclass
```

Figure 10: Setting Phase-Specific Sequences

3.3 Improving Verification Throughput

With long running simulations and the increasing times which are required to load up complex verification environments, it is desired to derive the maximum possible simulation throughput for each simulation.

3.3.1 Using the Command Line Manager

It is important to be able to propagate changes in the environment without having to recompile the design and also to ensure that these changes can also create new verification patterns. UVM provides for mechanism to capture values from the command line and again propagate them through the testbench through its configuration mechanism. This functionality can be leveraged appropriately by providing appropriate hooks in the testbench so that users are provided with a template to maximize the useful simulation data that can be extracted out of each simulation run.

Configuration Parameters: The Command Line Processor class provides a general interface to the command line arguments that are provided for the given simulation. The *uvm_cmdline_processor* class also provides support for setting various UVM variables from the command line, such as components' verbosities and configuration settings for integral types and strings. As far as overriding configurations settings are concerned, the command line option can only be used for setting the integer/string only in a class derived from *uvm_component*. The control of the testbench configuration of the scalar types through command line can be done as follows:

```
+uvm_set_config_int=<comp>,<field>,<value>
+uvm_set_config_string=<comp>,<field>,<value>
```

An example of the same would be :

```
+uvm_set_config_int=uvm_test_top.*.drv,delay,
10
```

It has to be ensured that the user must register the field which would be overridden using the field automation macros. There should not be any white spaces between the strings passed to the command line argument. This can be very easily extended to the sequence or sequence library specific

parameters which can thus end up saving a lot of simulation cycles. In this case, the user has to explicitly do a 'get' in the respective sequences as shown below.

```
typedef uvm_config_db #(uvm_bitstream_t) seq_cfg_phase;
void'(seq_cfg_phase::get(m_sequencer, <phase_name>,
        "default_sequence.min_random_count",min_random_count));

void'(seq_cfg_phase ::get(m_sequencer, <phase_name>,
      "default_sequence.max_random_count",  max_random_count));

void'(seq_cfg_phase::get(m_sequencer, <phase_name>,
          "default_sequence.selection_mode",  selection_mode));
```

Figure 11: Controlling the Parameters of the Sequence Library

Subsequently, the following command line will change the total number of sequences executed for a specific sequence library to be 25 instead of the default 10.

```
+uvm_set_config_int=uvm_test_top.tb.sequencer
,default_sequence.max_random_count,25
```

Thus, for specific values which are used as knobs and other discrete configuration parameters, additional tests do not have to be written. Different permutations can be tried out from the command line without requiring any recompiles. Debug modes can also be enabled through the same mechanism.

Factory Overrides from the command line: The changes that can be propagated from the command line extends to factory overrides as well. The testbench objects which are registered with the UVM factory can be overridden globally or in a given hierarchy specified by an instance path in the command line.

```
+uvm_set_inst_override=<req_type>,<override_t
ype>,<inst_path>
+uvm_set_type_override=<req_type>,<override_t
ype>
```

An example of the above overrides would be :

```
+uvm_set_inst_override=driver,NewDriver,*.drv
0
+uvm_set_type_override=packet,newPacket
```

3.3.2 Randomization Control and Performance

The cost that one has to pay for constrained randomization is the potential of degradation of simulation performance. The UVM Resources can conveniently be leveraged in the effort of cranking up simulation performance by avoiding redundant randomizations. A complex set of interleaved constraints can cause a significant impact on runtime performance. As the number of random variables and the associated constraints increase, the demand on the constraint solver can increase exponentially.

The UVM Resource mechanism is sometimes used for overriding values of 'random' parameters in the Configuration classes. When defining these classes, the user might not know whether a specific property would be through the Resource mechanism. If it is not overridden, the user might want that property to take a random value. If it is overridden, the user would want to ensure that the value is not subsequently randomized. How does he get the best of both worlds?

For dynamic objects extending from *uvm_object*, the *uvm_config_db::get()* should typically be in the constructor or a method which will subsequently be invoked post the creation of the object. Whenever a 'get' on a specific property succeeds, we know that the property should not be randomized with the other members of the class whenever the class is randomized at a later point in time. This can be ensured by checking for the return value of the *uvm_config_db#::get()*. If the return value denotes a successful retrieval, the rand_mode() of that specific parameter can be turned to '0', so that the variable no longer participates in the randomization.

```
if(uvm_resource_db# (int)::read_by_name("*","
A",value,this))
    A.rand_mode(0);
```

3.3.3 Dynamic Feedback, Coverage Convergence and Modifying Cover Groups

***Coverage Convergence*:**
The SystemVerilog language provides capabilities to query the functional coverage results on the fly. An intelligent reactive testbench can thus use this information obtained through constructs like *get_coverage()* to change its stimulus. Thus, without quitting simulation, the testbench in an automated way identifies holes that it needs to target and changes the constrained random stimulus accordingly. One of the pre-requisites to achieve this is to create your constraints in a way that it can also be modified during simulation. This is made possible if variables are used to model the testbench constraints. Thus by dynamically changing the value of the variables during simulation, the overall stimulus sample space can be expanded or shrunk. Thus, an intelligent reactive testbench can help raise the productivity, by helping the user to cut down on the number of tests to reach his or her coverage goals.

The coverage model itself can be 'set' in the Configuration Database. It can be retrieved easily in the sequences through the Resource Mechanism, queried and based on the coverage query results, the generation of the subsequent sequence items can be biased by modifying specific constraint variables as well as testbench parameters across the environment [4].This can be done by biasing the weights on a *randsequence* branch, or by changing the distributions modeled through variables.

Covergroup manipulation:
The enabling or disabling a covergroup can be controlled through the UVM Resource mechanism. The coverage attributes like 'weight', 'at_least', coverpoint ranges can be changed as well.

```
typedef enum {NOP, LOAD, . ..} op_code;

class transfer extends uvm_sequence_item;
    op_code op;
    . . .
endclass

class coverage_cfg extends uvm_object;
  int enable_coverage;
  int nop_code_weight;
  int nop_code_goal;
 …..
endclass

class coverage_collector extends uvm_component;
 coverage_cfg cfg;

 covergroup instr_cg;
 op_nop : coverpoint instr_word[15:12] { bins op = { nop_op }; }
 op_load : coverpoint instr_word[15:12]{bins op = { load_op }; }
 ...
 endgroup

 function void build_phase(uvm_phase phase);
    if(!uvm_config_db#(coverage_cfg)::get(this, "", "cfg", cfg))
        `uvm_fatal(…)
    if(cfg.enable_coverage) instr_cg cg = new();
 endfunction
 . . .
endclass
```

Figure 12: Modifying coverage shapes

In the code snippet in figure 12, we can see that the *coverage_collector* class is retrieving the Configuration Coverage Model from the Resource Database. The Coverage Model is dumped into the Resource Database from the agent (figure 13) in the *build_phase*. The user sequence then queries the current

coverage results and tweaks the constraint variables so that redundant stimulus is not generated.

```
class my_agent extends uvm_agent;
 coverage_collector cov_db;
 ...
 function void build_phase(uvm_phase phase)
   cov_db = coverage_collector::type_id::create("cov_db", this);
   uvm_resource_db#(coverage_collector)::
                  set("COVERAGE", "cov_db", cov_db, this);
 endfunction
endclass

class rand_sequence extends uvm_sequence;
  coverage_collector cov_db;
  int weight_nop = 1, weight_load = 1;
  rand op_code local_op_code;
  constraint valid_op_code {
         local_op_code dist {
                      NOP := weight_nop;
                      LOAD := weight_load; .  . . }; }
  task body();
       `uvm_do(req, {op = local_op_code;})
  endtask
  . . .
  virtual task post_body();
    if(!uvm_resource_db#(coverage_collector)::
     read_by_name("COVERAGE", "cov_db", cov_db, this))
            `uvm_fatal(...)
    if(cov_db.cg.op_nop.get_coverage() == 100)
        weight_nop = 0;
    . . .
    endtask
endclass
```

Figure 13: Stimulus Guidance through Coverage Feedback

3.4 Synchronizing RTL and Testbench Configuration

3.4.1 Verifying RTL Configurations

Verification engineers have to come up with testbenches to verify RTL which itself is configurable. The 'configurability' of the RTL can vary from a few bus width parameters, or a configurable IP block with optional features to a whole chip with optional interfaces. Some of these can affect the physical interface between the testbench and RTL, for example, Bus widths, number of instances of external interfaces or the number of interrupts. In specific cases, it has to support partially specified RTL configurations, multiple instances of the same RTL module with different configurations, and configurations which does not affect the physical interface, but does affect the way the test bench has to interact with the DUT functionally. For example, FIFO Depth, QoS algorithms. Finally, it is important to track the functional coverage across all these configurations too.

In a UVM testebench, this can be mapped as a UVM RTL configuration class like any testbench configuration class. This can be used to encapsulate the RTL configuration parameters for the design such as bus widths, FIFO sizes, number of I/O ports, etc.

```
class sram_config extends uvm_object;
 rand int num_sram_devices;
 constraint cst_sram_config_valid {
     num_sram_devices inside {1, 2, 4};
  }
 `uvm_object_utils_begin (sram_config)
      `uvm_field_int (num_sram_devices, UVM_DEFAULT)
 `uvm_object_utils_end
…
endclass
```

Figure 14: Class Containing Variable RTL Parameters

The RTL Configuration class itself can be registered to the Configuration Database. It can be retrieved in the Testbench Configuration class which can then subsequently set testbench parameters. The additional methods to load or store the RTL configuration from an external file can be provided. This configuration object is then instantiated in the UVM environment as shown in Figure 15.

```
class sram_subenv extends uvm_env;
  `uvm_component_utils (sram_subenv)
   sram_config   cfg;
   sram_model    rams[];
   // used in build_ph phase because it affects  the
   // structural content of the verification environment
   function void build_ph(uvm_phase phase);
     uvm_config_db#(uvm_object)::get( this,"", "cfg",cfg));
     this.rams = new [cfg.num_sram_devices];
     for (int i = 0; i < cfg.num_sram_devices; i++) begin
        this.rams[i] = sram_model::type_id::create ( .. );
     end
   endfunction
endclass
```

Figure 15: Building Configurable Number of SRAM Models

Now, it is important that the RTL Configuration class which is accessed across the verification environment is used during the compilation of the RTL code as well. Also, the testbench should then exercise its stimulus on the newly configured RTL. This basically calls for a two parse process. The first parse will generate the RTL configurations and the second will apply the configurations. The first parse will involve executing a UVM test which will insert a phase before the *build_phase*. In this phase, the RTL configuration will be randomized, the RTL configuration data packed and written out into a RTL Configuration text file.. Then the test will cause simulation to skip all the remaining phases and jump to the final phase.

```
class my_test extends uvm_test;
    //Add new phase gen_rtl_cfg_phase
    //Insert gen_rtl_cfg_phase before build_phase in the UVM
common domain.
endclass

class rtl_config_gen extends my_test;
   sram_config cfg;
   . . .
   function void gen_rtl_cfg_phase(uvm_phase phase);
       cfg = sram_config::type_id::create("cfg");
       cfg.randomize();
       cfg.byte_pack();
       //Dump the randomized values in a text file
       $fwrite(.....);
   endfunction
   function void build_phase(uvm_phase phase);
       super.build_phase(phase);
       //Jump to the final_phase
       uvm_domain::jump_all(uvm_final_phase::get());
   endfunction
endclass
```

Figure 16: Generating the Randomized RTL Configuration

```
% simv +UVM_TESTNAME=rtl_config_gen
```

The generated file in the user defined phase can be suitably parsed to generate the RTL parameters or compile time macros which can be fed back to the RTL compilation. For the actual testbench simulation, the base test will read back the RTL configuration from the generated file, unpack the data and then 'set' the RTL configuration in the Resource Database.

```
class my_test extends my_test;
   sram_config cfg;
   . . .
   function void gen_rtl_cfg_phase(uvm_phase phase);
     //Read the text file dumped in the first parse
     $fread(.....);
     cfg.byte_unpack(...);
   endfunction
  function void build_phase(uvm_phase phase);
     super.build_phase(phase);
     uvm_config_db#(sram_config)::set(this, "env*", "cfg", cfg);
      ….
    endfunction
  . . .
endclass
```

Figure 17: Loading the Randomized RTL Configuration

In projects where highly configurable IP has to be verified, it is convenient to control and observe the progress of verification across, not only the modes of operation, but also those modes relating to specific RTL configurations. The added benefit of using this method is that constrained randomization and functional coverage collection can be now be used for RTL configurations.

3.4.2 Leveraging Dynamic Reconfiguration

As designs grow in size, memory size and simulation speed are becoming critical issues for many customers. It is becoming increasingly difficult to accommodate large designs in 32-bit memory. Although the entire design may need a 64-bit for simulation, only a portion of the design may be necessary to run many tests. The simulation speed can be improved by removing or replacing part of the design hierarchy with alternate models. The removing portions of the hierarchy by replacing modules with empty modules, simpler modules or higher-level models allows the design size to be reduced, resulting in decreased memory requirements. This has the additional benefit of faster simulation due to the pruned design size.
The Dynamic Reconfiguration provides a flexible mechanism to replace portions of the design hierarchy at runtime. This enables many design configurations to be used without the need to re-compile the design.

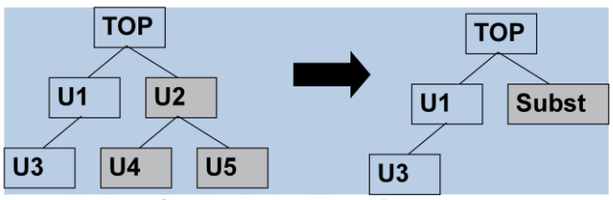

Figure 18: Reconfiguring Design Hierarchy

This requires passing in a configuration file at compile time specifying all the instances that can be substituted or black boxed with empty shells. At run time, one or more 'configuration' file gets passed to the executable which specifies the actual instance hierarchies for that simulation. As in the case of modeling RTL configurations, a similar file dumped out from the first parse can generate the appropriate 'configuration' file which can be used for stubbing out specific hierarchies at runtime. This ensures that the testbench is aware of the changed hierarchy. Additionally, all different permutations and combinations of the actual RTL instances can be tracked in the testbench.

3.5 Effective Resource Management (Bringing it All Together)

For System Level verification, the complexities and the permutations of functionalities that have to be verified increases manifold. Some of these include:

- Exercising modes of the peripherals with all the possible modes of system configuration
- Bandwidth issues in the system when multiple peripheral devices contest for system resources
- IRQ Handling
- Power Modes and how that affects different components
- Memory Allocation Management

Typically, the verification environments need to have their own resource management infrastructure which enables coordination across different components and would have APIs to help the components be aware of all the changes in the verification environment. Mailboxes, TLM ports or shared queues are used for communication across threads but then any form of bi-directional communication is typically restricted to the subscriber and the producer. Also, creating such an infrastructure would typically require a parameterized interface to manage a set of resource descriptors which are doled out and returned as needed by the tests. Creating such an infrastructure is a significant investment in time and resource. With the UVM Resource classes and the associated APIs, the additional investment in creating a Resource Manager is no longer required. As different testbench elements are registered in the Configuration Database, their current status and values can be queried from across the testbench.

The following APIs helps to ensure that different components can reconfigure themselves at any time when there is a relevant change in any of the attributes registered in the Configuration Database.

- *"wait_modified"*: Waits for a configuration setting to be set for a testbench attribute in a specific context and for a specific instance name.
- "*exists":* Checks if a value for a specific testbench attribute is available in an instance with a specific context as a starting point.

As described earlier, the UVM sequences would look up the Resource Database to generate appropriate stimulus based on the current configuration. They can be reactive when the need arises. The coverage model would sample the appropriate cover bins, the scoreboard and protocol checkers would verify and validate the traffic based on the dynamic parameters. With all the components feeding in relevant statistics back to the resource manager, different performance metrics at the system level can be assessed and improved upon. Thus, a significantly smaller yet configurable verification setup can target a bigger chunk of DUT functionality and avoid wasteful debug cycles to rule out false negatives.

4. DEBUG Infrastructure for UVM Resources

There is a lot of functionality and convenience that the UVM Resources provides. Along with the functionality, it is important that appropriate debug hooks are provided so that the user can easily decipher any unexpected behavior is seen in the Testbench.

UVM Configuration debug API's enables the recording of all reads and writes done to the Database. It stores information about the accesses. It records the number of times a resource is read or written at what times from which hierarchical component. A mechanism to turn off recording is provided as simulation performance can be affected because of the same. Based on the information analyzed from the resource dump, one can improvise on the usage of UVM configuration API's and use them more optimally

Figure 19 shows the hierarchy of functions that are used in debugging UVM Resources. The user is not expected to invoke all of these and the pictorial representation basically illustrates the internal debug process. Invoking the *dump()* function of the *uvm_resource_db* triggers the entire process provided the user has set the auditing parameter as arguments when invoking the Resource APIs.

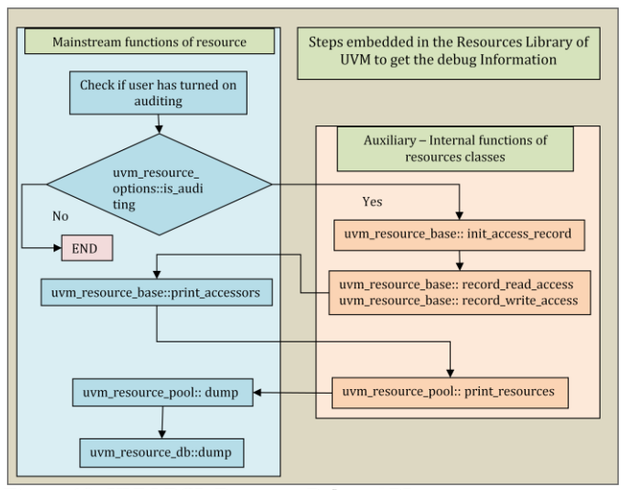

Figure 19: UVM BCL Resource Debug API

The Command line arguments +UVM_CONFIG_DB_TRACE and +UVM_RESOURCE_DB_TRACE would also help trigger this process and the user can find out from the trace messages in STDOUT as to which overrides succeeded,

which failed and what has been registered to the Configuration Database.

```
UVM_INFO
<path_to_uvm_install>/src/base/uvm_resource_db.svh(129) @ 0: reporter [CFGDB/SET] Configuration 'uvm_test_top.env.*.A' (type logic signed[4095:0]) set by uvm_test_top.env = (uvm_bitstream_t) 1011

UVM_INFO
<path_to_uvm_install>/src/base/uvm_resource_db.svh(129) @ 0: reporter [CFGDB/GET] Configuration 'uvm_test_top.env.leaf1.A' (type logic signed[4095:0]) read by uvm_test_top.env.leaf1 = (uvm_bitstream_t) 1011
```

Additionally, after setting all the testbench configurations, the user is provided with API's to check whether the topology of the environment mimics what was expected based on the configuration overrides. This can be done through a simple call to "*print_topology()"* for the structural correctness.

```
uvm_top.print_topology();
factory.print();
```

9. Conclusion

The objective was to present an overview of the UVM Configuration mechanism and demonstrate how the functionality provided in the UVM base classes can be used to model configurable testbench environments. This understanding is then applied to come up with relevant usage scenarios and to demonstrate how complex verification requirements can be met. We have seen very encouraging results with the different applications. With the save-restore flow, we were able to have multiple runs to share single initialization. This gave us a 1.5x Regression Farm Speedup for initialization sequence taking 33% of the complete simulation. For the application of UVM resources in the dynamic reconfiguration flow, there was a considerable reduction in the verification cycle time. The disk space requirement was reduced by around 90% for the example design. Compute requirement was also reduced by approximately 85%. Additionally, we brought in a lot of configurability to our coverage and constraint models to enable faster convergence. The testbench configuration and resource management is one piece of the overall puzzle, but to tackle this well using robust and powerful built-in functionality ensures that the verification engineer is well positioned to tackle the rest of the challenges in SoC verification.

10. REFERENCES

[1] UVM User Guide
[2] UVM Reference Manual
[3] Accellera Verification IP Technical Subcommittee Documents
http://www.accellera.org/apps/org/workgroup/vip
[3] UVM World Website http://www.uvmworld.org/
[4] A Practical Look @ SystemVerilog Coverage – Tips, Tricks, and Gotchas, Doug Smith, John Ansley
[5] Synopsys UVM CES Training
[6] http://www.vmmcentral.org/vmartialarts
[7] Advanced Testbench Configuration with Resources, Mark Glasser, Mohamed Elmalaki