Object Orientation Revisited

Accellera VHDL-TC Extensions-SC White Paper 012

Peter Ashenden

Ashenden Designs

peter@ashenden.com.au

Revision number: 1

Revision Date: April 4, 2007

Revision Author: Peter Ashenden

Revision Remark: Initial version.

1 Introduction

In an earlier White Paper, ESC-WP-001, I discussed extensions to VHDL for object orientation and genericity. The genericity extensions, in simplified form, were adopted in VHDL-2007. The motivation for object orientation was to support verification through features for high-level modeling and for constrained randomization. Since there was insufficient time to develop detailed specifications for such features, they were deferred to later revision.

This White Paper revisits the discussion of object-oriented language features. Much of what is presented here is material from the previous White Paper, revised to reflect subsequent consideration in the intervening period. This discussion takes the genericity features adopted in VHDL-2007 as given, and suggests ways in which object-oriented features can integrate with genericity features.

Object-oriented language features provide class types with three aspects:

- Encapsulation of state
- Inheritance of state and operations from superclasses

- Polymorphism with dynamic dispatch of operations

Since VHDL is strongly based on the Ada programming language, and Ada has been extended to provide object-oriented features, it is appropriate to consider adapting the Ada features to VHDL. This was the approach taken in the SUAVE extensions (www.ashenden.com.au/suave.html). However, since those extensions were developed, VHDL was revised to include protected types, which provide a mechanism for encapsulation with mutual exclusion for concurrent processes. Rather than providing separate language features that both provide for encapsulation, it is now more appropriate to base object-oriented features on protected types. This has the added advantage that protected types are similar in form to class types seen in various programming languages, such as C++ and Java. Thus, they will be more familiar to users than the features of Ada.

2 Encapsulation

In this section, we first review protected types as a mechanism for encapsulation, and then consider how class types can be defined in a similar way.

2.1 Review of Protected Types

A protected type, as specified in VHDL-2002, is intended to encapsulate variables that are to be shared between processes. The variables themselves are not made visible. Rather, the protected type provides sub-programs, called methods, that have visibility over the variables and that enforce mutual exclusion. Only one process at a time can invoke any method of an object of a protected type.

The protected type definition consists of two parts. First, the protected type declaration just defines the method signatures. Second, the protected type body includes declarations of variables and other items, as well as the bodies of the methods. The protected type forms a declarative region (in two disjoint parts) that is instantiated by elaborating a variable of the protected type. Only the method names are made visible (by selection) beyond the scope of the protected type definition. The prefix of a selected name denoting a method is the variable of the protected type.

Example 1: Protected type for network packets

The following is a protected type declaration to represent network packets:

```
type network_packet is protected

  procedure set_source (src : natural);
  impure function get_source return natural;
  procedure set_dest (dst : natural);
  impure function get_dest return natural;
  procedure compute_crc;

end protected network_packet;
```

The protected type has five methods that form the interface to the type. The implementation details are encapsulated within the protected type, and are not visible to the user. They are defined within the protected type body as follows:

```
type network_packet is protected body

  variable src, dst, crc : natural;

  procedure set_source (src : natural) is
  begin
    network_packet.src := src;
  end procedure set_source;

  impure function get_source return natural is
  begin
    return src;
  end function get_source;

  procedure set_dest (dst : natural) is ...

  impure function get_dest return natural is ...

  procedure compute_crc is
  begin
    crc := ...;
  end procedure compute_crc;

end protected body network_packet;
```

Within an architecture, we can declare a shared variable of the protected type:

```
shared variable packet_buf : network_packet;
```

This creates a variable that has, within it, state consisting of the `src`, `dst` and `crc` variables. Then, within processes in the architecture, we can access the variables through the methods of the protected type. For example:

```
traffic_gen : process is
begin
  packet_buf.set_source(192); packet_buf.set_dest(193);
  packet_buf.compute_crc;
  ...
end process traffic_gen;

traffic_mon : process is
  variable src, dst : natural;
begin
  ...
  src := packet_buf.get_source; dst := packet_buf.get_dest;
end process traffic_mon;
```

When the `traffic_gen` process calls a method, it first gains mutually exclusive access to the `packet_buf` variable. While the process is executing the method, any other process calling a method of `packet_buf` blocks until the call from `traffic_gen` returns.

While the mutual exclusion semantics are required for shared variables, they are not required for non-

shared variables of a protected type. Such non-shared variables can be declared within processes or subprograms. Without mutual exclusion, the encapsulation aspect of protected types is semantically similar to that of class definitions. Hence, protected types meet the encapsulation requirements of object-orientation in VHDL.

2.2 Shared Variable Ports

VHDL generally has good correspondence between declartions of objects and interface objects. That is, we can declare constants, variables, signals, and files; and we can have interface constants, variables, signals, and files, limited to where such interface objects make sense. In the case of signals, we can declare signals for communication and synchronization between parts of a design, we can access signals within processes, we can declare interface signals for subprograms, components and entities, and we can associate actual signals with interface signals. Shared variables of protected types similarly can be declared for communication and synchronization between parts of a design, and we can access shared variables within processes. We can declare interface variables of protected types for subprograms, and we can associate actual shared variables with interface variables of subprograms. What is missing is a facility to declare interface variables of protected types for components and entities, and to associate actual shared variables with such interface variables. Without this facility, we are limited to using a given shared variable only within a single design entity, or to making a shared variable global by declaring it in a package. With the extensions proposed in this White Paper, shared variables will become much more useful for abstract communication between parts of a design. Rectifying the anomaly will allow significantly enhance the usability of shared variables.

The extension required to rectify the anomaly is to allow declaration of interface shared variables in port lists of component and entity declarations. (There is precedent for ports of class other than signal in VHDL-AMS. There, ports can be terminals and quantities.) A shared variable port must be of a protected type, and, like a variable parameter of a protected type, be of mode inout. The actual associated with such a port must be a shared variable of the same type, or another such port of the same type. The semantics of association would be the same as those for variable parameters of protected type, namely, passing by reference. The reference would be determined at the time of elaboration of the association element.

Example 2: Shared counter port

Suppose we have a protected type for a shared counter declared in a package as follows:

```
package CounterPkg is
  type CounterType is protected
    procedure reset;
    procedure increment(by : integer := 1);
    impure function get return integer;
  end protected CounterType;
end package CounterPkg;
```

We can declare an entity that has a port of the protected type:

```
entity ArithmeticUnit is
  port ( ...
         shared variable arithCounter : work.CounterPkg.CounterType );
end entity ArithmeticUnit;
```

Within an architecture of another design entity, we can declare a shared variable and associate it with instances of components that also have ports of the protected type:

```
architecture topLevel of Accelerator is
  use work.CounterPkg.CounterType;
  shared variable opsCompleted : CounterType;

  component LoadStoreUnit is
    port ( ...
           shared variable loadStoreCounter : work.CounterPkg.CounterType );
  end component LoadStoreUnit;

begin
  arith : entity work.ArithmeticUnit(pipelined)
    port map ( ..., arithCounter => opsCompleted );
  loadStore: component LoadStoreUnit
    port map ( ..., loadStoreCounter => opsCompleted );
end architecture topLevel;
```

2.3 Class Types Based on Protected Types

One way in which protected types differ from classes in some languages is in the visibility of encapsulated variables. Other languages provide a way of specifying whether the variables are publicly visible or not, whereas protected types only allow for private (non-visible) variables. Were the variables to be visible, mutually exclusive access to them could not readily be guaranteed.

One way of addressing this difference would be to separate the concurrency control aspects of protected types from the encapsulation aspects. We could consider a protected type, as currently specified in VHDL-2002, to be a class type with concurrency control. In such a class type, variables must remain private. Moreover, a shared variable of a class type must be of a protected class type. We could also allow definition of a class type without concurrency control. Variables in such a class type could be declared in the class type declaration part (as opposed to the body). Such an unprotected class type would only be allowed for non-shared variables. We could relax the restrictions on use of access-type parameters for methods of unprotected class types, since objects of such a class type could not be used as a covert channel to move access values between processes. We would also allow access types to designate class types, and thus allow a name denoting an access value to be the prefix of a method call.

Syntactically, we would change a protected type definition to a class type definition, using the keyword class. We would allow the optional keyword protected before class to add concurrency control. For backward compatibility, we could allow the keyword protected by itself to mean the same as protected class.

Example 3: Class type for network packets

The protected type for network packets from the previous example could be revised to use unprotected class types as follows:

```
type network_packet is class
  variable src, dst : natural;
  procedure compute_crc;
end class network_packet;
```

```
type network_packet is class body

  variable crc: natural;

  procedure compute_crc is
  begin
    crc := ...;
  end procedure compute_crc;

end class body network_packet;
```

Since the `src` and `dst` variables are declared in the class type declaration, they are visible, and so we don't need to include accessor methods for them. A process that declares a variable of the class type could access the variables by selection, for example:

```
packet_proc : process is
  variable saved_packet : network_packet;
  variable saved_dst : natural;
begin

  saved_packet.src := 192;
  saved_dst := saved_packet.dst;
  ...
end process packet_proc;
```

Example 4: Class for random number state

Jim Lewis suggested use of a protected type to encapsulate the state of a random number generator. We can use a class type for this purpose. A simplified class type based on Jim's example is

```
type RandomType is class
  procedure initialize(seed1, seed2 : positive);
  impure function uniform return real;
end class RandomType;

type RandomType is class body
  variable randomSeed1 : positive := 7;
  variable randomSeed2 : positive := 1;

  procedure initialize(seed1 : positive := 7; seed2 : positive := 1) is
  begin
    randomSeed1 := seed1; randomSeed2 := seed2;
  end procedure initialize;

  impure function uniform return real is
    variable randomVal : real;
  begin
    ieee.math_real.uniform(randomSeed1, randomSeed2, randomVal);
    return randomVal;
  end function uniform;
end class RandomType;
```

We can then declare variables of the `RandomType` class, with each variable encapsulating the state of a separate random number stream. Successive calls to the `uniform` method on each stream generate the next number in the

stream.

```
variable stream1, stream2 : RandomType;
variable independent1, independent2 : real;
...

stream1.initialize;
stream2.initialize(1, 100);
while ( ... ) loop
  independent1 := stream1.uniform;
  independent2 := stream2.uniform;
  ...
end loop;
```

2.4 Operator Methods and Class-Typed Parameters

VHDL currently allows protected types to declare methods using operator symbols, but it does not provide a way to invoke such methods as operators. Instead, they must be invoked as functions calls. A method for a unary operator is declared with no parameters, the intention being that the object on which the method is invoked implicitly be the operand. Similarly, a method for a binary operator is declared with one parameter, the intention being that the object on which the method is invoked implicitly be the left operand and the parameter be the right operand.

We can provide for operator invocation of such a method by specifying that a class declaration defining a method with an operator symbol is implicitly followed by a function declaration for the operator symbol. For a unary operator, the parameter of the function declaration is anonymous and is of the class type. A call to the implicit function is equivalent to a call to the method with the operand as the prefix. For a binary operator, the first parameter of the function declaration is anonymous and is of the class type, and the second parameter is the same as the parameter of the method declaration. A call to the implicit function is equivalent to a call to the method with the left operand as the prefix and the right operand as the parameter.

For this extension to work, provision must be made for a function to have a class-type parameter. Currently, VHDL requires protected type parameters to be variable parameters of mode inout. We can relax this to allow class-type variable and constant parameters parameters and to allow modes in and out. A variable parameter of mode out would have the same semantics as one of mode inout, since passing is by reference. A parameter of mode in would be restricted to be read-only. We will return to this shortly.

To support read-only usage of class-typed parameters, we can define pure function methods for a class. Currently, a pure function cannot access or modify state declared outside the function's declarative region. Thus, it always returns the same results when invoked with a given set of actual parameter values. A function method should be able to access the state of the prefix object, but not modify it. Thus, when invoke on an object with a given state, it should always return the same value. We can ensure this by requiring a pure-function method follow the exiting rules for pure functions, except that it can read declarations in the class body, provided any declaration read is not of an access type and does not contain a subelement of an access type. The restriction on reading access values prevents the method from being affected by external changes to designated variables. With this provision, a pure-function method can access the state of an object, but cannot modify it.

Support for function parameters of class types involves allowing constants (including interface constants)

to be of class types. Since constants are read-only, once initialized, only pure-function methods can be invoked on them. For constant parameters, the actual parameter can take the form of an object name, in which case a reference to the object is passed. In general, a means of denoting a class-typed value is required. The value would need to be constructed so that a reference to it could be provided. This is analogous to the way in which an aggregate constructs an array or record value for use as a constant value. In the case of class types, we can invoke a constructor (see Section 4 on page 16) to specify the initial value of an explicitly declared constant, the default value of an interface constant, or an operand in a class-valued expression.

A subprogram with a class-type in-mode parameter would be required to treat the object reference by the parameter as read-only. That would mean only invoking pure-function methods on it, since any other kind of method (impure function or procedure) could potentially modify the state of the object as a side-effect. In the case of operator methods, since they would be deemed equivalent to function declarations, the implicit parameter would be of mode in. That means the operator method must be pure. This is a backward incompatibility with VHDL-2002. However, given that operator methods are virtually unknown, the impact on existing models is negligible.

A related issue is that a method that has a parameter of the class type cannot access the encapsulated declarations of the parameter. The only way to use the parameter is to invoke its methods. Thus, operations that would require access to the internal state of the parameter would have to be supported by methods to access that state. Those methods would be publicly visible (unless a means for declaring private methods were provided), and would incur the overhead of a method call and the associated concurrency control.

In other object-oriented languages, method bodies have access to the internal state of any object that is of the encapsulating class. We can extend the visibility rules of VHDL similarly by specifying that a declaration in a class type body be visible by selection at the place of the suffix of a selected name occuring within the class type body, where the prefix of the selected name is appropriate for the class type. That is, the prefix denotes an object of the class type, or an access value designating the class type, and so on.

Example 5: Clone method

A class can declare a method to make an object a clone of another object of the class:

```
type C is class
  ...
  procedure clone ( o : inout C );
end class C;

type C is class body
  variable i, j : integer;
  ...
  procedure clone ( o : inout C ) is
  begin
    i := o.i; j := o.j;
  end procedure clone:
end class body C;
```

We can allow a method of class to have an in-mode parameter of that class and to access the class body declarations for that parameter, provided any such declaration is only read and is not of an access type and does not contain a subelement of an access type. Moreover, any method invoked on the parameter must be a pure function method.

Example 6: Class for a point abstract data type

An abstract data type for a three-dimensional point can be defined as follows:

```
type Point is class
  new Point(x, y, z : real); -- constructor
  pure function "abs" return real;
  pure function "*" (R : real) return Point;
  pure function "+" (R : Point) return Point;
end class Point;

function "*" (L : real; R : Point) return Point is
begin
  return R * L;
end function "*”;

type Point is class

  variable x, y, z : real;

  new Point(x, y, z : real) is
  begin
    this.x := x; this.y := y; this.z := z;
  end new Point;

  pure function "abs" return real is
  begin
    return ieee.math_real.sqrt(x**2 + y**2 + z**2);
  end function "abs";

  pure function "*" (R : real) return Point is
  begin
    return Point(x * R, y * R, z * R);
  end function "*";

  pure function "+" (R : Point) return Point is
  begin
    return Point(x + R.x, y + R.y, z + R.z);
  end function "+";

end class Point;
```

We can use the class as follows:

```
constant origin : Point := Point(0.0, 0.0, 0.0);
variable here, there, elsewhere : Point;

here := Point(1.0, 1.0, 1.0);
there := origin + here * 5.0;
elsewhere := 2.0 * there + here;
```

2.5 Assignment and Predefined Equality

Assignment can be defined for class types (other than protected types). The right-hand-side value must be an object of the same class type as the assignment target. The effect of assignment is to copy the encapsulated variables of the right-hand-side value to the encapsulated variables of the target. For classes that encapsulate dynamically allocated structures, this form of assignment is commonly referred to as “shallow copy,” since only the pointers are copied. If “deep copy” is required, copying the allocated data, appropriate methods should be defined in the class.

Equality operators (and consequently inequality operators) can be predefined for class types (other than protected types). The predefined equality operator compares corresponding encapsulated variables using the predefined equality operators for the variables’ types. If all encapsulated varaibles are pairwise equal, the two class-typed objects are equal. The inequality operator is the logical negation of the equality operator.

Assignment and predefined equality and inequality cannot be predefined for protected types, since an appriate order for acquiring exclusive access to the two objects cannot be determined *a priori*. If these operations are required for protected types, methods must be defined in the type definition.

2.6 Static Class Members

Some object-oriented languages, such as C++ and Java, provide for static members (variables and methods) of classes, that is encapsulated declarations within a class that are shared by all instances of the class. Static variables are used for state that is shared by all objects of a class. Static methods provide utility operations involving objects of the class. Static members are required in these languages as there is no place other than the class declaration to declare the variables or methods. The languages do not have the hierarchical static scope structure that VHDL has.

A similar effect to static data members can be achieved in VHDL by declaring related variables and subprograms in the declarative region containing a class declaration.

Example 7: Static variable and method for random number class

We can augment the random number class of Example 4 on page 6 to keep a count of the number of times any stream is re-initialized. In the declarative region containing the declaration of the class type, we declare:

```
variable initialization_count : natural := 0;

type RandomType is class
  ...  -- as before
end class RandomType;

type RandomType is class body
  ...

  procedure initialize(seed1 : positive := 7; seed2 : positive := 1) is
  begin
    randomSeed1 := seed1; randomSeed2 := seed2;
    initialization_count := initialization_count + 1;
```

```
  end procedure initialize;

  impure function uniform return real is ...
end class RandomType;
```

At any place in which `RandomType` is visible, `initialization_count` is also visible, so it can be read directly. This has the same effect as a static class variable. If we want to hide the variable to prevent inadvertent corruption, and only provide access to it with what amounts to a static method, we could wrap the class type, the variable, and an accessor function in a package:

```
package RandomPkg is
  type RandomType is class
    ...  -- as before
  end class RandomType;
  impure function get_initialization_count return natural;
end package RandomPkg;

package body RandomPkg is

  variable initialization_count : natural := 0;

  impure function get_initialization_count return natural
  begin
    return initialization_count;
  end function get_initialization_count;

  type RandomType is class body
    ...

    procedure initialize(seed1 : positive := 7; seed2 : positive := 1) is
    begin
      randomSeed1 := seed1; randomSeed2 := seed2;
      initialization_count := initialization_count + 1;
    end procedure initialize;

    impure function uniform return real is ...
  end class RandomType;

end package body RandomPkg;
```

Recall that in VHDL-2007, we can declare a package in any declarative region. Provided the package is declared within a process or subprogam, we can declare a local non-shared variable in the package, as we have done here. Were this not in a process or subprogram, we would have to make the counter variable shared and of a protected type.

Many of the use cases for static methods can also be expressed using subprogram declarations in an enclosing declarative region. If data is declared in an enclosing region, a subprogram can be provided to access the data. If a binary operator for a class type is required, with the left operand being of some other type, the operator can often be expressed as a function that calls a class method with the operands in the opposite order.

One use case that presents difficulty is the case of an operator that is not commutative. In that case, it may be necessary for the class to provide a method specifically for the purpose of supporting the external operation.

3 Concurrency Control in Protected Classes

In this section, we review some limitations of the concurrency control provided by protected types and suggest some extensions to overcome the limitations.

3.1 Conditional Waiting in Methods

Protected types are a limited form of Hoare monitors. As they are currently defined, they provide concurrency control only by mutual exclusion on method invocation. Hoare's monitors also provided for condition testing within a method. A method could suspend, releasing mutual exclusion, until a condition became true. This provided a means of guarding concurrent operations. For example, a protected type implementing a bounded FIFO needs to guard the insert operation with the condition that there is room in the FIFO, and to guard the remove operation with the condition that the FIFO is not empty. This cannot currently be expressed in VHDL's protected types without busy waiting or using signals.

We can extend protected class types to provide for conditional concurrency control using a new form of wait statement within methods. A wait statement would be allowed with no sensitivity to signals, but waiting on a condition. When the wait statement is executed, mutual exclusion on the shared variable would be released and the parent process blocked. That would allow some other process to enter the monitor by executing a method call, or to resume executing a method in which it had been blocked.

The question to consider is when the condition of a wait statement in a method should be tested. In common use cases, the condition would involve variables internal to the protected class type. Those variables would be assigned by other methods, possibly causing the condition to become true. Thus, the condition should be tested when some other process either completes a method call for the object or blocks on a wait statement in a method of the object.

If several processes are blocked on wait statements or method entry for an object, there is no *a priori* reason to prefer any of them over any others for resumption. The language should not specify any choice of blocked process, but should leave the choice implementation dependent. A particular implementation might try to schedule processes in such a way as to promote fairness or deadlock avoidance.

Note that the condition on which a method blocks might become true as a consequence of some action other than method execution. This might happen if the condition involves values of objects external to the protected object. Nonetheless, the condition would not be tested until the next method exit or wait within the protected object. This is analogous to a situation in the current language of a wait statement with a sensitivity clause and a condition clause, in which the sensitivity clause does not include all signals mentioned in the condition.

The semantics of a wait statement would be augmented to specify that, for a wait statement in a method of a protected object or in a procedure whose parent is such a method, exclusive access to the object is rescinded when the process blocks. The suspended process can resume as a result of exclusive access by another process being rescinded and exclusive access being secured by the suspended process. In that case, the condition is evaluated. If it is true, the process is resumed; otherwise, the process resuspends. Note that exclusive access must be held while the condition is checked, otherwise a race condition may arise.

Example 8: Bounded FIFO

A FIFO has methods to put and get data. If the FIFO has bounded capacity, a call to the `put` method should block when the FIFO is full, and remain blocked until data is taken out using the `get` method. Similarly, a call to the `get` method should block when the FIFO is empty, and remain blocked until data is supplied using the `put` method. A protected class type for such a FIFO is

```
type BoundedFIFO is protected class
  procedure put ( e : in element_type );
  procedure get ( e : out element_type );
end protected class BoundedFIFO;

type BoundedFIFO is protected class body
  constant size : positive := 20;
  type element_array is array (0 to size-1) of element_type;
  variable elements : element_array;
  variable head, tail : natural range 0 to size-1 := 0;
  variable count : natural range 0 to size := 0;

  procedure put ( e : in element_type )
  begin
    if count = size then wait until count < size; end if;
    elements(head) := e;
    head := (head + 1) mod size;
    count := count + 1;
  end procedure put;

  procedure get ( e : out element_type ) is
  begin
    if count = 0 then wait until count > 0; end if;
    e := elements(tail);
    tail := (tail + 1) mod size;
    count := count - 1;
  end procedure get;
end protected class body BoundedFIFO;
```

The wait statement in each method needs to be guarded by the test, since a wait statement always waits when executed, even if the condition is initially true. When the `put` method blocks due to a full FIFO, another processes can call the `put` method, but it too will block. Eventually, when a process calls the `get` method, it will make space in the FIFO. When that process completes the method, a blocked process can be chosen. That could be another process waiting for entry to either a `put` or `get` method entry, or a process blocked on the `put` method's wait statement. The implementation would resume one of the blocked methods for resumption at some stage during the current delta cycle. A similar sequence of actions occurs when a process blocks on the wait statement in the `get` method due to the FIFO being empty.

Note that there might be several processes blocked at the wait statement in the `put` method. Each time any method of the FIFO object blocks or any process completes a method of the FIFO object, the implementation may check the conditions of the blocked processes. For example, a second process blocking on the `put` method's wait statement may cause the implementation to check the condition for the first process and discover that it is false. Similarly, another process completing the `get` method may cause the implementation either to check the condition of the first process, discover that it is true, and resume the first process; or to allow another process entry into one of the FIFO's methods.

Example 9: Barrier Synchronization

A barrier is a synchronization construct that allows a number of processes to wait until all a ready, and then to proceed. Each process performs its pre-barrier activity, then calls a synchronization method. The method causes the process to block until all processes have called the method. Each process can then resume. A protected class type for a four-entry barrier synchronizer is:

```
type barrier_type is protected class
  procedure synch(id : in natural range 0 to 3);
end protected class barrier_type;

type barrier_type is protected class body
  variable entered : boolean_vector(0 to 3) := (others => false);
  procedure synch(id : in natural range 0 to 3) is
  begin
    entered(id) := true;
    if not and entered then wait until and entered; end if;
    entered(id) := false;
    if nand (not entered) then wait until and (not entered); end if;
  end procedure synch;
end protected class body barrier_type;
```

A barrier object would be instantiated as a shared variable and used by processes as follows:

```
shared variable barrier : barrier_type;
...

p0 : process is
  constant my_id : natural range 0 to 3 is 0;
begin
  ... -- do activity
  barrier.synch(my_id);  -- wait for all to be ready
  -- all ready, so loop back to do activity again
end process p0;
...
```

Within the barrier object, there is a vector of Boolean values, one per barrier entry. The elements are all initialized to false. When a process enters the barrier, it sets its element to true. If not all elements are true, not all processes have entered, so the process waits until all have entered. The last process to enter does not wait. Instead, it clears its element to false and waits for all elements to become false. That process waiting unblocks another process, which clears its element and waits for all elements to become false. Eventually, the last blocked process unblocks and clears its element, so that all elements are now false. The process does not block, but instead completes the method. That causes another process to unblock and complete, and so on.

Note that the second wait is necessary to avoid a race condition. Once all processes have entered the barrier, the last one in clears it’s element. If it then exited, it might then re-enter before the rest have exited. Perhaps a more likely scenario would be for two to exit and then re-enter before any others can exit. The second wait in the method ensures that all elements are cleared before any process can re-enter the barrier.

3.2 Locking Other Objects

A method of a protected class type executes with exclusive access to the object on which the method is

invoked. If the method only needs access to that objects's state, the exclusive access given is sufficient. However, the method may also need exclusive access to another object, such as a parameter of the method. For example, the `clone` method in Example 5 on page 8, if implemented in a protected type, would need exclusive access to the object `o` as well as to the object in which the method is invoked. Even though the state of the parameter object is visible by selection within the method, some other process may have access to the actual object and interfere with access by the `clone` method.

We can provide a mechanism for acquiring exclusive access to an object for use in a sequence of statements, similar to the synchronized statement in Java. A possible syntax is:

```
with object_name protected
  sequential_statement; ...
end protected;
```

The object name must denote an object of a protected type. The executing process acquires exclusive access to the object, and then executes the sequence of statements. When they are complete, exclusive access to the object is rescinded.

Example 10: Protected clone method

We can revise the class in Example 5 on page 8 to make it protected. Doing so requires that the clone method ecquire exclusive access to the parameter object.

```
type C is protected class
  ...
  procedure clone ( o : inout C );
end protected class C;

type C is protected class body
  variable i, j : integer;
  ...
  procedure clone ( o : inout C ) is
  begin
    with o protected
      i := o.i; j := o.j;
    end protected;
  end procedure clone:
end protected class body C;
```

Note that there is scope for deadlock using this mechanism. For example, using the class of Example 10 on page 15, we might declare two shared variables:

```
shared variable a, b : C;
```

One process might call `a.clone(b)`, and at the same time another process might call `b.clone(a)`. The first process might acquire exclusive access to `a`, and then the second process might acquire exclusive access to `b`. From there, neither process can proceed. This potential for deadlock is not specifically a consequence of the proposed synchronization statement. The same situation can arise in protected types as they currently stand, using method calls within method bodies.

4 Constructors

A constructor in a class type definition allows an object to be explicitly initialized, to ensure that its encapsulated state is initially consistent with the class type's intended semantics. A constructor is a special form of method declared in a class type declaration:

```
type class_name is class
  ...
  new class_name ( parameters );
  ...
end class class_name;

type class_name is class body
  ...
  new class_name ( parameters ) is
    ...
  begin
    ...
  end new class_name;
  ...
end class body class_name;
```

A class type may define multiple constructors with different signatures (including a constructor with no parameters). Overload resolution is used to select the appropriate constructor to invoke. The parameter list for a constructor should conform to the rules for function parameter lists.

A constructor for a class type is invoked as a form of expresson primary. It can be used in the initial or default value expression of a declaration:

```
variable var_name : class_name
  := class_name ( actual_parameters );
```

or in an allocator:

```
ptr := new class_name ( actual_parameters );
```

The constructor is executed after the declarative region of the class type is elaborated to create the object.

Example 11: Class for random number state with constructor

We can modify the class type described in Example 4 on page 6 for random numbers to include a constructor that initializes the encapsulated seeds. This obviates a separate call to the `initialize` method. The class type declaration is revised as follows:

```
type RandomType is class
  new RandomClass (seed1, seed2 : positive);
  ...
end class RandomType;

type RandomType is class body
```

```
  variable randomSeed1 : positive := 7;
  variable randomSeed2 : positive := 1;

  new RandomType(seed1 : positive := 7; seed2 : positive := 1) is
  begin
    randomSeed1 := seed1; randomSeed2 := seed2;
  end new RandomClass;

  ...
end class RandomType;
```

A variable of the class type could be declared and initialized as follows:

```
variable randStream : RandomType := RandomType(1, 100);
```

Similarly, a variable could be allocated and initialized:

```
type RandomPtr is access RandomType;
variable nextRandStream : RandomPtr;
...

nextRandStream := new RandomType(2, 2);
```

In the case of a class type having a nested class-typed object, the constructor for the nested object's class type is invoked when the nested object is elaborated. The order of initialization is first to elaborate the enclosing object. As part of that elaboration, the variable declaration for the nested object is elaborated, invoking the constructor of the nested object's class type. Next, the constructor of the enclosing object's class type is invoked. As part of initializing the variables of the enclosing object, the constructor can invoke methods of the nested object.

4.1 Default Constructors

If a class type includes no explictly declared constructor with no parameters, then a default constructor with no parameters is implicity declared. If the class is an extension of a superclass (see Section 5 on page 18), the default constructor simply invokes the superclass constructor with no parameters, whether that superclass constructor be a default constructor, an explicitly declared constructor with no parameters, or an explicitly declared constructor with all parameters having default values. This ensures that superclass initialization is performed properly. If the class is not an extension of a subclass, the statement part of the default constructor is empty.

The constructor with no parameters, whether explictly or implicitly declared, is used to construct the default initial value for a declared object of the class type.

4.2 Invoking Constructors using `this`

The first statement in a constructor body may be an invocation of another constructor of the same class, written using the keyword `this`, as follows:

```
new class_name( actual_parameters ) is
begin
  this( actual_parameters );
  ...
end new class_name;
```

The constructor invoked must not be the same constructor, either directly or indirectly, since there is no mechanism to terminate constructor recursion. Alternatively, the first statement may be an invocation of a superclass constructor (see Section 5.2 on page 21).

4.3 Destructors

C++ provides for destructors in class definitions, allowing resources such as storage created for an object to be reclaimed. However, other languages, includign Java and SystemVerilog, have no explicit deallocation of class-typed objects, instead relying on garbage collection to reclaim storage resources when no references to allocated storage remain. This form of memory management is less error prone. Moreover, modern garbage collection techniques have good performance. Hence, we prefer to deprecate explicit deallocation of storage.

Implicit storage reclaimation can be extended to all access types in VHDL, not just access types designating class-typed objects. The semantics of the implicit `deallocate` procedure can be revised simply to update the access-typed parameter with the value `null`, and not to specify reclaimation of storage. An implementation may be permitted to use automatic storage reclaimation.

5 Inheritance

Inheritance allows a new class type, a subclass, to be defined as an extension of a superclass. All of the declarations of the superclass are included implicitly in the subclass. The subclass can also include additional declarations. An operation whose profile conforms with one inherited from the superclass overrides the declaration from the superclass.

We can extend class type declarations as described in Section 2.3 on page 5 to allow naming of a superclass to be extended. Possible syntax for a class type declaration is

```
type subclass_name is class extends superclass_name
  ...
end class subclass_name;
```

In earlier proposals, the keyword “new” was used in place of “extends.” However, introduced one more different use for the keyword “new” in addition to its use for allocators, constructors, and instantiation. The keyword “extends” is used in other languages, including Java and SystemVerilog, so its use in VHDL would be familiar to users.

The declarative part of the superclass declaration is implicitly included in, and extended by, the declarative part of the subclass declaration. Homographs of method declarations included from the superclass are

allowed in the subclass, in which case they hide the included declaration in the extension. Such a homograph must have a profile that conforms with that of the hidden declaration, to ensure that it can be called in the same way as the hidden method. Moreover, a pure-function method of a superclass must only be overridden by a pure function, to ensure that it also can be invoked on a read-only object. The scope of use clauses implicitly included is limited to the included declartive part in which the use clause originally appeared. That is, the scope of a use clauses in a superclass declaration does not extend into the declartive part of a subclass.

The declarative part of the superclass body is similarly implicitly included in, and extended by, the declarative part of the subclass body. All names declared in the included declarative part are hidden in the extension. Thus, declarations in the extension may be homographs of declarations in the included part. A declaration in the extension must not, however, be a homograph of a visible declaration from the subclass declaration (including any that are visible by inclusion from the superclass declaration).

Example 12: Network packet extension

The `network_packet` class type from Example 3 on page 5 can be extended to define different subclasses of packets:

```
type control_packet is class extends network_packet
  procedure compute_crc;
  procedure set_control (ctrl : natural);
end class control_packet;

type data_packet is class extends network_packet
  variable length : positive;
  procedure compute_crc;
  procedure set_data (index : natural; data : byte);
end class data_packet;
```

The subclass `control_packet` inherits the `src` and `dst` variables and overrides the `compute_crc` method. It adds the `set_control` method. The `compute_crc` declaration from the superclass is hidden in the subclass body. The subclass body must include a body for the overriding `compute_crc` method, as well as for the `set_control` method.

The subclass `data_packet` likewise inherits the `src` and `dst` variables and overrides the `compute_crc` method. It adds the `length` variable and the `set_data` method. The subclass body must include method bodies for `compute_crc` and `set_data`.

5.1 Subclass Visibility

If a subclass type is declared within a subprogram, it must be directly visible or visible by selection at every place in which the superclass is directly visible or visible by selection. (This is a refinement of a similar rule in Ada specifying accessibility levels of subclass and superclass type declarations.) The rationale is that at a place where the superclass type is visible, a polymorphic reference to the superclass type can be declared (see Section 7 on page 26). For example, an object of the subclass type can be referenced by a pointer designating the superclass type, and so a method of the subclass invoked by dynamic dispatch using the pointer. If the subclass is declared nested inside a subprogram (and hence not visible at the outer level), the method can have in its closure items declared within the subprogram. After the subprogram returns, subclass object persists, but the items in its closure do not. Precluding declaration of a subclass in

the subprogram avoids this problem.

Example 13: Problems due to invalid accessibility level for a subclass

The following architecture body illustrates the problem that could arise were a subclass declaration nested inside a subprogram allowed to extend a subperclass declared outside the subprogram:

```
architecture a is

    type C is class
      procedure M;
    end class C;

    type C_ptr is access C; -- polymorphic reference

    procedure p ( variable v : out C_ptr ) is

      variable local : integer;

      type SubC is class extends C
        procedure M; -- overrides M from C
      end class SubC;

      type SubC is class body
        variable i : integer;
        procedure M is
        begin
          i := local;
        end procedure M;
      end class body SubC;

    begin
      v := new SubC;
    end procedure p;

  begin

    proc : process is
      variable my_C : C_ptr;
    begin
      p(my_C);
      my_C.M; -- variable i in p is no longer live!
      …
    end process proc;

  end architecture a;
```

In the architecture declarative region, the class type `C` is declared, as is an access type `C_ptr` designating `C`. A pointer of type `C_ptr` can designate an object of type `C` or any subclass of `C`. Within the procedure `p`, there are a local variable `local` and declaration of a subclass `SubC` of `C`. The method `M` of `SubC` overrides the method `M` of the superclass. The overriding method body copies the value of the procedure's variable `local` to the subclass object's variable `i`. The procedure `p` has a parameter `v` to which it assigns a pointer initialized to designate an object of the subclass type.

The process proc in the architecture body calls the procedure to allocate a subclass object and return a pointer to it. The process then calls the method `M`, causing dynamic dispatch to the subclass method `M`. That method tries to copy the value of `local`, whose lifetime has now expired.

This could be made to work by specifying that activation records referenced in the closure of live objects be retained. Some languages take this approach, though the run-time cost of doing so is high. Among other considerations, it precludes a simple stack discipline for subprogram activation records. The proposed rule limiting the places where subclasses can be declared obviates retention of activiation records.

5.2 Constructors and Inheritance

A subclass does not inherit constructors from its superclass. That is, the constructors declared in the superclass declarative part that is included in the subclass are not directly visible in the subclass. However, the first statement in a subclass constructor body may be an invocation of a superclass constructor, written using the keyword `super`, as follows:

```
new subclass_name( actual_parameters ) is
begin
  new subclass_name( actual_parameters ) is
  begin
    super( actual_parameters );
    ...
  end new subclass_name;
  ...
end new subclass_name;
```

If the first statement in a subclass constructor body is not a constructor invocation written using `this` or `super`, an implicit superclass constructor invocation with no actual parameters is assumed. This is similar to the superclass constructor invocation in a default constructor (see Section 4.1 on page 17). It is illegal to use `super` in this way in a constructor of a class that does not extend a superclass.

5.3 Accessing Superclass Declarations using `super`

The keyword `super` can be used as a name within a method body to denote the object on which the method is invoked. The type of the name is the superclass of the class containing the method. The name denotes a view of the object as an object of the superclass type. Thus, the name can be used as a prefix to access superclass variables or methods, even if they are hidden by declarations within the subclass. In the case of method invocation using `super`, dynamic dispatching does not occur, since the type of `super` is a specific class type, not a class-wide type. It is illegal to use the name `super` in a method of a class that does not extend a superclass.

5.4 Abstract Class Types and Methods

In many object-oriented applications, it is desirable to factor out common variables and methods into a superclass. If that superclass does not represent anything concrete in the application, it is an “abstract”

class type and should not be used as the type of an object. Rather, only its subclasses should be used as object types.

We might also include declarations of methods that are to be provided in all subclasses, but whose bodies are different in each subclass. Such “abstract” methods should not have a body in the superclass. As a corollary, a superclass containing abstract methods would have to be an abstract superclass.

We can extend class type declarations to define abstract class types and abstract methods. A possible syntax for an abstract class type is

```
type class_name is abstract class
  ...
end class class_name;
```

and for an abstract class type that extends a superclass:

```
type subclass_name is abstract class extends superclass_name
  ...
end class subclass_name;
```

Within an abstract class type declaration, a possible syntax for abstract methods is:

```
procedure proc_name ( parameters ) is abstract;

function func_name ( parameters ) return result_type is abstract;
```

A class type that contains one or more abstract methods must be an abstract class, since such a calls cannot be instantiated. An abstract method is inherited by a subclass and can be overridden by a non-abstract method in the subclass. If all abstract methods of the superclass are overridden and the subclass declares no further abstract methods, the subclass can be non-abstract; otherwise, it too must be abstract.

Example 14: Abstract network packet class

Referring back to the network packet examples, the `network_packet` class type should be abstract and provide `compute_crc` as an abstract method. Both the `control_packet` and `data_packet` class types make the superclass concrete by overriding the `compute_crc` abstract method with a concrete body. The declarations could be revised to:

```
type network_packet is abstract class
  variable src, dst : natural;
  procedure compute_crc is abstract;
end class network_packet;
```

5.5 Assignment and Type conversions

An object of a given class type can be assigned to a variable of that class type, but cannot be assigned directly to a variable of a superclass or subclass. We can extend the semantics of a type conversion to provide a view of the object as being of the superclass. In that case, only those public variables and methods declared by the superclass are visible for the object. The type converted object can be assigned to a variable of the superclass, in which case just the state of the superclass is copied from the type-converted

object to the assignment target. Type conversion of an object to a subclass would not be appropriate, since the object would not have state corresponding to the subclass.

We can extend selected names to allow a type conversion as a prefix. That would allow a type-converted object name to be used as the prefix in a method call, allowing a superclass method to be invoked on the object.

We can also extend alias declarations that declare object aliases to allow for a view conversion of the aliased object. We would do so by including a subtype indication denoting a superclass in the alias declaration. However, assignment to a target that is a view of an object, should not be allowed. The viewed type is potentially different from the specific class type of the object. Assignment to the view would be a partial assignment if the specific class of the object is a subclass of the class used in the alias declaration. A partial assignment could lead to inconsistent state for the object.

Example 15: Network packet assignment and type conversion

Suppose we extend the `control_packet` class type as follows:

```
type counting_control_packet is class extends control_packet
  variable hop_count : natural;
  procedure compute_crc;
end class counting_control_packet;
```

Within a process, we have variables declared as:

```
variable ctrl : control_packet;
variable counting_ctrl : counting_control_packet;
```

We could then make the following assignment to copy the variables defined for the `control_packet` class type from `counting_ctrl` to `ctrl`. The `hop_count` variable (and any variables declared in the body of `counting_control_packet`) would not be copied.

```
ctrl := control_packet(counting_ctrl);
```

We could perform the following method call:

```
control_packet(counting_ctrl).compute_crc;
```

This would invoke the method defined in `control_packet` on the `counting_ctrl` object.

Note that assignment to shared variables of protected type is prohibited, since the assignment operation does not imply any concurrency control.

6 Interfaces

We can adopt the Java notion of interfaces to provide contracts for class types. An interface declares the specifications for methods that a class must subsequently implement. The interface methods are implicitly abstract methods, though they may be explicitly declared abstract. A possible syntax for an interface declaration is

```
type interface_name is interface
  ...  -- method declarations
end interface interface_name;
```

No body is permitted for an interface declaration, since all of the methods are abstract. No constructors are permitted, since there is no concrete implementation to construct.

An interface can also extend one or more another interfaces, called superinterfaces, as follows:

```
type subinterface_name is interface
  extends superinterface_name1, superinterface_name2, ...
  ...  -- method declarations
end interface subinterface_name;
```

As with class extension, the subinterface inherits the method declarations of the superinterfaces, can hide them with homographs (which must have conforming profiles), and can add further methods. If two homographs are inherited from different superinterfaces, they must have conforming profiles. The subinterfaces can be considered to inherit intersecting contracts from the superinterfaces, resulting in a single abstract method being defined in the subinterface.

A class can implement one or more interfaces, in which case the class body must provide bodies for the methods inherited from the interfaces. A possible syntax for specifying implementation of interfaces is:

```
type class_name is class
  implements interface_name1, interface_name2, ...
  ...
end class class_name;
```

A subclass may extend a superclass as well as implementing interfaces:

```
type subclass_name is class extends superclass_name
  implements interface_name1, interface_name2, ...
  ...
end class subclass_name;
```

Inheritance of methods from interfaces is similar to inheritance of abstract methods from a superclass. If the class is declared abstract, a method inherited from an interface can remain abstract in the class, thus deferring implementation to a subclass.

A method inherited from an interface can also be implemented by a method inherited from a superclass. If multiple interfaces implemented by a class all define a method with a given parameter and result type profile, those methods must all have conforming profiles. In that case, a single method with a conforming profile, either declared in the class or inherited from a superclass, can implement all of the methods. The interfaces, in such a case, can be considered to impose intersecting contracts on the implementing class.

Example 16: Interfaces for interprocess communication

We can define interfaces for abstract communications channels. One process can put data to a channel, and another can get data from the channel. The two interfaces are:

```
type putable is interface
```

```
  procedure put ( e : in element_type );
  procedure put_if_not_full ( e : in element_type;
                              ok : out boolean );
end protected interface putable;

type getable is interface
  procedure get ( e : out element_type );
  procedure get_if_not_empty ( e : out element_type;
                               ok : out boolean );
end protected interface getable;
```

Different forms of communications channel can implement these interfaces differently. One example is a single-entry mailbox:

```
type mailbox is protected class implements putable, getable
  function flag_up return boolean;
  procedure put ( e : in element_type );
  procedure put_if_not_full ( e : in element_type;
                              ok : out boolean );
  procedure get ( e : out element_type );
  procedure get_if_not_empty ( e : out element_type;
                               ok : out boolean );
end protected class mailbox;

type mailbox is protected class body

  variable element : element_type;
  variable full : boolean := false;

  function flag_up return boolean is
  begin
    return full;
  end function flag_up;

  procedure put ( e : in element_type ) is
  begin
    if full then wait until not full; end if;
    element := e; full := true;
  end procedure put;

  procedure put_if_not_full ( e : in element_type;
                              ok : out boolean ) is
  begin
    if full then
      ok := false; return;
    end if;
    element := e; full := true; ok := true;
  end procedure put_if_not_full;

  procedure get ( e : out element_type ) is
  begin
    if not full then wait until full; end if;
    e := element; full := false;
  end procedure get;
```

```
  procedure get_if_not_empty ( e : out element_type;
                               ok : out boolean ) is
  begin
    if not full then
      ok := false; return;
    end if;
    e := element; full := false; ok := true;
  end procedure get_if_not_empty;

end protected class body mailbox;
```

Another example is the bounded FIFO, described in Example 8 on page 13. It could be revised as follows:

```
type BoundedFIFO is protected class implements putable, getable
  procedure put ( e : in element_type );
  procedure put_if_not_full ( e : in element_type;
                              ok : out boolean );
  procedure get ( e : out element_type );
  procedure get_if_not_empty ( e : out element_type;
                               ok : out boolean );
end protected class BoundedFIFO;
```

A different example is a random value generator that just implements the getable interface. It would appear as a channel that is never empty.

```
type RandomElementGenerator is class implements getable
  procedure get ( e : out element_type );
  procedure get_if_not_empty ( e : out element_type;
                               ok : out boolean );
end class RandomElementGenerator;
```

7 Polymorphism and Dynamic Dispatch

Polymorphism allows an object of a subclass to be treated as an object of a superclass. For statically typed objects, type conversion (see Section 5.5 on page 22) allows invocation of superclass methods of an object, even if the methods are overridden in the subclass. However, object-oriented languages also provide mechanisms for a name to denote an object of a given class type or any subclass of that class type, and for a method call to invoke the overriding method for the specific subclass of the object. The determination of the method is done at runtime, a mechanism called dynamic dispatch. This allows code to be written that works for any subclass of the given class type, even a subclass written after the invoking code and not visible at the place of the method call.

We can allow a variable name to be polymorphic provided its declaration doesn’t imply creation of storage for the denoted variable. The places where this occurs are interface variables (where the name denotes a reference to an actual object), and access variables (where the name denotes a pointer to allocated storage). For these cases, we can specify that the name be a polymorhpic reference to an object of a given class type, called the root class of the polymorphic reference. The root class can be an abstract or concrete class type, or can be an interface type.

Declaring an interface variable to be of class type `C` means that the actual can be of class type `C` or any subclass. Within the subprogram or entity, we can access any publicly visible variable of the actual variable declared in class type `C`, since all subclasses inherit such variables. We can invoke any method declared in class type `C`, including abstract methods, since all subclasses either inherit or override such methods. In this case, dynamic dispatch is used to invoke the method visible for the specific class of the actual variable.

Declaring an access type with class type `C` as the designated type means that an access value of the access type can point to an object of class type `C` or any subclass. We can use the access value as a prefix for a publicly visible variable, declared in class type `C`, of the designated object. We can similarly use the access value as a prefix in a method call to invoke any method declared in class type `C`, in which case dynamic dispatch would be used to select the appropriate method based on the specific class type of the designated object.

For an access-type variable `P` with designating class type `C`, `P.all` would denote the object designated by `P`. The type of `P.all` would not be statically known, beyond the fact that it is `C` or a subclass; that is, `P.all` would be a polymorphic reference. Thus, publicly visible variables declared in class type `C`, could be accessed using `P.all` as the prefix, and methods declared in class type `C` would be invoked using dynamic dispatch.

Example 17: Dynamic dispatch for network packets

Given the definitions of classes for network packets in previous examples, we can declare a procedure as follows:

```
procedure munge_packet (pkt : inout network_packet) is
begin
  pkt.compute_crc;
  ...
end procedure munge_packet;
```

If we declare variables as before:

```
variable ctrl : control_packet;
variable counting_ctrl : counting_control_packet;
```

we can call the procedure with these variables:

```
munge_packet (ctrl);
munge_packet (counting_ctrl);
```

In the first call, the formal parameter refers to an actual of class type `control_packet`, so the call to `compute_crc` dispatches to the overriding version declared in `control_packet`. In the second call, the formal parameter refers to an actual of class type `counting_control_packet`, so the call to `compute_crc` dispatches to the overriding version declared in `counting_control_packet`.

We can declare an access type and a variable of the access type as follows:

```
type packet_ptr is access network_packet;
variable ptr : packet_ptr;
```

We can then create a new `control_packet` typed object and invoke the `compute_crc` method declared in `control_packet` as follows:

```
ptr := new control_packet;
```

```
ptr.compute_crc;
```

Subsequently, we can assign to the access variable a pointer to an object of a different class type:

```
ptr := new data_packet;
ptr.compute_crc;
```

In this case, the call to the `compute_crc` invokes the version declared in the class type `data_packet`. We can refer to `ptr.src`, since all subclasses of `network_packet` have that variable. However, we cannot refer to `ptr.length`, since not all subclasses of `network_packet` have that variable.

7.1 Type Conversions and Aliases

We can use a type conversion to view an object denoted by a polymorphic reference as being of a specific class. Method invocation with such a type conversion as a prefix avoids dynamic dispatch, instead invoking the method defined for the specific class type. In some cases, a check is required to ensure that the target type is appropriate for the operand. The check may be required at run-time or elaboration time, depending on when the specific type of the operand can be determined.

Suppose we have a class type `C`, a superclass `C_super`, and a subclass `C_sub`. A polymorphic interface variable `V` with root class `C` can be converted to type `C` or `C_super` without a check. Similarly, for an access-typed variable `P` with `C` as the designated type, the designated object `P.all` can be converted to type `C` or `C_super` without a check. The variable `V` or the designated object `P.all` can be converted to type `C_sub`, but it is an error if the referenced object is not of type `C_sub` or a subclass of `C_sub`.

Conversion of access values is also allowed, and yields a polymorphic reference whose root type is the desginated class type of the target access type. Such a conversion involves a run-time check if the conversion between designated types requires a run-time check.

Similarly, an object alias declaration can include a subtype indication that implies a type conversion. If the conversion would require a check, then a check must be performed when the alias declaration is elaborated.

7.2 Dispatch using `this`

The keyword `this` can be used as a name within a method body to denote a polymorphic reference to the object on which the method is invoked, with the root type of the reference being the class type containing the method. Thus, the name can be used as a prefix to access variables or methods, even if they are hidden by declarations within the method. In the case of method invocation using `this`, dynamic dispatching occurs, since the reference is polymorphic.Type conversion can be used to obtain a view of the object. Note that if `this` is used directly or indirectly in a constructor to dynamically dispatch to a subclass method, construction of the subclass state may not be complete.

7.3 Assignment to Polymorphic References

Assignment to a target denoting a polymorphic reference must only be allowed when the right-hand-side value is of the same specific class type as the target. The reason is that the target ultimately refers to a variable, either explicitly declared to be of some specific class type, or allocated and of some specific class type. The right-hand-side value must match that specific type exactly so that the target is not partially assigned, possibly leading to inconsistent state for the object.

Assignment to a target that is a view of a polymorphic reference, declared using an alias, should not be allowed. The viewed type is potentially different from the specific class type of the object. Assignment to the view would be a partial assignment if the specific class of the object is a subclass of the class used in the alias declaration.

Assignment of access values is permitted in certain cases. If the root class of the target is the same as or a superclass of the root class of the right-hand-side value, assignment is permitted and no check is needed. If the root class of the target is a subclass of the root class of the right-hand-side value, a check is needed to ensure that the right-hand-side value is either null or designates an object whose specific class type is the same as or a subclass of the root class of the target. No other access value assignments are allowed.

8 Inheritance and Protection

An issue to consider is whether protection can be added or removed as part of extension of a superclass to a subclass. The effect of protection is for method invocation to require exclusive access to an object, and to permit a method to block at a conditional wait statement. Protection also involves limitations on the class. Specifically, a protected class cannot have variables in its declaration, since atomic access to publicly visible variables cannot be guaranteed. Further, methods cannot have parameters or result types that are of or include access values, since that would provide a means of communicating access values between processes, and atomic access to the designated variables cannot be guaranteed. Lack of protection on a class implies that the class cannot be used for a shared variable, since no concurrency control is assumed. Inclusion of protection means that the class can be used for both shared and non-shared variables. In the case on non-shared variables, exclusive access is guaranteed, since such variables can only be declared in regions accessible to a single process.

Keeping protection when extending a protected class type to give a protected subclass presents no problems. Superclass methods that assume concurrency control do not have that assumption violated in the subclass, and subclass methods can assume concurrency control for inherited methods. Likewise, not adding protection when extending a non-protected class type to give a non-protected subclass presents no problems. Concurrency control is not an issue for superclass or subclass methods.

It would be desirable to be able to extend a non-protected class by adding protection. This would allow, as an example, a container class developed for single-process use to be used for a shared variable in a concurrent context. The effect would be to require acquisition on method calls so that the methods could update the object’s state without interference. The difficulty is that concurrency control imposes some overhead on execution of a method call. It would be desirable to be able to extend the non-protected superclass, but to leave use of the superclass unchanged. That is, the semantics should be defined in such a way that methods of a non-protected superclass can be invoked without overhead of having to allow for possibly concur-

rency control in subclasses.

If this is not feasible, an alternative is to wrap an object of the non-protected class in a protected class, and for the methods of the protected class to delegate to the encapsulated object. This pattern would be simplified if both the non-protected and the protected classes abstracted their set of methods into an interface that they both implemented.

More work to do here ...

9 Class Types and Signals

In principle, a signal could be declared to be of a class type. The kernel variable defining the driving and effective values for the signal would hold objects of the class type. Likewise, the value part of transactions on drivers for the signal would hold objects of the class type. During signal assignment, the predefined equality operator would be applied to determine which transactions to mark. During signal update, the predefined equality operator would be used to determine whether an event occurs. The signal name, denoting the current-value object, could be used as a prefix to read any publicly visible variables declared in the class or to invoke pure-function accessor methods.

An important restriction on signals is that their type must not be an access type or a composite type that has an access-typed subelement. The rationale is to avoid communication of pointers between processes, resulting in shared access to the designated object. Were class types permitted for signals, similar restrictions would be required. Encapsulated variables and method parameters must not be of or contain access types.

Signal parameters are currently passed by reference. For consistency with variable parameters, class-typed signals would be polymorphic. A similar correspondence would be established between class-typed signal ports and variable ports.

Given these considerations, the complexity involved in their semantics, and the unlikely use cases, it is probably best not to allow class-typed signals.

10 Class Types and Generics

In VHDL-2007, the generic mechanism was extended by allowing generic lists in packages and subprograms and by allowing generic types, subprograms, and packages. The considerations motivating these changes also apply to class types. We discuss the interactions between class types and the extended generics in this section.

10.1 Defining Classes in Packages

One way to make a class type generic is to define it within a package that has a generic list. Different instances of the class can then be defined by instantiating the package with different actual generics. If *post hoc* extension is required, a second package can be declared with a formal generic package referencing the first package.

Example 18: Packages for interprocess communication

In Example 16 on page 24 we defined interfaces and classes for abstract communications channels. We can make the interfaces generic with respect to the type of the communicated data by placing the interface declarations in an uninstantiated package:

```
package comms_interface_pkg is
  generic ( type element_type );

  type putable is interface
    procedure put ( e : in element_type );
    procedure put_if_not_full ( e : in element_type;
                                ok : out boolean );
  end protected interface putable;

  type getable is interface
    procedure get ( e : out element_type );
    procedure get_if_not_empty ( e : out element_type;
                                 ok : out boolean );
  end protected interface getable;

end package comms_interface_pkg;
```

We can then define a class that implements the interfaces generic by placing it in a package also. That package must have a formal generic package referring to an instance of the package defining the interfaces:

```
package mailbox_pkg is
  generic ( type element_type;
            package element_comms_interface_pkg is
              new work.comms_interface_pkg
                generic map ( element_type => element_type ) );

  type mailbox is protected class
    implements element_comms_interface_pkg.putable,
               element_comms_interface_pkg.getable
    function flag_up return boolean;
    procedure put ( e : in element_type );
    procedure put_if_not_full ( e : in element_type;
                                ok : out boolean );
    procedure get ( e : out element_type );
    procedure get_if_not_empty ( e : out element_type;
                                 ok : out boolean );
  end protected class mailbox;

end package mailbox_pkg;
```

In order to use the mailbox class, we would instantiate the interfaces package and then name the instance as the actual generic in an instance of the mailbox class:

```
type msg_type is ...

package msg_comms_interface_pkg is new work.comms_interface_pkg
  generic map ( element_type => msg_type );

package msg_mailbox_pkg is new work.mailbox_pkg
  generic map ( element_type => msg_type,
                element_comms_interface_pkg => msg_comms_interface_pkg );

component producer is
  port ( shared variable data : inout msg_comms_interface_pkg.putable );
component consumer is
  port ( shared variable data : inout msg_comms_interface_pkg.getable );
shared variable msg_mbx : msg_mailbox_pkg.mailbox;
```

10.2 Uninstantiated Class Types

Work in progress...

The earlier proposal for class types in VHDL included uninstantiated classes, that is, classes that include generic lists. Such a class must be instantiated with actual generics before being used as the type of a variable. Given that the same effect can be achieved by encapsulating the class type in an uninstantiated package, as described in Section 10.1 on page 31, is there a need for uninstantiated class types as well?

10.3 Formal Generic Class Types

Work in progress...

The earlier proposal for class types in VHDL included formal generic class types, that is, formal generic types that are defined to be class types with specified ancestry. The proposal illustrated how this mechanism could be used to express a form of mix-in inheritance. Is this mechanism required? What real-user use models are there motivating it?

11 Validation Through Use Cases

In this white paper, we have explored a number of detailed aspects of the proposed language extensions, including interaction with existing language features. Nonetheless, the language design should be validated through case studies. Two such validation exercises are proposed:

- A class library for collections and interprocess communication communication

- A verification framework, such as those defined in the the Advanced Verification Methodology (AVM) and the Verification Methodology Manual (VMM).

These studies will expose any language design issues overlooked in this white paper and will demonstrate application of the features for real use cases.