Mantis 1356

P1800-2012

Motivation

This Mantis item enables the use of Java style interfaces in the place of true multiple inheritance (MI) as implemented in C++. Please see Dave Rich's paper titled "The Problems with Lack of Multiple Inheritance in SystemVerilog and a Solution" for a good history and need for interfaces. We have chosen the Java approach, with some subtle variations needed for SVTB, because of the integration complexity associated with a full MI solution and because we see the Java solution as meeting the needs of an MI approach for SytemVerilog.   The restrictions that we have chosen basically limit interface classes to classes with pure virtual methods. WRT the diamond name resolution issue highlighted in Dave's paper, we choose to "hide", or in other words not inherit, parameters and other name scoped tokens of the interface class. These types can still be accessed with the class scope operator '::', they are just not inherited. We choose to introduce the keyword 'interface' and the concept of 'interface classes" rather than Dave's suggested 'virtual <classname>' as this best represents the intent of this new functionality. We do not believe this will conflict with SV interfaces or overuse that keyword as this new functionality will be introduced and discussed in the context of being an 'interface class'.

(NOTE: BNF will be written once the details of this spec are complete)

Add sub-clause 8.25 (and increment 8.25→8.27 by one number) to the Classes Clause as follows

8.25 Interface Classes

SystemVerilog introduces a type of class called an *interface class*. This is not to be confused with the interface construct. An *interface class* can be thought of as a prototype or skeleton of a class where the methods within it are outlined, but not defined thereby setting up a framework of how the class should be implemented. An interface class shall contain only methods of pure virtual type (see 8.20). Other unrelated classes can implement the interface class through the *implements* keyword but must fully define the methods with the exception of virtual classes. A virtual class that is implementing an interface class is not required to fully define the methods, however they must be fully defined by descendent classes.

A class may be declared to directly *implement* one or more interface classes, meaning that any instance of the class implements all the pure virtual methods specified by one or more interface classes. A class must implement all of the pure virtual methods that are prototyped by these interface classes. This (multiple) interface inheritance allows classes to support (multiple) common behaviors without sharing any implementation.

An interface class makes it unnecessary for related classes to share a common abstract superclass or for that superclass to contain all method definitions needed by child classes. An interface may be declared to be an *extension* of one or more other interfaces, meaning that it implicitly specifies all the member types, pure virtual methods and constants of the interfaces it extends, except for any member types and constants that it may hide.

A variable whose declared type is an interface class type may have as its value a reference to any instance of a class which implements the specified interface (see 8.21 Polymorphism). It is not sufficient that the class happens to implement all the pure virtual methods of the interface; the class or one of its superclasses must actually be declared to implement the interface through the *implement* keyword, or else the class is not considered to implement the interface.

The following is a simple example of interface classes.

```
interface class PutImp#(type T = logic);
  pure virtual task void put(T a);
endclass

interface class GetImp#(type T = logic);
  pure virtual task T get();
endclass

class Fifo#(type T = logic, DEPTH = 1) implements PutImp#(T), GetImp#(T);
   T [DEPTH-1:0] myFifo;
   virtual task void put(T, a);
      // Put implementation
   virtual task T get();
      // Get implementation
endclass
```

The example has two interface classes, `PutImp` and `GetImp`, which contain prototype pure virtual methods `put` and `get`. The Fifo class then uses the keyword *implements* to consume the prototypes of `put` and `get` and then fully define them.

8.25.1 Extension versus Implements

There is a difference between how classes, virtual classes, and interface classes inherit each other. The following highlights these differences:

- A interface class can extend one or more interface classes
- A virtual class can implement one or more interface classes
- A class can implement one or more interface classes
- A class or virtual class may extend one class and/or implement one or more interface classes

Conceptually an extension is considered a means to extend the content of the parent class while an *implements* is considered a contract on behalf of the implementing class to supply the definition of the interface class. Whenever the keyword *implements* is used the implementing class shall supply a definition or an error will be issued. An interface class may be extended to a interface class, meaning that the sub interface class can have additional methods outlined but may not define any of them. A virtual class can extend one class and/or implement one or more interface classes. Because virtual classes are abstract they may or may not choose to fully define the methods from their parent class. Therefore virtual classes may provide a means to create partial implementations of classes (See 8.25.5 Partial implementations). A class can only implement interface classes. It shall be an error to extend a class from an interface class.

The following example shows the case where a class is both extending a base class and implementing two interface classes:

```
class MyQueue(type T = logic, DEPTH = 1);
   T [DEPTH-1:0] PipeQueue[$];
   virtual function void deleteQ();
       // Delete implementation
   endfunction
endclass

class Fifo#(type T = logic, DEPTH = 1) extends MyQueue#(T, DEPTH), implements
PutImp#(T), GetImp#(T);
   virtual task void put(T, a);
      // Put implementation
   virtual task T get();
      // Get implementation
endclass
```

In this example, the `Fifo` child class is extending the `MyQueue` base class which has a parameterized queue and an associated `delete()` method. This property and method are inherited in the `Fifo` class. In addition the `Fifo` class is also implementing the `PutImp` and `GetImp` interface classes and defining the `put` and `get` methods respectively.

8.25.2 Type Access

Parameters, constants, typedefs, nested classes (see 8.22), and cover groups within a interface class are not inherited into the scope of descendent classes. All parameters, constants, nested classes and typedefs within a interface class are implicitly static and can be accessed through the class scope resolution operator :: (see 8.22)

For example: (Note to the committee – we obviously need to prune down the number of examples. They are here for clarifications only for now. Let's decide which to keep and which to remove - Tom).

```
interface class fooIntf;
  typedef enum {ONE, TWO, THREE} t1_t;
  interface virtual function t1_t fooFunc();
endclass : fooIntf

class fooClass implements fooIntf;
  t1_t t1_i; // error,  t1_t is not inherited from fooIntf
  virtual function fooIntf::t1_t fooFunc(); // correct.  The scoping operator ::
                                            // is used to access type t1_t
    return (fooIntf::ONE);
  endfunction : fooFunc
endclass : fooClass

interface class interfaceClassA #(type T1 = logic)
  typedef T1[1:0] T2;
  pure virtual function T2 foo();
endclass : interfaceClass

interface class interfaceClassB #(type T = int) extends interfaceClassA #(T);
  …
endclass : interfaceClassB

interface class interfaceClassC #(type T = int) extends interfaceClassA #(T);
  pure virtual function T1 bar(); // illegal, type T1 is not inherited into the
scope of interfaceClassC
endclass : interfaceClassC


class derivedClass1 implements interfaceClassB #(bit);
  virtual function T2 foo(); // illegal, type T2 is not inherited into the scope
of derivedClass0
    // implement foo
endclass : derivedClass1

class derivedClass1 implements interfaceClassB #(bit);
  virtual function T1[1:0] foo(); // illegal, type T1 is not inherited into the
scope of derivedClass0
    // implement foo
endclass : derivedClass1

class derivedClass2 implements interfaceClassB #(bit);
```

```
  virtual function interfaceClassA::T2 foo(); // illegal, the return type is
logic[1:0]; the inherited prototype return type is bit[1:0]
    // implement foo
endclass : derivedClass2

class derivedClass3 implements interfaceClassB #(bit);
  virtual function interfaceClassA#(bit)::T2 foo();  // legal, proper
parameterization makes the types agree
    // implement foo
endclass : derivedClass3

class derivedClass4 implements interfaceClassB #(bit);
  virtual function interfaceClassB#(bit)::T2 foo();  // illegal, interfaceClassB
did not inherit member T2 from interfaceClassA
    // implement foo
endclass : derivedClass4

class derivedClass5 implements interfaceClassB #(bit);
  virtual function bit[1:0] foo(); // legal, the return type bit[1:0] agrees with
the inherited prototype
    // implement bar
endclass : derivedClass5
```

8.25.3 Casting and Object reference assignment

There are a handful of relationships that must be clearly defined in order for interface classes to work properly with SystemVerilog. In order to maintain the OOP and polymorphism semantics, it shall be legal to assign an interface class handle to a child object that implements it.

```
    PutImp #() put_ref;
    Fifo#() fifo_obj = new;
    put_ref = fifo_obj;
```

It is also legal to cast implemented objects onto their prototype interface class handles

```
    $cast(fifo_obj, put_ref); // This is legal
    $cast(put_ref, fifo_obj); // Legal, but casting is not required
```

Like abstract classes, a variable of an interface class type cannot be instantiated.

```
    put_ref = new(); // This is illlegal
```

8.25.4 Name Scoping Conflicts and Resolution

With interface classes, there are many scenarios that can cause name collisions so it’s important to properly define their resolution. The resolution of name collisions within an interface class will either produce an error or resolve to the same method. If the interface class method prototypes are different then an error shall be issued. If the interface class method prototypes are identical then the collision resolves to one of the identical methods.

Let’s take the following examples of resolution:

```
interface class interfaceBase;
  pure virtual function bit foo();
endclass

interface class interfaceExt extends interfaceBase;
  virtual function bit bar();
endclass
```

```
virtual class A implements interfaceBase;
  virtual function bit foo();
endclass

class derivedClass extends A implements interfaceExt;
  virtual function function bit foo();
    return (0);
  endfunction
  virtual function function bit bar();
    return (0);
  endfunction
endclass
```

In the above example, `interfaceBase` has a method named `foo` which is not implemented. `interfaceExt` is an extension of `interfaceBase` and adds `bar` as another unimplemented method. Class `A` then implements `interfaceBase` but is virtual so no implementation is required. Finally `derivedClass` extends `B` and implements `interfaceExt` forcing it to provide full definitions to both `foo` and `bar`. But notice the name collisions seen by `derivedClass`. It sees foo from the `A` class extension and the `foo` from the implements of `interfaceExt`. Because the prototype of the `foo` method is identical through both the extend and implement paths, `derivedClass` will only need to create the full definition of the prototyped method named `foo`.

The same example now shows how a collision can occur that results in an error:

```
interface class interfaceBaseA;
  pure virtual function bit foo();
endclass

interface class interfaceBaseB;
  virtual function int foo();
endclass

class A implements interfaceBaseA, interfaceBaseB;
  virtual function function bit foo();
    return (0);
  endfunction
endclass
```

In this case, `foo` is defined in both `interfaceBaseA` and `interfaceBaseB` both are of a different types, bit and int respectively. Because they are not identical prototypes, an error will be issued. The same will occur if a prototype with the same name is defined twice with difference parameter values:

```
interface class PutImp#(type T = logic, WIDTH = 1);
  pure virtual task void put(T [WIDTH-1:0] a);
endclass

interface class PutGetImp#(type T = logic);
  pure virtual task void put(T a);
  pure virtual task T get();
endclass

class FiFo#(type T = logic, WIDTH = 1, DEPTH = 1) implements PutImp(T, WIDTH),
PutGetImp(T);
  T [DEPTH-1:0] myFifo;
  virtual task void put(T a);
    return (0);
  endfunction
  virtual task T get();
    return (0);
  endfunction
endclass
```

This example has the `PutImp` class with 2 parameters, `T` and `WIDTH`, feeding the `put` method within it. The `PutGetImp` class also has a definition of `put` but only has one parameter, `T`. When the `Fifo` class implements both of these interface classes it will recognize the parameter difference between the two versions of `put` and will issue an error.

It is possible to have a method definition and a conflicting pure method definition that collide in a derived class but which are merged because the contract of the pure method is met in the fully defined method. Here is an example:

```
class baseClass;
  virtual function bit foo();
    return (1);
  endfunction
endclass

interface class interfaceClass;
  pure virtual function bit foo();
  pure virtual function bit bar();
endclass

class derivedClass extends baseClass implements interfaceClass;
  // the “contract” to implement foo is fulfilled
  // by the inheritance from baseClass, even though baseClass
  // never declared that it implements interfaceClass
  // and in this example, it can’t claim to implement interfaceClass because it
  // doesn’t provide a definition for bar()
  virtual function function bit bar();
    return (0);
  endfunction
endclass
```

In this example `baseClass` fully defines `foo`, but `foo` is also prototyped in `interfaceClass`. `derivedClass` then extends `baseClass` and implements `interfaceClass` but because the prototype of `foo` in `interfaceClass` matches the prototype of the full definition of `foo` in `baseClass`, the name collision resolves without an error and `derivedClass` uses `foo` defined in `baseClass`.

8.25.5 Partial implementation

It is possible to create classes that are not fully defined and which take advantage of interface classes through the use of virtual classes (see 8.20 Abstract Classes and pure virtual methods). Because virtual classes do not have to fully define their implementation, they are free to partially define their methods. The following is an example of a partially implemented virtual class.

```
class baseClass;
  virtual function bit foo();
    return (1);
  endfunction
endclass

interface class interfaceClass;
  pure virtual function bit bar();
endclass

virtual class derivedClass extends baseClass implements interfaceClass;
endclass
```

In this case `derivedClass` is virtual. It both extends `baseClass` which contains a fully defined method `foo` and implements `interfaceClass` which has only a prototype of `bar`.

8.25.6 Method default argument values

Method declarations within Interface Classes may have default argument values. The default expression is evaluated in the scope containing the subroutine declaration each time a call using the default is made. See section 13.5.3 (Default argument values) for more information.

8.25.6 Constraint Blocks

Constraint blocks are allowed within an interface class but shall be empty. The implementing class shall implement the constraint. This will allow references to and use of constraint block names in the polymorphism usage models of the interface class.