White Paper

SYNOPSYS®
Accelerating Innovation

Hierarchal Testbench Configuration Using uvm_config_db

June 2014

Authors

Hannes Nurminen
Professional Services, Synopsys

Satya Durga Ravi
Professional Services, Synopsys

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.

```
uvm_config_db#(<type>)::set(uvm_component    cntxt,
                            string           inst_name,
                            string           field_name,
                            <type>           value)

uvm_config_db#(<type>)::get(uvm_component    cntxt,
                            string           inst_name,
                            string           field_name,
                            ref              value)
```

Figure 1: set() and get() function syntax

“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.

```
typedef enum {single,incr,wrap4,incr4,wrap8,incr8,wrap16,incr16}hburst_t
hburst_t      hburst;
uvm_config_db#(int)::set(this,“a”,“hburst”,incr);
```

Figure 2: config_db for enum type

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#(int)::set(this,“my_subblock_a”,“max_cycles”,max_cycles)
uvm_config_db#(int)::get(this,“”, “max_cycles”,max_cycles)
```

Figure 3: set() and get() typical use

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.

```
uvm_config_db#(int)::exists(this,“my_subblock_a”,“max_cycles”)
uvm_config_db#(int)::wait_modified(this,“my_subblock_a”,“max_cycles”)
```

Figure 4: exists() and wait_modified() typical use

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()**

```
class agents extends uvm_agent;
 int  i4;
 `uvm_component_utils_begin (agent)
   `uvm_field_int (i4, UVM_ALL_ON)
 `uvm_component_utils_end

 virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    ...
  endfunction
...
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.

```
class agent extends uvm_agent;
 int  i4;
 'uvm_component_utils_begin(agent)
    `uvm_field_int (i4, UVM_ALL_ON)
 'uvm_component_utiles_end
  virtual function void build_phase(uvm_phase phase);
    apply_config_settings(1); //No super.build_phase(phase)
    ...
  endfunction
 ...
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:

```
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:

```
function void build_phase (uvm_phase phase);
  agent_1 = agent::type_id::create("name_agent_1", this);
  uvm_component_db#(int)::set(this, "name_agent_1", "i4", 1111);
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>,<filed>,<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:

```
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_NOTSUCC”, “Get is not successful for a ….”);
   if(!uvm_config_db #(string)::get(this, “”, “color”, color))
    `uvm_fatal(“GET_NOTSUCC”, “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

class test extends uvm_test;
int a = 2;
string color =”blue”;
env env_i;
…
…
virtual function void build_phase(uvm_phase phase);
 super.build_phase (phase);
 env_i = env:type_id::create(“env_i”, this);
 uvm_config_db#(uvm_bitstream_t)::set(this, “env_i”, “a”, a);
 uvm_config_db#(string)::set(this, “env_i”, “color”, color);
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:

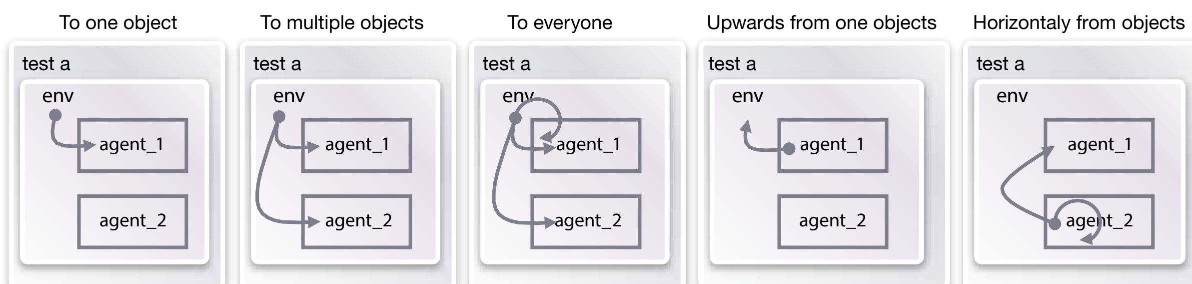
```
UVM _ INFO @ 0: reporter [UVM _ CMDLINE _ PROC] Applying config setting from the
command line: +uvm _ set _ config _ int=uvm _ test _ top.env _ i, a, 6

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`**.



Set to one	**uvm_config_db#(int)::set(uvm_root::get(),**	"uvm_test_top.env.name_agent_1",	"i_of_env", my_var);
	uvm_config_db#(int)::set(this,	"name_agent_1",	"i_of_env", my_var);
	uvm_config_db#(int)::set(agent_1,	"",	"i_of_env", my_var);
Set to multiple	**uvm_config_db#(int)::set(this,**	"name_agent_*",	"i_of_env", my_var);
	uvm_config_db#(int)::set(uvm_root::get(),	"*agent*",	"i_of_env", my_var);
Set to everyone	**uvm_config_db#(int)::set(uvm_root::get(),**	"*",	"i_of_env", my_var);
	uvm_config_db#(int)::set(null,	"*",	"i_of_env", my_var);
Get to me	**uvm_config_db#(int)::get(this,**	"",	"i_of_env", my_var);
Get to subblock	**uvm_config_db#(int)::get(this,**	"driver",	"i_of_env", my_var);
Get to other	**uvm_config_db#(int)::get(uvm_root::get(),**	"other_agent",	"i_of_env", my_var);
Get global	**uvm_config_db#(int)::get(uvm_root::get(),**	"*",	"i_of_env", my_var);

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 `*,+,?`. The table below determines the significance of each metacharacter:

Character	Meaning
*	0 or more characters
+	1 or more characters
?	Exactly 1 character

The illustration below shows using these metacharacters for the same object in **uvm _ config _ db**.

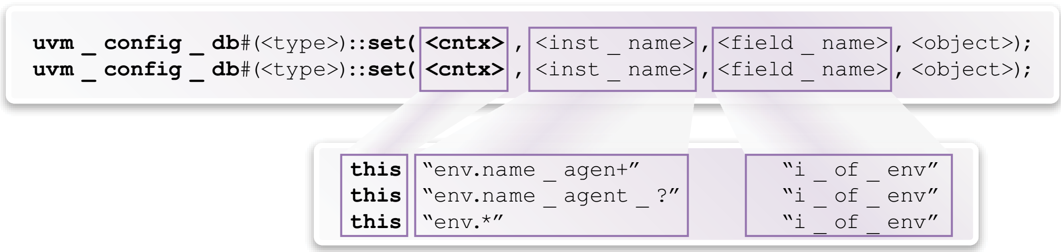


Figure 6: Different path notations to the one and same object

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.

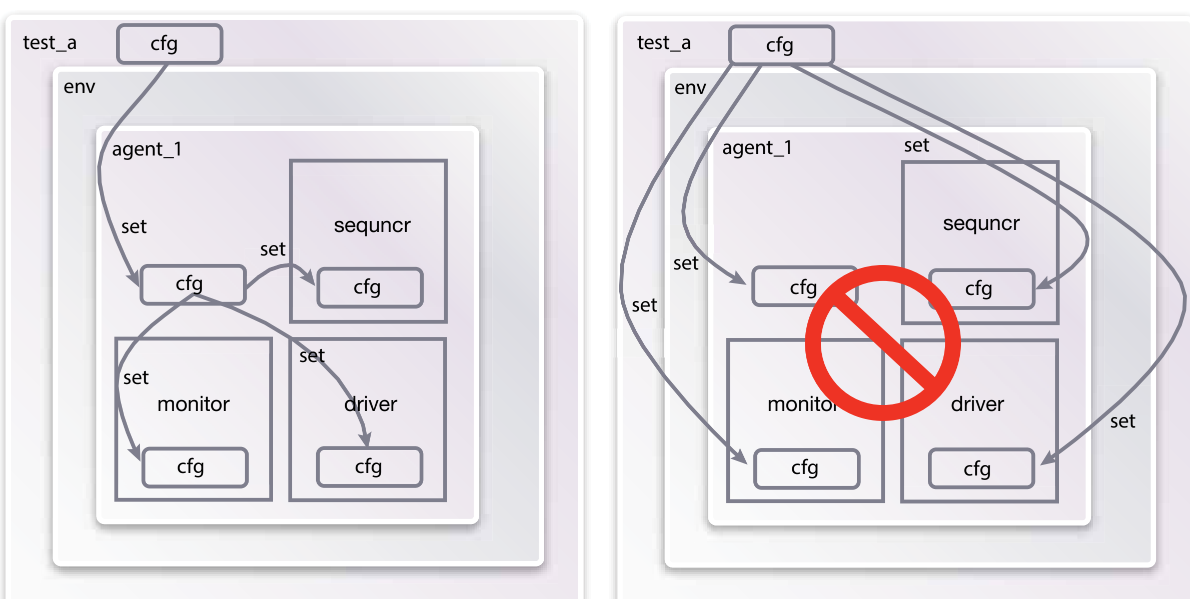


Figure 7: Passing configuration to agent and sub-components

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

```
module tb_top();
svt_axi_if vif();
…
…
initial begin
  uvm_config_db #(virtual svt_axi_if)::set(null, "uvm_test_top.env.m_agent_0", "vif", vif);
end
endmodule

class axi_agent extends uvm_agent;
  virtual svt_axi_if vif();
  …

  virtual function_void build_phase(uvm_phase phase);
    super.build_phase(phase);
     …
     if(!uvm_config_db#(virtual svt_axi_if)::get(this,"","vif", vif))
      `uvm_fatal("AXI_AGENT:NOVIF", "The virtual interface get is not successful");
     uvm_config_db#(virtual svt_axi_if)::set(this, "driver","vif",vif);
     uvm_config_db#(virtual svt_axi_if)::set(this,"monitor","vif",vif);
  endfunction
endclass
```

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.

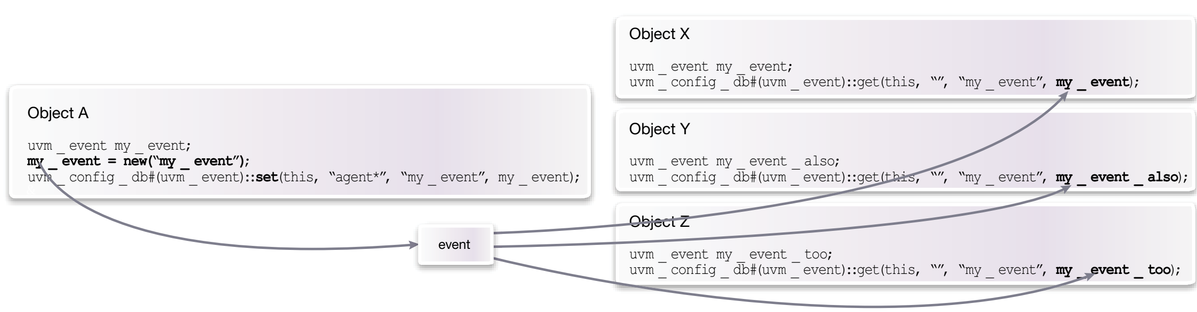


Figure 8: uvm_config_db shares handles to existing object

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`".

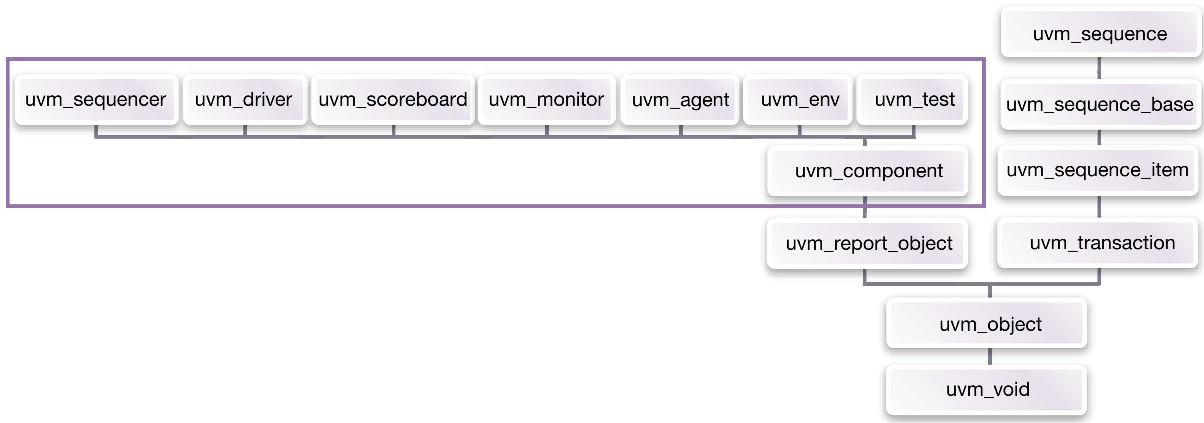


Figure 9: set() and get() functions need "cntxt" parameter of type uvm_component or null

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.

```
Error-[SV-IUOT] Illegal us of this
  test.sv, 9
  'this' can only be used in a class method.

Error-[ICTTFC] Incompatible complex type usage
  test.svh, 281
  Incompatible complex type usage in task or function call.
  The following expression is incompatible with the formal parameter of the
  function. The type of the actual is 'class
  my_pkg::bus_slave', while the type of the formal is 'class
  uvm_pkg::uvm_component'. Expression: this
  Source info: uvm_pkg_uvm_config_db_2116934237::get(this, "\000",
  this.get_full_name(), my_agent)
```

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.

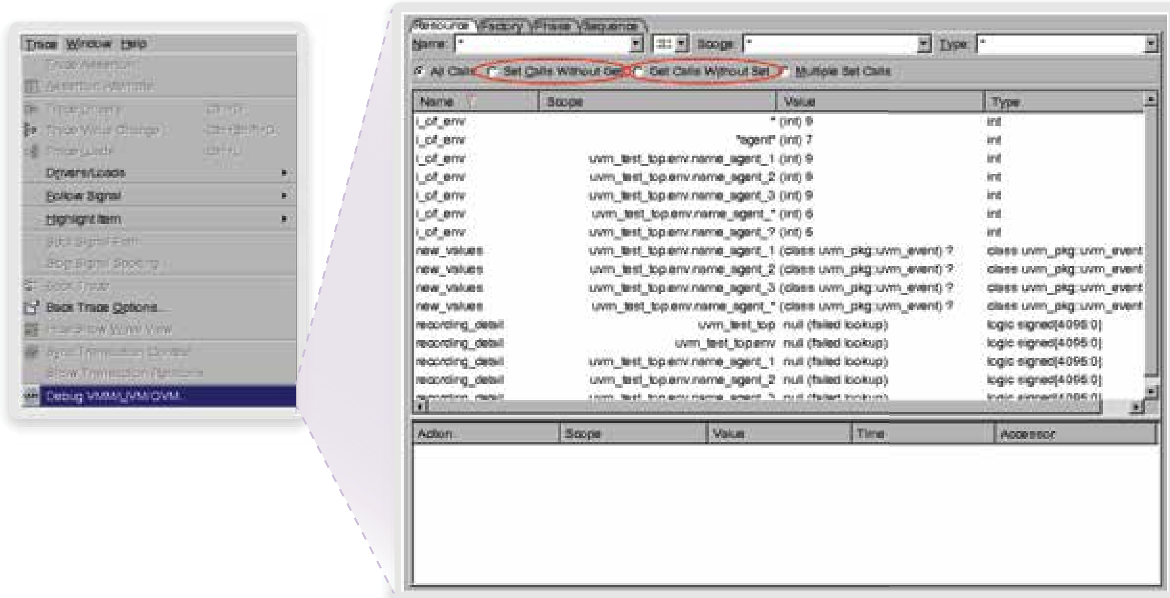


Figure 11: Synopsys' VCS DVE UVM debug dialog window

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.

```
UVM_INFO /global/apps4/vcs_2012.09-3/etc/uvm-1.1/base/uvm_resource db.svh(129) @ 0:
  reporter [CFGDB/SET]
  Configuration 'uvm_test_top.env.name_agent_1.i_of_env' (type int)
     set by uvm_test_top.env.name_agent_1 = (int) 1

UVM_INFO /global/apps4/vcs_2012.09-3/etc/uvm-1.1/base/uvm_resource_db.svh(129) @ 0;
  reporter [CFGDB/GET]
  Configuration 'uvm_test_top.env.name_agent_1.i_of_env' (type int)
     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.

```
$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.

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.

```
// Usage
// vcs -R -sverilog -ntb_opts uvm-1.1 -debug_all +vcs+vcdpluson
-l sim.log
// -q example.sv +UVM_TESTNAME=test_a
import uvm_pkg::*;

module dut;
  int dut_int = 10;
  initial uvm_config_db#(int)::set(uvm_root::get(),
                              “*”,
                              “from_dut”,
                              dut_int
                              );
endmodule

module top;
  initial run_test();
  dut i_dut();
endmodule

class agent extends uvm_component;
  int i1_agent, i2_agent, i3_agent; // to receive values from uvm_config_db
  int i4; // to use automatic configuration
  uvm_event new_values; // to signal new step in test (env->agent)
  `uvm_component_utils_begin (agent)
      `uvm_field_int(i4, UVM_ALL_ON)
  `uvm_component_utils_end
  function new (string name, uvm_component parent);
      super.new(name, parent);
  endfunction
  function void build_phase (uvm_phase phase);
      super.build_phase(phase);
      uvm_config_db#(uvm_event)::get(this, “”, “new_values”, new_values);
      if (new_values == null)
          `uvm_fatal(get_name(),
                  “new_values must be set in uvm_config_db”
                  );
      $display(“ Agent \”%0s\” got Automatic configuration: i4=%0d”,
              get_name(),
              i4
              );
  endfunction
  task run_phase (uvm_phase phase);
      while (1)
          begin
              uvm_config_db#(int)::get(this,
                                  “”,
                                  “i_of_env”,
                                  i1_agent
                                  );
```

```
                uvm_config_db#(int)::get(uvm_root::get(),
                              “uvm_test_top.env.name_agent_1”,
                              “i_of_env”,
                              i2_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
  task run_phase (uvm_phase phase);
      phase.raise_objections(this);
      //uvm_config: share data with one specific object
      $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();
        phase.drop _ objection(this);
  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