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 Methodoloveefic.6(i)-12Vinreleto create a scalable, robust and reusab 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 _ confg _ db**", which helps in passing different class properties across hierarchy testbench components. Through the use of examples, the usage, techniques, and limitations of **uvm _ confg _ 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 _ confg _ db**.

Using **uvm _ confg _ 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 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 _ confg _ db**. The uvm _ **confg _ db** class functions are static, so they must be called using the "::" operator.

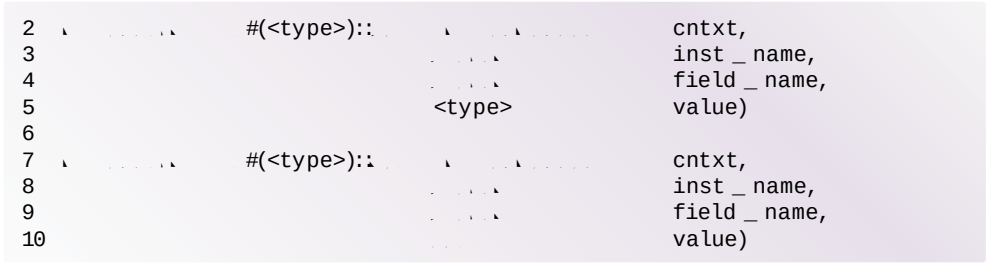

Figure 1: set() and get() function syntax

“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.

“ ¿ H O G B Q 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 _ confg _ db**.

“ Y D O”s the actual object handle shared through the **uvm _ confg _ db**. Multiple recipients accessing an object via **get()**, will access the same object.

“<type> ” is used as a parameter for the **uvm _ confg _ db** class to identify the object from the uvm _ **confg _ db**. “<type>” which may be either an integral or string, is the class name of the “ Y D O”s. The exception is with enumerated type variables which must use int otherwise the set() won’t work as expected.

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.

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

uvm _ confg _ db has two additional functions exists() and **wait _ modifed()**. exists() verifies that the defined variable is found in the **uvm _ confg _ db**. The **wait _ modifed()** function blocks execution until the defined variable is accessed with the set() call.

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

Automatic Configuration

UVM also offers build-time configuration of uvm _ component (and extended) classes utilizing **uvm _ confg _ 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 _ **confg _ 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 _ feld _ *** macros
- super() must be called in **build _ phase()**

```
3 class agents extends uvm _ agent ;
4 int i4;
5 ` uvm _ component _ utils _ begin (agent)
6 ` uvm _ feld _ int (i4, UVM _ ALL _ ON )
7 ` uvm _ component _ utils _ end
8
9
```

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:

```
XYP B VHW B FRQ¿J B LQW  FRPS!  ¿OHG!  YDOXH!
XYP B VHW B FRQ¿J B VWULQJ  FRPS!  ¿HOG!  YDOXH!
```

There is no way to override the object from the command line, because XYP B REMHFW cannot be passed to the simulation.

When using the command line argument to set the configuration, make sure that the "<type>" used in **uvm_confg_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 FRORU
 …
 …
 function new (string name, uvm_component SDUHQW
  super.new QDPH SDUHQW
 endfunction

 virtual function void build_phase (uvm_phase SDUHQW
  super.build_phase (phase);
  if (!uvm_confg_db #( uvm_bitstream_t):: get( this , "", "a", a))
  `  uvm_fatal !uvm_config_db #( get( this LV QRW VXFFHVVF
```

The log message generated during simulation is:

```
UVM_INFO @ 0: UHSRUWHU >8 90 B &0'/,1( B 35$2&(66@ $SSO\LQJ FRQ¿J setting IURP the
 FRPPDQGOLQH XYP B VHW B FRQ¿J B LQW XYP B WHVW B WRS HQY B L a,W6

UVM_INFO @ 0: UHSRUWHU >8 90 B &0'/,1( B 35$2&(66@ $SSO\LQJ FRQ¿J setting IURP the
 FRPPDQGOLQH XYP B VHW B FRQ¿J B VWULQJ XYP B WHVW B WRS FROOHFWRU HQY B L
```

Cross-Hierarchical Access

The set() and **get()** parameters “cntxt”, “inst_name” and “¿HOG B QDPH” make it possible to use a number of different paths to the same object. “cntxt” uses actual object hierarchy whereas “inst_name” and “¿HOG B QDPH” 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/

```
2 module WE B WRS
```

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

One common usage of **uvm _ confg _ db** outside uvm _ components , is delivering values from K G O B W0 R S the testbench, including access to interfaces instantiated on K G O B .T R S through K G O B W R S extended from any UVM class, **uvm _ confg _ 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 _ confg _ db** are logical and quite simple, **uvm _ confg _ 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 “ ¿ H O G Y D O X H
  - **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: Synopys' VCS DVE UVM debug dialog window

The UVM command line option +8 9 0 B &2 1),* B '% B 7 5 8 & 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 _ confg _ db** problem. Below is an example of log messages printed out when set() and **get()** functions are executed.

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

```
Q H Z B Y D O X H trigger(); #1;
$display ‡        W R  H Y H U \  D J H Q W   U H J H [ S  Q D P H  B  D J H Q W  B  "        ‡
uvm _ confg _ db#( int  )::set( this  ,
                  "name _ agent _ ?",
                  ‡ L  B  R I  B  H Q Y ·
                  i4 _ env
                  );
Q H Z B Y D O X H trigger(); #1;
$display ‡        W R  H Y H U \  D J H Q W   U H J H [ S  Q D P H  B  D J H Q W        ‡
uvm _ confg _ db#( int  )::set( this  ,
                  "name _ agent _ *",
                  ‡ L  B  R I  B  H Q Y ·
                  i5 _ env
                  );
Q H Z B Y D O X H trigger(); #1;
G L V S O D \  ‡        W R  H Y H U \  D J H Q W   U H J H [ S   D J H Q W        ‡
uvm _ confg _ db#( int  )::set( uvm _ root::get(),
                  "*agent*",
                      );
H p 0      Q H Z B Y D O X H trigger(); #1;–Ü$QªÊïïð••      uvm _ confg _ db#)  •  p(˜V hH  W    Pp ð W
```

Synopsys, Inc. • **700 East Middlefield Road** • **Mountain View, CA 94043** • **www.synopsys.com**

©2014 Synopsys, Inc. All rights reserved. Synopsys is a trademark of Synopsys, Inc. in the United States and other countries. A list of Synopsys trademarks is available at http://www.synopsys.com/copyright.html. All other names mentioned herein are trademarks or registered trademarks of their respective owners.

06/14.AP.CS3989.