Hierarchal Testbench Configuration Using uvm_config_db

June 2014

Abstract

SoC designs have become extremely complex as more and more IP blocks are integrated into them. This increases the verification challenge manifold in terms of configuration and data handling, as well as architecting and maintaining a large verification environment. Hence it has become very important to create a robust and reusable testbench using a proven methodology that does not just facilitate but also improves the efficiency in verifying different configurations of the device under test (DUT).

Accellera Systems Initiative’s Universal Verification Methodology (UVM), a stable and widely used methodology for architecting testbenches for verification of complex designs helps mitigate these verification challenges to create a scalable, robust and reusable test environment. UVM provides a vast set of macro, policy and base classes that help facilitate the creation of these testbenches, including an easy way to pass objects and variables across testbench hierarchy.

For engineers who are new to verification methodologies or are in the process of adopting UVM, this paper focuses on the UVM configuration mechanism “uvm_config_db”, which helps in passing different class properties across hierarchy testbench components. Through the use of examples, the usage, techniques, and limitations of uvm_config_db are explained.

Introduction

To address the needs of today’s verification architecture, a hierarchical setup of components is necessary to easily move or share configurations and other parameters across different testbench components. To enable this, UVM provides the infrastructure to maintain a database of objects and variables that can be populated and accessed using strings. This is achieved using the UVM syntax uvm_config_db.

Using uvm_config_db, objects can share the handle to their data members with other objects. Other testbench components can get access to the object without knowing where it exists in the hierarchy. It’s almost like making some class variables global or public. Any testbench component can place handles and get handles to objects. In the database, handles are identified by assigned ‘type’ and ‘name’.

Primarily there are two uvm_config_db functions, set() and get(). Any verification component using set() gives others access to the object it has created and controls which components have visibility to the object it has shared. The object can be shared globally or made available to one or more specific testbench components. Verification components using get() check if there is a shared handle matching the used parameters. The get() function defines the object type, the name and hierarchical path to the object searched for.

How to Use It – Different Syntax and Operation

Explicit set() and get() call functions are how you interact with the uvm_config_db. The uvm_config_db class functions are static, so they must be called using the “::” operator.
“cntxt” and "inst_name" are used to specify the storage location or address of the object handle. When used properly these parameters define the hierarchical path to the object data.

“field_name” is the name for the object. It does not have to match the object’s actual name in the source code. Objects using set() and get() must use exactly the same name, otherwise the receiving party (get()) will fail to find the object from uvm_config_db.

“value” is the actual object handle shared through the uvm_config_db. Multiple recipients accessing an object via get(), will access the same object.

"<type>" is used as a parameter for the uvm_config_db class to identify the object from the uvm_config_db. "<type>" which may be either an integral or string, is the class name of the "value". The exception is with enumerated type variables which must use int otherwise the set() won’t work as expected.

The set() specifies the “address” (cntxt & inst_name) where the object handle is stored to control the recipient(s) of the object. The get() has the same flexibility, and can freely select from where the information is to be fetched. In practice get() can be used to fetch an object destined to any component in the hierarchy. Typically for set() and get(), this is used in the “cntxt” field to specify the current instance/scope. set() uses “inst_name” to address the object to the appropriate sub-block in the hierarchy. get() often uses empty ("") inst_name, since it typically is getting the objects destined for itself.

uvm_config_db has two additional functions exists() and wait_modified(). exists() verifies that the defined variable is found in the uvm_config_db. The wait_modified() function blocks execution until the defined variable is accessed with the set() call.
**Automatic Configuration**

UVM also offers build-time configuration of `uvm_component` (and extended) classes utilizing `uvm_config_db`. In automatic configuration, it is sufficient to call `set()` from an upper layer in the hierarchy and the `get()` will automatically execute at build time without requiring an explicit call. Automatic configuration utilizes the `uvm_config_db` feature “under the hood” to pass the configuration values from higher level testbench components in the hierarchy to its lower level components.

For automatic configuration to work there are two important requirements:

- The variable or object must have the appropriate FLAG in `uvm_field_*` macros
- `super()` must be called in `build_phase()`

```verbatim
3 class agents extends uvm_agent;
4 int i4;
5 `uvm_component_utils_begin (agent)
6  `uvm_field_int (i4, UVM_ALL_ON)
7 `uvm_component_utils_end
8
9 virtual function void build_phase(uvm_phase phase);
10     super.build_phase(phase);
11     …
12 endfunction
13 …
14 endclass
```

Once the component properties have the `uvm_field_*` declaration(s) in place with the appropriate FLAG(s), the macro provides the `set_*_local` functionality and `super.build_phase()` calls the `apply_config_settings()` method under the hood. The `apply_config_settings()` method searches for all appropriate config settings matching this component’s instance and for each match, the appropriate `set_*_local` method is called using the matching `uvm_config_db` setting’s “field_name” and “value”.

The `super.build_phase()` method may be replaced with the `apply_config_settings()` method however it is recommended to use the `super.build_phase()` method.

```verbatim
15 class agent extends uvm_agent;
16 int i4;
17 `uvm_component_utils_begin(agent)
18   `uvm_field_int (i4, UVM_ALL_ON)
19 `uvm_component_utils_end
20 virtual function void build_phase(uvm_phase phase);
21     apply_config_settings(1); //No super.build_phase(phase)
22     …
23 endfunction
24 …
25 endclass
```

The implicit `get()` method call will not work in the following instances:

- Missing `uvm_field_*` macro
- FLAG is set to `UVM_READONLY`
- Missing `super.build_phase()` or `apply_config_settings()` in `build_phase()`

Below are log messages generated during the simulation phase because of an explicit `apply_config_settings()` function call:

```verbatim
UVM_INFO @ 0: env.name_agent_1 [CFGAPL] applying configuration settings
UVM_INFO @ 0: env.name_agent_1 [CFGAPL] applying configuration to field i4
```

To set the value for “i4” of the above agent, env would have the `build_phase()` below:

```verbatim
3 function void build_phase (uvm_phase phase);
4     agent_1 = agent::type_id::create("name_agent_1", this);
5     uvm_component_db::set(this, "name_agent_1", "i4", 1111);
6 endfunction
```
During the build phase of the simulation the agent object’s “i4” variable would get value 1111. It is important to note that automatic configuration happens only at build phase.

**Command Line**

Compilation and simulation time are the major contributors to verification overhead. The ability to change the configuration or parameters without being forced to recompile is critical. The UVM class `uvm_cmdline_processor` provides a mechanism to capture the command line argument and pass to verification components the testcase name, verbosity, configuration and other attributes.

Configuration overriding can only be done from the command line for integer and string using the following:

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

There is no way to override the object from the command line, because `uvm_object` cannot be passed to the simulation.

When using the command line argument to set the configuration, make sure that the “<type>” used in `uvm_config_db set()` and `get()` functions is `uvm_bitstream_t` for integer and the “<type>” for string is as shown below:

```vhdl
class env extends uvm_env;
  int a;
  string color;
  ...
  ...
  function new(string name, uvm_component parent);
    super.new(name, parent);
  endfunction
  virtual function void build_phase(uvm_phase parent);
    super.build_phase(phase);
    if(!uvm_config_db #(uvm_bitstream_t)::get(this, "", "a", a))
      `uvm_fatal("GET_NOTSUCCEED", "Get is not successful for a ...");
    if(!uvm_config_db #(string)::get(this, "", "color", color))
      `uvm_fatal("GET_NOTSUCCEED", "Get is not successful for color...");
    `uvm_info("GET_VALUE", $psprintf("The value of a = %d and color = %s", a,color),UVM_LOW);
  endfunction
  ...
endclass
```

The command line argument for the example above is:

```
<simulation command> +UVM_TESTNAME=test +uvm_set_config_int=uvm_test_top.env_i.a, 6 +uvm_set_config_string=uvm_test_top.env_i.color, red
```
The log message generated during simulation is:

```
UVM_INFO @ 0: reporter [UVM_CMDLINE_PROC] Applying config setting from the command line: +uvm_set_config_int=uvm_test_top.env_i, a, 6
UVM_INFO @ 0: reporter [UVM_CMDLINE_PROC] Applying config setting from the command line: +uvm_set_config_string=uvm_test_top.env_i, color, red
```

Cross-Hierarchical Access
The set() and get() parameters "cntxt", "inst _ name" and "field _ name" make it possible to use a number of different paths to the same object. "cntxt" uses actual object hierarchy whereas "inst _ name" and "field _ name" uses the hierarchy path with names given to the objects in create()/new() method. It is good practice to create the objects with the same name as the object name.

When referencing down in hierarchy, it should be enough to use this in "cntxt" and then provide the path and/or names in "inst _ name". "Field _ name" should be used just for the name of the object. When referencing upwards in hierarchy, utilize the uvm_root::get() function to get access to the hierarchy root, and then reference down from there using "inst _ name" parameter.

Figure 5 below clarifies and provides examples how objects can be referenced in uvm_config_db.

![Diagram of Cross-Hierarchical Access](image)

**Figure 5: Options for using “cntxt” and “inst _ name” parameters in set() and get()**

`uvm_config_db` does not actually limit how path field name is shared between "cntxt", "inst _ name" and "field _ name". UVM combines all three of these parameters into one "key" that is used to access the database. This feature makes it possible to reference the same object in multiple different ways using the 3 metacharacters *

### Character Meaning

<table>
<thead>
<tr>
<th>Character</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>0 or more characters</td>
</tr>
<tr>
<td>+</td>
<td>1 or more characters</td>
</tr>
<tr>
<td>?</td>
<td>Exactly 1 character</td>
</tr>
</tbody>
</table>
The illustration below shows using these metacharacters for the same object in `uvm_config_db`.

```plaintext
uvm_config_db#<type>::set(<cntx>, <inst_name>, <field_name>, <object>);
```

**Figure 6: Different path notations to the one and same object**

### Where To Use — Usage and Its Benefits

#### Passing Configuration

`uvm_config_db` is used often to configure agents of the testbench and to pass access to signal interfaces. Agent is a class encapsulating sequencer, driver and monitor. Agent usually takes care of generating and receiving data for an interface. The configuration variables or virtual interface are set at agent from top-level and later the agent is responsible for passing the virtual interface or configuration to other sub-components rather than passing it from top-level as shown in the figure below. Agents are often reused either as VIP blocks or across projects. This means that the receiver (`get`) of the information dictates "type" and "field_name", and source of the information (`set`) must use proper parameters when setting data into `uvm_config_db`. This is also part of the beauty of `uvm_config_db`: agents can be created without knowing where the parameters or signal interfaces are coming from, from where in the testbench hierarchy the agent object exists, what name it has, or how many instances there are in parallel.

**Figure 7: Passing configuration to agent and sub-components**

#### Passing Virtual Interface

Passing the virtual interface across the verification component is the most common requirement when creating the reusable verification environment. The preferred approach of doing this in UVM is to push the virtual interface into the configuration database from top-level. This is because top-level module is not `uvm_component` hence the context is "null" and the instance is the absolute hierarchal path of the component where the virtual interface is assigned.

The absolute hierarchal instance of the component starts with "uvm_test_top." Because the environment is usually instantiated by test and agent, it can extract the virtual interface from the configuration database as shown in the example on the next page.
Hierarchical Testbench Configuration Using uvm_config_db

2 module tb_top();
3 svt_axi_if vif();
4 ...
5 ...
6 initial begin
7   uvm_config_db #(virtual svt_axi_if)::set(null, "uvm_test_top.env.m_agent_0", "vif", vif);
8 end
9 endmodule
10
class axi_agent extends uvm_agent;
11   virtual svt_axi_if vif();
12 ...
13   virtual function _void build_phase(uvm_phase phase);
14     super.build_phase(phase);
15     ...
16     if(!uvm_config_db#(virtual svt_axi_if)::get(this, "", "vif", vif))
17       `uvm_fatal("AXI_AGENT:NOVIF", "The virtual interface get is not successful");
18   uvm_config_db#(virtual svt_axi_if)::set(this, "driver", "vif", vif);
19   uvm_config_db#(virtual svt_axi_if)::set(this, "monitor", "vif", vif);
20 endfunction
21 endclass
22

Event Synchronization

uvm_config_db is used to make the object available for others, it does not create new copies of the object. Figure 8 below shows how event-object created by Object A is also made available to Objects X, Y and Z through the uvm_config_db. When Object A chooses to use trigger() for the event object, others can detect it because they have access to exactly the same object. This demonstrates how the same object “event” is referenced from four different objects with three different instance names.

![Event Synchronization Diagram](image)

Some attention needs to be paid that set() is called before get() for a specific item, otherwise get() will fail. Values passed through uvm_config_db before run_phase() need to take into account that build_phase() constructs objects from top to bottom. This is often the desired order, since settings and configurations are usually set from higher levels to lower levels via agents. During the simulation, use of set() and get() need to be synchronized/timed by the normal testbench operation or by using events to create a synchronization mechanism.

Limitations

uvm_config_db can be used anywhere in the hierarchy. The first parameter of set() and get() functions, "cntxt", however needs to be of type class uvm_component (or extended from that); "cntxt" parameter is often given value utilizing class member this. So if set() or get() functions are used outside uvm_component extended object, "cntxt" parameter can be given value using uvm_root::get(), or just value "null".
Figure 9: set() and get() functions need “cntxt” parameter of type uvm_component or null

One common usage of uvm_config_db outside uvm_components, is delivering values from hdl_top to the testbench, including access to interfaces instantiated on hdl_top. Though hdl_top is not extended from any UVM class, uvm_config_db can still be utilized and communication with the UVM part of the testbench is possible.

If set() or get() function is used with “cntxt” parameter not pointing to object of uvm_component extended classes, there will be a compile error as shown below.

```
Figure 10: Example error messages when trying to use “this” for non-uvm_component in set/get
```

Problems, Errors and Debug

Even though operation and use of set() and get() functions with uvm_config_db are logical and quite simple, uvm_config_db related debugging is often needed. Some errors may stop the compile or simulation making them easy to find, as opposed to a coding error that simulates without error even though the get() function was receiving incorrect objects. Some common types of errors are:

- Compile time errors
  - Parameter type does not match provided T value
  - Trying to use this-pointer from class not extended from uvm_component
- Simulation time errors
  - get() does not find what was set using set() due to misspelling of “inst_name” or “field value”
  - null object access attributed to get() used before set()
Synopsys’ VCS Discovery Visualization Environment (DVE) has built-in support for UVM debug. Using the GUI, it is possible to get list of “Set calls without Get” and “Get calls without Set”. These lists help to find and detect errors in the testbench. Figure 11 below shows the DVE UVM debug dialog window.

The UVM command line option +UVM _ CONFIG _ DB _ TRACE makes all set() and get() calls visible in the simulation log. However doing this makes the log file too verbose and difficult to interpret. For this reason tracing is typically turned on only when finding a specific uvm _ config _ db problem. Below is an example of log messages printed out when set() and get() functions are executed.

```
39 UVM _ INFO /global/apps4/vcs _ 2012.09-3/etc/uvm-1.1/base/uvm _ resource _ db.svh(129) @ 0:
40 reporter [CFGDB/SET]
41 Configuration 'uvm_test_top.env.name_agent_1.i_of_env' (type int)
42   set by uvm_test_top.env.name_agent_1 = (int) 1
43
44 UVM _ INFO /global/apps4/vcs _ 2012.09-3/etc/uvm-1.1/base/uvm _ resource _ db.svh(129) @ 0;
45 reporter [CFGDB/GET]
46 Configuration 'uvm_test_top.env.name_agent_1.i_of_env' (type int)
47   read by uvm_test_top.env.name_agent_1 = (int) 1
```

Sometimes it may help just to print out the name of the object. UVM object has functions get _ name() and get _ full _ name(). By using these, it can be verified manually that names used in the source code and named objects at runtime match. Below is an example of how to print the object’s name.

```
49 $display("this.get_name=%0s, this.get_full_name= %0s", this.get_name(),
   this.get_full_name());
```

**Conclusion**

When using UVM you can’t avoid uvm _ config _ db. So it’s better to get a solid understanding about what the set() and get() functions of the uvm _ config _ db do and how you can use them more efficiently in building your testbench. Below are some do’s and don’ts found to be useful when using UVM.

Do’s:

- For simplicity and to avoid confusion, use the “field name” as the variable name
- Investigate an upshot warning/error on an unsuccessful get() method call
- Set the configuration variable needed across the verification component in the agent’s test environment to enable the agent to later set its sub-components
Don'ts:

- Avoid using the **uvm_config_db** mechanism excessively as it may cause performance issues.
- Avoid using the automatic configuration or implicit **get()** method call.

Apart from the above recommendations, it is recommended to use a UVM-aware GUI-based debugging tool such as Synopsys' VCS Discovery Visualization Environment (DVE). As part of Synopsys Professional Services we have used these concepts across multiple customer engagements to successfully deploy UVM. For more information on these services, see [www.synopsys.com/services](http://www.synopsys.com/services).

**Appendix**

Below is a sample UVM environment showcasing the examples presented earlier. **set()** and **get()** functions utilize only integers (**int**) though classes and interfaces that would normally be used.

```verilog
1 // Usage
2 // vcs -R -sverilog -ntb_opts uvm-1.1 -debug_all +vcs+vcdpluson -1 sim.log
3 // -q example.sv +UVM_TESTNAME=test_a
4 import uvm_pkg::*;
5
6 module dut;
7    int dut_int = 10;
8    initial uvm_config_db#(int)::set(uvm_root::get(),
9        "*",
10        "from_dut",
11        dut_int
12    );
13 endmodule
14
15 module top;
16    initial run_test();
17    dut i_dut();
18 endmodule
19
20 class agent extends uvm_component;
21    int i1_agent, i2_agent, i3_agent; // to receive values from uvm_config_db
22    int i4; // to use automatic configuration
23    uvm_event new_values; // to signal new step in test (env->agent)
24    `uvm_component_utils begin (agent)
25        `uvm_field_int(i4, UVM_ALL_ON)
26    `uvm_component_utils_end
27 function new (string name, uvm_component parent);
28    super.new(name, parent);
29 endfunction
30 function void build_phase (uvm_phase phase);
31    super.build_phase(phase);
32    uvm_config_db#(uvm_event)::get(this, ",", "new_values", new_values);
33    if (new_values == null)
34        `uvm_fatal(get_name(),
35            "new_values must be set in uvm_config_db"
36        );
37        $display(" Agent \"%0a\" got Automatic configuration: i4=%0d",
38            get_name(),
39            i4
40        );
41 endfunction
42 task run_phase (uvm_phase phase);
43    while (1)
44    begin
45        uvm_config_db#(int)::get(this,
46            "",
47            "i_of_env",
48            i1_agent
49    );
```
Hierarchal Testbench Configuration Using \texttt{uvm\_config\_db}

```verilog
uvm\_config\_db\#(int)::get(uvm\_root::get(),
    "uvm\_test\_top.env.name\_agent\_1",
    "i\_of\_env",
    i\_agent
);
$display(" Agent \"%0s\" got: %0d (and stole %0d from agent1)",
    get\_name(),
i1\_agent,
i2\_agent
);

uvm\_config\_db\#(int)::get(this, "", "from\_dut", i3\_agent);
$display(" Agent \"%0s\" got %0d from DUT",
    get\_name(),
i3\_agent
);
new\_values.wait\_trigger();
end
endtask
endclass
class env extends uvm\_env;
`uvm\_component\_utils(env)
agent agent\_1, agent\_2, agent\_3;
int i1\_env=1, i2\_env=2, i3\_env=3,
i4\_env=4, i5\_env=5, i6\_env=6,
i7\_env=7, i8\_env=8;

uvm\_event new\_values;
function new(string name, uvm\_component parent);
    super\_new(name, parent);
endfunction
function void build\_phase (uvm\_phase phase);
    agent\_1 = agent::\_type\_id::create("name\_agent\_1, this);
    agent\_2 = agent::\_type\_id::create("name\_agent\_2, this);
    agent\_3 = agent::\_type\_id::create("name\_agent\_3, this);
    set\_config\_int("name\_agent\_1", "i4", 1111);
    set\_config\_int("name\_agent\_2", "i4", 2222);
    set\_config\_int("name\_agent\_3", "i4", 3333);
    new\_values = new("new\_values");
    //Share event through uvm\_config\_db with agents
    uvm\_config\_db\#(uvm\_event)::set(this,
        "name\_agent\_*",
        "new\_values",
        new\_values
    );
endfunction

//Share run\_phase event through uvm\_config\_db
//with one specific object

function void run\_phase (uvm\_phase phase);
    phase.raise\_objections(this);
    $display(" --- 1, 2, 3 to every agent separately --- ");
    uvm\_config\_db\#(int)::set(agent\_1,
        "", "i\_of\_env",
        i1\_env
    );
    uvm\_config\_db\#(int)::set(this,
        "name\_agent\_2",
        "i\_of\_env",
        i2\_env
    );
    uvm\_config\_db\#(int)::set(uvm\_root::get(),
        "uvm\_test\_top.env.name\_agent\_3",
        "i\_of\_env",
        i3\_env
    );
    // uvm\_config\_db: share data with multiple objects using regexp
```
new_values.trigger(); #1;
$display(" --- 4 to every agent, regexp name_agent_? --- ");
uvm_config_db$(int)::set(this,
    "name_agent_?",
    "i_of_env",
    i4_env
);
new_values.trigger(); #1;
$display(" --- 5 to every agent, regexp name_agent* --- ");
uvm_config_db$(int)::set(this,
    "name_agent_*",
    "i_of_env",
    i5_env
);
new_values.trigger(); #1;
$display(" --- 6 to every agent, regexp *agent* --- ");
uvm_config_db$(int)::set(uvm_root::get(),
    "*agent*",
    "i_of_env",
    i6_env
);
// uvm_config_db: share data with everyone
new_values.trigger(); #1;
$display(" --- 7 to everyone, regexp *--- ");
uvm_config_db$(int)::set(uvm_root::get(),
    "*", 
    "i_of_env",
    i7_env
);
new_values.trigger(); #1;
$display(" --- 8 to everyone, regexp *--- ");
uvm_config_db$(int)::set(null,
    "*",
    "i_of_env",
    i8_env
);
new_values.trigger();

endtask.
endclass

class test_a extends uvm_test;
  uvm_component_utils (test_a)
env env;
function new (string name="test_a", uvm_component parent=null);
  super.new (name, parent);
  env = new("env", this);
endfunction
function void end_of_elaboration();
  print();
endfunction
task run_phase(uvm_phase phase);
  #1000; global_stop_request();
endtask
endclass

References
1 Accellera Systems Initiative Universal Verification Methodology (UVM) 1.1 User’s Guide, May 18, 2011
2 Accellera Systems Initiative Universal Verification Methodology (UVM) 1.1 Class Reference Manual, June 2011