A Comparison of Alternative Extensions for Data Modeling in VHDL

Peter J. Ashenden <petera@cs.adelaide.edu.au>

Philip A. Wilsey <p.wilsey@ececs.uc.edu>

Technical Report TR-02/97

Department of Computer Science
The University of Adelaide, SA 5005
Australia

Technical Report TR-203/05/97/ECECS

Department of Electrical and Computer Engineering
and Computer Science
University of Cincinnati,
PO Box 210030
Cincinnati, OH 45221−0030
USA

This work was partially supported by Wright Laboratory
under USAF contract F33615−95−C−1638

Abstract

High-level modeling of digital systems involves behavioural descriptions that use abstract representations of the data manipulated by the system. For a hardware description language to be used successfully for high-level modeling, it must provide appropriate features, such as object-oriented features, for managing the complexity in a design.

A number of proposals for object-oriented extensions to VHDL deal with object-oriented extensions to support the modeling of data. They provide features for expressing data abstraction, encapsulation, and inheritance with polymorphism, features that are central to object oriented techniques. There are two approaches to object-oriented data modeling: *class-based* and *programming by extension*. This report compares the two approaches in the context of hardware modeling in VHDL. It outlines the two approaches and presents examples illustrating how they might be included in VHDL. It discusses the issue of integration with signal semantics, and compares the way in which the approaches deal with encapsulation, initialization or objects, and inheritance. The report concludes that, while both approaches are viable, the programming by extension approach is preferred.

1 Introduction

As the complexity of digital systems is increasing, designers are becoming more reliant on high-level modeling as a stage in the design flow. High-level modeling involves behavioural descriptions that use abstract representations of the data manipulated by the system. The focus is on capturing the required functionality of the system without committing to implementation details, such as binary encoding of data and communication protocols between components.

For a hardware description language to be used successfully for high-level modeling, it must provide appropriate features for managing the complexity in a design. The software-engineering community has adopted object-oriented techniques as the means of managing complexity in software systems [2]. Since the task of high-level modeling using a hardware description language is very similar to software development, it is appropriate to consider introduction of object-oriented features into a hardware description lanugage such as VHDL.

A number of proposals for object-oriented extensions to VHDL have been published, and are surveyed in [1]. A number of them [4, 8, 10, 11, 13] deal with object-oriented extensions to support the modeling of data. They provide features for expressing data abstraction, encapsulation, and inheritance with polymorphism, features that are central to object oriented techniques.

There are two approaches to object-oriented data modeling: *class-based* and *programming by extension*. These approaches are briefly outlined in [1]. The proposals of Radetzki *et al* [10] and Willis *et al* [13] are class-based, whereas those of Dunlop [4], Mills [8], and Schumacher and Nebel [11] are based on programming by extension.

This report compares the two approaches in the context of hardware modeling in VHDL. Section 2 first outlines the two approaches and presents examples illustrating how they might be included in VHDL. Section 3 then discusses the issue of integration with signal semantics. The next three sections compare the way in which the approaches deal with issues relating to object-oriented modeling of data: Section 4 compares encapsulation, Section 5 compares initialization of objects, and Section 6 compares inheritance. In Section 7, we conclude that, while both approaches are viable, the programming by extension approach is preferred.

2 Outline of the two approaches

2.1 The class-based approach

The class-based approach is influenced by programming languages such as Simula [3], Smalltalk [5], C++ [12] and Java [6], in which a there is a specific language construct to encapsulate the state and operations of a class. Derived classes inherit state and operations from superclasses, with possible augmentation or modification. An object is an instance of a class, and contains storage for the state specified in the class. Depending on the language, the state may or may not be directly accessed. It may be strongly encapsulated, in which case only the operations of the class may access the state of an instance. Operations are typically represented as subprograms that have an object instance as an implicit parameter. Invoking an operation involves identifying both the operation and the object upon which the operation is to be performed.

In a conventional programming language, this model of classes and objects is quite workable. Such languages only have one kind of object, namely one that is an abstraction of a machine storage

location. Such an object can be read, with the retrieved value being used in an expression. An object can be assigned to, in which case the stored state is immediately updated and subsequent reads return the updated value.

One of the main issues in a conventional programming language with classes is how to deal with assignment and equality. In general they cannot be predefined, since the semantics intended by the programmer may be different from simple shallow copy and element-wise comparison. Further, if the state includes dynamically allocated storage, overwriting an object that is the target of assignment may be undesirable. To do so would create garbage that, in many run-time environments, is irretrievable. C++ deals with these problems by requiring the programmer to include constructor and descructor functions in a class.

As an example of how a class-based extension might be used, consider an ADT for complex numbers.

```
type complex is class
    private variable re, im: real;
    public function real_part return real is
    begin
        return re;
    end function real_part;
    public function imag_part return real is
    begin
        return im;
    end function imag_part;
    public procedure complex ( new_re : real; new_im : real := 0.0 ) return complex is
    begin
        re := new_re;  im := new_im;
    end procedure complex;
    public procedure ":=" ( value : in complex) is
    begin
        re := value.re;  im := value.im;
    end procedure ":=";
    public function "=" ( right : in complex ) return boolean is
    begin
        return re = right.re and im = right.im;
    end function "=";
    public function "+" ( right : in complex ) return complex is
    begin
        return complex ( re + right.re, im + right.im );
    end function "+";
    ...    -- other arithmetic operations
    public procedure clear_to_zero is
    begin
        re := 0.0;  im := 0.0;
    end procedure clear_to_zero;
end class complex;
```

In this particular example, shallow-copy assignment and element-wise comparison would suffice. However, user-defined operations are shown to illustrate the possible mechanism. The procedure complex is intended to be a constructor, allowing creation of anonymous complex values. For example:

```
constant c_zero : complex := complex(0.0, 0.0);
constant c_1 : complex := complex(1.0 );
constant c_i : complex := complex(0.0, 1.0);
```

Examples of instantiation of the class are shown later in Section 3.

2.2 The programming by extension approach

Several of the proposals for object-oriented extensions to VHDL follow the approach of Ada-95 [7] and for defining classes. The approach used by Oberon-2 [9] is similar. The Ada-95 approach is based on extensible tagged record types defined in packages. A package contains the declaration of a tagged record type and the primitive operations for the type. The primitive operations include one or more parameters of the declared type, or have a function result of the declared type. In the same or another package, a new tagged record type is derived from the original type, inheriting the elements of the original type and adding further elements. The primitive operations from the original type are also inherited for the new type. They may be overridden with alternate versions, and additional primitive operations added.

Rather than defining a class which encapsulates a stored object, the Ada-95 approach allows definition of an abstract data type class. The user then creates objects which belong to the abstract data type and invokes operations defined for the type, supplying the object as an actual parameter. If the concrete details of the type are made private within the defining package, the user may generally only manipulate the object using the defined operations. The assignment operation is predefined, implementing shallow copy of record elements. If that is inappropriate, the type may be made limited, in which case assignment is not allowed and the type definition must include operations to copy objects. In the Ada-95 approach, the equality operator is also predefined, implementing element-wise comparison. If that is inappropriate, the type definition can include an overloaded equality operator.

As an example, consider the complex number type recast as an ADT using the programming by extension approach. (The type in this example is not tagged, since inheritance is not relevant.)

```
package complex_numbers is

    type complex is private;

    function real_part ( c : in complex ) return real;

    function imag_part ( c : in complex ) return real;

    function make_complex ( new_re : real; new_im : real := 0.0 ) return complex;

    function "=" ( left, right : in complex ) return boolean;

    function "+" ( left, right : in complex ) return complex;

    ...    -- other arithmetic operations

    procedure clear_to_zero ( variable c : out complex );

    procedure clear_to_zero ( signal c : out complex );

private

    type complex is record
        re, im : real;
    end record complex;

end package complex_numbers;
```

Note that the current VHDL rules for overloading prevent the two declarations of clear_to_zero as shown, since they are considered to be homographs. A possible extension assumed here is to take account of the class of parameters in determining homographs. Since a formal variable parameter can only be associated with a variable, and a formal signal parameter can only be associated with a signal, the context is sufficient to resolve the overloading.

A possible package body for the ADT is:

```
package body complex_numbers is
    function real_part ( c : in complex ) return real is
    begin
        return c.re;
    end function real_part;
    function imag_part ( c : in complex ) return real is
    begin
        return c.im;
    end function imag_part;
    function make_complex ( new_re : real; new_im : real := 0.0 ) return complex is
    begin
        return (new_re, new_im);
    end procedure make_complex;
    function "=" ( left, right : in complex ) return boolean is
    begin
        return left.re = right.re and left.im = right.im;
    end function "=";
    function "+" ( left, right : in complex ) return complex is
    begin
        return ( left.re + right.re, left.im + right.im );
    end function "+";
    ...    -- other arithmetic operations
    procedure clear_to_zero ( variable c : out complex ) is
    begin
        c := (0.0, 0.0);
    end procedure clear_to_zero;
    procedure clear_to_zero ( signal c : out complex ) is
    begin
        c <= (0.0, 0.0);
    end procedure clear_to_zero;
end package body complex_numbers;
```

In this particular example, element-wise comparison would suffice. However, a user-defined operation is shown to illustrate the possible mechanism.

The function make_complex is intended to be a constructor, allowing creation of anonymous complex values. For example:

```
constant c_zero : complex := make_complex(0.0, 0.0);
constant c_1 : complex := make_complex(1.0 );
constant c_i : complex := make_complex(0.0, 1.0);
```

Examples of instantiation of the class are shown later in Section 3.

3 Integration with signals

One of the main ideas of a class is to define an ADT. It would seem reasonable to want to use ADT values as values for signals. The problems in doing so stem from the fact that signals in VHDL repre-

sent a different kind of storage object from variables and have quite different semantics. Whereas a variable stores a single value of a given type, a signal is much more complicated. It involves one or more drivers, each of which stores a trajectory of values for the current and future times. Existing language features relating to signals are:

- Signal assignment: requires a value of the signal's type, a delay time value, a pulse rejection time value and selection between transport and inertial delay mechanisms. Multiple waveform elements can be decomposed into a sequence of signal assignments each with one waveform element. Assignment involves a process of editing the trajectory for a driver.
- Signal update: involves determining driving values on a net (invoking resolution functions, type conversions and conversion functions towards the root of the net), then determining effective values (invoking type conversions and conversion functions away from root)
- Transactions and events: a transaction involves updating a signal with a newly determined effective value; if the new value is not equal to the old value, an event occurs. (Requires implicit assignment to update effective value, and implicit call of an equality operator to test for an event.)
- Sensitivity: involves a process specifying a signal upon which it may wait for an event.

3.1 Using a class for signal objects

The difficulty arising in the class-based approach is integrating a class-based ADT with the existing mechanisms relating to signals. An initial attempt at relating classes and signals might be to allow a class to encapsulate signals. For example:

```
type complex is class
    private signal re, im : real;
    ...    -- operations on a complex signal
end class complex;
```

The problem here is that a user of the class should only be permitted to declare signals of this type:

```
signal c : complex;
```

If the user wanted to declare variables of the type, they would have to declare a separate class to encapsulate variables. Mixing signal and variable objects would become unwieldy. For example, for a model to assign to a complex signal the sum of a complex signal value and a complex variable value would require several operations in each of the two classes to handle the different kinds of operands.

A better way of integrating ADT classes with signals is to require a class to have its encapsulated state specified as a variable, but to treat the variable as an abstract storage object. If the class is instantiated as a variable object, the value of the encapsulated state forms the value of the variable object, as in a conventional programming language. If the class is instantiated as a signal object, the value of the encapsulated state forms the value of a transaction in a driver. In this case, the class would have to provide value assignment, equality and deallocation operations to be used in signal assignment and update. Alternatively, it could rely on predefined shallow-copy for assignment, ele-

ment-wise comparison for equality and no-operation for deallocation. It is important to note that, if a class is instantiated as a signal, any operations that modify the encapsulated state cannot be called directly by the user, since to do so would violate the semantics of signals.

To illustrate how the class-based approach may be integrated with signals, consider again the class-based formulation of the complex-number type, shown in Section 2.1. A model might declare a variable and a signal of the complex class:

```
variable x : complex;
signal s : complex;
```

It might update the variable:

```
v := complex(0.0, −1.0);
```

This would involve invoking the ":=" procedure with v as the implicit parameter. The model might make an assignment to s in a process:

```
s <= reject 5 ns inertial c_zero after 20 ns, c_1 after 50 ns;
```

The algorithm for signal assignment would be followed as specified in the LRM. During transaction editing, the transactions after the time of the first new one are deleted, involving invocation of the deallocate operation to delete the transaction values. Next, a new transaction is created with the value initialized to default initial values. The ":=" operation is then invoked to copy the first waveform value to the new transaction. Then values of preceding transactions within the pulse rejection interval are compared to the value of the new transaction using the "=" function. Any that need to be deleted have the deallocate procedure invoked on their values. Finally, new transactions are created for waveform elements after the first, the values initialized to default initial values, and the ":=" procedure invoked for each one to copy the waveform value to the transaction value.

When a signal needs to be updated, the effective value is copied from the driving value transaction using the ":=" procedure. The "=" function is invoked to determine if there is an event. Any read of the signal is treated as a read of the effective value. Since the state is encapsulated, the only way to read a signal value is to invoke an operation on the signal. The operation must not modify the state, since that state is the effective value of the signal, and must only be modified as a result of the signal update algorithm. An example of a signal being used is:

```
process is
    variable v : complex;
begin
    wait until s /= c_zero;
    x <= s + c_i;
    y := s.real_part;
    s.clear_to_zero;  -- illegal!!
    v.clear_to_zero;  -- ok
    ...
end process;
```

The wait statement is sensitive to s, and waits until there is an event on s as described above. The expression s /= c_zero calls the "=" operator with s as the implicit parameter and c_zero as the actual parameter, then negating the result. Simililarly, the expression s + c_i calls a "+" function with s as the implicit parameter and c_i as the actual parameter. The expression s.real_part calls the real_part function with s as its implicit parameter and no other parameters. The procedure call s.clear_to_zero is illegal, since the only way to update a signal should be through signal assignment. Compare this with v.clear_to_zero, which is acceptable, since a variable can be updated immediately.

3.2 Using an abstract data type for signal objects

There are two aspects of integrating Ada-95 style ADTs with signals. The first relates to the way in which ADT values are passed as parameters to the ADT operations. Since an ADT may be instantiated as a variable or a signal, the operations should be written to deal with both possibilities. The simplest approach is to leave this to the ADT developer. They simply include variants of each operation with parameters of the appropriate kind (variable or signal). This is only relevant for operations which update the parameter. Operations that only read the parameter value have a constant parameter, which takes on the *value* of the actual parameter, be it a variable or a signal object.

The second aspect of integrating Ada-95 style ADTs with signals relates to signal assignment and update for signals of an abstract data type. Since the signal assignment and update algorithms rely on assigning and comparing values of the signal's type, copy and equality operations need to be defined. If the predefined operations are acceptable, they could be used. Alternatively, if an overloaded equality operator exists, it would be used. The Ada-95 approach does not provide for user-defined assignment, so if shallow copy were inappropriate, the type should be limited and would not be allowed for signals. Since the usual situation in which a type would be limited is if it contained access types, it would not be an appropriate type for a signal anyway, hence there is no loss of generality.

To illustrate how the programming by extension approach may be integrated with signals, consider again the formulation of the complex-number type, shown in Section 2.2. A model might declare a variable and a signal of the complex class:

```
variable x : complex;
signal s : complex;
```

It might update the variable:

```
v := make_complex(0.0, −1.0);
```

The model might make an assignment to s in a process:

```
s <= reject 5 ns inertial c_zero after 20 ns, c_1 after 50 ns;
```

The algorithm for signal assignment would be followed as specified in the LRM. During transaction edit, the transactions after the time of the first new one are deleted. Next, a new transaction is created and the predefined assignment operation is invoked to copy the first waveform value to the new transaction. Then values of preceding transactions within the pulse rejection interval are compared to the value of the new transaction using the "=" function to determine which need to be deleted. Finally, new transactions are created for waveform elements after the first and the predefined assignment operation invoked for each one to copy the waveform value to the transaction value.

When a signal needs to be updated, the effective value is copied from the driving value transaction using the assignment operation. The "=" function is invoked to determine if there is an event. Any read of the signal is treated as a read of the effective value.

An example of a signal being used is:

```
process is
    variable v : complex;
begin
    wait until s /= c_zero;
    x <= s + c_i;
```

```
        y := real_part(s);
        clear_to_zero(s);
        clear_to_zero(v);
        ...
    end process;
```

The wait statement is sensitive to s, and waits until there is an event on s as described above. The expression s /= c_zero calls the "=" operator with the values of s and c_zero as the actual parameters, then negates the result. Similarly, the expression s + c_i calls a "+" function with the values of s and c_i as the actual parameters. The expression real_part(s) calls the real_part function with the value of s as its actual parameter. The procedure call clear_to_zero(s) invokes the clear_to_zero procedure that has a signal parameter, and passes a reference to the driver on s. The procedure uses signal assignment to schedule a transaction on the driver. Compare this with clear_to_zero(v), which invokes the clear_to_zero procedure that has a variable parameter. That procedure uses variable assignment to update the variable.

3.3 Comparison of integration with signals

While both approaches can be integrated with signals, the Ada-like approach does so more simply. The class-based approach requires the restriction that the encapsulated state be declared as one or more variables. For a signal object of a given ADT, only operations that do not modify the instance variables are allowed, since the variables represent the values in transactions and effective values of signals. To modify them would violate the semantics of signal assignment. The only kind of operation where it is guaranteed that the instance variables are not modified is a pure function. Hence, operations on signals would have to be restricted to pure functions and signal assignment.

The Ada-style approach simply involves defining an ADT that can be used for variables or signals. Operations take values of the ADT type as parameters. For operations that have out-mode parameters, two version are needed: one for variables and one for signals. The current VHDL rules for overload resolution imply that the two versions of an operation must have different names. A minor extension, allowing the kind of parameter to be used in resolving overloading, would result in allowing the two versions of an operation to have the same name.

In both approaches, predefined assignment (shallow copy) and equality (element-wise comparison) operations can be used in the signal assignment and update algorithms. If the concrete representation of an ADT includes access values, the ADT should not be used for signals. Both approaches should include a means of indicating whether an ADT includes access values, without exposing further details of the encapsulated type. If the predefined assignment operation is inappropriate, a mechanism for preventing assignment of ADT values should be provided. (The Ada limited type feature is such a mechanism.) The ADT can include copy operations to be used instead. Such an ADT should also not be used for signals.

3.4 Composite signal semantics

VHDL currently specifies that a signal that is of a composite type is equivalent to a collection of scalar signals. This allows things like sub-element association, association of a port with an element of a signal, and separate drivers for separate sub-elements in separate processes.

If a signal is to be a class instance as described in Section 3.1, then the internal structure of the state is hidden within the class. Thus, even though the signal may be composite, that fact is not vis-

ible outside the class. Hence the signal must effectively be treated as atomic (in the sense that Dunlop defines [4]). This fits with the class-based scheme, since the only way to assign to a signal is via the ":=" operation. Thus a process that assigns to a signal assigns to the whole signal, and a process that is sensitive to a signal senses the whole signal.

If a signal is to be of an abstract data type as described in Section 3.2, then the internal structure of the type is hidden. Thus, even though the signal may be composite, that fact is not visible outside the defining package. Hence, subelement association, etc., may not be used. However, if a subprogram defined in the package has a signal parameter of the private type, that subprogram can see the structure of the signal. Hence it can assign to subelements. Thus, the idea of separate drivers for scalar subelements needs to be maintained in the semantics.

3.5 Resolved signals

VHDL currently supports user-defined resolution of multiply-driven signals. The user defines an unconstrained array type with elements of the signal's type. During signal update, the kernel creates an array of values collected from drivers, calls the user-defined resolution function, and uses the function result as the driving value for the signal.

If class-based ADTs are introduced as described above, the problem of creating the vector of contributions arises. This can be handled by having the kernel create an array of class instances, then calling the assignment operation for each element to copy a driver value to the vector. The kernel then calls the resolution function, which must then return a class instance as its result. The kernel then invokes the assignment operation again to copy the result to the driving value of the signal. The resolution function written by the user may need to invoke operations of the class to compute its result. For example, a resolution function for the complex class described above might be defined as:

```
type complex_vector is array (natural range <>) of complex;

function resolve_complex ( v : complex_vector ) return complex is
    variable sum : complex := c_zero;
begin
    for index in v'range loop
        sum := sum + v(index);
    end loop;
    return sum;
end function resolve_complex;

signal s_r : resolve_complex complex;
```

No changes are required to the resolution mechanism if ADTs are introduced using the programming by extension approach. The predefined assignment and equality operations may be used during the resolution process. The resolution function example for the complex number ADT is the same as for the class-based approach.

3.6 Conversion functions in port maps

The existing rules for conversion functions can be applied to conversion from a class type to some other non-class type. The conversion function is defined with an explicit parameter of the class type. For conversion from some source type to a class type, a construction operation could be used. For example:

```
function to_bit_vector_complex ( c : in complex ) return bit_vector_complex;

component cmp is
    port ( bvc : in bit_vector_complex; r : out real; ... );
end component cmp;

signal cs1, cs2 : complex;

u1 : cmp
    port map ( bvc => to_bit_vector_complex(cs1),
               complex(r) => cs2, ... );
```

Here, the conversion function to_bit_vector_complex is invoked when a new effective value is calculated on cs1. The function takes the effective value and returns a result to be used as the effective value of the component port bvc. When there is a new driving value of type real on the component port r, a constructor that can be called with one real actual parameter is called to determine the complex driving value for the signal cs1.

Similarly, the existing rules can be applied to conversion functions in the programming by extension approach. The conversion function may be defined in the ADT package or separately. Only the port map need be modified in the above example to use the make_complex function rather than the complex constructor.

```
u1 : cmp
    port map ( bvc => to_bit_vector_complex(cs1),
               make_complex(r) => cs2, ... );
```

4 Encapsulation

One of the main aspects of object-oriented data modeling is the ability to define abstract data types (ADTs). This requires encapsulating the concrete representation of an ADT value and providing operations for manipulating the value. The class approach uses a storage model for the encapsulated value. The concrete values of an object are declared in the class declaration and are viewed as instance variables by the operations. The particular object to be operated upon is implicit within the body of the operation. In the Ada-style approach, an ADT is defined as a type, encapsulated using the package feature of the language. A type name is provided by a package to denote the abstract type, and the concrete details of the type are made private to the package. Operations are provided in the form of subprograms that take explicit parameters of the type.

VHDL currently has some of the features required for Ada-style ADTs. It allows a type and operations on the type to be declared in a package. The details of implementation of the operations are hidden by being written in a separate package body. What VHDL lacks is a feature for information hiding: making the concrete details of the type invisible to users of the package. Such a feature would have to be added as an extension for VHDL to qualify as an object-oriented language. (This is a necessary, but not sufficient condition.) An appropriate path would be to extend the analogy between VHDL and Ada packages, and add provision for private types and private parts in packages.

Both classes and Ada-style packages provide means for declaring a set of related items. If classes were adopted in VHDL, the language would have two features that provide largely similar capabilities. The language would be simpler if there were only one mechanism for grouping declarations. Hence, the Ada-style features for defining ADTs are prefered.

One remaining aspect in which class-based proposals and VHDL packages extended with privacy features differ is the places where an ADT may be declared. Class-based proposals treat a class

as an additional kind of type definition. Hence, a class can be declared in any declarative region and is thus local to that region. In Ada, a package can likewise be declared in any declarative region. Hence, an ADT in Ada can be made local to a particular region. VHDL currently only allows a package to be declared as a library unit, so an ADT declared using a VHDL package is globally visible. Allowing packages to be declared in other declarative regions, as in Ada, would avoid this limitation.

5 Initialization of objects

In both the class-based and the programming by extension approaches, the rules for default initial values can be extended to the state encapsulated by an ADT. In the class-based approach, each instance variable declared in a class should take on the default initial value implied by its type. Initialization expressions might be allowed, and interpreted as initial values to be used in place of the default initial values for the instance variable. Such initial values effectively specify the default initial value for objects of the class. This would make class types different from other types, where there is no mechanism for specifying a default initial value in the type. In the Ada-like approach, the default initial values follow from the current default initial value rules for record types. The default initial value for each element is implied by the element type. Thus, tagged record types are no different from other types with respect to initialization.

In some circumstances it may be desirable to define constructors for initializing the value of an object. In the class-based approach, a constructor could simply take the form of an ordinary operation that assigns values to instance variables. However, there are two problems with this. First, a constant object could not be initialized with such a constructor. Second, a signal could not be initialized with such a constructor, since updating a signal using an operation must not be allowed (for reasons discussed below). Thus, a special form of constructor operation would be required, as described in Note 97−01.

In the Ada-like approach, a constructor need only be a function that returns a value of the ADT. The function can be called in the initialization expression for a constant, variable or signal object. This is simpler, being based on existing language mechanisms and not requiring special cases.

6 Inheritance

Both the class-based and Ada-style approaches allow inheritance of ADT state and operations. In the class-based approach, a child class is derived from a parent class by identifying the parent class in the declaration of the child class. The child class then inherits the instance variables and the operations of the parent, and may add further instance variables and operations and may override inherited operations. In the Ada-style approach, a child tagged record type is derived from a parent tagged record type and extends the parent type by adding record elements. The child type then inherits all of the primitive operations of the parent type. If the child type is defined within a package, additional primitive operations may be defined for the child type, and primitive operations may be defined hiding operations inherited from the parent type.

Both approaches allow definition of ADT operations whose implementation is deferred to a descendent in the inheritance hierarchy (“pure virtual”, or “abstract”). Both approaches also allow specification that an ADT is not to be instantiated directly, but only used for inheritance.

In both approaches, an object can be of a class-wide type, meaning that it can store a value of a nominated type or any of its descendants. When an operation is invoked on the object, the actual type of the object is used at run-time to determine which operation to use (dynamic dispatching).

Whereas some class-based proposals allow multiple inheritance, the Ada-style approach only admits of single direct inheritance. While multiple inheritance may have some applications, its general necessity is still widely debated. (As an illustration, although C++ includes multiple inheritance, Java only includes single inheritance.) Furthermore, allowing multiple inheritance requires complex rules in the language to resolve conflicting inherited instance variables and operations, and complicates implementation of the language. One use of multiple-inheritance, container classes, can be addressed by "template" (or generic) classes. Another use, mixin inheritance, can be addressed in the Ada-style approach by use of generic ADT packages that have a tagged type as a formal generic type parameter.

The comparison of inheritance capabilities of the two approach leads to a preference for the Ada-style approach. In most aspects, the two approaches provide similar capabilities, provided inheritance is restricted to a single parent. For multiple inheritance, the class-based approach would require a general multiple inheritance mechanism with the associated complexity and costs. The Ada-style approach, however, provides a simpler mechanism for mixin inheritance, which is the most common form of multiple inheritance.

Adopting the Ada-style of inheritance in VHDL would require adopting the notions of derived types. It would be beneficial to adopt the notion for all types, not just tagged record types, since derived types are a generally useful safety feature and can enhance the benefit of strong type checking. In order to support specification of container classes and mixin inheritance, the generic mechanism of VHDL should be extended to packages and subprograms, and allow a wider range of formal generic parameter kinds than just constants. Following the Ada model would be appropriate. This extension of the generics mechanism would be widely beneficial, as it would also allow generic design entities whose port types could vary between instances.

7 Conclusion

The comparison of the class-based approach and the programming by extension approach to extending VHDL for data modeling shows that both alternatives are viable. However, there are a number of factors that lead to choosing the programming by extension approach. That approach integrates more simply with mechanisms relating to signals. It extends the existing encapsulation mechanism, rahter than duplicating or replacing it. It provides a simpler initialization mechanism. When integrated with generics, it provides for simpler implementation of mix-in inheritance. These factors all stem from the requirement that extensions must integrate well with existing language mechanisms.

One final issue that leads to the Ada-like approach as the preferred alternative is stylistic integration. VHDL already borrows many features from Ada. Since the Ada style of object-oriented data modeling is based on these features, incorporating them in VHDL ensures that the extensions are in keeping with the the existing style of VHDL. To paraphrase the Ada 9X Rationale: "The solution should be conceptually consistent with existing [VHDL] programming models. Intuitions about object, types, subprograms, generic units, and so on should be preserved."

References

[1] P. J. Ashenden and P. A. Wilsey, "Considerations on Object-Oriented Extensions to VHDL," *Proceedings of VHDL International Users Forum Spring 1997 Conference*, Santa Clara, CA, 1997.

[2] G. Booch, *Object-Oriented Analysis and Design with Applications*. Redwood City, CA: Benjamin/Cummins, 1994.

[3] O. J. Dahl and K. Nygaard, "Simula: An Algol Based Simulation Language," *Communications of the ACM*, vol. 9, pp. 671–678, 1966.

[4] D. D. Dunlop, *VHDL Structure Varying Signals and OO Extensions to the VHDL Type System*. IEEE DASC OO-VHDL Study Group, working paper, ftp://vhdl.org/vi/oovhdl/papers/structure-varying-signals.txt, 1995.

[5] A. Goldberg and D. Robson, *Smalltalk-80: The Language*. Reading, MA: Addison-Wesley, 1989.

[6] J. Gosling, B. Joy, and G. L. Steele, *The Java Language Specification*. Reading, MA: Addison-Wesley, 1996.

[7] ISO/IEC, *Ada 9X Reference Manual*. Draft International Standard 8652, Cambridge, MA: Intermetrics, 1994.

[8] M. T. Mills, "Proposed Object Oriented Programming (OOP) Enhancements to the Very High Speed Integrated Circuits (VHSIC) Hardware Description Language (VHDL)," Wright Laboratory, Dayton, OH, Tech. Report WL-TR-5025, 1993.

[9] H. Mössenböck, *Object-Oriented Programming in Oberon-2*, 2nd ed. Berlin, Germany: Springer-Verlag, 1995.

[10] M. Radetzki, W. Putzke, W. Nebel, S. Maginot, J.-M. Bergé, and A.-M. Tagant, "VHDL Language Extensions to Support Abstraction and Re-Use," *Proceedings of Workshop on Libraries, Component Modeling, and Quality Assurance*, Toledo, Spain, 1997.

[11] G. Schumacher and W. Nebel, "Inheritance Concept for Signals in Object-Oriented Extensions to VHDL," *Proceedings of Euro-DAC '95 with Euro-VHDL '95*, Brighton, UK, 1995.

[12] B. Stroustrup, *The C++ Programming Language*. Reading, MA: Addison-Wesley, 1986.

[13] J. C. Willis, S. A. Bailey, and R. Newschutz, "A Proposal for Minimally Extending VHDL to Achieve Data Encapsulation Late Binding and Multiple Inheritance," *Proceedings of VHDL International Users Forum Fall '94 Conference*, McLean, VA, 1994.