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 interfaces or overuse that keyword as this new functionality will be introduced and discussed in the context of being an ‘interface class’.

(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).

**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** interface\_class\_type {**,** interface\_class\_type }]**;**

 { class\_item }

**endclass** [ **:** class\_identifier]

interface\_class\_type ::= ps\_class\_identifier[parameter\_value\_assignment]

……

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

**8.25 Interface classes**

A set of classes may be created that can be viewed as all having a common set of behaviors. Such a common set of behaviors may be created using *interface classes*. A non-interface class can be declared as implementing one or more interface classes. This creates a requirement for the non-interface class to provide implementations for a set of methods which shall satisfy the requirements of a virtual method override (see 8.19).

An **interface class** shall only contain pure virtual methods (see 8.20), type declarations (see 6.18) and parameters (see 6.20, 8.24). Constraint blocks, covergroups and nested classes (see 8.22) shall not be allowed in an interface class. An interface class shall not be nested within another class. An interface class can inherit from one or more interface classes through the **extends** keyword, meaning that it inherits all the member types, pure virtual methods and parameters of the interface classes it extends, except for any member types and parameters that it may hide. In the case of multiple inheritance, name conflicts may occur which must be resolved (see 8.25.6).

Classes can implement one or more interface classes through the **implements** keyword. No member types or parameters are inherited through the **implements** keyword. A subclass implicitly **implements** all of the interface classes implemented by its superclass. In the following example, class C implicitly implements interface class A and has all of the requirements and capabilities as if it explicitly implemented interface class A:

**interface class** A;

**endclass**

**class** B **implements** A;

**endclass**

**class** C **extends** B;

**endclass**

All pure virtual methods from an interface class shall have overridden implementations in order to be implemented by a non-abstract class. When an interface class is implemented by a class, the required implementations of interface class methods may be provided by inherited virtual method implementations. A **virtual class** shall define or inherit a **pure virtual** method prototype or **virtual** method implementation for each **pure virtual** method prototype in each implemented **interface class**. The keyword **virtual** shall be used unless the **virtual** method is inherited.

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 class (see 8.21 Polymorphism). It is not sufficient that a class provides implementations for all the pure virtual methods of an interface class; the class or one of its superclasses shall be declared to implement the interface class through the **implements** keyword, or else the class does not implement the interface class.

The following is a simple example of interface classes.

**interface class** PutImp#(**type** PUT\_T = **logic**);

 **pure virtual function void** put(PUT\_T a);

**endclass**

**interface class** GetImp#(**type** GET\_T = **logic**);

 **pure virtual function** GET\_T get();

**endclass**

**class** Fifo#(**type** T = **logic**, **int** DEPTH = 1) **implements** PutImp#(T), GetImp#(T);

 T myFifo [$:DEPTH-1];

 **virtual function void** put(T a);

 myFifo.push\_back(a);

 **endfunction**

 **virtual function** T get();

 get = myFifo.pop\_front();

 **endfunction**

**endclass**

**class** Stack#(**type** T = **logic**, **int** DEPTH = 1) **implements** PutImp#(T), GetImp#(T);

 T myFifo [$:DEPTH-1];

 **virtual function void** put(T a);

 myFifo.push\_front(a);

 **endfunction**

 **virtual function** T get();

 get = myFifo.pop\_front();

 **endfunction**

**endclass**

The example has two interface classes, PutImp and GetImp, which contain prototype pure virtual methods put and get. The Fifo and Stack classes use the keyword **implements** to implement the PutImp and GetImp interface classes and they provide implementations for put and get. These classes therefore share common behaviors without sharing a common implementation.

**8.25.1 Interface class syntax**

interface\_class\_declaration ::=

**interface class** class\_identifier [ parameter\_port\_list ]

[ **extends** interface\_class\_type {**,** interface\_class\_type }] **;**

 { interface\_class\_item }

**endclass** [ **:** class\_identifier]

interface\_class\_item ::=

 type\_declaration

 | { attribute\_instance } interface\_class\_method

 | local\_parameter\_declaration **;**

 | parameter\_declaration7**;**

 | **;**

interface\_class\_method ::=

 **pure** **virtual** method\_prototype

**NOTE to the editor: The footnote for parameter\_declaration should refer to the same footnote that is #7 in the 1800-2009 LRM:**

**7) In a parameter\_declaration that is a class\_item, the parameter keyword shall be a synonym for the localparam keyword.**

**8.25.2 Extends versus implements**

Conceptually **extends** is a mechanism to add to or modify the behavior of a superclass while **implements** is a requirement to provide implementations for the pure virtual methods in an interface class. When a class is extended, all members of the class are inherited into the subclass. When an interface class is implemented, nothing is inherited.

An interface class may extend, but not implement, one or more interface classes, meaning that the interface subclass inherits members from multiple interface classes and may add additional member types, pure virtual method prototypes and parameters. A class or virtual class may implement, but not extend, one or more interface classes. Because virtual classes are abstract they are not required to fully define the methods from their implemented classes (see 8.25.7). The following highlights these differences:

* An interface class
	+ may extend zero or more interface classes
	+ may not implement an interface class
	+ may not extend a class or virtual class
	+ may not implement a class or virtual class
* A class or virtual class
	+ may not extend an interface class
	+ may implement zero or more interface classes
	+ may extend at most one other class or virtual class
	+ may not implement a class or virtual class
	+ may simultaneously extend a class and implement interface classes

In the following example, a class is both extending a base class and implementing two interface classes:

**interface class** PutImp#(**type** PUT\_T = **logic**);

 **pure virtual function void** put(PUT\_T a);

**endclass**

**interface class** GetImp#(**type** GET\_T = **logic**);

 **pure virtual function** GET\_T get();

**endclass**

**class** MyQueue #(**type** T = **logic**, **int** DEPTH = 1);

 T PipeQueue[$:DEPTH-1];

 **virtual** **function** **void** deleteQ();

 PipeQueue.delete();

 **endfunction**

**endclass**

**class** Fifo #(**type** T = **logic, int** DEPTH **= 1**)

 **extends** MyQueue#(T, DEPTH)

 **implements** PutImp#(T), GetImp#(T);

 **virtual function void** put(T a);

 PipeQueue.push\_back(a);

 **endfunction**

 **virtual function** T get();

 get = PipeQueue.pop\_front();

 **endfunction**

**endclass**

In this example, the PipeQueue property and deleteQ method are inherited in the Fifo class. In addition the Fifo class is also implementing the PutImp and GetImp interface classes so it shall provide implementations for the put and get methods respectively.

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

**virtual class** XFifo#(**type** T\_in = **logic**, **type** T\_out = **logic**, **int** DEPTH = 1)

 **extends** MyQueue#(T\_out)

 **implements** PutImp#(T\_in), GetImp#(T\_out);

 **pure virtual function** T\_out translate(T\_in a);

   **virtual function void** put(T\_in a);

     PipeQueue.push\_back(translate(a));

 **endfunction**

   **virtual function** T\_out get();

      get = PipeQueue.pop\_front();

 **endfunction**

**endclass**

An inherited virtual method can provide the implementation for a method of an implemented interface class. Here is an example:

**interface class** IntfClass;

  **pure virtual function bit** funcBase();

  **pure virtual function bit** funcExt();

**endclass**

**class** BaseClass;

  **virtual function bit** funcBase();

    **return** (1);

  **endfunction**

**endclass**

**class** ExtClass **extends** BaseClass **implements** IntfClass;

  **virtual function bit** funcExt();

    **return** (0);

  **endfunction**

**endclass**

ExtClass fulfills its requirement to implement IntfClass by providing an implementation of funcExt and by inheriting an implementation of funcBase from BaseClass.

An inherited non-virtual method does not provide an implementation for a method of an implemented interface class.

**interface class** IntfClass;

 **pure virtual function void** f();

**endclass**

**class** BaseClass;

 **function void** f();

 $display("Called BaseClass::f()");

 **endfunction**

**endclass**

**class** ExtClass **extends** BaseClass **implements** IntfClass;

 **virtual function void** f();

 $display("Called ExtClass::f()");

 **endfunction**

**endclass**

The non-virtual function f() in BaseClass does not fulfill the requirement to implement IntfClass. The implementation of f() in ExtClass simultaneously hides f() from base class and fulfills the requirement to implement IntfClass.

**8.25.3 Type access**

Parameters and typedefs within an interface class are inherited by extending interface classes, but are not inherited by implementing interface classes. All parameters and typedefs within an interface class are static and can be accessed through the class scope resolution operator :: (see 8.22). It shall be illegal to access interface class parameters or localparams through an interface class select (a dotted reference).

Example 1: types and parameter declarations are inherited by extends

**interface** **class** IntfA #(**type** T1 = **logic**);

 **typedef** T1[1:0] T2;

 **pure virtual function** T2 funcA();

**endclass** : IntfA

**interface** **class** IntfB #(**type** T = **int**) **extends** IntfA #(T);

 **pure virtual function** T2 funcB(); // legal, type T2 is inherited

**endclass** : IntfB

Example 2: type and parameter declarations are not inherited by implements and must be specified with the scope resolution operator

**interface class** IntfC;

  **typedef enum** {ONE, TWO, THREE} t1\_t;

  **pure virtual function** t1\_t funcC();

**endclass** : IntfC

**class** ClassA **implements** IntfC;

  t1\_t t1\_i; // error,  t1\_t is not inherited from IntfC

  **virtual function** IntfC::t1\_t funcC(); // correct

    **return** (IntfC::ONE); // correct

  **endfunction** : funcC

**endclass** : ClassA

**8.25.4 Type usage restrictions**

A class shall not implement a type parameter, nor shall an interface class extend a type parameter; even if the type parameter or defined type resolves to an interface class. The following examples illustrate 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) **extends** T;

A class shall not implement a forward typedef for an interface class. An interface class shall not extend from a forward typedef of an interface class. An interface class shall be declared before it is implemented or extended.

**typedef interface class** IntfD;

**class** ClassB **implements** IntfD #(**bit**); // illegal

 **virtual function bit**[1:0] funcD();

**endclass** : ClassB

// This interface class declaration must be declared before ClassB

**interface class** IntfD #(**type** T1 = **logic**)

 **typedef** T1[1:0] T2;

 **pure virtual function** T2 funcD();

**endclass** : intfD

**8.25.5 Casting and object reference assignment**

It shall be legal to assign an object handle to a variable of an interface class type that the object implements.

    **class** Fifo #(**type** T = **int**) **implements** PutImp#(T), GetImp#(T);

 **endclass**;

    Fifo#(**int**) fifo\_obj = **new**;

 PutImp#(**int**) put\_ref = fifo\_obj;

It shall be legal to dynamically cast between interface class variables if the actual class handle is valid to assign to the destination.

    GetImp#(**int**) get\_ref;

    Fifo#(**int**) fifo\_obj = **new**;

 PutImp#(**int**) put\_ref = fifo\_obj;

 **$cast**(get\_ref, put\_ref);

In the above, put\_ref is an instance of Fifo#(**int**) which implements GetImp#(**int**). It shall also be legal to cast from an object handle to an interface class type handle if the actual object implements the interface class type.

 **$cast**(fifo\_obj, put\_ref); // legal

 **$cast**(put\_ref, fifo\_obj); // legal, but casting is not required

Like abstract classes, an object of an interface class type shall not be constructed.

 put\_ref = **new**(); // illegal

Casting from a source interface class handle that is null is handled in the same manner as casting from a source class handle that is null. (See section 8.15 Casting)

**8.25.6 Name conflicts and resolution**

When a class **implements** multiple interface classes, or when an **interface class** **extends** multiple interface classes, identifiers are merged from different name spaces into a single name space. When this occurs, it is possible that the same identifier name can be inherited from multiple name spaces simultaneously creating a conflict which must be resolved.

**8.25.6.1 Method name conflict resolution**

It is possible that an interface class may inherit multiple methods, or a class may be required through implements to provide an implementation of multiple methods, where these methods have the same name. This is a method name conflict. A method name conflict shall be resolved with a single method prototype or implementation which simultaneously

1. is a virtual method override (see 8.19) for any inherited method and
2. provides an implementation for all pure virtual methods of any implemented interface classes

of the same name.

Example:

**interface class** IntfBase;

  **pure virtual function bit** funcBase();

**endclass**

**interface class** IntfExt **extends** IntfBase;

  **pure** **virtual function bit** funcExt();

**endclass**

**virtual class** ClassBase;

  **pure** **virtual function bit** funcBase();

**endclass**

**class** ClassExt **extends** ClassBase **implements** IntfExt;

  **virtual function bit** funcBase();

    **return** (0);

  **endfunction**

  **virtual function bit** funcExt();

    **return** (0);

  **endfunction**

**endclass**

Class ClassExt provides an implementation of funcBase which simultaneously overrides the pure virtual method prototype from ClassBase and provides an implementation for funcBase from IntfBase.

There are cases in which a method name conflict cannot be resolved. Example:

**interface class** IntfBaseA;

  **pure virtual function bit** funcBase();

**endclass**

**interface class** IntfBaseB;

  **pure virtual function string** funcBase();

**endclass**

**class** ClassA **implements** IntfBaseA, IntfBaseB;

  **virtual function bit** funcBase();

    **return** (0);

  **endfunction**

**endclass**

In this case, funcBase is prototyped in both IntfBaseA and IntfBaseB but with different return types, bit and string respectively. Although the implementation of funcBase is a valid override of IntfBaseA::funcBase, it is not simultaneously a valid override of the prototype of IntfBaseB:: funcBase so an error shall occur.

**8.25.6.2 Parameter and type declaration inheritance conflicts and resolution**

Interface classes may inherit parameters and type declarations from multiple interface classes. A name collision will occur if the same name is inherited from different interface classes. The subclass shall provide parameter and/or type declaration which override all such name collisions.

Example:

**interface class** PutImp#(**type** T = **logic**);

 **pure virtual function void** put(T a);

**endclass**

**interface class** GetImp#(**type** T = **logic**);

 **pure virtual function** T get();

**endclass**

**interface class** PutGetIntf#(**type** TYPE = **logic**)

 **extends** PutImp#(TYPE), GetImp#(TYPE);

 **typedef** TYPE T;

e**ndclass**

In the above example, the parameter T is inherited from both PutImp and GetImp. A conflict occurs despite the fact that PutImp::T matches GetImp::T and is never used by PutGetIntf. PutGetIntf overrides T with a type definition to resolve the conflict.

**8.25.6.3 Diamond inheritance**

Diamond inheritance occurs if the same interface class is inherited by the same class or interface class in multiple ways. Diamond inheritance never causes a name conflict. For example:

**interface class** IntfBase;

  **parameter** SIZE = 64;

**endclass**

**interface class** IntfExt1 **extends** IntfBase;

  **pure** **virtual function bit** funcExt1();

**endclass**

**interface class** IntfExt2 **extends** IntfBase;

  **pure** **virtual function bit** funcExt2();

**endclass**

**interface class** IntfExt3 **extends** IntfExt1, IntfExt2;

**endclass**

In the above example, the class IntfExt3 inherits the parameter SIZE from IntfExt1 and IntfExt2. Since these parameters originate from the same interface class, IntfBase, it shall not be considered a conflict.

Each unique parameterization of a parameterized interface class is an interface class specialization. It shall be a conflict if a parameterized interface class is inherited from different paths using different parameterizations. For example:

**interface class** IntfBase #(**type** T = **int**);

  **pure virtual function bit** funcBase();

**endclass**

**interface class** IntfExt1 **extends** IntfBase#(**bit**);

  **pure** **virtual function bit** funcExt1();

**endclass**

**interface class** IntfExt2 **extends** IntfBase#(**logic**);

  **pure** **virtual function bit** funcExt2();

**endclass**

**virtual class** ExtBase **implements** IntfExt1, IntfExt2;

 **typedef** **bit** T; // Override the conflicting identifier name

  **pure virtual function bit** funcBase();

**endclass**

In the above example, there are two different parameterizations of the interface class IntfBase. Each of these parameterizations of IntfBase is a specialization, therefore there is no diamond inheritance and there is a conflict of the parameter T which must be resolved.

**8.25.7 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.

**interface class** IntfClass;

  **pure virtual function bit** funcA();

  **pure virtual function bit** funcB();

**endclass**

// Partial implementation of IntfClass

**virtual class** ClassA **implements** IntfClass;

  **virtual function bit** funcA();

    **return** (1);

  **endfunction**

 **pure virtual function bit** funcB();

**endclass**

// Complete implementation of IntfClass

**class** ClassB **extends** ClassA;

  **virtual function bit** funcB();

    **return** (1);

  **endfunction**

**endclass**

It shall be illegal to use an interface class to partially define a virtual class without fulfilling the interface class prototype requirements. In other words, when an interface class is implemented by a virtual class, the virtual class must do one of the following for each interface class method prototype:

* Provide a method implementation
* Re-declare the method prototype with the pure qualifier.

In the example above ClassA fully defines funcA, but re-declares the prototype funcB.

**8.25.8 Method default argument values**

Method declarations within Interface Classes may have default argument values.  The default expression shall be a constant expression and is evaluated in the scope containing the subroutine declaration.  The value of the constant expression shall be the same for all the classes that implement the method. See section 13.5.3 (Default argument values) for more information.

**8.25.9 Constraint blocks, covergroups, and randomization**

Constraint blocks and covergroups shall not be declared in interface classes.

A **randomize** method call shall be legal with interface class handles. While inline constraints shall also be legal, interface classes cannot contain any data meaning that  inline constraints will only be able to express conditions related to state variables and are therefore of very limited utility.  Use of rand\_mode and constraint\_mode is not be legal as a consequence of the name resolution rules and the fact that interface classes are not permitted to contain data members.

***Change sub-clause 1.8 as follows:***

FROM:

Clause 8 describes the object-oriented programming capabilities in SystemVerilog. Topics include defining classes, dynamically constructing objects, inheritance and subclasses, data hiding and encapsulation, polymorphism, and parameterized classes.

TO:

Clause 8 describes the object-oriented programming capabilities in SystemVerilog. Topics include defining classes, interface classes, dynamically constructing objects, inheritance and subclasses, data hiding and encapsulation, polymorphism, and parameterized classes.

***Change sub-clause 8.15 post-Mantis item 3293 update as follows:***

FROM:

1. The type of the source expression is cast-compatible with the destination type, that is, the type of the source expression is a superclass of the destination type and the source is an object that is assignment compatible with the destination type. This type of assignment requires a run-time check as provided by $cast.

TO:

1. The type of the source expression is cast-compatible with the destination type, that is, either
	* the type of the source expression is a superclass of the destination type or
	* the type of the source expression is an interface class (see section 8.25)

and the source is an object that is assignment compatible with the destination type. This type of assignment requires a run-time check as provided by $cast.

***Change sub-clause 20.6.2 as follows:***

FROM:

The $bits system function returns 0 when called with a dynamically sized expression that is currently

empty. It shall be an error to use the $bits system function directly with a dynamically sized data type

identifier.

TO:

The $bits system function returns 0 when called with a dynamically sized expression that is currently

empty. It shall be an error to:

* Use the $bits system function directly with a dynamically sized data type identifier.
* Use the $bits system function on an object of an interface class type (see section 8.25 Interface Classes)

***Change sub-clause 21.2.1.7 as follows:***

FROM:

A chandle, class handle, event, or virtual interface shall print its value in an implementation dependent format, except that a null handle value shall print the word null.

TO:

A chandle, class handle, interface class handle, event, or virtual interface shall print its value in an implementation dependent format, except that a null handle value shall print the word null.

***Change sub-clause 6.22.5 as follows:***

FROM:

**6.22.5 Type incompatible**

*Type incompatible* includes all the remaining nonequivalent types that have no defined implicit or explicit

casting rules. Class handles, and chandles are type incompatible with all other types.

TO:

**6.22.5 Type incompatible**

*Type incompatible* includes all the remaining nonequivalent types that have no defined implicit or explicit

casting rules. Class handles, interface class handles, and chandles are type incompatible with all other types.

***Change sub-clause 6.18 as follows:***

FROM:

Sometimes a user-defined type needs to be declared before the contents of the type have been defined. This

is of use with user-defined types derived from the basic data types: **enum**, **struct**, **union**, and **class**. Support

for this is provided by the following forms for a *forward typedef*:

**typedef enum** type\_identifier;

**typedef struct** type\_identifier;

**typedef union** type\_identifier;

**typedef class** type\_identifier;

**typedef** type\_identifier;

TO:

Sometimes a user-defined type needs to be declared before the contents of the type have been defined. This

is of use with user-defined types derived from the basic data types: **enum**, **struct**, **union**, and **class**. Support

for this is provided by the following forms for a *forward typedef*:

**typedef enum** type\_identifier;

**typedef struct** type\_identifier;

**typedef union** type\_identifier;

**typedef class** type\_identifier;

**typedef interface class** type\_identifier;

**typedef** type\_identifier;

***Change sub-clause 6.18 as follows:***

FROM:

type\_declaration ::= *// from A.2.1.3*

**typedef** data\_type type\_identifier { variable\_dimension } **;**

| **typedef** interface\_instance\_identifier constant\_bit\_select **.** type\_identifier type\_identifier **;**

| **typedef** [ **enum** | **struct** | **union** | **class**] type\_identifier **;**

TO:

type\_declaration ::= *// from A.2.1.3*

**typedef** data\_type type\_identifier { variable\_dimension } **;**

| **typedef** interface\_instance\_identifier constant\_bit\_select **.** type\_identifier type\_identifier **;**

| **typedef** [ **enum** | **struct** | **union** | **class** | **interface class**] type\_identifier **;**

***Change sub-clause A.2.1.3 as follows:***

FROM:

type\_declaration ::=

**typedef** data\_type type\_identifier { variable\_dimension } **;**

| **typedef** interface\_instance\_identifier constant\_bit\_select **.** type\_identifier type\_identifier **;**

| **typedef** [ **enum** | **struct** | **union** | **class**] type\_identifier **;**

TO:

type\_declaration ::=

**typedef** data\_type type\_identifier { variable\_dimension } **;**

| **typedef** interface\_instance\_identifier constant\_bit\_select **.** type\_identifier type\_identifier **;**

| **typedef** [ **enum** | **struct** | **union** | **class** | **interface class**] type\_identifier **;**

***Change sub-clause 10.10 as follows:***

FROM:

* An item of any other type, or an item that has no self-determined type, shall be illegal except that the literal value **null** shall be legal if the target array's elements are of class type

TO:

* An item of any other type, or an item that has no self-determined type, shall be illegal except that the literal value **null** shall be legal if the target array's elements are of class or interface class type

***Change sub-clause 11.4.5 as follows:***

FROM:

The logical equality (or case equality) operator is a legal operation if either operand is a class object or the

literal **null**, and one of the operands is assignment compatible with the other. The logical equality (or case

equality) operator is a legal operation if either operand is a **chandle** or the literal **null**. In both cases, the

operator compares the values of the class objects or chandles.

TO:

The logical equality (or case equality) operator is a legal operation if either operand is a class object, interface class object or the literal **null**, and one of the operands is assignment compatible with the other. The logical equality (or case equality) operator is a legal operation if either operand is a **chandle** or the literal **null**. In both cases, the operator compares the values of the class objects, interface class objects or chandles.

***Change sub-clause 11.4.6 as follows:***

FROM:

The wildcard equality operator is equivalent to the logical equality operator if its operands are class objects,

chandles or the literal **null**.

TO:

The wildcard equality operator is equivalent to the logical equality operator if its operands are class objects, interface class objects, chandles or the literal **null**.

***Change sub-clause 11.4.11 as follows:***

FROM:

The conditional operator can be used with nonintegral types (see 6.11.1) and aggregate expressions (see

11.2.2) using the following rules:

* If both the first *expression* and second *expression* are of integral types, the operation proceeds as defined.
* If the first *expression* or second *expression* is an integral type and the opposing expression can be implicitly cast to an integral type, the cast is made and proceeds as defined.
* If the first *expression* or second *expression* is a class data type, the condition expression is legal in the following cases:
1. If both first *expression* and second *expression* are the literal value **null**, the result of the entire conditional expression is as if the expression were the literal **null**.
2. else, if either first *expression* or second *expression* is the literal **null**, the resulting type is the type of the non-null expression.
3. else, if the first *expression* is assignment compatible with the second *expression*, the resulting type is the type of the second *expression*,
4. else, if the second *expression* is assignment compatible with the first *expression*, the resulting type is the type of the first *expression*,
5. else, if the first *expression* and second *expression* are of a class type deriving from a common base class type, the resulting type is the closest common inherited class type.

In the above cases, the resulting value is the value of the first *expression* if the condition evaluates to

TRUE or the value of the second *expression* if the condition evaluates to FALSE.

* For all other cases, the type of the first *expression* and second *expression* shall be equivalent (see 6.22.2).

For nonintegral and aggregate expressions, if *cond\_predicate* evaluates to an ambiguous value, then:

* If the first *expression* and the second *expression* are of a class data type and if the conditional operation is legal, then the resulting type is determined as defined above and the result is null.
* Otherwise, both the first *expression* and second *expression* shall be evaluated, and their results shall be combined element by element. If the elements match, the element is returned. If they do not match, then the default-uninitialized value for that element’s type shall be returned.

TO:

The conditional operator can be used with nonintegral types (see 6.11.1) and aggregate expressions (see

11.2.2) using the following rules:

* If both the first *expression* and second *expression* are of integral types, the operation proceeds as defined.
* If the first *expression* or second *expression* is an integral type and the opposing expression can be implicitly cast to an integral type, the cast is made and proceeds as defined.
* If the first *expression* or second *expression* is a class or interface class data type, the condition expression is legal in the following cases:
1. If both first *expression* and second *expression* are the literal value **null**, the result of the entire conditional expression is as if the expression were the literal **null**.
2. else, if either first *expression* or second *expression* is the literal **null**, the resulting type is the type of the non-null expression.
3. else, if the first *expression* is assignment compatible with the second *expression*, the resulting type is the type of the second *expression*,
4. else, if the second *expression* is assignment compatible with the first *expression*, the resulting type is the type of the first *expression*,
5. else, if the first *expression* and second *expression* are of a class type deriving from a common base class type, the resulting type is the closest common inherited class type.

In the above cases, the resulting value is the value of the first *expression* if the condition evaluates to

TRUE or the value of the second *expression* if the condition evaluates to FALSE.

* For all other cases, the type of the first *expression* and second *expression* shall be equivalent (see 6.22.2).

For nonintegral and aggregate expressions, if *cond\_predicate* evaluates to an ambiguous value, then:

* If the first *expression* and the second *expression* are of a class data type or interface class and if the conditional operation is legal, then the resulting type is determined as defined above and the result is null.
* Otherwise, both the first *expression* and second *expression* shall be evaluated, and their results shall be combined element by element. If the elements match, the element is returned. If they do not match, then the default-uninitialized value for that element’s type shall be returned.