SystemVerilog Virtual Classes, Methods, Interfaces and Their Use in Verification and UVM

Clifford E. Cummings

Sunburst Design, Inc.
Provo, UT, USA

Heath Chambers

HMC Design Verification
Roswell, NM, USA

www.sunburst-design.com

ABSTRACT

This paper describes SystemVerilog virtual classes, virtual methods and virtual interfaces, how they can be used in verification and how they are used in UVM. This paper then describes how virtual methods enable polymorphism for use in verification including the concepts of upcasting and downcasting. This paper also describes pure virtual methods and why the pure keyword is useful, including an example from the UVM base classes. This paper includes useful guidelines regarding the use of virtual classes, methods and interfaces.

Table of Contents

Table of Figures

Table of Examples

1. Introduction

Virtual classes, virtual methods and virtual interfaces are important tools in the construction of powerful verification environments. The Accellera 2003 SystemVerilog Standard[9] added these capabilities to the Verilog/SystemVerilog language, and they officially became part of the IEEE SystemVerilog language starting with the IEEE Std 1800-2005[3].

The Universal Verification Methodology (UVM)[6] is a class library and methodology that takes full advantage of virtual classes, methods and interfaces. UVM does not add any new keywords or capabilities to the SystemVerilog language, but uses existing capabilities to put together a powerful verification methodology. If a simulator supports the full SystemVerilog class-based capabilities, an engineer can run UVM without any additional licenses.

To fully understand how UVM works, engineers need to understand the SystemVerilog "virtual" features described in this paper.

1.1 Tools and OS versions

The examples in this paper were run using the following Linux and Synopsys tool versions:

64-bit Linux laptop: CentOS release 6.5

VCS version K-2015.09-SP1_Full64

> Running **`vcs`** required the command line switch **`-full64`**
>
> Without the **`-full64`** command line switch, VCS compilation would fail with the message:
>
> ```
> g++: /home/vcs/linux/lib/ctype-stubs_32.a: No such file or directory
> make: *** [product_timestamp] Error 1
> Make exited with status 2
> ```

2. Classes

SystemVerilog classes can have data members (often called properties in other class-based languages) and methods (or built-in subroutines). The SystemVerilog class method types are **`function`**, **`void function`** and **`task`**. When considering which method type to use, engineers should generally use **`function`** and **`void function`** methods unless the method consumes simulation time, then a **`task`** should be used. All three method types can call a **`function`** or **`void function`**, but **`task`** methods can only be called another **`task`**, so if **`task`** methods are used, only another **`task`** can call them. A quick scan of the UVM class libraries shows that only UVM methods that consume time are declared as **`task`** methods.

Guideline #1: Class methods should be **`function`** and **`void function`**. Only use **`task`** when the method consumes simulation time.

3. Virtual classes

A **`virtual class`** is a class declaration that has the **`virtual`** keyword preceding the **`class`** keyword.

Virtual classes cannot be constructed (**`new()`**-ed) directly. Any attempt to **`new()`**–construct a **`virtual class`** object will give a compilation error. This is true of any computer language that allows the equivalent of **`virtual class`** declarations.

Virtual classes are classes that must be extended in order to use the virtual class functionality. A

virtual class could be a top-level base class or an extension of another virtual class.

Virtual class libraries are assembled to serve as templates for a specific purpose. Most of the classes in the UVM class library are defined as **virtual** classes, which means a user cannot both declare handles of the UVM base classes and construct a local object of the same. There are a few notable exceptions in the UVM class library, including the **uvm_sequencer** (which is sometimes directly declared and constructed in UVM testbench examples) and the **uvm_driver** (which probably should have been a **virtual** class since we can think of no useful examples that could directly construct and reasonably use the **uvm_driver** base class object).

In practice, almost all user-defined UVM testbenches are built from user-defined classes that are extensions of the UVM component classes. All user-defined transactions are extensions of the **uvm_sequence_item** virtual class and all user-defined sequences are extensions of the **uvm_sequence** virtual class.

The UVM base class library is a set of template files that the user extends to build a UVM testbench and set of tests (transactions and sequences). If a user had to build up the entire UVM testbench environment from scratch, it would never happen. There is so much functionality included in the UVM base classes that was put there by very smart verification engineers from multiple end-user and EDA companies, and most users will never fully understand everything that the UVM class library makes possible.

4. Methods

Methods in classes are built-in subroutines and as already noted they are defined as functions, void functions and tasks.

An extended class can override the base class methods, but when an extended class handle is assigned to a base class handle, calling the method using the base class handle will still execute the original base-class method, unless the base-class method was defined to be a virtual method (see Section 5).

A common misconception is that virtual class methods are virtual by default, but that is not true. In order to define a virtual method in a virtual class, the method still has to use the **virtual** keyword.

5. Virtual Methods

SystemVerilog class methods can be either non-virtual or virtual. Virtual methods include the **virtual** keyword before the **function** or **task** keyword. Classes with non-virtual methods fix the method code to the class object when constructed. Virtual method functionality is set at run-time, which allows extended class handles to be assigned to base class handles and run-time method calls using the base class handle will execute the extended class method functionality. This is a powerful feature that is called polymorphism.

Every class method has something called the method prototype, which is basically the header of the **function** or **task**. The **uvm_object** base class includes the **virtual function** definition shown in Figure 1.

```
virtual function bit do_compare (uvm_object rhs, uvm_comparer comparer);
  return 1;
endfunction
```

Figure 1 - UVM virtual do_compare() method included in the uvm_object base class

The prototype of the **virtual function** in Figure 1 is the header code:

```
virtual function bit do_compare (uvm_object rhs, uvm_comparer comparer);
```

This virtual method prototype includes five required elements: (1) the return type is **bit** , (2) the argument directions for both arguments are **input** (without a direction keyword in SystemVerilog, **input** is the default), (3) there are two input arguments, (4) the argument types are **uvm_object** and **uvm_comparer** , (5) the argument names are **rhs** and **comparer** . Any method extended from this virtual function must use these same exact five elements. Extending a virtual method requires strict method argument compatibility. The extended virtual method prototype must be identical to the base class virtual method prototype. The easiest way to ensure that the extended method prototype is correct is to copy the base class virtual prototype into the derivative class.

Because virtual method prototypes are identical between base classes and derivative classes, it is guaranteed that any call to a base class virtual method will succeed, even if a derivative handle was assigned to the base class handle.

Guideline #2: Declare SystemVerilog class methods to be **virtual** methods unless there is a very good reason to prohibit method-polymorphism.

A quick scan of UVM class library methods shows that the vast majority of methods are **virtual** methods.

Although not a common practice, a base class can have a non- **virtual** method and an extended class can override that method with a **virtual** method. On the other hand, if a base class has a **virtual** method, it is not possible to override the **virtual** method with a non- **virtual** method in an extended class. Once a method is declared to be **virtual**, all extended overrides of that method will still be **virtual**, with or without the **virtual** keyword. This can be a bit confusing since we cannot find a single example in the UVM base classes where an overridden **virtual** method in an extended class uses the **virtual** keyword.

In UVM, although not universally true, it is a pretty good assumption that all extended methods are virtual methods, even though the **virtual** keyword is not present.

Many important **uvm_object** methods, such as **copy()**, **do_copy()**, **compare()**, **do_compare()**, **convert2string()**, etc., are non-virtual methods. Users are highly discouraged from overriding the **copy()** , **compare()** and similar common transaction methods. The **copy()**, **compare()** and other common transaction methods call user-defined **do_copy()**, **do_compare()**, and other pre-defined **do_method()** methods which are called by the **copy()**, **compare()**, etc. methods. Users should not override the non-**do** variety of these methods with virtual methods. See Cummings[1] for 72 pages of mind-numbing fun that describes how these methods and field macros work!)

With this basic understanding of virtual methods, the question is, what makes virtual methods valuable and why do we care? An important part of understanding the value of virtual methods requires the user to understand upcasting and downcasting described in Section 7.

6. Extended and derivative classes

An extended class uses the **extends** keyword to extend the specified base class. If there is an extended class from another extended class, the second-level extended class is referred to as a derivative class, or a derivative of the base class.

An example from the UVM class library is shown below.

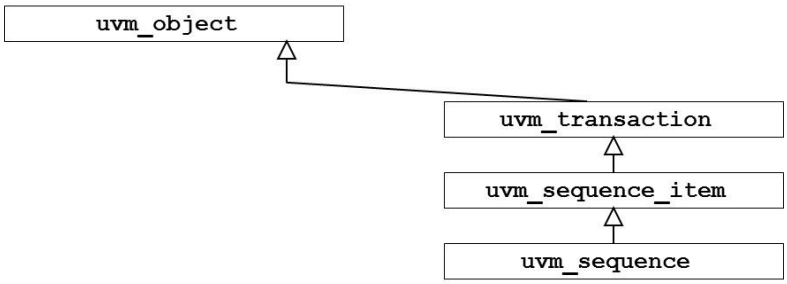

Figure 2 - UVM transaction classes

uvm_transaction is an extended class of the **uvm_object** base class. At the same time, **uvm_transaction**, **uvm_sequence_item** and **uvm_sequence** are all derivatives of **uvm_object**. It should also be clear that **uvm_sequence** is both an extended class and a derivative class of **uvm_sequence_item**.

7. Upcasting & Downcasting

"Upcasting is casting to a supertype, while downcasting is casting to a subtype. Upcasting is always allowed, but downcasting involves a type check and can throw a **ClassCastException**."[7]

To paraphrase this description, any extended or derivative class handle can be copied to a base class handle, but only some base class handles can be copied to an extended class handle and SystemVerilog requires a type-check before allowing the latter.

Base class handles can only access data members and methods that are declared in the base class, even if an extended handle, with additional data members and methods, is copied to the base class handle. Since extended classes already contain the base class data members and methods, it is given that calling methods in the base class could find an equivalent method from the extended handle copied to the base handle.

A base class handle cannot be copied to an extended class handle. Any attempt to copy a base handle to an extended handle will cause a compilation error similar to, "** Error: Illegal assignment … Types are not assignment compatible."

The following **test_classes** package, shown in Example 1, contains a **class base** definition and a **class ext** definition. The class definitions in this package are used in the **test** module shown in Example 2.

```
package test_classes;
  class base;
    bit [7:0] a;

    function void showit;
      $display("BASE(%m): a=%2d", a);
    endfunction

    function void seta(bit [7:0] din);
      a = din;
    endfunction

    function bit [7:0] geta();
      return(a);
    endfunction
  endclass

  class ext extends base;
    bit [7:0] data;

    // Inherit: base::bit [7:0] a;
    // Inherit: base:: function void seta(bit [7:0] din);
    // Inherit: base:: function bit [7:0] geta;

    function void showit;  // Overrides base::showit
      $display(" EXT(%m): data=%2d   a=%2d", data, a);
    endfunction

    function void setdata(bit [7:0] din);
      data = din;
    endfunction

    function bit [7:0] getdata();
      return(data);
    endfunction
  endclass
endpackage
```

Example 1 – test_classes package

The module test code shown in Example 2 includes a few lines that have been commented out. Each of these lines caused compilation errors as described in the comments shown at the end of the same line. This example shows upcasting and downcasting in practice. Snippets of this code along with explanations are shown in Figure 3 - Figure 10.

```
module test;
  import test_classes::*;
  base b1, b2;
  ext  e1, e2;
  bit  e1good, e2good;
  bit [7:0] m_data;

 initial begin
    b1 = new();
    b1.seta(4);
    b1.showit();
    // e1 = b1;                // Error ... types not assignment compatible
    e1good = $cast(e1, b1);    // ILLEGAL assignment - $cast fails-returns 0
    if (!e1good) $display("b1->e1 cast failed");
    e1 = new();
    e1.seta(2);
    e1.setdata(6);
    e1.showit();
    b2 = e1;
    b2.showit();
    e1 = new();
    e1.seta(9);
    e1.showit();
    // b2.setdata(9);          // Error ... setdata() not in b2
    // m_data = b2.getdata(); // Error ... getdata() not in b2
    // e2 = b2;                // Error ... types not assignment compatible
    e2good = $cast(e2, b2);    // LEGAL assignment - $cast passes
    if (e2good) $display("b2->e2 cast PASSED!");
    e2.showit;
  end
endmodule
```

Example 2 - Upcasting and downcasting test module

Figure 3 shows the declaration of two base class handles (**b1** and **b2**) and two extended class handles (**e1** and **e2**). These handles are initially null.

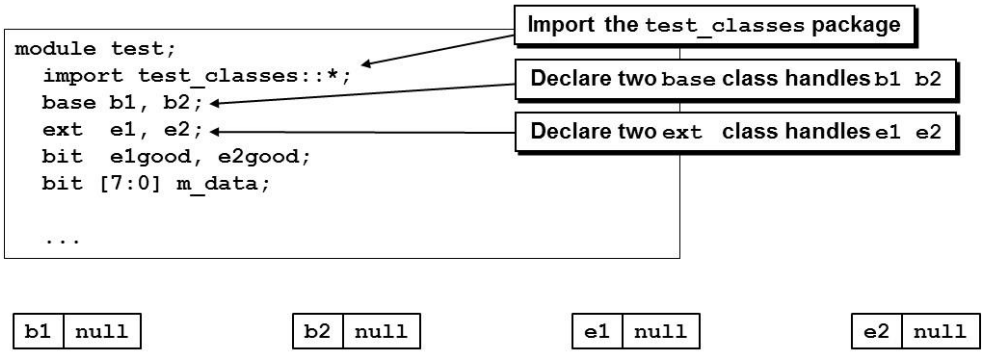

Figure 3 - Declare two base class handles and two extended class handles

In Figure 4, the **b1** base class object is constructed, and the **a**-variable is set to 4. Then the **showit()** method is called and displays the output shown at the bottom of the figure.

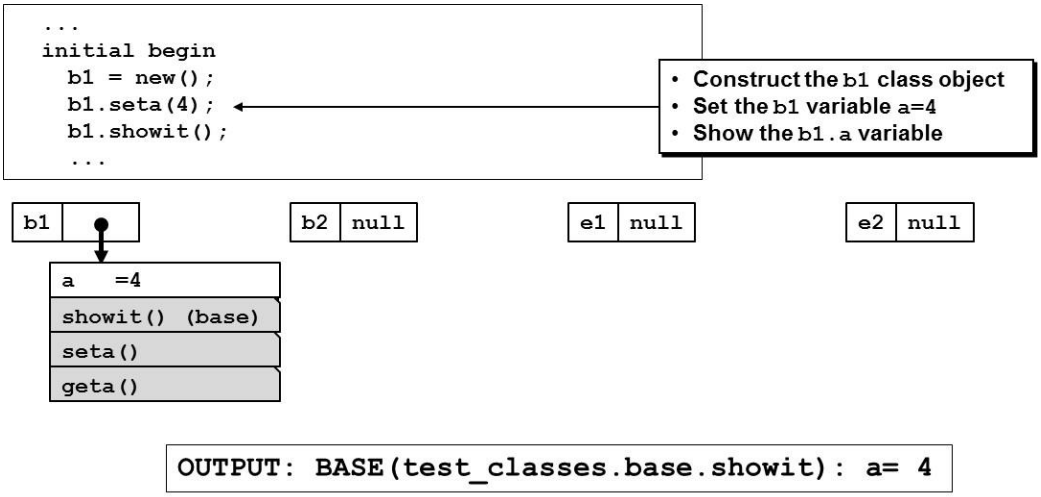

Figure 4 - Construct the b1 base class and set the a value

In Figure 5, an attempt was made to copy the **b1** base class handle to the **e1** extended class handle. This is an illegal attempt at downcasting and is a compilation error. Then an attempt was made to **$cast** the **b1** handle to the **e1** handle but the **$cast** fails, so the **if**-test causes the displayed failure message at the bottom of the figure to be shown.

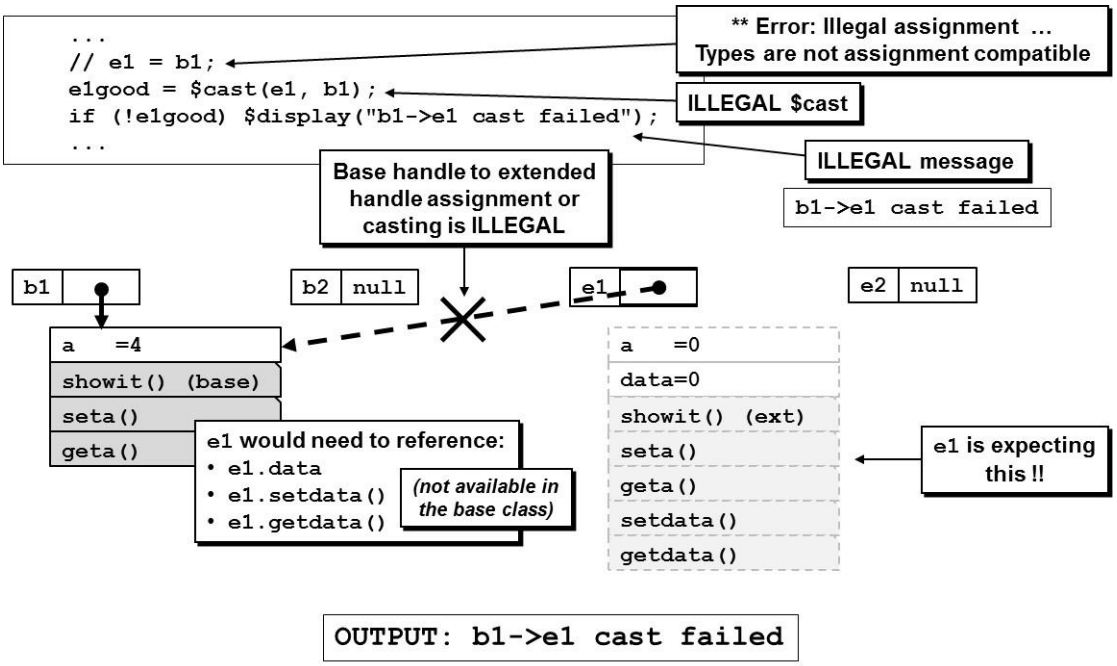

Figure 5 - Illegal to copy or $cast a base class handle to an extended class handle

In Figure 6, the **e1** object is constructed and the **a**-value and **data**-value are set, then the contents of the **e1** object are displayed using the **showit()** method and those contents are shown at the bottom of the figure.

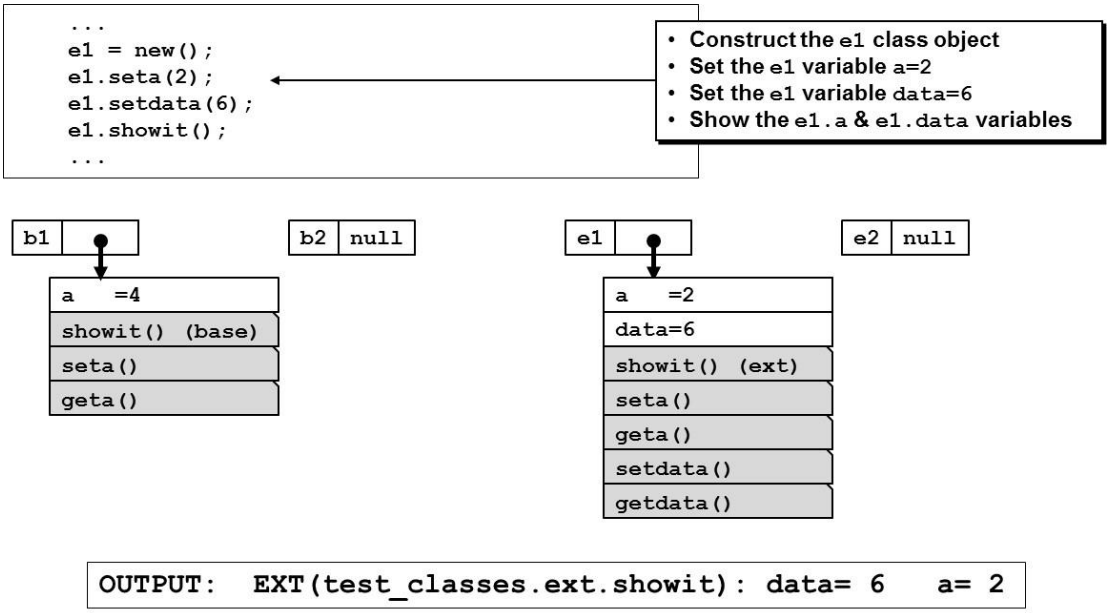

Figure 6 - Construct the e1 extended class and set the a and data values

In Figure 7, the **e1** extended object handle is copied to the **b2** base class handle. It is always possible to copy an extended handle to a base class handle because base class data members and methods have been inherited by the extended class, so it is possible to use the base class handle to call inherited base class data members and methods. This is upcasting and is always legal.

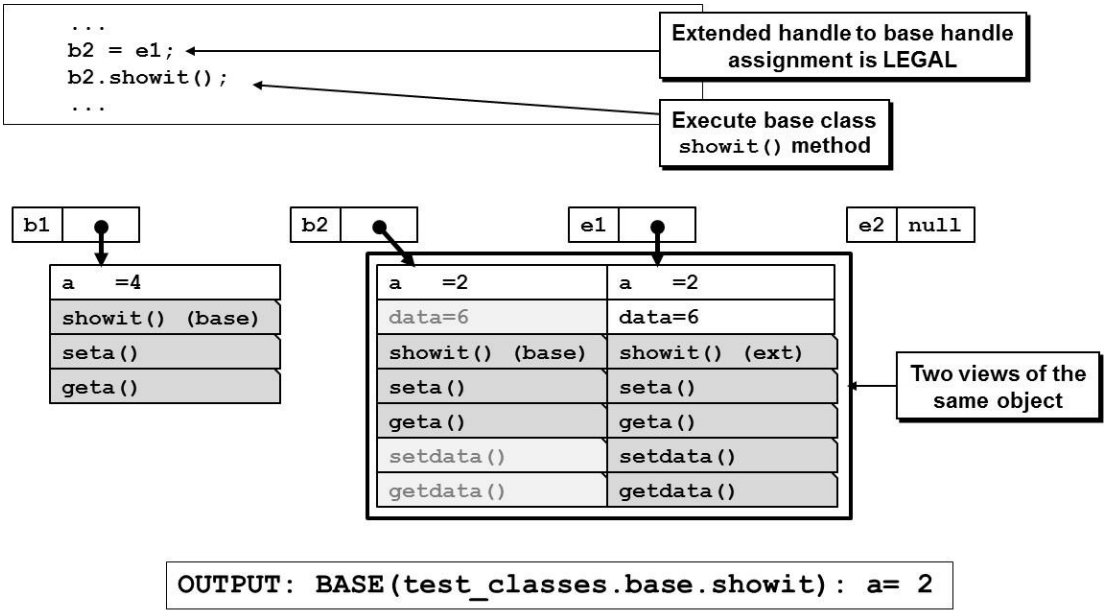

Figure 7 - Copy the e1 extended object handle to the b2 base class handle

In Figure 8, the **e1** object is re-constructed and the **a**-value is set, while the **data**-variable retains its default value of **0**. Then the contents of the **e1** object are displayed using the **showit()** method and those contents are shown at the bottom of the figure.

Note that the old **e1** object still exists and is pointed to by the **b2** class handle.

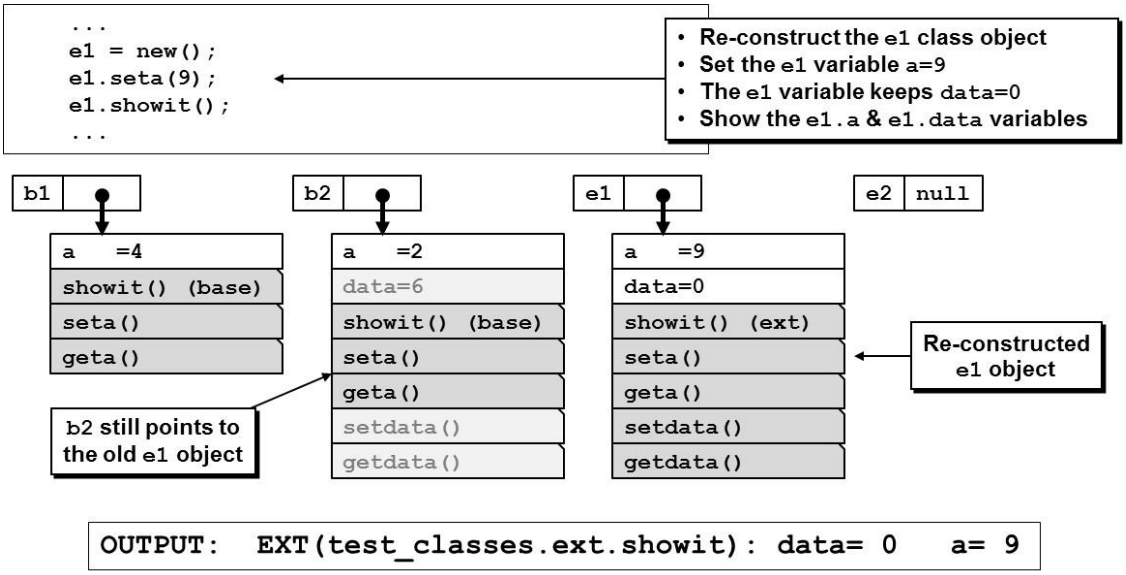

Figure 8 - Re-construct the e1 extended class object - b1 still points to the old e1 object

It can be seen in Figure 9 that any attempt to access the extended class **data** variable or any attempt to call the extended methods will cause a compiler error. The **data** variable, **setdata()** and **getdata()** methods all exist in the object pointed to by the **b2** handle, but a base class handle does not know about extended members and methods, so the compiler will not allow the base handle to make the commented-out access commands.

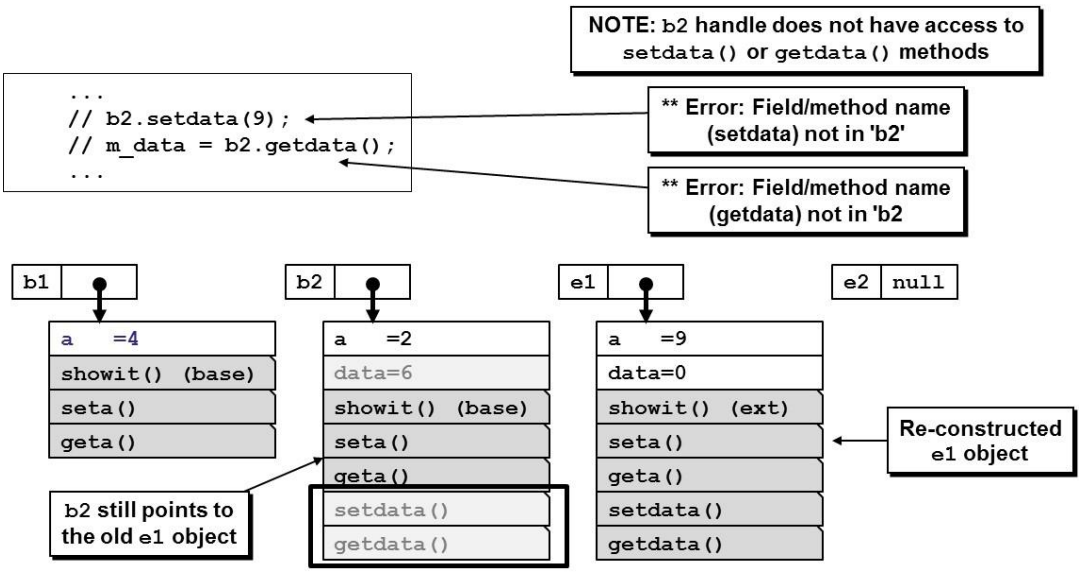

Figure 9 - The b2 base handle cannot be used to access extended object data or methods

In Figure 10, an attempt is made to again copy a base class handle (**b2**) to an extended class handle (**e2**), but this is still illegal and flagged as a compiler error.

But in Figure 10, it is now legal to **$cast** the **b2** base class handle to the **e2** extended class handle, because the **$cast** operation first checks to see if the object pointed to by **b2** is an extended object, and once the compiler confirms that the base handle is indeed pointing to an extended object the **b2** base handle can now be copied (**$cast**) to the **e2** extended handle. This is an example of legal downcasting, as described at the beginning of Section 7.

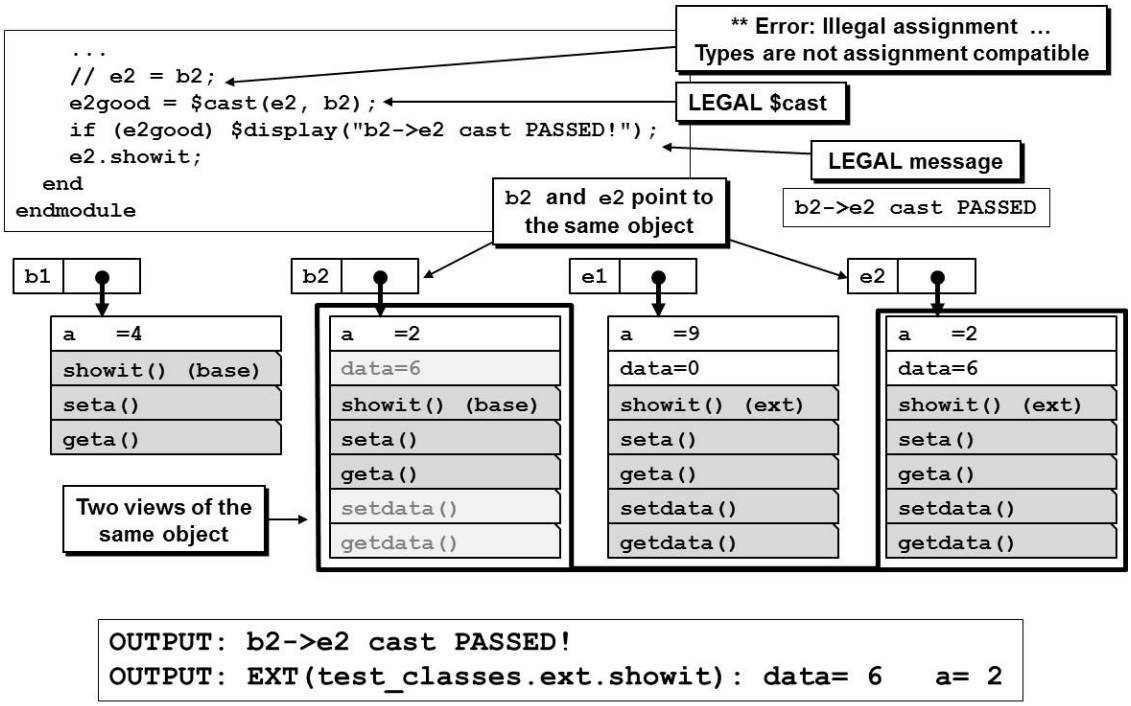

Figure 10 - $cast the b2 handle to the e2 handle - This is now legal

How do engineers use this capability in a UVM testbench? One common example is when overriding the **do_copy**() method in the user's transaction class definition.

User defined transaction classes are typically extensions or derivatives of the **uvm_sequence_item** class, and as was shown in Figure 2, **uvm_sequence_item** is a derivative of **uvm_object**; therefore, user defined transactions are also derivatives of **uvm_object**. This means that any user defined object transaction handle can be copied or passed to a **uvm_object** handle, as is the case when overriding the **do_copy()** method.

The **do_copy()** method as defined in **uvm_object** is shown in Figure 11.

```
virtual function void uvm_object::do_copy (uvm_object rhs);
  return;
endfunction
```

Figure 11 - UVM do_copy() virtual method

When overriding the **do_copy**() method, it is important to declare a handle of the user's defined transaction class type (in this example, **trans tr**). When the **do_copy**() method is called with a user defined transaction handle, it is upcast to the **uvm_object** type. Any attempt to access transaction fields from the **rhs** handle will fail because the **rhs** handle is of the **uvm_object** base class type, but the **tr** object fields are still there, just not accessible. To gain access to the object fields, the **rhs** handle needs to be downcast to the transaction type (**$cast(tr, rhs);**). The **copy**() defined in **uvm_object** calls the **do_copy()** method and by defining the **do_copy()** method to **$cast** the argument handle **rhs** to the desired **trans** handle **tr** , we know that any call to the transaction **tr.copy(new_tr)** method will succeed in copying the fields of the **new_tr** handle to the local fields of the **trans tr** object, where **new_tr** is the handle of another object of the same transaction type.

```
function void do_copy (uvm_object rhs);
  trans tr;
  super.do_copy(rhs);
  $cast(tr, rhs);
  field_1 = tr.field_1;
  field_2 = tr.field_2;
  ...
endfunction
```

Example 3 - do_copy() method implemented with downcasting

When a user overrides the virtual **do_copy()** or **do_compare()** methods, it is required to declare a local transaction handle in the functions and downcast the **uvm_object rhs** handle to the locally declared transaction handle.

Many industry UVM testbench examples name the local transaction handle **rhs_** , but we believe this is confusing and therefore a poor practice. Using **rhs_** means that casting is done in the form **$cast(rhs_, rhs);** and then fields are referenced as **rhs_.field_1** , etc. It is easy to confuse the **uvm_object rhs** handle with the transaction class **rhs_** handle. We believe it is a better practice to make sure the local transaction handle has a very distinct non- **rhs** handle name such as **tr** , to avoid confusion about which handle is being referenced inside the method.

Guideline 3: Declare local transaction handles using distinct names such as **tr** and avoid local transaction handle names such as **rhs_**.

Partial **do_copy()** and **do_compare**() **trans** class methods with **trans tr** handle declarations and **$cast(tr, rhs);** downcasting commands are shown at the beginning of Section 7.

```
class trans extends uvm_sequence_item;
  ...
  int f1;
  ...
  virtual function void do_copy (uvm_object rhs);
    trans tr;
    //---------------------------------
    super.do_copy(rhs);
    $cast(tr, rhs);
    f1 = tr.f1;
    ...
  endfunction

  virtual function bit do_compare (uvm_object rhs, uvm_comparer comparer);
    trans tr;
    bit   eq;
    //---------------------------------
    eq = super.do_compare(rhs, comparer);
    $cast(tr, rhs);
    eq &= comparer.compare_field_int("f1", f1, tr.f1);
    ...
    return(eq);
  endfunction
  ...
Endclass
```

Example 4 - Transaction example with do_copy() implemented using downcasting

8. Pure virtual methods

SystemVerilog-2009[4] added the **pure** keyword to the SystemVerilog language. The **pure** keyword is only legal in a virtual class. The **pure** keyword is not legal in a non-virtual class.

Pure virtual methods serve two important purposes:

(1) Pure virtual methods can only be a method prototype. Pure virtual methods are a place-holder.
(2) Pure virtual methods ***must*** be overridden in a non-virtual class.

To summarize these purposes, **pure** virtual methods create a method place-holder that imposes the requirement that it must be implemented in the non-virtual class.

As a **pure virtual** method placeholder, the **pure virtual** method in a **virtual** class imposes the following restrictions:

(1) The **pure virtual** method can only be a prototype.
(2) The **pure virtual** method cannot have a body, it is not allowed to have any of the implementation code.
(3) The **pure virtual** method is NOT even allowed to have an "**end**" keyword (**endfunction** / **endtask**).

Question: If a virtual class declares a pure virtual method, does the first derivative non-virtual class have to override the pure method? If no other derivative virtual classes have overridden the pure method with an implementation, the answer is yes. This is shown in Figure 12.

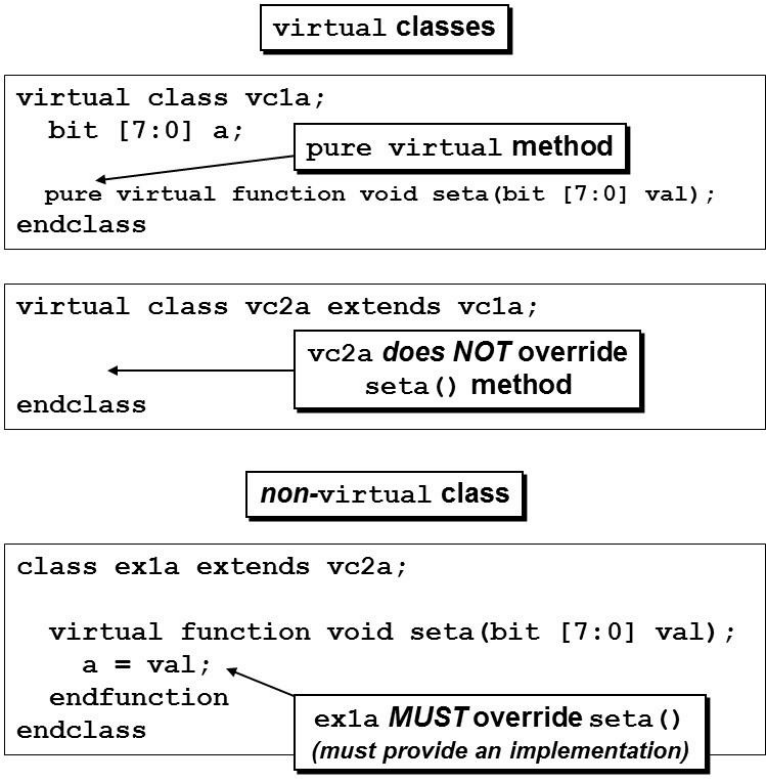

Figure 12 - Non-virtual class must override a pure virtual method from a virtual class

If a derivative extended virtual class has overridden the pure method with an implementation, the first derivative non-virtual class does not have to override the pure method. This is shown in Figure 13.

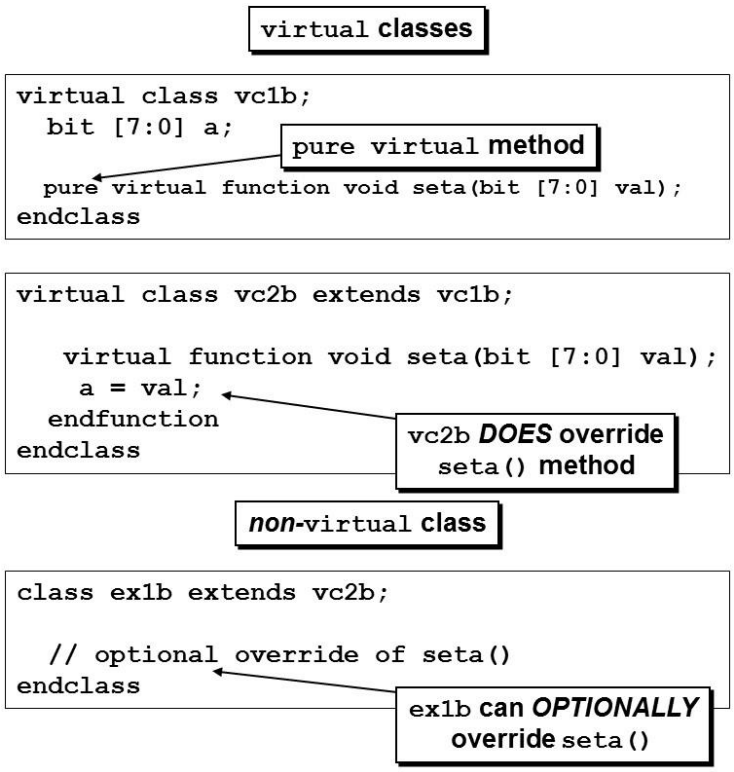

Figure 13 - Non-virtual class not require to override a pure virtual method if overridden in a virtual class

8.1 UVM pure virtual method example

One common place where UVM verification engineers encounter pure virtual methods is in the **uvm_subscriber**. This UVM base class is often extended to create a coverage collector component and is sometimes used to help create components that are part of a testbench scoreboard.

The **uvm_subscriber** has a built-in **uvm_analysis_imp** port with port name **analysis_export**. To use the **uvm_analysis_imp** port requires a **write** function, so the **uvm_subscriber** also includes the declaration of a **pure virtual function write (** … **)**

Extending the **uvm_subscriber** to create a testbench coverage collector means that the coverage collector inherits the **uvm_analysis_imp** port and requires verification engineers to implement the **write()** function.

Partial code from the **uvm_subscriber** base class is shown in Example 5.

```
virtual class uvm_subscriber #(type T=int) extends uvm_component;

  typedef uvm_subscriber #(T) this_type;

  // Port: analysis_export
  uvm_analysis_imp #(T, this_type) analysis_export;

  function new (string name, uvm_component parent);
    super.new(name, parent);
    analysis_export = new("analysis_imp", this);
  endfunction

  // Function: write
  //
  // A pure virtual method that must be defined in each subclass. Access
  // to this method by outside components should be done via the
  // analysis_export.

  pure virtual function void write(T t);
endclass
```

Example 5 - Partial uvm_subscriber code

Example 6 shows part of a scoreboard predictor, commonly used in a UVM scoreboard. Since the **sb_predictor** extends the uvm_**subscriber**, the predictor is required to override the **write** method with a full implementation.

```
class sb_predictor extends uvm_subscriber #(trans1);
  `uvm_component_utils(sb_predictor)

  // Inherit the uvm_analysis_imp port declared in uvm_subscriber
  // uvm_analysis_imp #(T, this_type) analysis_export;

  ...

  // MUST override the write method
  function void write(trans1 t);
    ...
  endfunction
endclass
```

Example 6 - Partial common scoreboard predictor code

Since the **sb_predictor** is a parametereized class of **type trans1**, the **write()** method must also use **trans1** as the input type. The **type T** parameter of the **uvm_subscriber** was the input type of the **write()** method (as shown in Example 5), so whatever type is used in the extended **sb_predictor** must also be the input type of the **write()** function (as shown in Example 6).

Guideline #4: Declare a **pure** method whenever it is important to force a derivative class to implement the method, as is done by the **uvm_subscriber** virtual class.

9. Virtual Interfaces

Virtual interfaces are class data member handles that point to an interface instance. This allows a dynamic class object to communicate with a Design Under Test (DUT) module instance without using hierarchical references to directly access the DUT module ports or internal signals. This is important given two facts: (1) class definitions are typically gathered into one or more packages, and (2) items that are defined inside of a package are not permitted to make hierarchical references to items outside of that package. This SystemVerilog package restriction exists because the package might be used in a design that does not include the hierarchical paths that might be referenced in the package.

In Figure 14, it can be seen that the class methods access the virtual interface (using the normal dot notation, *<virtual interface>.<signal>*). Since the virtual interface is pointing at the interface instance, the class methods are in effect accessing the interface instance. With the DUT instance connected to the interface instance, the class methods are indirectly accessing the DUT instance without using a hierarchical reference outside of the class.

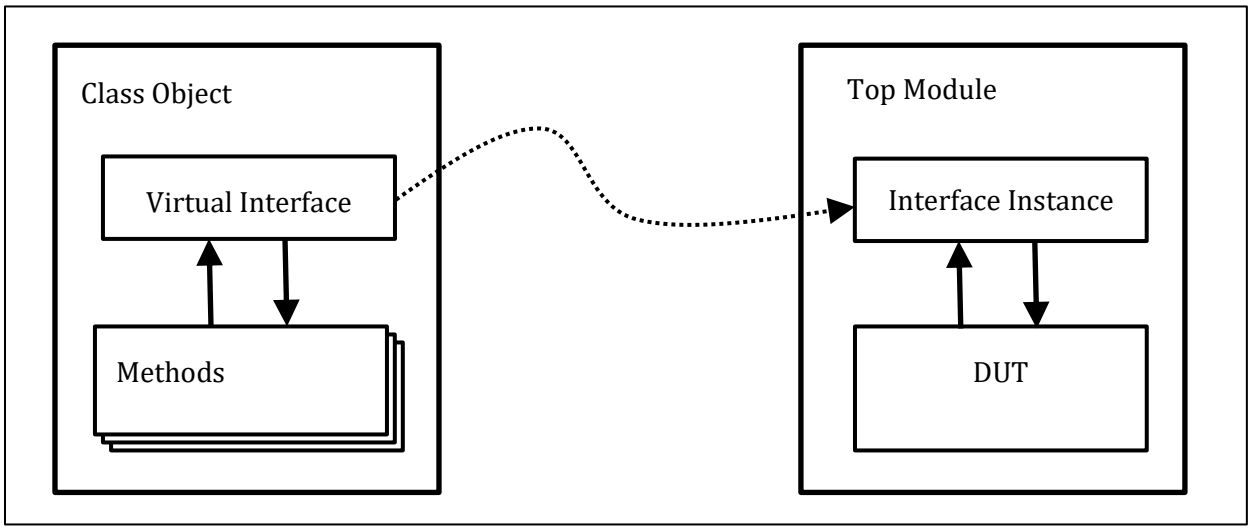

Figure 14. Virtual Interface Block Diagram

To successfully use a virtual interface there are three requirements:

- An instantiated interface must be correctly connected to the module instance.
- A virtual interface handle must be declared in a class.
- An assignment must be made to set the class virtual interface handle to point to the module interface instance.

Example 7 shows the interface instantiated as **dif** and hierarchically connected to the module instance **dut** to satisfy the first requirement above (this is a common style of connecting an interface to a module instance for use with a non-UVM class based test environment). If the module had used the interface as a port, the connection of the interface to the module instantiation is shown in the commented out instantiation at the bottom.

```
module top();
  …
  // The actual instantiation of the Interface (note: this is the
  // only physical copy of the interface in the environment).
  dut_if dif(.*);

  // DUT instantiation with hierarchical port connections with the
  // .* at the end to do port-type checking for connection errors.
  simple_dut dut(.clk(dif.clk),       .rst_n(dif.rst_n),
                 .valid(dif.valid), .in_char(dif.in_char),
                 .ack(dif.ack),      .out_char(dif.out_char), .*);

  // If the DUT used the interface as a port rather than a discreet
  // port list, then the instantiation would be similar to this.
  // simple_dut dut(.lif(dif), .*);
  …
endmodule
```

Example 7 - Interface Instantiation and Connection

Example 8 shows the second two requirements. First is the declaration of the virtual interface, **vif**, inside of the **Testbench** class. Second is the assignment of the interface instance handle, **dif**, to the virtual interface, **vif**, using the **new()** constructor of the class. This is a relatively common non-UVM style of connecting the virtual interface to the interface instance (section 9.2 will show the UVM style).

```
class Testbench;
  …
  // The virtual interface declaration.
  virtual dut_if vif;

  // When constructing the testbench object, take the passed in
  // handle to the actual interface instance and assign it to the
  // virtual interface of the testbench.
  function new (virtual dut_if nif);
    vif = nif;
  endfunction
  …
endclass


module top();
  …
  // The testbench class object handle and construction.
  Testbench tb = new(dif);
  …
endmodule
```

Example 8 - Virtual Interface Declaration and Assignment

Once the virtual interface is connected correctly, the class objects can communicate with the module instance by accessing the signals inside of the interface that are connected to the module, as shown in Example 9.

```
class Testbench;
  …
  task run_test();
    vif.rst_n <= '0;
    vif.valid <= '0;
    @(negedge vif.clk);
    vif.rst_n = '1;
    @(negedge vif.clk);

    // Drive the test string into the DUT.
    foreach (test_str[i]) begin
      vif.in_char = test_str[i];
      vif.valid   = '1;
      @(negedge vif.clk);
      vif.valid   = '0;
    end

    repeat (3) @(negedge vif.clk);
    $finish(2);
  endtask

endclass
```

Example 9 - Virtual Interface Accesses

9.1 Virtual Interfaces vs Static Interfaces

When dealing with interfaces, there are three different perspectives that should be considered. The first two are static in nature and are constructed and connected during compile/elaboration of the design and environment. The third is created and assigned dynamically during run-time.

The first is the static interface instance itself. This was shown as the instantiation of **`dif`** in Example 7, which is created during compilation/elaboration and cannot be changed during run-time. The instantiation of an interface is the only place that an actual interface exists. All other references to an interface are handles pointing to the actual interface instantiation.

The second is the static handle to an interface from a module port list. This was shown in the commented out module instantiation in Example 7 where the port **`lif`** was connected to the static handle **`dif`**. This handle is set and connected to the port statically during compilation/elaboration and cannot be changed during run-time.

The third is the dynamic virtual interface within classes. The creation and setting of the **`vif`** virtual interface was shown in Example 8 and the access of interface signals was show in Example 9. This dynamic field is set and used during run-time. Since it is dynamic, an environment could be designed that had multiple interfaces instantiated, but only one virtual interface that pointed at the different interface instances at different times during run-time (this is just being mentioned as a theoretical possibility, not as a recommendation as it would probably create a very complex and convoluted environment that would be a nightmare to create and maintain effectively).

Notice that the instance name of an interface instantiation is used to represent both the actual instance and a handle. The handle is treated as a static handle when it is connected to a module

instance port and as a dynamic handle when passed into a class to set a virtual interface handle.

9.2 Virtual Interface Usage in UVM

Since UVM is a class-based verification methodology and it is recommended to group the environment into a package. One or more virtual interfaces are required for the UVM testbench to access the DUT. The UVM driver and UVM monitor are typically the only two locations that need to access the DUT, so they are the only two locations within UVM that require a virtual interface.

The most common UVM style of instantiating and connecting the virtual interface to the DUT was shown in Example 7 using the named port connections that hierarchically connect to the internal signals of the interface.

Where the UVM differs from the basic virtual interface usage, is in the connection of the interface instance to the virtual interface. There are three ways to do this connection and the first two are generally not recommended.

The first way that is not recommended was shown in Example 8 by passing the interface handle into the **`new()`** constructor of the class. This style in UVM creates spaghetti-like coding as you have to pass the interface handle to each level of the UVM component structure until you get it to the driver and monitor where they are actually used.

The second way, which is no longer recommended, is the older OVM style using a wrapper class and the config object table to store and retrieve that wrapper. This style was required in OVM and will still work for UVM, but is not recommended for UVM since it requires the creation of an extra wrapper class for the sole purpose of storing the interface handle inside of a class object that can then be stored into a **`uvm_object`** (config object) table using the now-deprecated[1] **`set_config_object()`** / **`get_config_object()`** commands. The wrapper class was necessary because a static interface handle cannot be directly placed into a class-based storage table.

[1] **`set_config_object()`** and **`get_config_object()`** commands were deprecated in UVM 1.2. These commands were replaced by the **`uvm_config_db#(…)::set(…)`** and **`uvm_config_db#(…)::get(…)`** commands.

Example 10 shows how the wrapper class has to be created, its virtual interface pointed to the actual interface, and then the wrapper class object is stored inside of the config object table before the call to **run_test()**. Then in the driver and monitor classes (Example 10 only shows the driver, but the monitor would use the same coding), the wrapper object must be retrieved from the config object table and downcast from the **uvm_object** class back into the wrapper class before the stored virtual interface handle can be retrieved and assigned to the local **vif** handle. This assignment is typically done in the **build_phase()** method and once it is done, the testbench has access to the virtual interface for the driver and monitor classes to indirectly communicate with the DUT.

```
module top;
  …
  dut_if         dif;
  dut_if_wrapper w;  // dut_if_wrapper is derived from uvm_object

  initial begin
    w = new(dif);
    set_config_object("*", "vif", w, 0);
    run_test();
  end
endmodule

class dut_if_wrapper extends uvm_object;
  virtual dut_if vif;

  function new (virtual dut_if nif);
    vif = nif;
  endfunction
endclass

class tb_driver extends uvm_driver #(trans1);
  …
  virtual dut_if vif;
  …
  function void build_phase(uvm_phase phase);
    super.build_phase(phase);

    uvm_object     obj;
    dut_if_wrapper w;
    //------------------------------
    if (!get_config_object("vif", obj, 0))
      `uvm_fatal("NOVIF",{"virtual interface must be set for:",
                          get_full_name(),".vif"});
    if (!$cast(w, obj))
      `uvm_fatal("NOVIF",{"bad dut_if_wrapper handle for",
                          get_full_name()});
    vif = w.vif;
  endfunction
  …
endclass
```

top module with interface and interface wrapper

dut_if_wrapper class definition

tb_driver class that accesses the wrapper class to retrieve the interface handle

Example 10 - OVM Style Virtual Interface Connection

The third, and recommended, way to set virtual interfaces in UVM is a cleaner and easier to use approach as it basically does the wrapper handling automatically. Example 11 first shows that the actual interface handle, **dif**, is stored into the **uvm_config_db** at string location **"vif"** using the **set()** command just before the **run_test()** call in the top level module. Then it shows the use of the **get()** function of the **uvm_config_db** to retrieve the virtual interface handle from the same **"vif"** location and assign it to the local **vif** virtual interface handle in the driver class (the same code will be used inside of the monitor class).

```
module top;
  …
  dut_if         dif;
  …
  initial begin
    uvm_config_db#(virtual dut_if)::set(null, "*", "vif", dif);
    run_test();
  end
endmodule

class tb_driver extends uvm_driver #(trans1);
  …
  virtual dut_if vif;
  …
  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    // Get the virtual interface handle that was stored in the
    // uvm_config_db and assign it to the local vif field.
    if (!uvm_config_db#(virtual dut_if)::get(this, "", "vif", vif))
      `uvm_fatal("NOVIF", {"virtual interface must be set for: ",
                  get_full_name(), ".vif"});
  endfunction
  …
endclass
```

Example 11 - UVM Style Virtual Interface Connection

This third technique eliminates the need to create a separate wrapper class to store the static interface handle, and eliminates the need to declare a **uvm_object** base class handle and downcast that handle to a wrapper handle before retrieving the virtual interface handle.

Guideline #5: Use **uvm_config_db#(virtual dut_if)::set(…)** and **uvm_config_db#(virtual dut_if)::get(…)** commands to store and retrieve an interface for use by a UVM testbench.

10. Summary & Conclusions

This paper gave a quick introduction to classes, class extension, class methods, virtual classes and virtual methods.

A guideline for choosing how to implement class methods was given:

Guideline #1: Class methods should be **`function`** and **`void function`**. Only use **`task`** when the method consumes simulation time.

This paper described how virtual methods are frequently used in UVM and how upcasting and downcasting work and are also used in UVM. This lead to the second guideline:

Guideline #2: Declare SystemVerilog class methods to be **`virtual`** methods unless there is a very good reason to prohibit method-polymorphism.

UVM uses the important virtual methods **`do_copy()`** , **`do_compare()`** and other virtual methods that are called by the **`copy()`** , **`compare()`** and other standard transaction methods, which require that the **`uvm_object`** method prototypes be copied and used. The prototype methods take **`uvm_object`** handles as inputs, which upcasts all transaction handles into **`uvm_object`** handles. These methods require the user to declare handles of the transaction type and to downcast the handles inside of these methods to recover the transaction data members. This paper described why it might be confusing to declare transaction handles named **`rhs_`** and recommended that distinct names be given to the local transaction handles in these methods.

Guideline #3: Declare local transaction handles using distinct names such as **`tr`** and avoid local transaction handle names such as **`rhs_`**.

This paper also discussed the keyword **`pure`** and how it is used in **`virtual`** classes to impose a strict method prototype requirement that also eventually requires a derivative class to implement the full method. A **`uvm_subscriber`** class example was included to show how this is commonly used in UVM testbenches.

Guideline #4: Declare a **`pure`** method whenever it is important to force a derivative class to implement the method, as is done by the **`uvm_subscriber`** virtual class.

This paper also showed how virtual interfaces are used to drive signals from a class-based testbench to a real testbench interface and that the real interface signals touch the pins of the DUT, effectively allowing the class-driven signals to touch the DUT pins in a testbench.

It was also shown that the new **`uvm_config_db#(…)::set(…)`** and **`uvm_config_db#(…)::get(…)`** commands greatly simplify the process of connecting a class based testbench to a DUT.

Guideline #5: Use **`uvm_config_db#(virtual dut_if)::set(…)`** and **`uvm_config_db#(virtual dut_if)::get(…)`** commands to store and retrieve an interface for use by a UVM testbench.

11. Acknowledgements

We are grateful to our colleagues John Dickol, Don Mills and Jeff Vance for their reviews and suggested improvements to this paper. Their contributions have ensured the presented material includes explanations and insight that we would have otherwise neglected to include.

12. References

[1] Clifford E. Cummings, “UVM Transaction - Definitions, Methods and Usage,” SNUG 2014 (Silicon Valley). Available at www.sunburst-design.com/papers

[2] IEEE Standard Verilog Hardware Description Language, IEEE Computer Society, IEEE, New York, NY, IEEE Std 1364-2001.

[3] "IEEE Standard For SystemVerilog: Unified Hardware Design, Specification and Verification Language," IEEE Computer Society and the IEEE Standards Association Corporate Advisory Group, IEEE, New York, NY, IEEE Std 1800™-2005

[4] "IEEE Standard For SystemVerilog - Unified Hardware Design, Specification and Verification Language," IEEE Computer Society and the IEEE Standards Association Corporate Advisory Group, IEEE, New York, NY, IEEE Std 1800™-2009

[5] "IEEE Standard For SystemVerilog - Unified Hardware Design, Specification and Verification Language," IEEE Computer Society and the IEEE Standards Association Corporate Advisory Group, IEEE, New York, NY, IEEE Std 1800™-2012

[6] "IEEE Standard For Universal Verification Methodology Language Reference Manual," IEEE Computer Society Sponsored by the Design Automation Standards Committee, IEEE, New York, NY, IEEE Std 1800.2™-2017

[7] https://stackoverflow.com/questions/23414090/what-is-the-difference-between-up-casting-and-down-casting-with-respect-to-class

[8] https://www.tutorialcup.com/cplusplus/upcasting-downcasting.htm

[9] "SystemVerilog 3.1 – Accellera’s Extensions to Verilog®," 2003, Accellera Organization, Inc., Napa, CA 94558