2015 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

An Easy VE/DUV Integration Approach

Uwe Simm
Software Architect
Cadence Design Systems, Inc.
Email: uwes@cadence.com

***Abstract*- This paper outlines a simple and clear approach to connect a device under verification (DUV) and a verification environment by utilizing an *interface registry*. At the same time, it has the flexibility to handle environments ranging from the module to the system level. The paper includes an example that demonstrates that module to system reuse is significantly simplified.**

I. INTRODUCTION

Most of the current methodology enhancements in the module verification context target the creation of the environment itself. The traditional verification methodologies eRM/OVM/UVM focus on the creation aspect of the verification environment and provide several building blocks that simplify the implementation of a verification environment (VE). However, when it comes to the integration of the verification environment into the testbench, which includes the DUV and verification environment, the integration and building the connectivity is not as structured or simple as for the environment creation itself.

Several methods exist for each methodology/language, each with different characteristics in terms of horizontal/vertical reuse potential, simplicity, or coding effort. Generally, there is a growing coding effort when module-level environments, or universal verification components (UVCs) are reused in subsystem or system contexts. This coding effort typically arises from the fact that paths to interfaces, number of instances, and so on, have to be recoded or reworked, and/or special infrastructure/testbenches need to be created for each level.

Traditional methods to connect a DUV and testbench include:

- Using interface instance(s) at the testbench top-level, with the connection by way of DUV module ports, and propagation into the testbench with a custom propagation function: `assign_vi`.

  This method relies upon the fact that all *interesting* interfaces are externally visible (or are already embedded) and that the propagation function `assign_vi`, is properly implemented. Reusing the testbench at a different level (subsystem or system, or in different configuration) requires several manual adjustments.

- Using interface instance(s) at the testbench top-level, with the connection by way of DUV module ports (similar to the previous approach). However, instead of using a custom propagation function, the distribution of the interfaces is done with a configuration object, or with a configuration database setting.

- Using a *Hierarchical interface ports* pattern in the DUV. In this pattern, every DUV module has an interface instance in the port list, and all interfaces of interest are grouped and routed through this interface port. As a result, a large hierarchical interface tree is maintained. Distribution into the testbench is done either hierarchical, using a function call, or a configuration object/configuration database.

- Using a direct specification of out-of-module-reference (OOMR) to the DUV interfaces inside the testbench. This is the least reusable method, but is still used by some engineers.

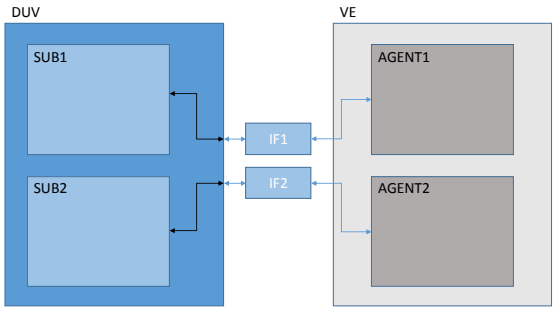

Figure 1: A direct connection between the DUV and the VE.

Interfaces instantiated at the Testbench level are to the DUV Top only, and are passed as a Virtual IF to the Testbench.

II. BUILDING BLOCKS

The *interface registry* approach we present here, builds upon a set of well-known building blocks that orchestrate a simple solution for the typical interface hookup task in verification projects. Testbench creators can hide some of the complexity with non-intrusive `bind`, interface self-registration, and the registry. This reduces the amount effort required for users to perform this task.

The interface registry approach requires just two specific blocks of functionality, as follows:

A. Interface self-registration

SystemVerilog interfaces used in the DUV and testbench need to be enhanced so that each instance automatically registers itself with a database. As a result, all instantiated interfaces are available for connection to the testbench just by making the instance. This enhancement is only required for the interfaces that should be used/connected by the VE. Since there is no downside to having the self-registration active, one can add the self-registration code into any new interface.

For SystemVerilog-LRM experts, it should be noted that self-references to interfaces (similar to `this` for class instances) are not yet covered by the SystemVerilog-LRM. However, all major vendors do support this concept. Unfortunately, each vendor's syntax is different from each other, but since self-registration is only a single line of code and in a central place, conditional code can avoid incompatibilities until the LRM narrows down the syntax definition. The following figure illustrates this concept.

```
interface clk_intf(output  bit clk);
   // custom signals go here
   // all interface instances will register itself with ~ifname~
   import cdns_vif_registry::*;
   const string ifname = "protocol";
   function automatic void register();
      virtual     clk_intf vif;
      vif = clk_intf; // interface self reference
      cdns_vif_registry::cdns_vif_db#(virtual
      clk_intf)::register_vif(vif,$sformatf("%m.%s",ifname));
   endfunction

   initial register();
endinterface
```

Figure 2: Example code with self-registration of the interface.
The interface registers itself with the instance name appended with `protocol`.

B. Database to store the references to the interfaces

The database itself provides two main methods: `register_vif()` and `retrieve_vif()`. The `register_vif()` is operated by the interface instances, and the `retrieve_vif()` is operated by the testbench/VIP, as illustrated in the following figure.

The registry is provided as a Cadence add-on source code package, and builds upon the `uvm_config_db` capabilities. A thin layer over `uvm_config_db` provides simplified handling for module-to-system scenarios. You could view this as if the VE hierarchy (`uvm_component` hierarchy) and the DUV hierarchy (module instance hierarchy) were tied to each other. When re-using the same DUV and the same VE in a different context/hierarchy, all that needs adjustment is the *single* path to the previously setup hierarchy.

```
class cdns_vif_db#(type T=int) extends uvm_config_db#(T);
   static function void register_vif(T vif, string vifName);
   static function void retrieve_vif(ref T vif,input uvm_component
         cntxt,string path, bit validate=1);
endclass
```

Figure 3: Public Core API of the Interface Registry.

The database can also expose additional introspection, debug, checks, or messaging facilities. These facilities are optional for the overall operation, but can contribute to debug efficiency and/or usability.

III. USE MODEL

To use the registry, the verification component creator just needs to query the database for a particular interface. The input parameters for that query are: the *interface type*, the *relative interface name*, and the *DUV context* that this VE component should utilize. The retrieval method returns a reference to the selected interface, which can be used inside the testbench code as usual. Since the retrieval code is encapsulated in one place, checking the requirements such as, the interface must exist, needs to be non-`null`. This common code, provided in one place, leads to better and more uniform code, and to error reporting.

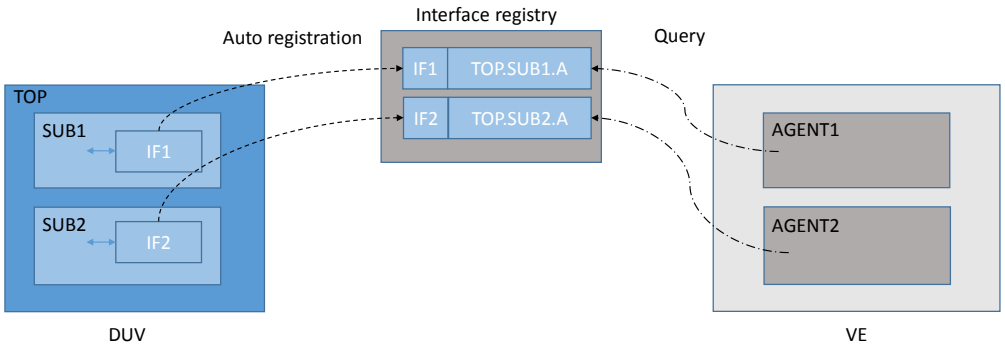

Figure 4 Separation of the DUV and VE using an Interface Registry.
The IF Instance is inside the DUV using bind, and the VE retrieves the IF with a query from the Registry.

In the following example "`clk_intf_i.protocol`" is the name of the interface relative to the current HDL context set for the particular `uvm_component`.

```
function void build_phase(uvm_phase phase);
   super.build_phase(phase);
   cdns_vif_db#(virtual clk_intf)::retrieve_vif(vif,
         this,"clk_intf_i.protocol");
endfunction
```

Figure 5: VIP/UVC Retrieving an interface from within the build phase of a UVM Component.

From a testbench creator or user point of view, only two tasks are required to hook *interface registry aware* components into the environment:

1. First, one has to instantiate an *interface registry aware* interface. The easiest way to do this is to use the SystemVerilog `bind` capability, which enables users to instantiate interfaces into a module. The `bind` construct is particular useful here because it allows users to create an instance of the interface without changing the source code of the DUV. That way, it allows the VE to pull together whatever signals are needed into an interface, and hook it up to the DUV. With the auto-register approach, the interface instances will self-register, which means no additional steps are required to inform the registry about interface instances. No interfaces are lost or forgotten, and at the same time, the registry can provide a condensed view of all the interfaces that are present in the system. In addition, the registry can later report which interfaces have been retrieved/used by the testbench. The following figure illustrates this.

```
// -*- binds an instance of the clk_intf into sub_block
// bind signals by name
bind sub_block clk_intf clk_intf_i(.*);
```

Figure 6: Using `bind` to make an instance in the Sub-block Module.

A DUT can instantiate more interface instances than are actually required to hook up the VE. Such instances could be used to add embedded assertions for checking, for example. Nothing needs to be changed or added for this scenario, and it will not cause a warning/error. However, should the testbench require more interfaces in the setup/configuration than are actually available, the registry can raise a warning/error indicating the mismatch during the construction of the testbench. This setup ensures that all interfaces required by the testbench are provided and connected properly. Misalignments will be reported in a common way even for components that are *buried somewhere deep* in the verification environment.

2. The second and final step is to configure the DUV HDL context for the verification component(s). This defines which part of the DUV the particular testbench/VIP instance belongs. Configuration is done using the `uvm_config_db`, as shown in the following figure.

```
// tell the TB component that its hdl paths are inside sub block_i1
uvm_config_db#(string)::set(tb1,"","HDLContext","sub_block_i1");
```

Figure 7: Configuration of VE Interface in a Module Verification Context.

For a module-to-system scenario, only the HDL context for the enclosing component or add-on modules needs to be set. The full HDL context name is constructed through the concatenation of all "`HDLContext`" attributes of the component (and its parent components) and retrieving the interface and the relative name for the interface (which is, `clk_intf_i.protocol`). The construction of the full HDL context path follows the same patterns as the construction of `uvm_reg` `hdl_path` fragments. This enables the reuse of the same VE and DUV in a larger setup without changes, but also allows for changes by way of the configuration database.

```
// tell the top component that its hdl context paths are inside top_block_i
uvm_config_db#(string)::set(uvm_root::get(),"","HDLContext","top_block_i");
// tell the TB2 component that its hdl paths are inside sub_block_i2
uvm_config_db#(string)::set(tb2,"","HDLContext","sub_block_i2");
```

Figure 8: Configuration of the context for both VE instances

IV. IMPLEMENTATION

The interface registry itself has been implemented as small package on top of the `uvm_config_db`, which offers further flexibility and debug improvements. In addition, advanced use models, such as overriding a configuration from the outside, and tracing and logging of accesses are also enabled. The interface registry has been made an add-on class in order to provide for better handling of module-to-system scenarios. The other elements are implemented in the user-space as outlined earlier. This package is available for download by way of the Accellera forums (http://forums.accellera.org/files/).

V. SUMMARY

The use of an interface registry is a methodology approach, which relies on a few SystemVerilog capabilities and a little bit of generic code, rather than special tooling. Most of the power of the interface registry methodology comes from the combination of interface self-registration, non-intrusive interface instance creation using `bind`, and the registry itself. This combination makes the DUV/VE integration simpler and more flexible especially in module-to-system contexts, or in changing, or multiple DUV configuration setups.

REFERENCES

IEEE-1800-2012, SystemVerilog LRM
UVM-1.1d Reference Guide