



World Class SystemVerilog & UVM Training



SystemVerilog's Virtual World - An Introduction to Virtual Classes, Virtual Methods and Virtual Interface Instances

Clifford E. Cummings

Heath Chambers

Sunburst Design, Inc.
Beaverton, OR, USA

HMC Design Verification, Inc.
Roswell, NM, USA

www.sunburst-design.com

hmcdiv.iwarp.com

ABSTRACT

*The SystemVerilog keyword **virtual** is used in three very distinct ways within the language. This paper introduces the fundamentals that are required to understand how virtual is used and behaves with virtual classes, virtual methods and virtual interface instances, and how it adds polymorphism within a SystemVerilog context. This paper also introduces **pure virtual** methods and **pure** constraints, features added to the IEEE-1800-2009 SystemVerilog Standard and how **pure virtual** methods are already in use today.*

*Prepare to enter SystemVerilog's new **virtual** reality!*

Table of Contents

1	Introduction.....	5
1.1	Example code.....	5
2	Quick intro to classes and methods.....	5
2.1	What is a class?.....	5
2.2	What is an object handle.....	6
2.3	Handles -vs- pointers.....	6
2.4	What is an object?.....	7
2.5	Reconstructing or removing objects.....	8
3	Class extension.....	9
3.1	Override versus overload.....	10
3.2	Super keyword.....	11
3.3	This keyword.....	14
4	virtual classes.....	15
5	Class methods.....	16
6	Non-virtual methods.....	16
7	virtual methods.....	16
7.1	Once virtual, always virtual.....	22
8	The pure keyword.....	23
9	pure virtual methods.....	23
9.1	Extending pure virtual methods.....	25
9.2	Work-around for pure virtual methods.....	33
10	pure constraints.....	34
11	Polymorphism.....	34
12	virtual interfaces.....	36
12.1	Interfaces.....	36
12.2	Dynamically connecting to virtual interfaces.....	37
12.3	Step #1 - Build the real interface.....	37
12.4	Step #2 - Instantiate a virtual interface into the transactor class.....	39
12.5	Step #3 - Create a new() constructor to tie the real interface to the virtual interface.....	41
13	Conclusions.....	44
14	References.....	44
15	Author & Contact Information.....	44
16	Appendix.....	46

Table of Figures

Figure 1 - alu_reg block diagram	38
Figure 2 - Top-level module with design (DUT), interface and clock generator	38
Figure 3 - alu_if block diagram	39

Table of Examples

Example 1 - Simple class declaration: base1	6
Example 2 - Class declaration with handle declaration: base1 b1;.....	6
Example 3 - Construction of the b1 object handle using the new() constructor	7
Example 4 - Construction of the b1 object handle using the new() constructor at time 10	7
Example 5 - Reconstruction of the b1 object handle removes the first b1 object.....	8
Example 6 - BAD: null b1 object handle causes fatal run-time null-object-handle access for show() method	9
Example 7 - Cmd base class base and extended NewCmd class	10
Example 8 - Example of extended pre-method functionality	11
Example 9 - Example of extended post-method functionality.....	12
Example 10 - Use of "this" command to access class data member rather than method argument	14
Example 11 - Valid virtual class declaration with subsequent virtual class handle declaration... ..	15
Example 12 - BAD - attempt to construct "new();" a virtual class handle	15
Example 13 - Valid virtual method - matching argument and return types.....	17
Example 14 - BAD virtual method - argument name does not match	18
Example 15 - BAD virtual method - argument type does not match.....	19
Example 16 - BAD virtual method - argument direction does not match	20
Example 17 - BAD virtual method - number of arguments does not match.....	21
Example 18 - BAD virtual method - return type does not match	22
Example 19 - Valid virtual class with pure virtual method and declared class handle.....	24
Example 20 - BAD pure virtual method with "endfunction" declaration	24
Example 21 - BAD use of pure - pure can only be used with virtual methods.....	24
Example 22 - Valid pure virtual methods must be extended in a non-virtual class.....	25
Example 23 - BAD - pure virtual methods must be extended with the same argument and return types	26
Example 24 - BAD - pure virtual methods MUST be extended in the first non-virtual class	27

Example 25 - BAD - pure virtual methods cannot be defined in a non-virtual class.....	28
Example 26 - Valid - an extended virtual class does not have to extend a pure virtual method...	29
Example 27 - BAD - First non-virtual class must override all inherited base class pure virtual methods	30
Example 28 - BAD - First non-virtual class must override all declared base class pure virtual methods	31
Example 29 - Valid - non-virtual class does not have to override non-pure methods from virtual base class.....	32
Example 30 - Pure virtual function work-around: (1) empty function, (2) add endfunction.....	33
Example 31 - VMM pseudo-pure virtual method work-around	33
Example 32 - Pure constraint with override in non-virtual class	34
Example 33 - Polymorphism example showing base class handle accessing extended classes ...	35
Example 34 - alu_reg.sv SystemVerilog source code.....	38
Example 35 - alu_if.sv SystemVerilog source code	39
Example 36 - clkgen.sv SystemVerilog source code	39
Example 37 - top.sv SystemVerilog source code.....	39
Example 38 - Testbench class SystemVerilog source code with virtual interface.....	40
Example 39 - Real interface handle aif assigned to new() handle nif, assigned to virtual interface handle vif	41
Example 40 - Non-virtual method overrides another method with matching argument and return types	46
Example 41 - Valid - Non-virtual method overrides another method with non-matching argument type.....	47
Example 42 - Valid - Non-virtual method overrides another method with non-matching argument direction	48
Example 43 - Valid - Non-virtual method overrides another method with different number of arguments	49
Example 44 - Valid - Non-virtual method overrides another method with different return type .	50

1 Introduction

The SystemVerilog keywords **virtual** and **pure** impose coding requirements on a class-based verification environment to help implement high-level verification methodologies such as those found in the VMM and OVM. And yet there are a number of misconceptions and misunderstandings regarding the use of these constructs in a SystemVerilog environment.

This paper will detail how the **virtual** keyword is used to create abstract classes, runtime and polymorphic class methods and interfaces between a static design and a dynamic verification environment.

This paper will also introduce the SystemVerilog 2009 keyword **pure**, how pure methods and constraints work, how pure methods have already been added to some SystemVerilog simulator implementations and the simple work-around to make pure methods work in all SystemVerilog-2005 implementations.

It is always easier to use a language feature if one knows how the feature works and the problems the feature solves. It is the intent of this paper to provide this background education on the powerful virtual and pure features of SystemVerilog-2005 and 2009.

1.1 Example code

The examples shown in this paper are full working examples, or full examples that will not compile due to the errors described in the examples. It would have been easier and less verbose to just include code snippets but it is our experience that full examples do a better job of answering questions that might arise if the full example were not present.

2 Quick intro to classes and methods

For those who are new to SystemVerilog, this section gives a quick introduction to the basic concepts needed to understand the declaration and basic usage of SystemVerilog classes and methods.

If you are already familiar with SystemVerilog class basics, you can skip to section 3.

2.1 What is a class?

A class is a dynamic type definition that includes data members (commonly referred to as properties or field members in Object Oriented Programming) and SystemVerilog tasks and functions that are intended to interact with the data members. Tasks and functions defined within a class are often referred to as methods, both in Object Oriented Programming and in SystemVerilog.

Example 1 shows a simple **base1** class declaration with one data member named **a** and one method, a virtual void function named **set_show**.

```
class base1;
  bit [7:0] a;

  virtual function void set_show (bit [7:0] i1);
    a = i1;
    $display("base1 : a = %2h", i1);
  endfunction
endclass
```

Example 1 - Simple class declaration: base1

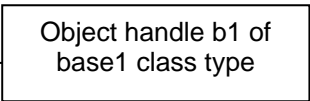
In SystemVerilog, we typically avoid calling class data members "properties" because **property** is a SystemVerilog keyword and has a different meaning when referring to SystemVerilog assertion based design and verification.

2.2 What is an object handle

An object handle is basically a "safe" pointer to an object. A handle to an object can be declared without actually having constructed an object. More on this later.

```
program example2;
  class base1;
    bit [7:0] a;

    virtual function void set_show (bit [7:0] i1);
      a = i1;
      $display("base1 : a = %2h", i1);
    endfunction
  endclass

  base1 b1; ← 
endprogram
```

Example 2 - Class declaration with handle declaration: base1 b1;

In Example 2, the handle **b1** is declared for the **base1** class type, but no object has been constructed so the **b1** handle is currently assigned the value **null**. For this example, there is a null handle of the **base1** type.

Handles can be constructed or they can be assigned the values of extended class handles of the same base class type. More on this later.

2.3 Handles -vs- pointers

In C and C++, a pointer can be mathematically manipulated. If the wrong value is added to a pointer, the pointer might not point to valid data and the program behaviour referencing the invalid pointer can execute with completely unexpected behaviour.

Like Java, SystemVerilog handles cannot be mathematically manipulated; therefore, they do not allow the execution of unexpected program code, which is possible in C and C++.

2.4 What is an object?

An object is an instantiation of a class. Making an instantiation is referred to as construction of an object and construction is done using the built in `new()` constructor.

```
program example3;
class base1;
  bit [7:0] a;

  virtual function void set_show (bit [7:0] i1);
    a = i1;
    $display("base1 : a = %2h", i1);
  endfunction
endclass

base1 b1 = new(); ← b1 object constructed

initial b1.set_show(8'h55);
endprogram
```

Example 3 - Construction of the `b1` object handle using the `new()` constructor

In Example 3, the `b1` handle of `base1` type is constructed using the `new()` constructor. Once an object is constructed, the handle can be used to call class methods. The `set_show` method is called in the `initial` block using the `b1` handle.

Classes are dynamic types and class objects can be constructed during run time. In Example 3, the object was constructed during declaration of the `b1` class handle at time 0. In Example 4, the `b1` class handle was declared as a stand-alone statement, then the `b1` handle was constructed in the `initial` block at time 10.

```
program example4;
class base1;
  bit [7:0] a;

  virtual function void set_show (bit [7:0] i1);
    a = i1;
    $display("base1 : a = %2h", i1);
  endfunction
endclass

base1 b1;

initial begin
  #10 b1 = new(); ← b1 object constructed at time-10
  b1.set_show(8'h55);
end
endprogram
```

Example 4 - Construction of the `b1` object handle using the `new()` constructor at time 10

2.5 Reconstructing or removing objects

SystemVerilog objects cannot be explicitly destroyed; there are no object destructors like in C++. It is possible to "destroy" a SystemVerilog object by either reconstructing a handle using the same name as an existing handle, or by explicitly setting a handle to **null** (**null** is a SystemVerilog keyword).

Constructing an object with the name of an existing object handle destroys the previous object. In Example 5, the **b1** handle is reconstructed at time 10, causing any reference to the original **b1** handle to be lost. The first call to the **b1.show()** method returns the value of **55**. The second call to the **b1.show()** method returns the default, uninitialized value for the **a** variable, which is **00**.

```
program example5;
  class base1;
    bit [7:0] a;

    virtual function void set (bit [7:0] i1);
      a = i1;
    endfunction

    virtual function void show;
      $display("base1 : a = %2h", a);
    endfunction
  endclass

  base1 b1 = new();
  initial begin
    b1.set(8'h55);
    b1.show();
    #10
    b1 = new();
    b1.show();
  end
endprogram
```

The diagram illustrates the reconstruction of the **b1** object handle. It shows two boxes with arrows pointing to the corresponding lines in the code. The first box, labeled "b1 object constructed at time-0", points to the line `base1 b1 = new();`. The second box, labeled "b1 object reconstructed at time-10", points to the line `b1 = new();` inside the `initial begin` block.

Example 5 - Reconstruction of the **b1** object handle removes the first **b1** object

If there is only one handle pointing to an object, then setting that object handle to **null** removes all access to the original object. In Example 6, the **b1** handle is set to **null** at time 10, causing any reference to the original **b1** handle to be lost. The first call to the **b1.show()** method returns the value of **55**. The second call to the **b1.show()** method causes a fatal run-time null-object-handle error.

```

program example6;
class base1;
  bit [7:0] a;

  virtual function void set (bit [7:0] i1);
    a = i1;
  endfunction

  virtual function void show;
    $display("base1 : a = %2h", a);
  endfunction
endclass

base1 b1 = new();

initial begin
  b1.set(8'h55);
  b1.show();
  #10
  b1 = null;
  b1.show();
end
endprogram

```

Example 6 - BAD: null b1 object handle causes fatal run-time null-object-handle access for show() method

Garbage collection is automatic in SystemVerilog. When there are no longer any handles that point to the object, there is no longer a way to access the object data and method so the object storage space is removed and recovered from use. This is called garbage collection, or in other words, recycling the unused memory. If the unused memory space were not recycled, it would be possible to eventually use up all of the available simulation memory by leaving chunks of memory lying around until we run out of memory. This condition is referred to as memory leaks; we keep leaking chunks of memory by not recycling the storage during simulation.

3 Class extension

To create a second copy of a base class with modifications, we use the keyword **extends** to copy all of the base class data members and methods into the extended class and then add-to or modify the base class data and methods.

Example 7 shows two command classes that will be used in another example later in the paper. The **Cmd** class is the base class with four data members, **a**, **b**, **opcode**, **cycle**, and one virtual void function (class method). In the **Cmd** class, the **printmsg** method displays the two randomized data members **a** and **b**, concatenated together and displayed as a single hex value. The

`printmsg` method also displays the hex-numeric value of the `opcode` instead of the `opcode` name.

```
class Cmd;
  rand bit  [15:0] a, b;
  rand op_e  opcode;
  rand bit   cycle;

  virtual function void printmsg;
    $display("Cmd:  cmd=%h  opcode=%h", {a,b}, opcode);
  endfunction
endclass

class NewCmd extends Cmd;
  virtual function void printmsg;
    $display("NewCmd: data1=%h  data2=%h  opcode=%s", a, b, opcode.name());
  endfunction
endclass
```

Example 7 - Cmd base class base and extended NewCmd class

To make a second copy of this class type with all of the same data members but with a modified (overridden) `printmsg` method, the `Cmd` class is extended using the `extends` keyword to form a `NewCmd` class that displays the `printmsg`-method data as two separate hex values for `a` and `b` respectively, along with the a display of the `opcode` name as opposed to its hex value.

3.1 *Override versus overload*

In SystemVerilog, an extended class can override a base class method by the same name, but there is no such thing as overloaded methods.

The distinction between override and overload is this: override replaces the base class method completely in the extended class. Overloading a method would allow more than one version of a method by the same name to exist and the method called would be based on the data types of the arguments passed to the method. Due to the implementation difficulty of working with loose data types in Verilog and SystemVerilog, method and function overloading in SystemVerilog is not supported.

3.2 Super keyword

The **super** keyword is used to reference a data member or method from the base class. The **super** keyword is often used to extend the functionality of a base class method.

When extending a method in an extended class, it is legal to add functionality *before* the corresponding method functionality in the base class. Example 8 shows an extended method that uses a **super**.<method> call in an extended class.

```
program example8;
  class Base1;
    bit [7:0] d1, d2, csum;

    virtual function void set (bit [7:0] val1='0, bit [7:0] val2=8'hAA);
      d1 = val1;
      d2 = val2;
    endfunction

    virtual function void calc_csum (bit [7:0] id = '0);
      csum = d1 + d2;
    endfunction

    virtual function void show;
      $display("base1 : data = %2h:%2h csum = %2h", d1, d2, csum);
    endfunction
  endclass

  class Ext1 extends Base1;
    virtual function void calc_csum (bit [7:0] id = '0);
      d1 = id;
      super.calc_csum();
    endfunction

    virtual function void show;
      $write("ext1 : ");
      super.show();
    endfunction
  endclass

  Base1 b1 = new();
  Ext1 e1 = new();

  initial begin
    b1.set(8'h01);
    b1.calc_csum(8'h03);
    b1.show();
    e1.set(8'h01);
    e1.calc_csum(8'h03);
    e1.show();
  end
endprogram
```

Example 8 - Example of extended pre-method functionality

In Example 8, the `Basel calc_csum` method calculates a simple checksum on the data items of the class. In the `Ext1` extended class, the extended `calc_csum` method sets the first class data member (`d1`) to match the method argument value (`id`) and then calls the base class `calc_csum` method to do the checksum calculation. After the `e1` object is constructed, an `initial` procedure calls the `e1.calc_csum` method with an `id` value of `8'h03`. The `e1.calc_csum` method first updates the `d1` data member from a value of `8'h01` to `8'h03` and then calls `super.calc_csum` to use the parent class checksum calculation. This allows the reuse of complex calculations in parent class methods on extended class data values without having to rewrite the calculation code.

```

program example9;
  class Basel;
    bit [7:0] d1, d2, csum;

    virtual function void set (bit [7:0] val1='0, bit [7:0] val2=8'hAA);
      d1 = val1;
      d2 = val2;
    endfunction

    virtual function void calc_csum (bit [7:0] id = '0);
      csum = d1 + d2;
    endfunction

    virtual function void show;
      $display("basel : data = %2h:%2h csum = %2h", d1, d2, csum);
    endfunction
  endclass

  class Ext1 extends Basel;
    virtual function void calc_csum (bit [7:0] id = '0);
      super.calc_csum();
      csum += 8'h0f; // Corrupt the checksum.
    endfunction

    virtual function void show;
      $write("ext1 : ");
      super.show();
    endfunction
  endclass

  Basel b1 = new();
  Ext1 e1 = new();

  initial begin
    b1.set(8'h01);
    b1.calc_csum(8'h03);
    b1.show();
    e1.set(8'h01);
    e1.calc_csum(8'h03);
    e1.show();
  end
endprogram

```

Example 9 - Example of extended post-method functionality

Example 9 shows that the **super** .<method> call in the extended class can also add functionality *after* the base class method functionality. The base class functionality is the same as in Example 8. The extended class **calc_csum** method calls the **super.calc_csum** first to calculate the valid checksum using the parent class functionality and then corrupts the checksum to create an “error” data object.

There are a few more details about the **super** keyword that need to be understood.

The **super** keyword cannot be used in a top-level base class since there is no parent class that could be referenced with the **super** keyword.

In a UNIX file system, one often references a file or executable from a parent directory using the **../** notation. For example, to change directories to the parent directory, one would execute the command: **cd ..**

The **super** command is analogous to the **../** notation used in a UNIX file system, but unlike the UNIX file system, SystemVerilog does not permit reference to a data member or method up two or more levels of base classes. The following syntax would be illegal:

```
super.super.gen_crc
```

You are only allowed to immediately reference one-level of **super**. If you need to reference a method that is two levels up from the current class definition, you will have to reference the **super.<method>** and that method will have to reference its **super.<method>**.

3.3 This keyword

The **this** keyword refers to the current class definition and is used to remove ambiguity when accessing methods or data members (primarily data members) of the same name at different levels of naming scope within a class structure. This allows a sub-scope within the class that has a local name that is the same as a name at the top scope of the class, to access the top scope name item.

The **this** command is analogous to the `./` notation used in a UNIX file system to access current directory information. For example, to run the script **foo** that is in the local directory, one would execute the command: `./foo`

In Example 10, **this.i1** is used to access the data member of the current class definition rather than the argument **i1** of the set method (it is recommended to avoid coding styles that need to use the **this** command). Without using the **this** command, the method would only be able to access the argument **i1** passed in to it. The **bad_set** method takes the passed in argument value and writes it back to the argument so that when the show method is called after **bad_set**, it shows the same value for **i1** that was set using the previous set method call.

```
program example10;
  class base1;
    bit [7:0] i1;

    virtual function void set (bit [7:0] i1);
      this.i1 = i1; ← assign to base1
    endfunction                               i1 data member

    virtual function void bad_set (bit [7:0] i1);
      i1 = i1; ← assign to bad_set
    endfunction                               i1 data member

    virtual function void show;
      $display("base1 : this.i1 = %2h", this.i1);
    endfunction
  endclass

  base1 b1 = new(); ← b1 object
                                     constructed

  initial begin
    b1.set(8'h55);
    b1.show();
    b1.bad_set(8'hAA); ← b1.badset
    b1.show();         method call does
                                     not update b1.i1
  end
endprogram
```

Example 10 - Use of "this" command to access class data member rather than method argument

4 virtual classes

A virtual class is a class that cannot be directly constructed.

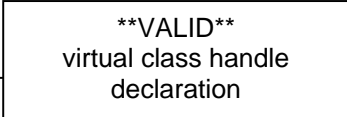
A **virtual class** is often referred to as an abstract class that cannot be directly constructed. The virtual class is intended to create a template for classes that are extensions of the virtual class. The virtual class is not intended to be constructed and used directly, but different forms of the extended class are frequently generated in a constrained random testbench and then assigned to a virtual class handle for consistent execution of the pre-defined legal class methods.

In Example 11, a **virtual base1 class** is declared, and subsequently a virtual class handle called **b1** is declared to be of the **base1** type. It is permitted to declare a virtual class handle and this is frequently done in advanced verification methodologies.

```
typedef bit [7:0] arg1_t;

program example11;
  virtual class base1;
    virtual function void showit (arg1_t i1);
      $display("base1 : arg1_t = %2h", i1);
    endfunction
  endclass

  base1 b1;
endprogram
```



The diagram shows a rectangular box containing the text ****VALID** virtual class handle declaration**. An arrow points from the right side of this box to the line `base1 b1;` in the code block above.

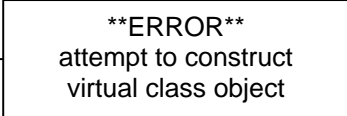
Example 11 - Valid virtual class declaration with subsequent virtual class handle declaration

In Example 12, the same **virtual base1 class** is declared but then an attempt is made to declare and construct a **b1** handle of the **virtual base1** class type. This is illegal.

```
typedef bit [7:0] arg1_t;

program example12;
  virtual class base1;
    virtual function void showit (arg1_t i1);
      $display("base1 : arg1_t = %2h", i1);
    endfunction
  endclass

  base1 b1 = new();
endprogram
```



The diagram shows a rectangular box containing the text ****ERROR** attempt to construct virtual class object**. An arrow points from the right side of this box to the line `base1 b1 = new();` in the code block above.

Example 12 - BAD - attempt to construct "new();" a virtual class handle

There is a common misconception that all methods declared in a virtual class are automatically virtual methods. This is not true. For method to be virtual, it must be declared virtual, even in virtual classes.

A virtual class is not useful until extended. Classes extended from the virtual base class inherit capabilities from the virtual base class and if the base class used virtual methods, the extended class methods can be used in a polymorphic way as described in Section 11.

5 Class methods

Class methods can be divided into three categories:

- Regular class methods (non-virtual methods).
- Virtual class methods.
- Pure virtual class methods.

Each of these method types is explored in the following sections.

6 Non-virtual methods

Any class function or task that does not include the **virtual** keyword is a non-virtual method. Non-virtual methods can be defined in both classes and virtual classes.

An extended class can override a non-virtual method without any restrictions.

Non-virtual methods do not generally lend themselves well to polymorphism and advanced verification methodologies, so the examples of non-virtual methods and accompanying legal examples of same are de-emphasized in this paper and therefore have been demoted to the appendix (see section 16 for Appendix examples).

7 virtual methods

Virtual methods require all arguments and types to be the same in the corresponding extended methods.

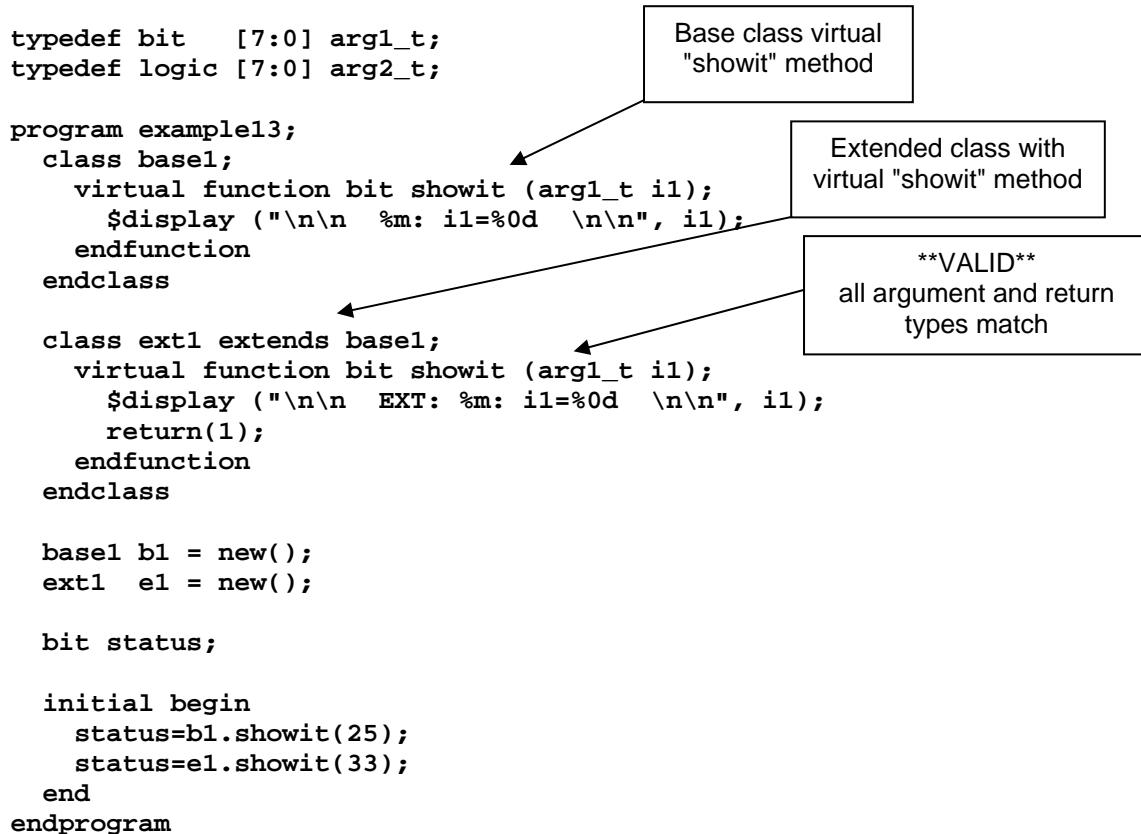
Virtual methods impose one very important restriction in extended classes: virtual methods force the use of the exact same number of arguments and the exact same argument and return types when the method is extended. Non-virtual methods do not.

Virtual methods can be defined in classes and virtual classes.

Virtual methods differ from non-virtual methods in that the number, types and names of the arguments, as well as the return type, must match the same arguments and types as the base class method, if overridden. An extended non-virtual method is allowed to change every calling and return detail of the base class method by the same name.

In Example 13, the **base1** class has a virtual method called **showit**. The **showit** method has (a) a 1-bit return type (a status bit), (b) one input argument named **i1**, and (c) the input argument is of type **arg1_t**, which was defined to be an 8-bit **bit**-type at the top of the example.

The **ext1** class is a valid extension of the **base1** class because it too has (a) a 1-bit return type, (b) just one argument that is also an input argument, and (c) the input argument is also the **arg1_t** type.



Example 13 - Valid virtual method - matching argument and return types

The virtual method in the extended class of Example 14 is illegal because the input argument is not the same name as used in the base class virtual method. In the base class virtual method, the argument name is `i1` while the extended class virtual method the argument type is `in1`. This program will not compile.

```
program example14;
  class base1;
    virtual function bit showit (arg1_t i1);
      $display ("\n\n %m: i1=%0d \n\n", i1);
    endfunction
  endclass

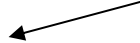
  class ext1 extends base1;
    virtual function bit showit (arg2_t in1);
      $display ("\n\n EXT: %m: i1=%0d \n\n", i1);
      return(0);
    endfunction
  endclass

  base1 b1 = new();
  ext1 e1 = new();

  bit status;

  initial begin
    status=b1.showit(25);
    status=e1.showit(33);
  end
endprogram
```

****ERROR****
argument name does not
match



Example 14 - BAD virtual method - argument name does not match

The virtual method in the extended class of Example 15 is illegal because the input argument is not the same type as the base class virtual method. In the base class virtual method, the argument type is **arg1_t** while the extended class virtual method the argument type is **arg2_t**. This program will not compile.

```
typedef bit    [7:0] arg1_t;
typedef logic [7:0] arg2_t;

program example15;
  class base1;
    virtual function bit showit (arg1_t i1);
      $display ("\n\n %m: i1=%0d \n\n", i1);
    endfunction
  endclass

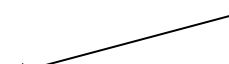
  class ext1 extends base1;
    virtual function bit showit (arg2_t i1);
      $display ("\n\n EXT: %m: i1=%0d \n\n", i1);
      return(0);
    endfunction
  endclass

  base1 b1 = new();
  ext1 e1 = new();

  bit status;

  initial begin
    status=b1.showit(25);
    status=e1.showit(33);
  end
endprogram
```

****ERROR****
argument type does not
match



Example 15 - BAD virtual method - argument type does not match

The virtual method in the extended class of Example 16 is illegal because the argument direction is not the same as the argument direction in the base class virtual method. In the base class virtual method, the argument direction is **input** while in the extended class virtual method the direction is **output**. This program will not compile.

```
typedef bit    [7:0] arg1_t;
typedef logic [7:0] arg2_t;

program example16;
  class base1;
    virtual function bit showit (arg1_t i1);
      $display ("\n\n  %m: i1=%0d  \n\n", i1);
    endfunction
  endclass

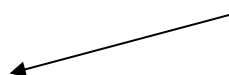
  class ext1 extends base1;
    virtual function bit showit (output arg1_t i1);
      $display ("\n\n  EXT: %m: i1=%0d  \n\n", i1);
      return(0);
    endfunction
  endclass

  base1 b1 = new();
  ext1  e1 = new();

  bit status;

  initial begin
    status=b1.showit(25);
    status=e1.showit(33);
  end
endprogram
```

****ERROR****
argument direction does
not match



Example 16 - BAD virtual method - argument direction does not match

The virtual method in the extended class of Example 17 is illegal because the number of arguments does not match the number of arguments in the base class virtual method. In the base class virtual method, there is one input argument while in the extended class virtual method there are two input arguments. This program will not compile.

```
typedef bit    [7:0] arg1_t;
typedef logic [7:0] arg2_t;

program example17;
  class base1;
    virtual function bit showit (arg1_t i1);
      $display ("\n\n %m: i1=%0d \n\n", i1);
    endfunction
  endclass

  class ext1 extends base1;
    virtual function bit showit (arg1_t i1, i2=55);
      $display ("\n\n EXT: %m: i1=%0d i2=%0d \n\n", i1, i2);
      return(0);
    endfunction
  endclass

  base1 b1 = new();
  ext1 e1 = new();

  bit status;

  initial begin
    status=b1.showit(25);
    status=e1.showit(33);
  end
endprogram
```

****ERROR****
number of arguments
does not match

Example 17 - BAD virtual method - number of arguments does not match

The virtual method in the extended class of Example 18 is illegal because the function return type is not the same type as the base class function return type. In the base class virtual method, the return type is **bit** while the extended class virtual method return type is **logic**. This program will not compile.

```

typedef bit    [7:0] arg1_t;
typedef logic [7:0] arg2_t;

program example18;
  class base1;
    virtual function bit showit (arg1_t i1);
      $display ("\n\n  %m: i1=%0d \n\n", i1);
    endfunction
  endclass

  class ext1 extends base1;
    virtual function logic showit (arg1_t i1);
      $display ("\n\n  EXT: %m: i1=%0d \n\n", i1);
      return(0);
    endfunction
  endclass

  base1 b1 = new();
  ext1 e1 = new();

  bit status;

  initial begin
    status=b1.showit(25);
    status=e1.showit(33);
  end
endprogram

```

****ERROR****
 funciton return type does
 not match

Example 18 - BAD virtual method - return type does not match

Virtual methods are an important part of polymorphism (see section 11). By requiring all virtual methods to exactly match the number, names and types of arguments used in the base class method, it is possible to make a base class handle point to an extended class object derived from the common base class and call the base class methods with full confidence that calling the method in either the base class or extended class will be valid, even if the base and extended class methods execute different code inside of the methods.

7.1 Once virtual, always virtual

If the base class declares a method to be **virtual**, the extended class method by the same name is also virtual, even if it is not declared to be virtual. The **virtual** keyword is essentially a "sticky" keyword that sticks to all extended classes with methods by that same name. There is no way to remove the virtual stickiness from an extended method by the same name.

Guideline: To avoid confusion, declare all virtual methods in extended classes with the **virtual** keyword.

Reason: Even though overriding a virtual base class method in an extended class is still virtual, if another engineer examines the extended class code but does not examine the base class code, that engineer could come to the erroneous conclusion that a virtual method is actually a non-virtual method. The **virtual** keyword on all virtual methods avoids this type of confusion.

8 The pure keyword

Note: the **pure** keyword as used with methods was not technically legal until SystemVerilog-2009, but you will find **pure virtual** methods commonly used in the OVM.

The best way to think of the **pure** keyword is that it imposes three immediate restrictions on a class based methodology, and these restriction are checked by the compiler:

- (1) The definition of **pure** methods and constraints is only allowed in an abstract class (**virtual class**).
- (2) A virtual method or constraint shall not have an implementation in the virtual class (the method or constraint must be empty).
- (3) The virtual class or method *MUST* be overridden in the first non-virtual extended class based on the abstract class.

9 pure virtual methods

It is illegal to use the **pure** keyword in a class. The **pure** keyword can only be used in an abstract class (**virtual class**).

The **pure virtual** method is either a **pure virtual task** or **pure virtual function** prototype that is not allowed to have an implementation (no body code allowed) and is not even allowed to have an **endtask** or **endfunction** keyword to close the pure virtual method.

In Example 19, a **pure virtual function** is properly defined within a **virtual class** and the pure virtual function has no function-body code, not even an **endfunction** statement.

```

typedef bit [7:0] arg1_t;

program example19;
  virtual class base1;
    pure virtual function bit showit (arg1_t i1);
  endclass

  base1 b1;
endprogram

```

Example 19 - Valid virtual class with pure virtual method and declared class handle

In Example 20 shows a common new user mistake. Even though the **pure virtual function** has no function-body code, the function erroneously includes an **endfunction** keyword. This program will not compile.

```

typedef bit [7:0] arg1_t;

program example20;
  virtual class base1;
    pure virtual function bit showit (arg1_t i1);
    endfunction
  endclass

  base1 b1;
endprogram

```

Example 20 - BAD pure virtual method with "endfunction" declaration

In Example 21, a **virtual class** is defined but the **pure** keyword was erroneously added to a non-virtual method. This program will not compile.

```

typedef bit [7:0] arg1_t;

program example21;
  virtual class base1;
    pure function bit showit (arg1_t i1);
  endclass

  base1 b1;
endprogram

```

Example 21 - BAD use of pure - pure can only be used with virtual methods

9.1 Extending pure virtual methods

As a prototype, a **pure virtual** method essentially imposes the requirement that any extension of an abstract class with a non-abstract class, ***MUST*** override the virtual method using the exact same arguments (number, size and type) and in the case of a **pure virtual function**, must use the exact same return argument type.

In Example 22, a virtual **base1** class includes a properly coded **showit** pure virtual method. The **ext1** non-virtual class extends the **base1** base class and is required to also extend the **showit** pure virtual method with actual method functionality as shown.

```
typedef bit [7:0] arg1_t;

program example22;
  virtual class base1;
    pure virtual function bit showit (arg1_t i1);
  endclass

  base1 b1;

  class ext1 extends base1;
    virtual function bit showit (arg1_t i1);
    $display ("\n\n EXT: %m: i1=%0d \n\n", i1);
    return(1);
  endfunction
  endclass

  ext1 e1 = new();

  bit status;

  initial begin
    status=e1.showit(33);
  end
endprogram
```

****VALID****
Non-virtual class extends virtual class and properly extends the pure virtual method

Example 22 - Valid pure virtual methods must be extended in a non-virtual class

In Example 23, the **showit** pure virtual method of the **base1** virtual class is erroneously overridden in the extended **ext1** class. As is true with any virtual method, pure virtual methods require declarations that use the exact same arguments (number, size and type), which is not the case with the extended **showit** method in this example. This program will not compile.

```
typedef bit [7:0] arg1_t;

program example23;
  virtual class base1;
    pure virtual function bit showit (arg1_t i1);
  endclass

  base1 b1;

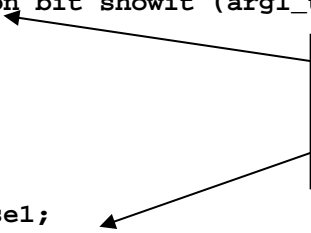
  class ext1 extends base1;
    virtual function logic showit (inout arg1_t i1, input int i2=55);
      $display ("\n\n EXT: %m: i1=%0d i2=%0d \n\n", i1, i2);
      return(0);
    endfunction
  endclass

  ext1 e1 = new();

  bit status;

  initial begin
    status=e1.showit(33);
  end
endprogram
```

****ERROR****
The pure virtual method is extended with the wrong return type, wrong number of arguments and wrong argument directions



Example 23 - BAD - pure virtual methods must be extended with the same argument and return types

In Example 24, because the **showit** virtual method of the **base1** virtual class was declared to be **pure**, all non-virtual class extensions of the **base1** class must provide an implementation for the **showit** method. The extended **ext1** class provides a new **showit2** method but does not provide a **showit** method implementation. This program will not compile.

```
typedef bit [7:0] arg1_t;

program example24;
  virtual class base1;
    pure virtual function bit showit (arg1_t i1);
  endclass

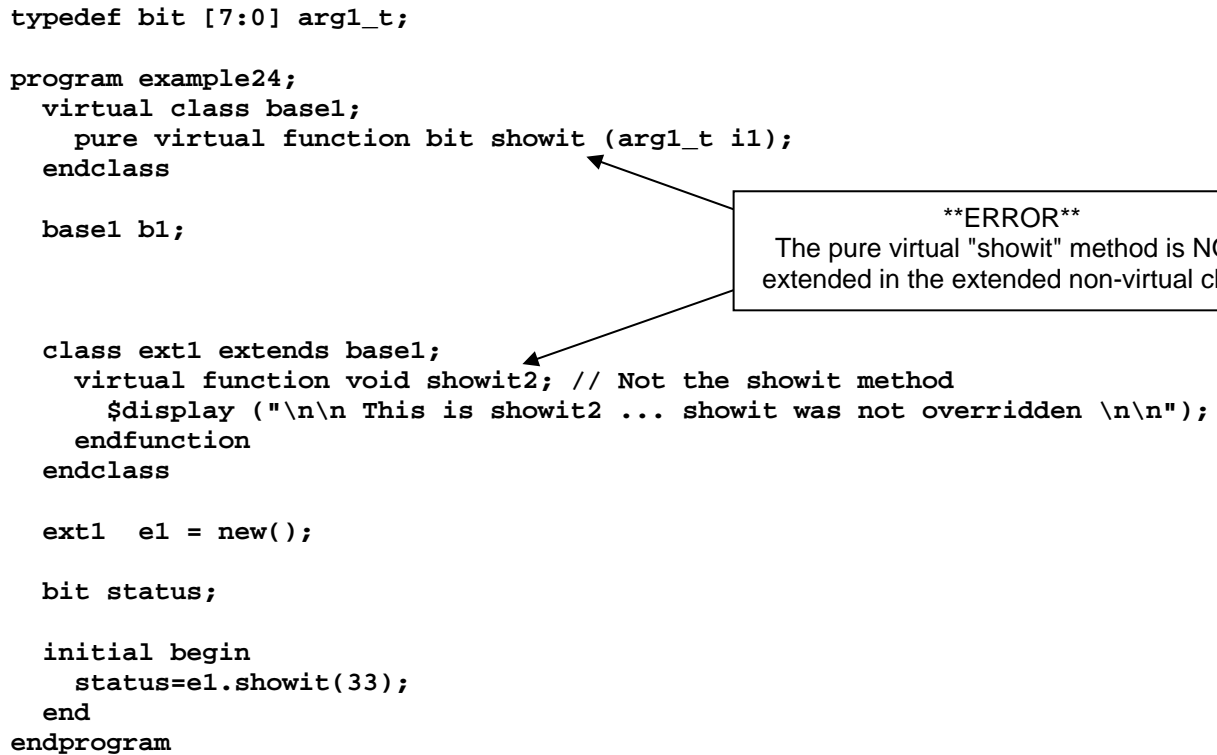
  base1 b1;

  class ext1 extends base1;
    virtual function void showit2; // Not the showit method
    $display ("\n\n This is showit2 ... showit was not overridden \n\n");
    endfunction
  endclass

  ext1 e1 = new();

  bit status;

  initial begin
    status=e1.showit(33);
  end
endprogram
```



The diagram shows a rectangular box with the text: ****ERROR****
The pure virtual "showit" method is NOT extended in the extended non-virtual class. Two arrows originate from the box: one points to the `showit` method declaration in the `base1` class, and the other points to the `showit2` method declaration in the `ext1` class.

Example 24 - BAD - pure virtual methods MUST be extended in the first non-virtual class

In Example 25 there are actually two problems in the non-virtual extended **ext1** class. The first problem is that there is no **showit** method provided to override the **pure virtual showit** method of the base class. The second problem is that the **ext1** extended class attempts to declare a **pure virtual showit2** function in a non-virtual class. It is only legal to declare **pure virtual** methods in a **virtual class**. This program will not compile.

```
typedef bit [7:0] arg1_t;

program example25;
  virtual class base1;
    pure virtual function bit showit (arg1_t i1);
  endclass

  base1 b1;

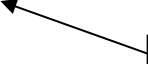
class ext1 extends base1;
  virtual function bit showit (arg1_t i1);
    $display ("\n\n EXT: %m: i1=%0d \n\n", i1);
    return(1);
  endfunction

  pure virtual function void showit2; // Not the showit method
endclass

ext1 e1 = new();

bit status;

initial begin
  status=e1.showit(33);
end
endprogram
```



****ERROR****
A pure virtual "showit2" method is NOT permitted in the extended non-virtual class

Example 25 - BAD - pure virtual methods cannot be defined in a non-virtual class

It should be noted that if a virtual class is extended by another virtual class, the extended virtual class is allowed to override the pure virtual method of the base class with another pure virtual method. It is only required to override the pure virtual methods with a non-pure virtual method in the first non-virtual extended class. This condition exists in Example 26.

```

typedef bit [7:0] arg1_t;

program example26;
  virtual class base1;
    pure virtual function bit showit (arg1_t i1);
  endclass

  virtual class ext1 extends base1;
    // No override of pure virtual function showit
  endclass

  class ext2 extends ext1;
    virtual function bit showit (arg1_t i1);
      $display ("\n\n EXT2: %m: i1=%0d \n\n", i1);
      return(1);
    endfunction
  endclass

  ext2 e2 = new();

  bit status;

  initial begin
    status=e2.showit(33);
  end
endprogram

```

****VALID****
 The pure virtual "showit" method is not extended in the extended virtual class

****VALID****
 The pure virtual "showit" method is finally extended in the extended non-virtual class

Example 26 - Valid - an extended virtual class does not have to extend a pure virtual method

In Example 27, a virtual **base1** class is extended in an **ext1** virtual class, which inherits the pure virtual method from the **base1** virtual class. The virtual **ext1** class has no visible methods (pure or non-pure), but it does have an inherited pure virtual method. The extended non-virtual **ext2** class is required to override the **showit** pure virtual function, but it does not. This program will not compile.

```
typedef bit [7:0] arg1_t;

program example27;
  virtual class base1;
    pure virtual function bit showit (arg1_t il);
  endclass

  virtual class ext1 extends base1;
    // extended virtual class inherits pure virtual method
    // from base class unless overridden
  endclass

  class ext2 extends ext1;
    // No override of virtual function showit;
  endclass

  ext2 e2 = new();

  bit status;

  initial begin
    status=e2.showit(33);
  end
endprogram
```

****VALID****
The pure virtual "showit" method is not extended in the extended virtual class

****ERROR****
First non-virtual class must override all base class pure virtual methods

Example 27 - BAD - First non-virtual class must override all inherited base class pure virtual methods

In Example 28, a virtual **base1** class is extended in an **ext1** virtual class, and the **ext1** virtual class overrides the **showit** pure virtual function of the **base1** class with its own **showit** pure virtual function. The extended non-virtual **ext2** class is required to override the **showit** pure virtual function, but it does not. This program will not compile.

```
typedef bit [7:0] arg1_t;

program example28;
  virtual class base1;
    pure virtual function bit showit (arg1_t i1);
  endclass

  virtual class ext1 extends base1;
    pure virtual function bit showit (arg1_t i1);
  endclass

  class ext2 extends ext1;
    // No override of virtual function showit;
  endclass

  ext2 e2 = new();

  bit status;

  initial begin
    status=e2.showit(33);
  end
endprogram
```

****VALID****
The extended virtual class overrides the pure virtual "showit" method from the base virtual class.

****ERROR****
First non-virtual class must override all base class pure virtual methods

Example 28 - BAD - First non-virtual class must override all declared base class pure virtual methods

In Example 29, a virtual **base1** class is extended in an **ext1** virtual class, and the **ext1** virtual class overrides the **showit** pure virtual function of the **base1** class with its own **showit** virtual function (not pure). The extended non-virtual **ext2** class is not required to override the **showit** virtual function. The **ext1 showit** virtual function is inherited by the **ext2** class and the example will compile and run without problems.

```

typedef bit [7:0] arg1_t;

program example29;
  virtual class base1;
    pure virtual function bit showit (arg1_t i1);
  endclass

  virtual class ext1 extends base1;
    virtual function bit showit (arg1_t i1);
      $display ("\n\n EXT1: %m: i1=%0d \n\n", i1);
      return(1);
    endfunction
  endclass

  class ext2 extends ext1;
    // No override of virtual function showit;
  endclass

  ext2 e2 = new();

  bit status;

  initial begin
    status=e2.showit(33);
  end
endprogram

```

****VALID****
 The pure virtual "showit" method IS extended as a virtual method (not pure) in the extended virtual class

****VALID****
 The non-virtual class is NOT required to override the virtual method (not pure) from the virtual class

Example 29 - Valid - non-virtual class does not have to override non-pure methods from virtual base class

9.2 Work-around for pure virtual methods

What if you are using a simulator that does not yet support pure virtual functions?

The simple work-around to replace a **pure virtual function**, as shown in Example 30, is to build an empty function or task, to place them in a **virtual class**, and to require any engineer who extends the virtual class with a non-virtual class to also extend the empty function or task with some content (rules imposed by methodology as opposed to syntax).

```
typedef bit [7:0] arg1_t;

program example30;
  virtual class base1;
    virtual function bit showit (arg1_t i1);
  endfunction
endclass

  base1 b1;
endprogram
```

(1) define a virtual method in a virtual class

(2) no function code but DO add "endfunction" to the virtual function

Example 30 - Pure virtual function work-around: (1) empty function, (2) add endfunction

The benefit of using a simulator that has fully implemented pure virtual method functionality and checking is that the compiler will ensure that the method is extended in all non-virtual classes. In the absence of this enforcement from simulation tools, engineers must follow a methodology "honor-system" to extend the pure virtual methods.

The VMM class library uses another trick to mimic a **pure virtual** method. In the **virtual class vmm_notification** from the **vmm.sv** file, shown in Example 31, the **virtual task reset()** writes out an error message and terminates the simulation if it is called during simulation.

```
virtual class vmm_notification;
...
  virtual task reset();
    $write("FATAL: An instance of vmm_notification::reset() was not ",
          "overloaded or super.reset() was called\n");
    $finish;
  endtask
endclass
```

Example 31 - VMM pseudo-pure virtual method work-around

The **reset()** task does not force a user to actually override the task when the virtual **vmm_notification** class is extended by a non-virtual class, which is the only feature that is missing that would have been enforced by a **pure virtual** method.

10 pure constraints

The use of the keyword **pure** with constraints is brand new to SystemVerilog-2009[2], and when this paper was first published, there were no known SystemVerilog simulators that had implemented this feature. For more information about this feature, see Mantis 2514[3].

Like **pure virtual** methods, a **pure constraint** must be defined in a **virtual** class and must be overridden in the any non-virtual derived class. In the Example 32, the **virtual class D** defines a data member that can be randomized, along with a **pure constraint** called **Test**. The same example contains a non-virtual **class E** that is an extension of class **D**. In the non-virtual **class E**, the constraint override is required.

```
virtual class D;
    rand bit [7:0] data;
    pure constraint Test;
endclass

class E extends D;
    constraint Test {data>8'hC0;}
endclass
```

Example 32 - Pure constraint with override in non-virtual class

Pure constraints obligate all non-virtual extensions of the virtual class with **pure constraints** to be overridden.

11 Polymorphism

Polymorphism is the concept of allowing an object or method to take on a different meaning based on the context in which it is used.

In the case of SystemVerilog, the polymorphism is based on subtypes (extensions) of classes. Subtype polymorphism requires that the objects or methods, being given a different meaning, be in subtypes of a common base type.

Usually a virtual class is used as the common base type and a handle of the base virtual class is declared to be used as the common interface to all of the extended classes.

An example of this, would be to create a virtual class of a network port that has various common pure virtual methods for manipulating data (e.g. tx_data, rx_data, process_data). Then multiple extensions (subtypes) of the network port class would be created, one for each required network type (e.g. ethernet, atm, isdn). A handle of the base virtual class would be declared in the testbench and the generic method names would be used to send, receive, and process data from the device under test.

Based on which network port type the current device under test is using, an object of the appropriate extended class would be created and the base class handle would be pointed to it. When the generic testbench calls one of the common methods through the base class handle, it is actually calling the extended class method, which can manipulate base class data fields and extended class data fields.

The base class handle is being treated as different data types based on the context in which it is used (context being the network port type of the current device under test).

```

program example33();
  // Virtual base class declaration.
  virtual class base1;
    bit [7:0] id;
    function new(int sid);
      id = sid;
    endfunction
    pure virtual function void print_id();
  endclass

  // First extension; add field and pure virtual override.
  class ext1 extends base1;
    bit [3:0] e1_type;

    function new(int sid, int sel);
      super.new(sid);
      e1_type = sel;
    endfunction

    virtual function void print_id();
      $display("ID %0d: E1 %0d\n", id, e1_type);
    endfunction
  endclass

  // Second extension; pure virtual override.
  class ext2 extends base1;
    function new(int sid);
      super.new(sid);
    endfunction

    virtual function void print_id();
      $display("ID %0d: E2\n", id);
    endfunction
  endclass

  base1 base;
  ext1 e1 = new(0, 3);
  ext2 e2 = new(1);

  initial begin
    base = e1;
    base.print_id();
    base = e2;
    base.print_id();
  end
endprogram

```

Polymorphically use a handle of base class to access data and methods of extended class objects

```

// Output:
//   ID 0: E1 3
//   ID 1: E2

```

Example 33 - Polymorphism example showing base class handle accessing extended classes

In Example 33, two extended class objects are created and sequentially assigned to the same base class handle. The base class handle is used to access the extended class' object data through the functionality of the extended pure virtual function.

12 virtual interfaces

To understand how virtual interfaces work, a very brief explanation of interfaces is warranted.

12.1 Interfaces

An **interface** is a bundle of signals, much like a **struct**, with a few important differences.

A **struct** can be passed across a port, but all of the struct members must move in the same direction. The entire struct is declared as an **input**, **output**, or **inout**. On the other hand, it is possible to declare the interface members to move in different directions by means of the **modport** (not explained in this paper).

An **interface** can also contain **tasks**, **functions**, **assertions**, and although not very common, they can also contain other continuous assignments, **initial** procedures and **always** procedures.

Declaring an interface is a lot like declaring a class. The interface is declared with signals (the data members) and they can be declared with methods (tasks and functions). One notable difference between an interface and a class is that interfaces can have ports. Another notable difference is that when instantiated, an interface is static; that is, an interface is compiled and then elaborated statically and permanently before the simulation runs. Classes are constructed dynamically after elaboration and can be constructed and deleted throughout the simulation.

After being declared, an interface is instantiated and the instance name is really a handle to the interface that allows the user to access the interface signals hierarchically using the interface instance/handle name.

Modules that will be "connected" to the interface must declare a static interface handle of the same type in the module header as an "interface port."

This allows the connecting module to reference the "connected" instantiated interface signals hierarchically.

At the top-level where the interface is instantiated, we often say that the interface is then "connected" to other modules, when in reality, we are simply assigning (passing) the interface instance/handle to the static handles of the "connecting" modules. Once we pass the actual interface instance/handle-name to the connecting modules, the connecting modules will now have a local static handle-name that can be used to reference the actual, instantiated interface signals.

If you understand the concept of passing actual interface handles to make "connections," then it is easier to grasp how virtual interfaces work, as described in the next section.

12.2 Dynamically connecting to virtual interfaces

Since classes are dynamic (and interfaces are static), and since classes do not have ports, making it impossible to pass a static interface handle to a class port, we need a different mechanism to "connect" to an interface from a class. That is the capability that virtual interfaces provide.

Declaring an interface to be virtual is a common method used to connect a dynamic object oriented verification environment to a static design under test.

When first learning higher level SystemVerilog verification methodologies, this is not an easy concept to grasp.

Here is the problem:

- (1) You will need a real interface (an instantiated copy of the interface) to communicate with the DUT.
- (2) You will need an interface (a virtual interface or dynamic interface handle) to communicate with class transactor methods.
- (3) You need a way to tie the real interface that communicates with the DUT to the virtual interface that communicates with the class transactor methods. You will need a way to pass a handle that is connected to the real interface to the handle that connects to the virtual interface so that the class transactors can access the interface signals via the virtual interface handle (remember, we are going to tie the real interface signals directly to the virtual interface signals, so any communication with the virtual interface signals will be exactly the same as communicating with the real interface signals). This is the tricky concept to grasp. Once you understand how the handles are connected, you will understand how communication between the virtual interface and the real interface works.

12.3 Step #1 - Build the real interface

Build an interface to communicate with the pin-level or RTL-level DUT. The big difference between a pin-level and RTL level DUT is that the RTL level typically has full-bus ports, such as `data[15:0]`, whereas the pin-level design will have individual data-bus ports connected to each bit of the data bus. If the top-level module has a data bus connected to each data-bus port, the same top-level interface can typically be used.

Consider the simple example of an `alu_reg` design as shown in Figure 1 with corresponding SystemVerilog source code shown in Example 34.

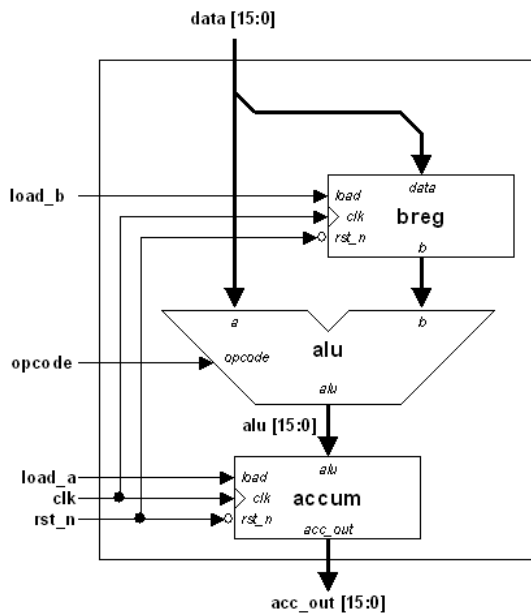


Figure 1 - alu_reg block diagram

```

module alu_reg (
    output logic [15:0] acc_out,
    input logic [15:0] data,
    input op_e          opcode,
    input logic         load_a, load_b,
    input logic         clk, rst_n);

    logic [15:0] b, alu;

    always_ff @(posedge clk)
        if (!rst_n) b <= '0;
        else if (load_b) b <= data;

    always_comb begin
        case (opcode)
            PASSA: alu = data;
            NAND : alu = ~(data & b);
            OR   : alu = data | b;
            ADD  : alu = data + b;
        endcase
    end

    always_ff @(posedge clk)
        if (!rst_n) acc_out <= '0;
        else if (load_a) acc_out <= alu;
    endmodule

```

Example 34 - alu_reg.sv SystemVerilog source code

In order for a testbench to communicate with this design, a physical interface must be connected to it and the design and interface should be enclosed within a top-level module as shown in Figure 2. This is Step #1 as listed in Section 12.3.

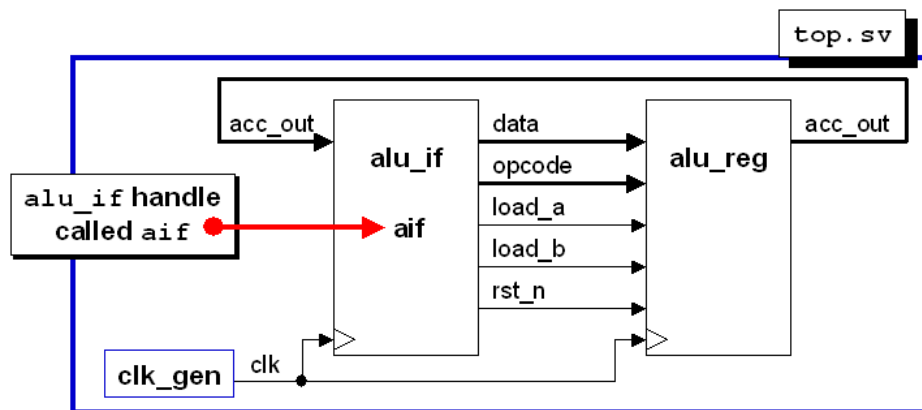


Figure 2 - Top-level module with design (DUT), interface and clock generator

The interface (block diagram shown in Figure 3) is easily coded by declaring all DUT outputs as interface inputs and all DUT inputs as interface outputs, except for the clock input, which is an input to both the DUT and interface.

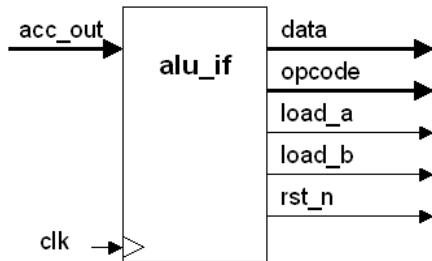


Figure 3 - alu_if block diagram

```
interface alu_if (
    input  logic [15:0] acc_out,
    output logic [15:0] data,
    output logic        op_e   opcode,
    output logic        load_a,
    output logic        load_b,
    output logic        rst_n,
    input  logic        clk);
endinterface
```

Example 35 - alu_if.sv SystemVerilog source code

The SystemVerilog interface code is shown in Example 35. All signals on the interface ports become part of the interface bundle of signals. All interface I/O signals are declared as logic variables in the interface, except for any bi-directional signals, which are declared as wires (follows good SystemVerilog data type usage guidelines).

```
`define CYCLE 10
`timescale 1ns/1ns
module clkgen (output logic clk);
    initial begin
        clk <= '0;
        forever #(`CYCLE/2) clk = ~clk;
    end
endmodule
```

Example 36 - clkgen.sv SystemVerilog source code

```
module top;
    logic [15:0] acc_out, data;
    op_e        opcode;
    logic       load_a, load_b,
    logic       rst_n, clk;

    clkgen clkgen (. *);
    alu_if aif    (. *);
    alu_reg alu_reg (. *);
endmodule
```

Example 37 - top.sv SystemVerilog source code

After declaring the interface code, a simple **clkgen** module can be coded as shown in Example 36, and the **clkgen**, **alu_if** and **alu_reg** modules are all instantiated into a top-level module as shown in Example 37. The top-level module has an instance of the DUT, an instance of the interface (this is the real interface, not a virtual interface), and the clock generator.

12.4 Step #2 - Instantiate a virtual interface into the transactor class

Instantiate a virtual copy of the same interface type into the transactor class. Until we get to Step #3, the virtual interface handle does not point to anything. We are going to make assignments to the variables declared in the virtual interface but we still need a way to tie the virtual interface handle to the real interface handle (this will happen in step #3).

The **Testbench** class in Example 38 includes the code statement **virtual alu_if vif;** which declares a handle (**vif**) to a virtual interface. This **vif** handle is used to make assign-

ments to the virtual interface variables as shown in the **drive1** and **drive2** tasks. These assignments are made with the expectation that the variables assigned will somehow be connected to the real interface.

```

class Testbench;
  Cmd c;

  virtual alu_if vif;

  function new(virtual alu_if nif, Cmd cmdin);
    vif = nif;
    c = cmdin;
  endfunction

  virtual task drive1;
    vif.data = c.a;
    vif.opcode = c.opcode;
    vif.load_a = '1;
    vif.load_b = '0;
    vif.rst_n = '1;
  endtask

  virtual task drive2;
    vif.data = c.a;
    vif.opcode = c.opcode;
    vif.load_a = '0;
    vif.load_b = '1;
    vif.rst_n = '1;
    @(negedge vif.clk);
    vif.data = c.b;
    vif.load_a = '1;
    vif.load_b = '0;
  endtask

  task stim;
    assert(c.randomize);
    c.printmsg();
    if (c.opcode==PASSA) drive1;
    else drive2;
    @(negedge vif.clk);
  endtask

  task FINISH (int cnt=1);
    repeat(cnt) @(negedge vif.clk);
    $finish;
  endtask
endclass

```

Example 38 - Testbench class SystemVerilog source code with virtual interface

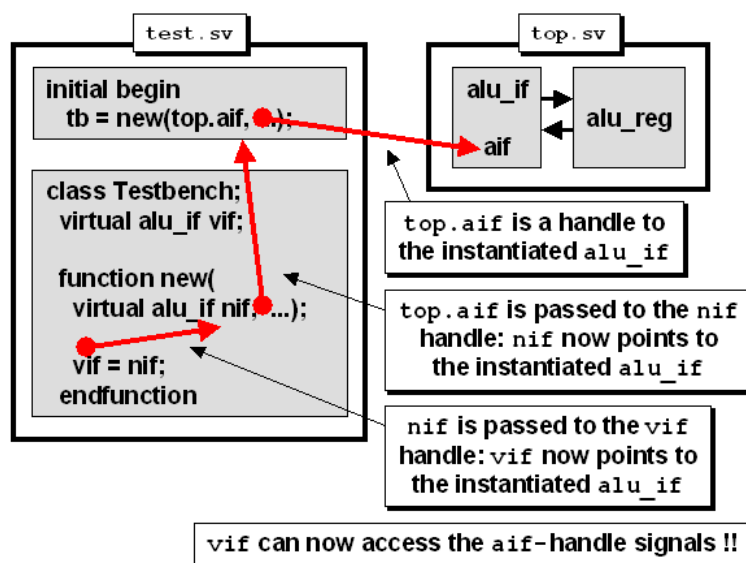
We now have a virtual interface (virtual interface handle) called **vif** that will be used to communicate with the real interface (**aif**). This is Step #2 as listed in Section 12.3.

12.5 Step #3 - Create a new() constructor to tie the real interface to the virtual interface

This is the magic step! The **Testbench** class in Example 38 includes the following snippet of code:

```
virtual alu_if vif;  
  
function new(virtual alu_if nif, Cmd cmdin);  
    vif = nif;  
    c    = cmdin;  
endfunction
```

The **new()** constructor includes an input argument of the **alu_if** interface handle type, with handle name **nif** (**new()** interface). The input interface handle will be assigned to the virtual interface handle **vif**, so the **new()** constructor input handle **nif** will be assigned to the virtual interface handle **vif**, which means that the virtual interface will point to whatever the constructor handle was pointing to as shown in Example 39.



Example 39 - Real interface handle `aif` assigned to `new()` handle `nif`, assigned to virtual interface handle `vif`

When the constructor is called to build a transactor object, the real interface handle (**aif**) will be the handle that is passed to the constructor (**nif**), which will then be assigned to the virtual interface handle (**vif**), which means that the virtual interface handle now points to the same interface signals as the real interface handle.

Now when the transactor class writes to and reads from the virtual interface handle, it is actually writing to and reading from the real interface handle, which is communicating directly with the DUT. This is step #3 - the class-based testbench is now communicating directly with the DUT.

The virtual interface is used to point to an instance of a real interface that communicates with the DUT. If a class had ports, the class itself could communicate with the DUT and a virtual interface would not be necessary.

This is the magic! This is how we get a virtual interface to point to the real interface, and now all class-based testing can be done over the virtual interface.

```

`timescale 1ns/1ns

`ifndef PKG1
`include "pkg1.sv"
`endif

class Cmd;
  rand bit  [15:0] a, b;
  rand op_e  opcode;
  rand bit   cycle;

  constraint c1 {(opcode==PASSA) -> cycle == '0;
                (opcode!=PASSA) -> cycle == '1;}

  //function void printmsg;
  virtual function void printmsg;
    $display("Cmd:  cmd=%h  opcode=%h", {a,b}, opcode);
  endfunction
endclass

class NewCmd extends Cmd;
  virtual function void printmsg;
    $display("NewCmd: data1=%h  data2=%h  opcode=%s", a, b, opcode.name());
  endfunction

  constraint c2 {opcode dist {PASSA:=10, NAND:=20, OR:=20, ADD:=50};}
endclass

program tb5;
  Testbench  tb;
  Cmd        b1=new();
  NewCmd     e1=new();

  initial begin
    tb = new(top4.aif, b1);
    repeat(5) tb.stim();
    tb = new(top4.aif, e1);
    repeat(5) tb.stim();
    tb.FINISH(2);
  end
endprogram

// Cmd:  cmd=a1d633b8  opcode=1
// Cmd:  cmd=d224a6e5  opcode=0
// Cmd:  cmd=01863daa  opcode=1
// Cmd:  cmd=41d8f441  opcode=3
// Cmd:  cmd=e5c00deb  opcode=1

```

```
// NewCmd: data1=29f9 data2=8afe opcode=ADD
// NewCmd: data1=f15a data2=b578 opcode=NAND
// NewCmd: data1=34b2 data2=a6a5 opcode=OR
// NewCmd: data1=23e4 data2=bdee opcode=PASSA
// NewCmd: data1=91c9 data2=76ff opcode=NAND
```

13 Conclusions

The virtual keyword provides two very important features to a high-level verification environment. (1) It allows constrained randomization of extended class objects that are then assigned to a base class handle (polymorphism) and all operations are then performed using virtual methods that were defined by the base class. (2) Using the virtual keyword restricts the number and types of arguments used by all extended classes, which makes the polymorphic assignments possible.

The **pure** keyword allows methods and constraints to be added to an abstract class and to impose the restriction that the functionality must be implemented to override the **pure virtual** methods and **pure** constraints in extended classes.

If an engineer always overrode the empty virtual methods and empty constraints defined in a base class, the **pure** keyword would never technically be needed, but the **pure** keyword allows the compiler to catch such omissions in a class based environment.

Although not technically part of the SystemVerilog language until 2009, there are and have been SystemVerilog simulators that have had the **pure virtual** methods implemented for a couple of years and the **pure virtual** methods are already part of the OVM verification methodology.

14 References

- [1] "IEEE Standard For SystemVerilog - Unified Hardware Design, Specification and Verification Language," IEEE Computer Society, IEEE, New York, NY, IEEE Std 1800-2005
- [2] "IEEE P1800/D9 Standard For SystemVerilog - Unified Hardware Design, Specification and Verification Language," To be published by the IEEE Computer Society, IEEE, New York, NY, IEEE
- [3] EDA.org Mantis Database for SystemVerilog P8100:
www.eda-stds.org/svdb (login: guest / password: guest)
- [4] Janick Bergeron, Eduard Cerny, Alan Hunter and Andrew Nightingale, "Verification Methodology Manual for SystemVerilog," (VMM), Springer, www.springeronline.com, 2005.
- [5] www.ovmworld.org - Version ovm-2.0.2 (June 2009)
- [6] www.vmmcentral.org - Version vmm-1.1.1 (August 2009)

15 Author & Contact Information

Cliff Cummings, President of Sunburst Design, Inc., is an independent EDA consultant and trainer with 27 years of ASIC, FPGA and system design experience and 17 years of SystemVerilog, synthesis and methodology training experience.

Mr. Cummings has presented more than 100 SystemVerilog seminars and training classes in the past six years and was the featured speaker at the world-wide SystemVerilog NOW! seminars.

Mr. Cummings has participated on every IEEE & Accellera SystemVerilog, SystemVerilog Synthesis, SystemVerilog committee, and has presented more than 40 papers on SystemVerilog & SystemVerilog related design, synthesis and verification techniques.

Mr. Cummings holds a BSEE from Brigham Young University and an MSEE from Oregon State University.

Sunburst Design, Inc. offers World Class Verilog & SystemVerilog training courses. For more information, visit the www.sunburst-design.com web site.

Email address: cliffc@sunburst-design.com

Heath Chambers, President of HMC Design Verification, Inc., is an independent EDA consultant and trainer with 13 years of ASIC and verification experience and 9 years of SystemVerilog, Verilog and methodology training experience and 5 years of synthesis training experience.

Mr. Chambers has participated on IEEE & Accellera SystemVerilog committees, and is still an active member of the IEEE SystemVerilog committees.

Mr. Chambers holds a BSCS from New Mexico Tech.

HMC Design Verification, Inc. offers verification consulting, and offers Verilog & SystemVerilog training through Sunburst Design. For more information, visit the hmcdv.iwarp.com web site.

Email address: hmcdvi@msn.com

An updated version of this paper can be downloaded from the web sites:

www.sunburst-design.com/papers

hmcdv.iwarp.com/Papers

(Last updated September 21, 2009)

16 Appendix

This appendix contains examples of extending non-virtual methods.

When extending non-virtual methods, almost anything is legal! The extended class simply overrides the base class method by name and there are no restrictions on the number, types and names of the arguments, as well as the return type.

When an extended class handle is assigned to a base class handle the base class methods are still referenced, which calls into question the value of making these extended-class to base-class handle assignments.

It is obviously legal to override a base class method with an extended class method that has the same number, types and names of the arguments, as well as the same return type, as shown in Example 40.

```
typedef bit    [7:0] arg1_t;
typedef logic [7:0] arg2_t;

program example40;
  class base1;
    function bit showit (arg1_t i1);
      $display ("\n\n %m: i1=%0d \n\n", i1);
    endfunction
  endclass

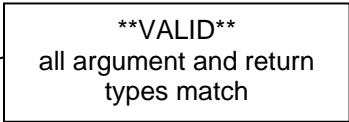
  class ext1 extends base1;
    function bit showit (arg1_t i1);
      $display ("\n\n EXT: %m: i1=%0d \n\n", i1);
      return(1);
    endfunction
  endclass

  base1 b1 = new();
  ext1 e1 = new();

  bit    status;
  arg1_t dummy;

  initial begin
    status=b1.showit(25);
    status=e1.showit(33);
  end
endprogram
```

****VALID****
all argument and return
types match



Example 40 - Non-virtual method overrides another method with matching argument and return types

It is legal to override a base class method with an extended class method that has the same number and names of the arguments, as well as the same return type, but different input argument types (**arg2_t** instead of **arg1_t**) as shown in Example 41

```
typedef bit    [7:0] arg1_t;
typedef logic [7:0] arg2_t;

program example41;
  class base1;
    function bit showit (arg1_t i1);
      $display ("\n\n %m: i1=%0d \n\n", i1);
    endfunction
  endclass

  class ext1 extends base1;
    function bit showit (arg2_t i1);
      $display ("\n\n EXT: %m: i1=%0d \n\n", i1);
      return(0);
    endfunction
  endclass

  base1 b1 = new();
  ext1 e1 = new();

  bit    status;
  arg1_t dummy;

  initial begin
    status=b1.showit(25);
    status=e1.showit(33);
  end
endprogram
```

****VALID****
argument types DO NOT
match

Example 41 - Valid - Non-virtual method overrides another method with non-matching argument type

It is legal to override a base class method with an extended class method that has the same number, types and names of the arguments, as well as the same return type, but a different argument direction (**output arg1_t** instead of default **input arg1_t**) as shown in Example 42.

```
typedef bit    [7:0] arg1_t;
typedef logic [7:0] arg2_t;

program example42;
  class base1;
    function bit showit (arg1_t i1);
      $display ("\n\n %m: i1=%0d \n\n", i1);
    endfunction
  endclass

  class ext1 extends base1;
    function bit showit (output arg1_t i1);
      $display ("\n\n EXT: %m: i1=%0d \n\n", i1);
      return(0);
    endfunction
  endclass

  base1 b1 = new();
  ext1 e1 = new();

  bit    status;
  arg1_t dummy;

  initial begin
    status=b1.showit(25);
    status=e1.showit(dummy);
  end
endprogram
```

****VALID****
argument direction DOES
NOT match

Example 42 - Valid - Non-virtual method overrides another method with non-matching argument direction

It is legal to override a base class method with an extended class method that has the same number, types and names of the arguments, as well as the same return type, (one **arg1_t** input instead of two **arg1_t** inputs) as shown in Example 43.

```
typedef bit    [7:0] arg1_t;
typedef logic [7:0] arg2_t;

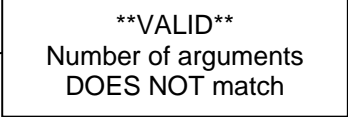
program example43
  class base1;
    function bit showit (arg1_t i1);
      $display ("\n\n %m: i1=%0d \n\n", i1);
    endfunction
  endclass

  class ext1 extends base1;
    function bit showit (arg1_t i1, i2=55);
      $display ("\n\n EXT: %m: i1=%0d i2=%0d \n\n", i1, i2);
      return(0);
    endfunction
  endclass

  base1 b1 = new();
  ext1 e1 = new();

  bit    status;
  arg1_t dummy;

  initial begin
    status=b1.showit(25);
    status=e1.showit(33);
  end
endprogram
```



Example 43 - Valid - Non-virtual method overrides another method with different number of arguments

It is legal to override a base class method with an extended class method that has the same number, types and names of the arguments, but a different return type (**logic** instead of default **bit**) as shown in Example 44.

```
typedef bit    [7:0] arg1_t;
typedef logic [7:0] arg2_t;

program example44;
  class base1;
    function bit showit (arg1_t i1);
      $display ("\n\n %m: i1=%0d \n\n", i1);
    endfunction
  endclass


  class ext1 extends base1;
    function logic showit (arg1_t i1);
      $display ("\n\n EXT: %m: i1=%0d \n\n", i1);
      return(0);
    endfunction
  endclass

  base1 b1 = new();
  ext1 e1 = new();

  bit    status;
  arg1_t dummy;

  initial begin
    status=b1.showit(25);
    status=e1.showit(33);
  end
endprogram
```

****VALID****
Return type DOES NOT
match



Example 44 - Valid - Non-virtual method overrides another method with different return type