The Problems with Lack of Multiple Inheritance in SystemVerilog and a Solution

David Rich
Dave_rich@mentor.com

Abstract – The concept of multiple class inheritance is a feature that many Object-Oriented Programming (OOP) languages have where a subclass can inherit behaviors (i.e. class methods and properties) from more than one superclass. As the language is currently defined, a SystemVerilog subclass (child class) can only extend from a single superclass (parent class). This paper presents some of the problems of having an OOP language lacking multiple inheritance, and then suggests a solution.

With only single class inheritance, class definitions tend to become bloated. That is, behaviors that should have been treated as separate concerns in individual superclasses are written into a single class. It becomes difficult to control access to these individual behaviors from within a single class because there is no way to distinguish a group of methods belonging to a single concern. The Adaptor/Wrapper design pattern is one solution that has been applied in many existing environments. However this pattern adds complexity and a performance overhead by creating additional levels of class reference indirection. The TLM class library from the OVM is one example of using this pattern that demonstrates the problem. Another example of the problem is observed when creating an environment that allows interoperability between the OVM and VMM base class libraries. Single class inheritance makes it much more difficult to create an interoperable class library without merging classes together.

Simply adding C++ style multiple inheritance to SystemVerilog may solve most of these problems, but creates many new problems. Some of these new problems include how to combine multiple constructors and dealing with name lookup resolution when the same identifiers appear in multiple superclasses (commonly referred to as the diamond problem). Other OOP languages have solved these problems by simplifying the behaviors allowed when inheriting from multiple super-classes. This paper will compare and contrast solutions from languages such as Java, C++ and PHP.

Finally, this paper recommends a solution for SystemVerilog by introducing the concept of a class interface, similar in nature to a virtual class with pure virtual methods. Multiple superclass interfaces could be extended as the base of a single subclass. Examples are presented to show the potential for reduced code complexity and improved performance.

1 INTRODUCTION

1.1 THE ROLE OF INHERITANCE

The role of inheritance in any Object-Oriented-Programming (OOP) language is simple: reuse code that is already written. The basic premise behind inheritance is that you start with one class, and then you create a new class based on the original class. That new class inherits all the data members and methods from the original *base* class.

Example 1

```
class A;
  int i1,i2;
  function void f1; endfunction
  function void f2; endfunction
endclass
class B extends A;
    int i3;
    function void f3; endfunction
endclass
```

The user of class `B` sees three members – `i1`, `i2`, `i3`, and three methods, `f1`, `f2`, `f3`. If not for other features, the user does not need to know that class `B` was derived from class `A`. Class `B` has all the members and methods of class `A` as if they were written there in the first place, without having to cut and paste the text from `A` as shown in Example 2.

Example 2

```
class B; // user's view of class B
    int i1,i2;
    int i3;
    function void f1; endfunction
    function void f2; endfunction
    function void f3; endfunction
endclass
```

The true power of inheritance is realized when you add the concept of dynamic polymorphism. This is when the user writes code thinking that they have an object of class `A`, but they actually have an object of class `B`.

Example 3

```
class A;
  int i1,i2;
    virtual function void print;
      $display(i1,i2);
    endfunction
endclass
class B extends A;
    int i3;
    function void print;
      super.print;  // displays i1,i2
      $display(i3);
    endfunction
endclass
function void f(A a_h);
   a_h.print();
endfunction
```

The function `f()` in Example 3 is written just knowing that it has been passed a handle to a class A object. The object that is passed into `a_h` could be of type class `A` or `B`. Because the `print()` method is virtual, the call to `a_h.print()` will dynamically chose either `A::print()` or `B::print()` based on the object that currently sits in `a_h`.

There are many other features associated with inheritance, but the concept of dynamic polymorphism is the key feature used as the basis in building most class libraries.

1.2 THE CLASS INTERFACE

The list of accessible class members and methods form the interface[1] to a class. The prototypes for the methods are typically the only items documented to the user as the API to the class. The user is not concerned with the details of the implementations.

In the case of an abstract class, things are reversed. An abstract class defines prototypes for pure virtual methods, and it is a user requirement to provide the implementation. In either case, this interface becomes a contract for communication.

[1] In this paper, an interface is strictly used as a term from the software engineering domain, not the SystemVerilog **interface** keyword and semantics.

1.3 Single versus Multiple Inheritance

The examples up to this point show the extension of a single class to form a new class. SystemVerilog does not currently provide a mechanism to inherit from more than one class as many other languages do. [1]

The ability to inherit from multiple classes seems useful when dealing with very separate concerns such as a reporting mechanism and a hierarchical component mechanism. You just create a list of classes you want to extend from, and your new class has the combined functionality of all those classes.

Example 4

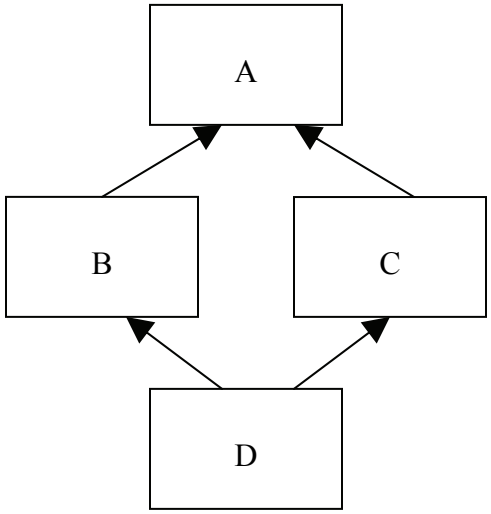
```
class reporter;
  virtual function void report
                      (string message);
    $display("report at %t: %s",
                     $time, message);
    endfunction
endclass
class hierarchy;
local hierarchy m_parent, m_children[$];
  function new(hierarchy parent);
    if ((m_parent = parent) != null)
     m_parent.m_children.push_back(this);
  endfunction
  pure virtual task run;
endclass
class my_object extends
    reporter, hierarchy; // hypothetical
    function new…();… endfunction
    task run;
      report("I'm running");
    endtask
endclass
```

This hypothetical syntax shown in the definition of `my_object` in Example 4 is allowed in languages like C++, but it was specifically excluded in SystemVerilog. Although very convenient and powerful, multiple inheritance presents many additional problems that took awhile for C++ to address.

1.4 Problems with Multiple Inheritance

One of the classic problems with multiple inheritance is the "diamond problem" that occurs with name resolution. The name comes from the diamond shape in the inheritance diagram that represents the problem

Figure 1



The code for the above diagram is shown in Example 5. Class A has a member or method M. Class B and C both override M. Class D makes a reference to M, which is ambiguous; is it B::M or C::M? There is another problem with casting and virtual method lookup. Suppose M is a virtual method. If an instance of class D is cast to a class A handle, which method gets called?

Example 5 (C++)

```
class A {
   public:
    virtual void M(int i) {
       m_i = i;
    }
    int m_i;
};
class B : public A {};
class C : public A {};
class D : public B, public C {
  public:
   virtual F(int i) {
      M(i);
  }
};
```

The call to M(i) in Example 5 is ambiguous because there are two possible paths, as shown in the diagram, to get to class A from D.

2 INHERITANCE IN SYSTEMVERILOG

The SystemVerilog class model is derived from the *Vera* language, which had adopted a class inheritance model similar to Java[2]. At the time of Vera's creation, C++ had not yet solved all the issues surrounding multiple inheritance. Java made the design decision to have only single inheritance, and has another mechanism, an interface, to address some of the applications that multiple inheritance would have addressed. However, this concept of an interface was never incorporated into Vera.
Without this concept, class libraries, such as the OVM and VMM, have dealt with the lack of multiple inheritance in a variety of ways as will be shown in the sections that follow.

2.1 EXTENDED CLASS TREE

With only single inheritance, all class inheritance diagrams become simple trees. The brute force way to compose any additional functionality from one class into a class tree is either to insert that class into the baseline (Figure 2), or extend every branch of the tree (Figure 3).

Figure 2

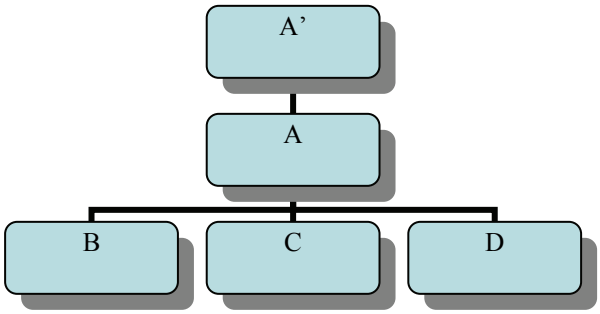
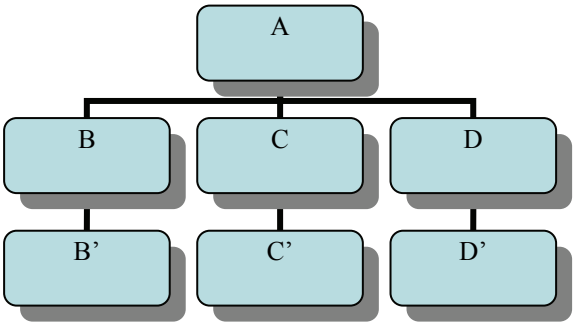


Figure 3



Looking at the reporter and hierarchy classes from Example 4, if we try to get the same functionality using single inheritance, one has to make an ordering decision: should all reporter classes have hierarchy, or should all hierarchy classes have reporters? Either way, some usages of these classes are bloated with additional functionality that results in unnecessary overhead.

2.2 ADAPTOR OBJECT PATTERN

A common software problem is how to make two unrelated classes communicate with each other. The easiest way to make this happen is for one class to call a method by referencing another class. However, this reference creates a dependency, which makes the class less reusable.

Example 6

```
typedef class B;
class A;
  B b_h;
    task A_do();
      repeat (100)
         b_h.B_put(Barg1, Barg2);
    endtask
  endclass
class B;
    task B_put(int Barg1, Barg2);
    endtask
endclass
class Top;
    A a_h; B b_h;
    function new;
      a_h = new();
      b_h = new();
      // make connection
      a_h.b_h = b_h;
    endfunction
endclass
```

Class A, the client, must have a handle to class B, the target, in order to call its method `B_put()`, as well as know the interface for all the arguments. If later we want to change type of class B to class C, class A must be modified for the new class.

The adaptor design pattern breaks this dependency by isolating these classes by defining an intermediary adaptor class.[3] This adaptor class preserves the original target interface that the client expects to see and maps it to the new target.

Example 7

```
virtual class B; // the interface for B
    pure virtual task B_put(int Barg);
    pure virtual task B_get(int Barg);
endclass
class A;
  B b_h;
    task A_do();
      repeat (100) begin
         b_h.B_put(Barg);
         b_h.B_get(Barg);
    endtask
endclass
class C;
    task C_put(real Carg);
    // assume this does something
    endtask
    task C_get(real Carg);
    // assume this does something
    endtask
endclass
class BCadapter extends B;
    C imp;
    task B_put(int Barg1);
      imp.C_put(Barg1*123);
    endtask
    task B_get(int Barg1);
      imp.C_get(Barg1*123);
    endtask
endclass
class Top;
    A a_h; C c_h;
    BCadaptor bc_h;
    function new;
      a_h = new();
      c_h = new();
      bc_h =new();
      // make connections
      a_h.b_h = bc_h;
      bc_h.imp = c_h;
    endfunction
endclass
```

This adaptor class adds one extra method call, and one extra class reference between the client and target. The TLM class library is an example of a set of adaptor classes that have been standardized. Any client or target class can now communicate using a TLM interface as an adaptor class.

The TLM standard library comes with many interfaces. The B class above is essentially the TLM blocking_slave interface modeled as an abstract class. That interface is made by combining a put and get TLM interface. But the combining has to be done by cut-and-paste because SystemVerilog does not allow inheriting from multiple classes.

3 LESSENS FROM OTHER LANGUAGES

Other languages have picked similar solutions to deal with these problems. Java, PHP and C# do not allow multiple inheritance, but have created a special interface construct which is essentially a limited form of an abstract class. This gets rid of the re-converging diamond by restricting multiple inheritance from simple base classes with only pure virtual methods. A Java version[4] of Example 7 is show below

Example 8 (Java)

```
public interface B {
    void B_put(int Barg);
    void B_get(int Barg);
}
class C {
    public void C_put(real Carg)
    {
    // assume this does something
    }
    public void C_get(real Carg)
    {
    // assume this does something
    }
}
class BCadaptor extends C implements B {
    public void B_put(int Barg) {
        C_put(Barg*123);
    }
    public void B_get(int Barg) {
        C_get(Barg*123);
    }
}
```

The key difference in this example is there is no extra level of indirection using an `imp` instances.

Other languages like C++ have resolved ambiguities by requiring explicit class scope references. In the original example explaining the diamond problem, a reference to the bare M would be illegal, and either `B::M` or `C::M` would be required. Perl and Python have simply used the listed order of extension to resolve the ambiguity.

C++ also distinguishes between *independent* inheritance and *virtual* inheritance. Virtual inheritance removes the ambiguity by sharing instead of creating independent copies of members. [5]

If you look into the history of C++ development, you will see that *independent* inheritance was the initial functionality proposed before ANSI standardization, and *virtual* inheritance came out of the resolution of the issues created by that initial functionality.

Independent inheritance is shown in Example 5. Two copies of class A's method and properties are created for each extension of classes B and C. Virtual inheritance is shown in Example 9 (C++).

Example 9 (C++)

```
class A {
   public:
    virtual void M(int i) {
       m_i = i;
    }
    int m_i;
};
class B : public virtual A {};
class C : virtual public A {};
class D : public B, public C {
  public:
   virtual F(int i) {
      M(i);
  }
};
```

Only one shared copy of class A is created, thus removing any ambiguity to the reference to M(i);

4 A PROPOSED SOLUTION

The majority of situations that have been hampered by the lack of multiple inheritance could be resolved by the Java interface concept. Unfortunately, the term `interface` is already in heavy use by a slightly different concept in SystemVerilog. If you look at the virtual inheritance of C++, you can achieve the same effect by limiting multiple inheritance to only virtual base classes that contain only pure virtual methods. A class interface is indicated by prefixing the keyword `virtual` in front of one or more base class extensions.

Example 10

```
// the interface for puts
virtual class tlm_put_if;
    pure virtual task put(int arg);
endclass
// the interface for gets
virtual class tlm_get_if;
    pure virtual task get(int arg);
endclass
virtual class tlm_slave_if extends
  virtual tlm_put_if, virtual tlm_get_if;
endclass
class A;
  tlm_slave_if slave_h;
    task A_do();
      repeat (100) begin
         slave_h.put(arg);
         slave_h.get(arg);
    endtask
endclass
class C;
    task put(real Carg);
    // assume this does something
    endtask
    task get(real Carg);
    // assume this does something
    endtask
  endclass
  class tlm_slave_imp extends
        virtual tlm_slave_if,
        C;  // extends non-virtual C
    task put(int Barg1);
      C::put(Barg1*123);
    endtask
    task get(int Barg1);
      C::get(Barg1*123);
    endtask
  endclass
  class Top;
    A a_h; C c_h;
    tlm_slave_imp slave_h;
    function new;
      a_h = new();
      c_h = new();
      slave_h =new();
      // make connections
      a_h.b_h = slave_h;
    endfunction
  endclass
```

Notice that the class `tlm_slave_imp` in Example 10 is declared in the same manner as any other abstract class – there no new syntax in the declaration of an class interface. Only when extending an abstract class as virtual does the semantic restriction that it contain only pure virtual methods come into play.

5 CONCLUSION

The introduction of multiple interface inheritance can greatly streamline the coding effort required to develop base class libraries, as well as produce more efficient implementations.

Although not presented in this paper, this proposed solution leaves the door open for future multiple inheritance extensions more inline with the current C++ standard.

6 REFERENCES

[1] "IEEE Standard for SystemVerilog – Unified Hardware Design, Specification, and Verification Language," IEEE Std 1800-2009

[2] D. Rich "The evolution of SystemVerilog" IEEE Design and Test of Computers, July/August 2003

[3] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns, Elements of Reusable Object-Oriented Software. Addison-Wesley Publishing Company, Reading Massachusetts, 1995

[4] "Interfaces." Developer Resources for Java Technology. Web. 10 Jan. 2010. <http://java.sun.com/docs/books/jls/third_edition/html/interfaces.html#9.5>.

[5] B. Stroustrup (1999). Multiple Inheritance for C++. Proceedings of the Spring 1987 European Unix Users Group Conference.