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 SystemVerilog. 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 interface classes'.

(NOTE: There were several mantis issues opened up to deal with resolving other aspects of the LRM that need attention before we can bottom out on Interface Class refinement. In most of these cases, this will lead to a restriction to specific Interface Class features until we can resolve them. Where pertinent, I have noted the issue, the restriction, and the open mantis ticket).

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

8.1 General

This clause describes the following: — Class definitions — Virtual classes and methods — Polymorphism — Parameterized classes — Interface Classes

8.3 Syntax

class_declaration ::= // from A.1.2
[virtual] class [lifetime] class_identifier [parameter_port_list]
[extends class_type [(list_of_arguments)]]
[implements class_type [(list_of_arguments) {, class_type [(list_of_arguments) }]]
[class_item };
endclass [: class_identifier]

Comment [t1]: I really dug through the class BNF but in the end really only saw this as the needed addition. Good exercise for all reviewers to double check this too.

Add sub-clause 8.25 (and increment 8.25 \rightarrow 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, string MyPutSignal = "a");
    pure virtual task void put(T MyPutSignal);
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 Interface Class Syntax

Comment [t2]: This BNF is all new. I only defined the strings that did not already exist in the LRM. I walked through all the downstream definitions and pulled things out like extern, constraints, etc.. Please walk through this with a fined tooth comb.

| local_parameter_declaration ; | parameter_declaration 7; | ; class_property ::= { property qualifier } data_declaration | const { class_item_qualifier } data_type const_identifier [= constant_expression] ; interface_class_method ::= { method_qualifier } method_prototype | { method_qualifier } class_constructor_prototype

8.25.2 Extension versus Implements

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

- An interface class
 - may extend zero or more interface classes;
 - may not implement an interface class
 - o may not extend a non-interface class.
- A class or virtual class
 - \circ may extend at most one other class or virtual class;
 - $\circ \quad \text{may implement zero or more interface classes;} \\$
 - may not extend an interface class.
 - o may both extend a class and implement 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 an 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:

In this example, the Fifo child class is extending the MyQueue base class which has a parameterized queue and an associated deleteQ() 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.

Comment [t3]: Are we allowing constants to have a default value in Interface Classes?

In the following example we demonstrate that multiple types can be can be parameterized in the class definition and the resolved types used in the implemented classes PutImp and GetImp.

8.25.3 Type Access

Parameters, constants and typedefs within a interface class are not inherited into the scope of descendent classes. All parameters, constants, nested classes and typedefs within an interface class are implicitly static and can be accessed through the class scope resolution operator :: (see 8.22). Nested classes (see 8.22) shall not be allowed in an interface class. An interface class additionally cannot nest within another interface class or within a class.

Some examples:

```
interface class fooIntf;
  typedef enum {ONE, TWO, THREE} t1 t;
 pure virtual function t1_t fooFunc();
endclass : fooIntf
class fooClass implements fooIntf;
  t1_t t1_i; // error, t1_t is not inherited from fooIntf
 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 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
```

Comment [t4]: But the next sentence does not allow them at all in interface classes. Not sure how this got in here, but I will remove it in the final draft.

Comment [t5]: Issue #3 resolution for 1/18/2011 mantis note.

```
// implement foo
endclass : derivedClass3
class derivedClass4 implements interfaceClassB #(bit);
virtual function bit[1:0] foo(); // legal, the return type bit[1:0] agrees with
the inherited prototype
// implement bar
endclass : derivedClass4
```

8.25.3.1 Type Usage Restrictions

There is a restriction placed on what type is passed along to the implementation of an Interface Class. An interface class type shall not be used in the parameterization of classes that implement said interface class. The implemented class type shall be known at the point of reference to be an interface class. A class shall not implement a parameter, even if it resolves to an interface class. The following examples demonstrate this restriction and are illegal:

```
class Fifo #(type T = PutImp) implements T;
virtual class Fifo #(type T = PutImp) implements T
interface class Fifo #(type T = PutImp) implements T;
```

Forward Typedef of interface classes shall not be allowed in this revision of the LRM, simply because of the complexities associated with its implementation. This means that all interface class definitions must be declared before the implementation of the interface class is reached.

typedef class interfaceClassA; // Illegal forward typedef of an interface class

```
class derivedClass4 implements interfaceClassA #(bit);
  virtual function bit[1:0] foo();
   // implement bar
endclass : derivedClass4
// This interface class declaration must be declared before derivedClass4
interface class interfaceClassA #(type T1 = logic)
  typedef T1[1:0] T2;
  pure virtual function T2 foo();
endclass : interfaceClass
```

8.25.4 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 polymorphic 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 shall also be possible to have multiple references of an interface class and use them to cast from one to other.

```
PutImp #() put_ref;
GetImp #() get_ref;
Fifo#() fifo_obj = new;
put_ref = fifo_obj;
$cast(get_ref, put_ref);
```

Comment [t6]: Issue #2 resolution from 1/18/2011 mantis note.

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

It shall be an error to cast from a source handle that is null. (See section 8.15 Casting)

8.25.5 Name Scoping Conflicts and Resolution

When any class *implements* an interface class, the names and prototypes of that interface class' methods are declared as method names within the implementing class. They may be accessed in all ways that it is legal to access other names in that scope. Any subsequent definition, or duplicate declaration of these names must have exactly the same prototype (except that a definition eventually replaces the "pure" modifier). It is an error to declare, define, or acquire (from a super class or from another interface class) a method with the same name that does not exactly match the method prototype as declared in the interface class being implemented. One exception allows for the virtual methods of derived interface classes to use a matching return type and not exactly match (see section 8.19)

Let's take the following examples:

```
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 bit foo();
   return (0);
  endfunction
 virtual 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 A 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;

Comment [t7]: This is to resolve the \$cast issue (Issue #4) in the mantis note from 1/18/2011.

Comment [t8]: For 3278, 3279

```
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 but with different result 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.

A class may implement its interface class contracts by either defining or inheriting (or both) a good implementation of each unique pure method it promises to provide. The class which defines the method need not be the one to claim that it implements an interface 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
```

7

```
// 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, derivedClass uses foo defined in baseClass.

8.25.6 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 derived Class is virtual. It both extends baseClass which contains a fully defined method foo and implements interfaceClass which has only a prototype of bar.

8.25.7 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. The actual value of the constant shall be the same on all classes that implement the method. See section 13.5.3 (Default argument values) for more information.

Comment [t9]: Issue #5 resolution from 1/18/2011 mantis note.

8.25.8 Constraint Blocks and Cover Groups

Constraint blocks are not allowed.

Cover groups are not allowed.