

A Comparison of SUAVE and Objective VHDL

Report for the IEEE DASC Object-Oriented VHDL Study Group

Peter J. Ashenden, The University of Adelaide
petera@cs.adelaide.edu.au

Martin Radetzki, Institute OFFIS, Oldenburg University
radetzki@offis.uni-oldenburg.de

December 1998

Introduction

The IEEE DASC OO-VHDL Study Group has received two proposals for extensions to VHDL to provide object-orientation: the SUAVE proposal from Peter Ashenden of the University of Adelaide and Phil Wilsey of the University of Cincinnati, and the Objective VHDL proposal from Wolfgang Nebel's group at Institute OFFIS, Oldenburg University. Detailed information on the SUAVE proposal can be found at

<http://www.cs.adelaide.edu.au/~petera/suave.html>

and on the Objective VHDL proposal at

http://babbage.informatik.uni-oldenburg.de/research/objective_vhdl.html

At the meeting of the OO-VHDL Study Group in Lausanne in September 1998, it was agreed that the group could not viably support two proposals, and it was suggested that a merger of the proposals be investigated. The SUAVE and Objective VHDL teams were asked to collaborate to compare the two proposals in detail, and to investigate if and how a merger might be achieved. This report is the result of an investigation undertaken by the authors at Adelaide in November 1998. The report compares the two language proposals, and will serve as the basis for consideration of a possible merger or for a choice between the two proposals.

The report is divided into the following sections:

- Section I: Overview of SUAVE Language Features (page 2)
- Section II: Overview of Objective VHDL Language Features (page 15)
- Section III: Comparison of SUAVE and Objective VHDL Language Features (page 30)
- Section IV: Considerations on Nebel's Proposal for Class Types (page 36)

Sections I to III are organized round the same set of headings, each addressing individual semantic mechanisms implemented by language features. Section IV discusses a proposal by Wolfgang Nebel to unify the two language proposals. He suggests a third proposal that draws upon aspects of both SUAVE and Objective VHDL, attempting to answer criticisms of each.

Section I

Overview of SUAVE Language Features

Peter J. Ashenden, The University of Adelaide

1 Object-oriented data modeling

1.1 Class interface definition

A class interface in SUAVE is defined using a combination of language features: a package declaration, type declarations, and subprogram declarations. These are combined in a particular idiom to provide the semantics of a class definition. In SUAVE, a package need not be a library unit; it may also be declared in other declarative parts. This allows definition of classes locally to declarative regions.

1.1.1 Type definition

A type representing a class is defined using a type declaration within a package declaration. There is no specific syntactic construct that identifies the type as representing a class. The elements of the defined type represent the data attributes (instance variables) of the class. While it is possible to define multiple types representing multiple classes within a single package, the idiom is to separate distinct class definitions into distinct packages. Note that visibility control for data attributes is provided through the encapsulation features described below in Section 1.2.

For the set of data attributes of a class to be extensible in inheriting classes, the type must be declared as a *tagged* record type (see Section 1.3). In that case, the record type includes a run-time tag that identifies the specific type of the record. The tag is immutable and is not visible as a record element. The tag value of an object or of a tagged type can be read using the 'tag' predefined attribute, which returns a value of the predefined type tag. This type is defined as a private type in package standard. The operations of type tag are the relational operators, which test whether tagged types are the same or bear ancestor/decendant relationships.

An example of a class describing a simple CPU instruction is:

```
package instructions is
  type instruction is
    tagged record
      opcode : opcode_type;
    end record instruction;
  -- methods defined here
end package instructions;
```

1.1.2 Method definition

Methods of a class are defined as subprograms that follow the class's type declaration in a package. These subprograms have a formal parameter or a function result of the class's type, and

are called *primitive operations* of the type. The set of primitive operations for a type also include the implicitly declared (predefined) operations for the type.

To illustrate, methods for the instruction class defined in Section 1.1.1 could be defined at the place indicated in the comment as follows:

```
function privileged ( instr : instruction;
                    mode : protection_mode ) return boolean;
procedure disassemble ( instr : instruction; L : inout line );
```

1.1.3 *Non-instantiable (abstract) classes*

A class represented by a tagged type may be made non-instantiable by declaring it to be abstract, using the reserved word **abstract** in the declaration. Methods can be declared for an abstract type, however, since no object of the abstract type can be declared, the methods can only be invoked if an object is type-converted to the abstract type.

An example is an abstract class representing memory reference instructions:

```
type memory_instruction is
  abstract new instruction with record
    base : register_number;
    offset : integer;
  end record memory_instruction;
```

This type includes addressing mode information for the instruction, but does not specify a register for the loaded or stored data. Nonetheless, a method can be defined to calculate the effective address:

```
function effective_address_of ( instr : memory_instruction ) return integer;
```

1.1.4 *Non-invocable (abstract) methods*

An abstract class definition can include a specification of a method that must be provided by inheriting classes; such a method is defined to be *abstract* by including the reserved word **abstract** in its specification. An abstract method can have no body. An inheriting non-abstract class must provide an overriding body for the inherited abstract method. Thus, an abstract class can specify a “protocol” (a set of methods) that must be implemented by an inheriting class.

An example is an abstract method to perform a memory reference instruction, defined for the abstract memory reference class of Section 1.1.3:

```
procedure perform_memory_instruction ( instr : memory_instruction ) is abstract;
```

1.2 Encapsulation and visibility control

1.2.1 *Encapsulation of attributes*

SUAVE provides for encapsulation and visibility control of attributes through a combination of two language features: private parts in packages, and private types. A package may be divided into a publicly visible part and a private part by using the reserved word **private**, as follows:

```
package name is
  -- publicly visible declarations
```

private

-- *private declarations not visible outside package*

end package name;

As described in Section 1.1.1, data attributes are defined as elements of a type declared in a package declaration. If the type is declared in the public part of the package declaration, the type structure and element names are publicly visible. Alternatively, the type may be declared as a *private type* in the public part of the package, and the full type declaration deferred to the private part. In this case, only a partial view of the type is visible within the public part of the package and outside of the package. The partial view includes the type name, thus allowing definition of methods for the class in the public part of the package. Other aspects that may be specified in the partial view are:

- whether the type is tagged, and can thus be extended on derivation,
- whether the type, if tagged, is abstract,
- whether the partial view of the type is *limited* (does not allow assignment and does not predefine the “=” operator), and
- whether the type includes an element of an access type (thus preventing its use as the type of a signal).

The following example shows a package declaration for a class representing a media-access-level network packet. The type `MAC_packet` is a private type, whose full declaration is as a tagged record type in the private part of the package. The data attributes (source and dest) of an instance are not visible to the user of the type outside of the package. The user can only manipulate the attributes indirectly by invoking the methods on the instance.

```
package MAC_level is
  ...
  type MAC_packet is abstract tagged private;
  procedure set_MAC_dest ( pkt : inout MAC_packet; dest : in MAC_level_address );
private
  type MAC_packet is
    tagged record
      source, dest : MAC_level_address;
    end record MAC_packet;
end package MAC_level;
```

A type may also be declared as a *private extension* of a tagged parent type. A private extension is a partial view of a record extension derived type (see Section 1.3.2). The partial view specifies that the derived type derives from the named parent type, but does not expose the elements in the extension. The full view, declared in the private part of the package, completes the partial view by providing the details of the new data attributes added as extension elements.

The following example shows a package declaration for a class representing a network-level packet. The type `network_packet` is a private extension derived from the type `MAC_packet`. In this case, the parent type is a private type, so its data attributes are not visible. The only information in the partial view of `network_packet` is that it is a tagged type derived from `MAC_packet`. The full view of `network_packet` is completed in the private part of the package, and includes the record extension.

```

package network_level is
...
type network_packet is new MAC_packet with private;
procedure set_network_dest ( pkt : inout network_packet; dest : in network_level_address );
private
type network_packet is
  new MAC_packet with record
    source, dest : network_level_address;
    control : control_type;
    time_to_live : hop_count;
  end record network_packet;
end package network_level;

```

1.2.2 Encapsulation of method implementation

The methods of a class are defined as subprograms in the same package as the type definition. SUAVE relies on the existing package mechanism of VHDL to encapsulate the implementation of methods. A package declaration can only contain subprogram specifications; the subprogram bodies must be deferred to the package body, and are not visible to the user of the package.

1.3 Inheritance

1.3.1 Specifying inheritance

Inheritance is specified in SAUVE using derived types. A derived type definition names a parent type, and defines a distinct new derived type that inherits the set of permissible values and the primitive operations from the parent. The parent type need not be a tagged type, but may be any type.

For example, the following declaration defines a new type for 32-bit words, distinct from other bit-vector types. It inherits all of the predefined primitive bit-vector operations.

```

type word is new bit_vector(0 to 31);

```

Where a class is defined using a type declaration and primitive operations in a package declaration, a subclass can be defined using a derived type declaration. While it is possible to include the subclass definitions in the same package as the parent type, the preferred idiom is to define the subclass in a separate package.

1.3.2 Augmenting inherited attributes

If the parent type is defined as a tagged record type, the derived type can be defined as a *record extension* containing additional record elements beyond those inherited from the parent type. Note that a record extension must not be declared at a deeper accessibility level than its parent. This avoids a problem that can otherwise arise as a result of dynamically dispatching—under certain circumstances, methods of the record extension could be invoked through dynamic dispatch even after the referencing environment of the methods had passed its lifetime.

As an example of a record extension, the type `memory_instruction` in Section 1.1.3 inherits the element `opcode` from its parent type, `instruction`, and adds new elements `base` and `offset`. A further type can be derived from this type, further extending it, as follows:

```
package load_instructions is
...
type load_instruction is
  new memory_instruction with record
    destination : register_number;
  end record load_instruction;
  -- methods for load_instruction
end package load_instructions;
```

1.3.3 *Augmenting inherited methods*

When a derived type (tagged or otherwise) is defined in a package, additional primitive operations can be defined for the derived type. This is illustrated in the package for network-level packets shown in Section 1.2.1: the primitive operation `set_network_dest` is added for the derived type `network_packet`.

1.3.4 *Overriding inherited methods*

A package that includes a derived type can override primitive operations inherited from the parent type. An overriding operation is declared as a subprogram with the same signature as that of the parent operation, but for each parameter of the parent type, a parameter of the derived type is included instead. In the case of an inherited function with the parent type as the result type, an overriding function has the derived type as the result type.

To illustrate overriding of primitive operations, consider the class for load instructions shown in Section 1.3.2. The type `load_instruction` inherits operations `perform_memory_instruction` and `disassemble` from its parents. These can be overridden by new operations as follows:

```
procedure perform_memory_instruction ( instr : load_instruction );
procedure disassemble ( instr : load_instruction; L : inout line );
```

1.4 Instantiation

1.4.1 *Instantiation of constant objects*

A class is instantiated as a constant using a constant declaration; the type representing the class is named as the type of the constant. If the type is a private type, the initial value expression for the constant must use functions or constants defined in the package that defines the type. Since only the partial view of the type is visible, the initial value expression cannot include literals or aggregates of the type. A constant cannot be declared of a limited private type outside the package defining the type, since assignment of the initial value is not allowed.

Apart from the consequences of the full view of a private-type constant not being visible, constant objects can be used and manipulated according to the normal VHDL rules.

1.4.2 Instantiation of declared variable objects

A class is instantiated as a variable using a variable declaration; the type representing the class is named as the type of the variable. If the type is a private type, an initial value expression for the variable must use functions or constants defined in the package that defines the type. Since only the partial view of the type is visible, the initial value expression cannot include literals or aggregates of the type. A variable of a limited private type cannot include an initial value expression, since assignment of the initial value is not allowed. Such a variable can only be updated by operations defined in the package containing the type definition.

Apart from the consequences of the full view of a private-type variable not being visible, variable objects can be used and manipulated according to the normal VHDL rules.

1.4.3 Instantiation of dynamically allocated variable objects

A class is instantiated as a dynamically allocated variable using a **new** allocator; the type representing the class is named in the subtype indication or as the type mark in the qualified expression. In the former case, the allocated variable takes on the default initial value for the type. In the latter case, the allocated variable takes on the value of the qualified expression. If the type is a private type, the qualified expression must use functions or constants defined in the package that defines the type. Since only the partial view of the type is visible, the qualified expression cannot include literals or aggregates of the type.

Apart from the consequences of the full view of a private-type dynamically allocated variable not being visible, dynamically allocated variable objects can be used and manipulated according to the normal VHDL rules.

1.4.4 Instantiation of signal objects

A class is instantiated as a signal using a signal declaration; the type representing the class is named as the type of the signal. If the type is a private type, an initial value expression for the signal must use functions or constants defined in the package that defines the type. Since only the partial view of the type is visible, the initial value expression cannot include literals or aggregates of the type. A limited type or a private type that includes the reserved word **access** in its definition cannot be used as the type of a signal.

Apart from the consequences of the full view of a private-type signal not being visible, signal objects can be used and manipulated according to the normal VHDL rules. Transaction list editing and event determination are performed using the predefined equality operator for the signal type.

1.4.5 Parameter passing

The rules for parameter passing are as for standard VHDL. In addition, parameters of tagged types are passed by reference.

1.4.6 Initialization

Objects are initialized according to the rules of standard VHDL. A tagged type behaves just as a standard record type with respect to default initial value. The default initial value for a private type is determined by the full view of the type.

1.4.7 Finalization

As in standard VHDL, there is no mechanism for user-defined finalization of objects in SUAVE. The lifetimes of storage objects follow the same rules as in standard VHDL.

1.4.8 Assignment and controlling copying

The rules for assignment of variables and signals in SUAVE are the same as those in standard VHDL, with the exception of assignment to objects of limited types. A private type can be declared to be limited by including the reserved word **limited** in its type definition. A record type can be declared to be limited either by including the reserved word **limited** in its type definition or by including a limited-type element.

The assignment operations (variable and signal assignment) are not allowed for objects of limited types, and limited types do not provide predefined equality or inequality operators. For a limited private type, the full view can be nonlimited. In this case, the only way to update an object declared to be of the private type is to pass it as a parameter to a primitive operation of the type. Since the operation's body is declared in the package body, where the full view of the type is visible, the operation can make assignments to the object.

Limited types are useful for types that are implemented using linked data structures and for which deep-copy assignment and deep-compare equality testing are desired. The predefined assignment and equality operators perform element-wise operations, and do not traverse access-value links. By declaring the type to be limited, the type provider can force use of type-provided copy and comparison operations.

1.5 Aggregation

1.5.1 Class instances as attributes of classes

A composite type representing a class may have an element that is of a type representing some other class. The only restrictions are those described in earlier sections that arise from use of types, such as use of private types containing access values as the types of signals.

1.5.2 Class instances as elements of composite data types

As in Section 1.5.1—there is no distinction between these two cases.

1.6 Method invocation

1.6.1 *Methods designated by identifiers*

A class method is simply a subprogram declared in a package. An object that is an instance of the type representing the class is passed as a parameter to the subprogram. The standard VHDL rules of subprogram invocation apply. (See also Section 1.7.3.)

1.6.2 *Methods designated by operator symbols*

Methods designated by operator symbols must conform to the standard VHDL rules for subprograms that overload operator symbols. They may be invoked in infix form with left and/or right operands or the result being of the type representing the class for which the method is defined. The standard VHDL rules of operator invocation apply. (See also Section 1.7.3.)

1.7 Polymorphism

1.7.1 *Class-hierarchy types*

For a tagged type T , SUAVE defines a *class-wide type*, denoted by T 'class, that is the union of T and all types derived directly or indirectly from T . T is called the *root* of the class-wide type T . Objects declared to be of a class-wide type are polymorphic, in that they can take on values of any of the specific types in the union. However, only elements and operations applicable for the root type are accessible for objects of the class-wide type.

As an example, a class-wide type rooted at the specific type instruction defined in Section 1.1.1 is denoted by instruction'class. A *class-wide operation* (see Section 1.7.2) on this type can be defined as follows:

```
procedure execute ( instr : instruction'class );
```

1.7.2 *Objects of class-hierarchy types*

SUAVE allows constants, dynamically allocated variables, signals, subprogram parameters and ports to be of class-wide types.

- A constant of class-wide type must be initialized with a value of a specific tagged type, and thereafter cannot be changed.
- An access type can be defined with a class-wide type as its designated type. An allocator for such an access type must have either a specific tagged type as a subtype indication, or a qualified expression of a specific tagged type. The dynamic variable thus created is of a specific tagged type, and is pointed to by the access value of class-wide type returned by the allocator.
- A signal of class-wide type can be assigned values of different specific types during its lifetime. There are a number of issues relating to signals of class-wide type that are yet to be resolved in the language definition (see Section 1.7.6).
- A subprogram with a parameter of a class-wide type is called a *class-wide operation*. The body may only refer to elements and apply operations that are defined for the root type of

the parameter. A class-wide constant or variable parameter is passed by reference. A class-wide signal is passed by reference (including a reference to the driver of the actual signal in the case of an out or inout signal parameter) as in standard VHDL.

- A class-wide port is a particular case of a class-wide signal.

1.7.3 *Dynamic dispatch*

The primitive operations of a tagged type are called *dispatching operations*. Where a dispatching operation is inherited and overridden for derived types within a class hierarchy, the specific types of the operands and the expected result in a call determine which version of the operation is invoked. If the specific types are statically determined, the particular method to be invoked is statically determined. However, if the specific types can only be determined dynamically (because the actual parameters are of a class-wide type), the method is dispatched dynamically. In this case, only methods that are defined for the root type of the class-wide type of the actual parameters can be invoked.

As an example, consider the class-wide operation `execute` shown in Section 1.7.1. The body of the operation might include the statement:

```
if privileged ( instr ) then ...
```

The operation `privileged` is defined for instruction's class, but may be overridden by types derived from instruction. The parameter `instr` may be of type `instruction` or any type derived from it; the specific type is not known until each call to `execute` is made. Thus, the particular version of `privileged` to be invoked must be determined dynamically.

1.7.4 *Subtype compatibility and type conversion*

A parent type and a derived type are distinct and are not assignment compatible. They are, however, related, and so type conversions can be used. SUAVE distinguishes two kinds of type conversion: value conversions and view conversions. A value conversion is the same as a type conversion in standard VHDL—it changes the value from one type to another. A view conversion, on the other hand, does not change the operand value, but simply treats it as being of the target type. Type conversions between specific tagged types are always view conversions, and must be to an ancestor type of the operand. This ensures that extension elements are not lost, merely ignored.

An object of a tagged type can be view-converted to an ancestor tagged type. Conversions to a descendent type are not allowed. Instead, an extension aggregate may be used. This is a new form of aggregate that includes a source value, and provides values for the record extension elements that are required for the target type. For example, given a value `base_instruction` of type `instruction`, a value of type `load_instruction` could be specified using the following extension aggregate:

```
load_instruction'(base_instruction with base => r0, offset => 0, destination => r31)
```

An object of a tagged type can be converted to a class-wide type if the root of the class-wide type is the same as, or an ancestor of, the source type. Such a type conversion need not be explicitly written. An object of a class-wide type can be converted to a specific tagged type if the specific type of the source object is the target type or one of its descendants. A run-time check

may be required. Conversion from a source class-wide type to a target class-wide type is allowed only if the root of one of the classes is the same as or an ancestor of the other. A run-time check is required if the source type is not a subclass of the target type.

Note that view a view conversion of a tagged type can be used to invoke a parent method of the type. For example, given a value `instr` of type `load_instruction`, the following statement invokes the `disassemble` method of the type `instruction` from which `load_instruction` is (indirectly) derived:

```
disassemble ( instruction(instr), L );
```

1.7.5 Staticness of types

Strong static typing is used for tagged types. Since tagged types related by derivation are distinct types, they are not subtype compatible (c.f. C++). However, type conversions can be used as required to provide parent-type views of objects. The same strong static typing applies to access types that designate tagged types: a value of an access type that designates a tagged type can only point to a value of that tagged type, not of any descendent type. Class-wide types provide the kind of run-time polymorphism that is seen in C++ for those circumstances where it is needed.

1.7.6 Semantics for signals of class-hierarchy types

SUAVE allows a signal to be declared of a class-wide type. Assignment to such a signal is performed using the semantics of standard VHDL. A new transaction is created for the driver, and the queue of transactions on the driver is processed to implement the required inertial or transport delay semantics. The values of transactions are compared using the predefined “=” operation, which includes comparison of the tags of the values. If the tags differ, the values are not equal. If the tags are the same, the elements of the signal are compared to determine equality of the values.

When a signal of class-wide type is referred to in a sensitivity list, only the whole signal or elements that are defined for the root type may be referred to. (This follows from the semantics of class-wide types.) Note that some limitations arise in port maps relating to association with subelements of a formal class-wide port or subelement association with an actual class-wide signal. Where VHDL requires that the whole of a signal or port be associated or that there be drivers for all elements, it is not possible to determine statically that all elements of a class-wide signal or port are dealt with. Hence a class-wide signal or port cannot be used in such contexts. Instead, the signal must be treated as an atomic whole.

Another limitation that arises from the semantics of class-wide types is that class-wide signals cannot be resolved. It is not possible to declare a resolution function for a class-wide signal, since a class-wide type cannot be used as the element type of an array.

Type conversions can be used in association lists as in standard VHDL, but the rules relating to type conversion for tagged and class-wide types apply (see Section 1.7.4). Extension aggregates cannot be used in associations, so it is not possible to associate, for example, a signal of a parent type with a port of a derived type. Instead, class-wide signals and ports should be used.

Conversion functions can also be used in association lists as in standard VHDL. In the case that a conversion function is a method in a class hierarchy and the object to be converted is of the class-wide type, dynamic dispatch is used to invoke the appropriate version of the function.

Where a signal is of a class-wide type, it is not possible to determine statically if there is a derived type in the class hierarchy that includes an element of access type. If the hierarchy includes such a derived type, the class-wide type is not legal as the type of the signal. This check must be deferred to elaboration time, when the full class hierarchy is known. (An alternative language design, yet to be fully considered, is to require a tagged record to include the word **access** in its definition if it is to be extended with an element of access type. This parallels the use of the word **access** in private type definitions and the use of the word **limited** in record definitions. This change would obviate the need for the elaboration-time check, since only a class-wide type whose root type does not include the word **access** would be allowed as the type of a signal.)

2 Object-oriented structure and behaviour modeling (component abstraction)

Whereas Objective VHDL allows new entities to be derived from parent entities and new architectures to be derived from parent architectures, SUAVE provides no language features to do so. We believe that many of the cases where use of entity/architecture inheritance has been proposed are better expressed as compositions of component instances using standard VHDL. In more complex cases, simply replacing inherited concurrent statements and adding new ones is not sufficiently powerful to achieve the level of reuse desired. The mechanism does not provide means of revising the connectivity of inherited processes and component instances so as to insert new processes and instances. Instead, inherited processes and component instances must be overridden with new versions that have the same functionality but that are interconnected differently to accommodate the really new processes and instances. This replication complicates the derived architecture. We have yet to see a convincing example that shows how entity/architecture inheritance can be successfully applied with significant benefits. Until such an example is forthcoming, we prefer not to add the complexity of entity/architecture inheritance to the language.

3 Genericity

3.1 Formal-constant genericity for types

SUAVE allows a package to include a generic interface list; such a package is generic, and must be statically instantiated to be used. A generic constant in a package that defines an abstract data type can be used within the definition of the data type. In this way, the effect of making the data type generic is achieved.

As an example, the following package defines an abstract data type for a stack whose size is specified by a generic constant:

```
package integer_stacks is
  generic ( stack_size : natural );
  type integer_stack is private;
  function is_empty ( s : integer_stack ) return boolean;
  function is_full ( s : integer_stack ) return boolean;
  ...
```

private

```

subtype stack_pointer_range is natural range 0 to stack_size;
type stack_store is array ( 1 to stack_size ) of integer;
type integer_stack is
  record
    top : stack_pointer_range;
    store : stack_store;
  end record integer_stack;
end package integer_stacks;

```

The package may be instantiated and used as follows:

```

package integer_stacks_small is new integer_stacks
  generic map ( stack_size => 10 );
package integer_stacks_large is new integer_stacks
  generic map ( stack_size => 100 );
variable small_stack : integer_stacks_small.stack;
variable large_stack : integer_stacks_large.stack;

```

3.2 Formal-type genericity

A generic interface list in SUAVE may include formal types. This allows type-parameterization of entities and components. In addition, generic interface lists may be included in package and subprogram declarations. Generic packages and subprograms must be instantiated before they can be used or called respectively. In order to make type-parameterized packages and subprograms more useable, SUAVE also allows formal subprograms (e.g., action routines) and packages to be included in generic interface lists. These features and examples of their use are described in more detail in papers and technical reports on SUAVE (see www.cs.adelaide.edu.au/~petera/suave.html). The reason for include type-genericity in SUAVE is to improve scope for reuse of units. This aids modeling at all levels of abstraction, from system-level down to gate-level, and aids development of complex test benches.

4 Abstract communication and concurrency

SUAVE includes language features to improve useability of the language for system-level modeling. These features include:

- channels and message passing as a more abstract form of communication and synchronization than signals and signal assignment, and
- process declarations with static and dynamic instantiation for modeling dynamic reactive systems.

The OO-VHDL and SID Study Groups have agreed that consideration of these features is outside the scope of the OO-VHDL Study Group, and falls within the scope of the SID Study Group. The features and examples of their use are described in more detail in papers and technical reports on SUAVE (see www.cs.adelaide.edu.au/~petera/suave.html).

5 Tool implications

5.1 Analyzer/elaborator implications

SUAVE adds a number of features to VHDL, so tools for analysis and elaboration of models must be extended to deal with the new features. The extensions to the syntax are straightforward. Most of the work in extending an analyzer is in static semantic analysis and code generation. The extensions for type genericity would also require extending an elaborator to handle instantiation of generic units. Since most of the new SUAVE features are modeled on those of Ada, implementation of semantic analysis and code generation is of similar complexity and scope.

5.2 Simulation implications

The object-orientation and genericity extensions will have little impact on simulator run-time support. The main extension needed is support for dynamic dispatch of operations and for class-wide signals. Most of the effort in run-time support for SUAVE would be for channels, message passing and synchronization, and dynamic process management.

5.3 Synthesis implications

Current synthesis technology is very limited in the allowable input subset of VHDL. If we are to conform to these limitations, most of the new features in SUAVE will be excluded from synthesis, and there will be negligible impact on synthesis tools. The main area of interest in synthesis from SUAVE input is in the area of high-level synthesis and software synthesis. In principle, most of the new language features of SUAVE are synthesizable. Signals of class-wide types may present some challenges due to the varying size of values within the class hierarchy. However, the maximum size of values can be determined during elaboration of a model, and this maximum size could be used for signal values in hardware. Dynamic process instantiation is probably not synthesizable into hardware, except in limited forms for dynamically reconfigurable hardware. Type-generic units should present no problems for synthesis since they are instantiated at elaboration time.

Section II

Overview of Objective VHDL Language Features

Martin Radetzki, Institute OFFIS, Oldenburg University

1 Object-oriented data modeling

1.1 Class interface definition

Objective VHDL defines a new type, the class type, in which the language extensions are concentrated. This is thought to minimize interference with other VHDL language features and to provide a single construct for the description of an abstract data type (and *only* the abstract data type).

1.1.1 Type definition

An object-oriented class data type is defined with a class type definition which is added in Objective VHDL to the VHDL type definitions such as file, access, record, or array types. A class type definition must only occur in a type declaration. Such class type declaration should be located in a package declaration so as to be globally visible. The Objective VHDL LRM currently permits class type declarations in other declarative parts, e.g., of architectures and subprograms, where the class types are only locally visible. This, however, causes problems related to polymorphism and dynamic dispatch, which may have to be resolved by additional semantic constraints on the declaration and use of class types and class-wide types. Alternatively, class type declarations could be limited to being in packages. This restriction currently applies when using the VHDL translator.

A class describing a CPU instruction could be declared:

```
type instruction is class
    ... -- declarations of class, see below
end class instruction;
```

1.1.2 Method definition

The user of Objective VHDL can define methods by declaring subprograms (procedures and functions) in a class type declaration. The syntax of the subprogram declarations is as in VHDL. Objective VHDL forbids subprogram bodies to be declared at this place in order to hide the implementation details in the interface view. As opposed to SUAVE, the methods do not have to have a formal parameter of the class type. Instead, they implicitly have access to certain portions of the class data (see ...)

For instance, a class type buffer with methods put, get, is_full, and is_empty could be declared as follows:

```
type buffer is class
    procedure put( item : item_type );
```

```

    procedure get( item : out item_type );
    function is_full return boolean;
    function is_empty return boolean;
end class buffer;

```

Note that subprogram `get` cannot be declared as a function since it modifies the state of the class, taking one item out of the buffer. A method which is declared as a function, be it pure or impure, has only read access to the class data.

Methods of the instruction class can be declared as follows:

type instruction **is** class

```

    function privileged( mode : protection_mode ) return boolean;

```

```

    procedure disassemble( L : inout line );

```

```

end class instruction;

```

Methods can be defined in so-called object configurations in order to implement them in a specific way for a given storage class (signal, variable, constant). An object configuration, to be specified inside a class definition, has the following syntax:

```

for signal -- also: variable, constant

```

```

    ... -- method declaration(s)

```

```

end for;

```

A method declared in an object configuration can only be invoked with an object of the specified storage class. The method implementation is allowed to use the operations defined for that storage class (and only these operations). For instance, a method „for signal“ can use signal assignment to the class attributes and may apply the ‘EVENT attribute to them. However, it may not use variable assignment which is only permitted to methods „for variable“. Methods „for constant“ must not use assignment to class attributes at all. Further restrictions apply to method calls inside object-specific methods. A method for signal can invoke other method for signal as well as the methods which are declared outside of an object configurations. It must, however, not invoke methods which were declared only „for variable“ or „for constant“.

1.1.3 *Non-instantiatable (abstract) classes*

A class type as well as a derived class type can be declared as abstract by including the keyword `abstract` in the type definition. Language mechanisms prevent the instantiation of such abstract classes, which is to be checked statically by an analyzer. Whereas a non-abstract class must have a class body, an abstract class may, but doesn't need to, have a class body.

For instance, we may choose to declare the buffer class type as abstract and leave the decision whether it is a FIFO or LIFO style buffer to the derived classes:

```

type buffer is abstract class

```

```

    ...

```

```

end class buffer;

```

Similarly, a memory instruction can be derived as an abstract class from class instruction. Derivation and (class) attributes will be explained later in the document:

type memory_instruction **is new abstract class** instruction **with**

class attribute base : register_number;

class attribute offset : integer;

function effective_address **return** integer;

end class memory_instruction;

Note that function `effective_address` can only be invoked with an object of a class derived from `memory_instruction`.

1.1.4 Non-invocable (abstract) methods

In Objective VHDL, a class which is not abstract must not have non-invocable methods. An implementation (subprogram body) must be provided for any method of a non-abstract class. In an abstract class, the user can choose to provide an implementation or not for each single method. A method which has no implementation is considered as being abstract. This property does not have to be (and cannot be) declared explicitly.

In the example of the abstract class `type buffer`, we may choose to provide no method implementations (and no class body) at all since all the methods depend on whether the class is a FIFO or a LIFO.

An abstract method `perform_memory_instruction` could be added to class `memory_instruction` by declaring the following in the class construct and not providing a subprogram body:

procedure perform_memory_instruction;

1.2 Encapsulation and visibility control

Method implementations and attribute declarations are / can be encapsulated in a class type body belonging to a class type declaration. The body must be declared in the same declarative region and with the same name as the class type as follows:

type C is class body

...

end class body C;

1.2.1 Encapsulation of attributes

Attributes can be declared in the class type declaration and in the class body. The syntax of an attribute declaration is like a variable declaration with the exception that the keywords `class attribute` be used instead of `variable` (keyword `class attribute` were chosen instead of only `attribute` in order to tell OO attributes apart from VHDL attributes). Attributes are never visible outside of the class and its childs. If declared in the class type declaration (interface), they are visible to derived classes. If declared in the class body, they are visible only to the class itself. By declaring attributes in the class type declaration and body, the user can choose to make them accessible directly in derived classes or only through delegation, respectively.

For instance, the abstract class `buffer` may have an attribute to store `n` items, visible also to the derived classes:

```
type buffer is abstract class
```

```
    type item_array is array( 0 to n-1 ) of item_type; -- Note: type, subtype, use clause,
                                                    -- and alias declarations may also be declared
                                                    -- inside a class for internal use.
```

```
    class attribute store : item_array; -- initialization is optional
```

```
    ... --method declarations
```

```
end class buffer;
```

The MAC / network package example, with attributes visible only to the class itself, can be coded as follows:

```
type MAC_packet is abstract class
```

```
    procedure set_MAC_dest( dest : in MAC_level_address );
```

```
end class;
```

```
type MAC_packet is class body
```

```
    class attribute source : MAC_level_address;
```

```
    class attribute dest : MAC_level_address;
```

```
end class body;
```

```
type network_packet is new class MAC_packet with
```

```
    procedure set_network_dest( dest : in network_level_address );
```

```
end class;
```

```
type network_packet is class body
```

```
    class attribute source : network_level_address;
```

```
    class attribute dest : network_level_address;
```

```
    class attribute control : control_type;
```

```
    class attribute time_to_live : hop_count;
```

```
end class body;
```

1.2.2 Encapsulation of method implementation

The user can provide method implementations, i.e., subprogram bodies corresponding to the subprogram declarations made in the class type declaration, in the class body. If the class is not abstract, these implementations *must* be provided. It is also possible to declare additional subprogram bodies in the class body for local use in the class. These subprogram bodies will not be invocable from derived classes or from outside the class (i.e., they are no methods).

1.3 Inheritance

1.3.1 Specifying inheritance

A new class (child, subclass, derived class) can be derived from a parent class using a derived class type declaration which allows to specify the one parent class which is to be derived from (i.e., single inheritance):

```
type D is new class C with
    ...
end class D;
```

The syntax for declaring a class body of a derived class is the same as for a non-derived class.

1.3.2 Augmenting inherited attributes

The derived class inherits all attributes from its parent class. Note that also the class attributes of the parent's class body are inherited and contribute to the state space of the child. They are, however, not visible (accessible) directly in the derived class.

If an attribute is declared in a derived class with a name that has already been used in the parent class for an attribute declaration, the new attribute will not override, but hide the inherited attribute. This means that both attributes co-exist. The new attribute is directly visible; the inherited one only by selection using as prefix the name of the class it was declared in.

For instance, a class `lifo` can be derived from class `buffer` as follows:

```
type lifo is new class buffer with
    class attribute top : integer range 0 to n := 0;
end class lifo;
```

A load instruction with additional destination register information can be declared:

```
type load_instruction is new class memory_instruction with
    class attribute destination : register_number;
end class load_instruction;
```

1.3.3 Augmenting inherited methods

A derived class can add new methods to the inherited ones by declaring them as subprograms in the derived class type declaration. VHDL overloading rules apply, allowing to use the same name for two different methods if their parameter and return type profiles differ.

Corresponding subprogram bodies can / have to be declared in the class type body of a derived class. They have access to all attributes of the derived class they are declared in and to the attributes declared in the interface of all parent classes. Regarding abstractness of classes and methods, the rules explained for non-derived classes apply to derived classes as well.

1.3.4 Overriding inherited methods

An inherited method can be overridden by declaring the subprogram again in the derived class.

The name of the subprograms and their parameter lists must be equal. It is not sufficient to declare the overriding method with the same parameter and return type profile. Also the names, modes (in, out, inout) and VHDL classes (signal, variable, or constant) of corresponding parameters must be equal. If they are not, and if the subprograms cannot be overloaded according to VHDL rules, the method declaration of the derived class is illegal.

The following declarations, when included into the declaration of the class type `load_instruction`, would override the respective inherited ones:

```
procedure perform_memory_instruction; -- overrides perform_memory_instruction
                                     -- of class memory_instruction

procedure disassemble( L : inout line ); -- overrides disassemble inherited from class instruction
```

1.4 Instantiation

Note that, when speaking of a class type in this section, the same applies to derived class types.

1.4.1 Instantiation of constant objects

A non-abstract class can be instantiated as a constant object by declaring a constant of the class type. An initialization expression must be provided. It is, however, not possible to construct a value of a class type by aggregating the values of its attribute (like a record aggregate) since such construction could lead to an inconsistent state of the class data and would break class encapsulation. Hence, the only way to provide the value is by a call to a user-defined function. This function can declare a variable of the class type, modify its state by calling methods, and finally return the class value.

1.4.2 Instantiation of declared variable objects

A variable can be declared with a class type. It may be initialized by the user. If not, the initial value is defined by the initial values of the class attributes.

1.4.3 Instantiation of dynamically allocated variable objects

It is possible to declare an access type of a non-abstract class type. As in VHDL, a variable of such access type can be declared. Such access variable is either null or designates an object which was dynamically allocated using the keyword `new` as in VHDL. Until it is assigned a user-defined value, the class attributes of the allocated object have their initial values. It is possible to provide a user-defined qualified initialization expression when allocating the new object. Later, the allocated space can be accessed and assigned to using the VHDL keyword **all**, which may be omitted in many cases.

```
type p_fifo is access fifo;
variable dynamic_fifo : p_fifo;
...
dynamic_fifo := new fifo; -- class attributes are initialized with their default values
dynamic_fifo := new fifo'( some_expression_resulting_in_value_of_type_fifo ) --user def'd
initialization
```

```
dynamic_fifo.all := some_expression_resulting_in_value_of_type_fifo; -- accessing the allocated
object
```

1.4.4 Instantiation of signal objects

A signal can be declared with a class type which has no class attribute (declared in the class itself, inherited, or in classes instantiated as class attribute) of access type. It may be initialized by the user. If not, the initial value is defined by the initial values of the class attributes.

1.4.5 Parameter passing

Subprogram parameters and port signals can be declared with a class type. Actual parameters are passed as defined for composite types in VHDL. I.e., if the formal parameter is a signal, a reference to the driver of the actual parameter is passed. If the formal parameter is a variable, the actual may be passed by reference or by value, depending on simulator implementation, and a model is erroneous if it relies on a particular mechanism. Finally, if the formal parameter is a constant, the actual is passed by value.

1.4.6 Initialization

Objects are initialized with the initialization expressions specified for the class attributes, and the VHDL default values (type'LEFT) for class attributes which do not have an initialization expression.

1.4.7 Finalization

Signal objects exist during the complete simulation. Declared variable objects and constant objects as well unless they are declared in subprograms. In this case, they are finalized by the simulator upon return from subprogram execution. Dynamically allocated variable objects have to be finalized by the user by calling the predefined deallocate procedure known from VHDL. In all these cases, finalization means freeing the allocated stack or heap space. It does not include, e.g., the traversal of references in a linked list to free all its elements. If users need such more sophisticated finalization, they can define appropriate procedures (methods) and call them explicitly. However, it is not possible to integrate such user-defined finalization into the automatic finalization performed by the simulator.

To explain finalization in more detail, it is useful to introduce the relationships *has-a* and *uses*. If a class C contains a class attribute A of another class type B, we say that an object of type C *has-an* object of type B. This sub-object is an integral part of C's state, it is fully encapsulated by C, and no object from outside can have a reference of the sub-object. The latter property is enforced by VHDL language mechanisms. If C is deallocated, B will automatically be deallocated as well in Objective VHDL. This is the same as with a record that has an element of a record type.

There is a use-relationship of C and B (C uses B) if C contains a reference to an object of type B (in VHDL terms, an access value designating an object of type B). C may have to share the object with others who also have a reference of B. Thus, if C is deallocated, B cannot be deallocated automatically by the simulator. The user has to care for deallocation of B and has to ensure that the deallocated object isn't used elsewhere.

1.4.8 Assignment and controlling copying

A value of a class type can be assigned to a signal or a variable of the same class type. The effect of the assignment is like an assignment of all class attribute values of the source to the respective class attribute of the target. If the assigned object is in a has-a relationship with a sub-object, also the sub-object will be assigned. The effect is a deep copy of the source created in the target of the assignment. On the other hand, an object used via a reference (an access type) will not be copied, i.e., in this case a flat copy is performed. Only the reference of the used object is copied, not the used object itself. If users need a replication of the used object, they have to implement and call their own copying method. All this corresponds to the finalization of exclusive sub-objects (has-a) and referenced objects (use). As with finalization, Objective VHDL does not provide a means to have such user-defined copy mechanism called by the system; in particular, it is not possible to override/-load the assignment operator. This can be compared to VHDL, which does not allow the user to overload the assignment. Also, it is not possible to forbid the use of the predefined assignment if user-defined copy is provided or if a class contains references (attributes of access type).

1.5 Aggregation

1.5.1 Class instances as attributes of classes

An attribute of a class can be of a class type. In this case, the class aggregates an exclusively owned sub-object. This is the has-a relationship mentioned before. See above for finalization and assignment / copying of aggregated objects.

1.5.2 Class instances as elements of composite data types

A class type can as well be used to declare an element of a record or as the element type of an array.

1.6 Method invocation

When invoking a method, the following has to be specified: The object with which the method is to be invoked, the method to be invoked, and its actual parameters (if any). In Objective VHDL, the object is usually used stated as a prefix of the method call, and the suffix of the method call is like a VHDL subprogram call. An exception applies when method shall be invoked from within a method with the same object. In this case, the prefix can be omitted. Alternatively, it is possible to use the prefix *this* which refers to the object with which a currently executed method has been invoked. Such method call prefixed with *this* will, however, be dispatched dynamically, whereas the non-prefixed method invocation is statically bound.

Another possibility to call a method from within a class is to prefix it with the VHDL selected name of a class type. This is a mechanism to call a method implementation which has been overridden and is therefore hidden in a derived class.

From outside the class it is declared in and its childs, a method can only be called with an object as a prefix. The object can be a scalar object (variable, signal, constant, or class attribute of a class type), a class-typed element of an array or record, a dynamically allocated object designated by an

access value, or a class-typed return value of a function.

1.6.1 *Methods designated by identifiers*

An invocation of method `is_full` from method `put` of class `lifo` can be written as follows:

```
type lifo is class body -- lifo is derived from buffer
  procedure put( item : item_type ) is
  ...
  assert not is_full report „buffer overflow“ severity failure;
```

It is equivalent to write:

```
assert not WORK.lifo_package.lifo.is_full report ...
```

We could also write:

```
assert not this.is_full report ...
```

In the last case, however, the call to `is_full` would be dynamically dispatched to the overridden version of `is_full` if the inherited `put` is used with an object of a derived class type in which `is_full` is overridden. In the first case, the `is_full` directly visible in the class type in which `put` was declared is called even if `is_full` is overridden later by inheritance. The second case is equivalent to the first in the above example. Assuming that `is_full` was implemented in `buffer` and overridden in `lifo`, the use of a selected name would allow to call the hidden implementation of `buffer`:

```
assert not WORK.buffer_package.buffer.is_full report ...
```

To invoke the `put` method with an object, we can write:

```
my_buf.put( my_item ); -- if my_buf is of type lifo
my_rec.buf_element.put( my_item ); -- if buf_element is an element of type lifo in record my_buf
my_buf(1).put( my_item ); -- if my_buf is an array of lifo and 1 is in the legal index range
```

Assuming that function `give_buf` returns type `lifo`, the previous explanations suggest another possibility:

```
give_buf( some_parameter ).put( my_item ); -- illegal!
```

This, however, is illegal since only function methods that use the object like a constant can be invoked with a return value of a function. Everything else would be clueless because a modification of the returned object has no effect if it is stored nowhere. The following, however, is permitted; note that the modifications are stored in `my_buf` whereas they were simply disposed in the previous example:

```
my_buf := give_buf( some_parameter );
my_buf.put( my_item );
```

A legal example of a method call prefixed by a return value is:

```
my_bool := give_buf( some_parameter ).is_full;
```

1.6.2 *Methods designated by operator symbols*

The Objective VHDL LRM (currently) does not mention the use of operator symbols to declare methods. Since VHDL rules apply for the details of method subprogram declarations, operator

symbols should be permitted; this may have to be made more explicit in the LRM. Currently, it does not appear to be well-defined. Particularly, the number and kind of parameters required for an operator method, as well as the possibility to define an infix notation, e.g., $a + b$ instead of $a.+(b)$, deserve more attention.

1.7 Polymorphism

1.7.1 Class hierarchy types

Objective VHDL defines the class-wide type C'CLASS for each class (or derived class) type, also for abstract ones. C'CLASS is a type which is assignment-compatible with C and all children of C.

1.7.2 Objects of class-hierarchy types

A signal, variable, constant, or class attribute can be declared with type C'CLASS. It is possible to assign to such class-wide object a value of type C or derived. By multiple assignments during run-time (except for constants), the exact class membership of the contained value can change. Therefore, run-time information called tag is managed with each individual class-wide object.

1.7.3 Dynamic dispatch

It is allowed to call the methods which are declared in C with an object of class-wide type C'CLASS since all classes derived from a class C inherit the attributes and methods declared in C. Additionally, it is possible that the method is overridden in derived classes. Then, the subprogram call will be dispatched to the implementation visible in the class to which the object belongs. Since this membership may change dynamically, the mechanism is called dynamic dispatch.

1.7.4 Subtype compatibility and type conversion

As previously mentioned, an object of class-wide type C'CLASS can be assigned a value of type C or derived. In the following, we will consider also assignment targets of class type and assigned expressions of class-wide type.

An assignment of a class-wide expression (e.g., a function returning C'CLASS) is potentially compatible with a target of type C or derived. It succeeds if the tag of the assigned expression indicates that the expression is of type C, and fails otherwise.

A class-wide expression of type C'CLASS can always be assigned to a target of type P'CLASS if P is a parent of or equals C. Should P be derived from C, the assignment is potentially compatible, succeeding if the tag of the expression indicates type C or derived, and failing otherwise.

Two class types, even if they are in an inheritance relationship, are never subtypes of each other; they are individual types instead. The relationship among class-wide types has more similarities to subtypes: The values represented by a class-wide type C are a superset of those of a class-wide type D if D is derived from C; thus, D can be considered a subtype of C. However, whereas VHDL allows the assignment between any two subtypes that have the same base type, static checking of assignments involving class-wide type is more restrictive and catches more errors before simulation (see next section).

Assignments involving a class-wide type can be understood as a kind of implicit type conversion. There are no other implicit nor pre-defined conversions available. In particular, it is not possible to map, e.g., a formal subprogram parameter of type `C'CLASS` to an actual of type `C` or derived. (This would be desirable in some cases, but it would affect the overloading resolution mechanism, and the implicit conversion might be difficult where VHDL doesn't allow them, e.g., for subprogram parameters of class signal and variable.) The user can, however, call user-defined conversion functions wherever appropriate. There will be dynamic dispatch of these functions only if they are methods in the Objective VHDL sense and invoked with a prefix which is an object of class-wide type.

1.7.5 Staticness of types

All VHDL types are checked statically, and subtype compatibility is checked during simulation (run-time). Class-wide types of Objective VHDL are checked statically as well; however, if a potential compatibility (see section above) is detected, the assignment passes analysis and may fail during run-time. In that sense, class-wide type are less restrictively checked during analysis than VHDL types, yet they are more restrictive than subtypes.

1.7.6 Semantics for signals of class-hierarchy types

It is possible to declare a resolved signal of class type or class-wide type. The resolution function has to be written like in VHDL, i.e., it is not a method declared in the class. To access the driving values, it can (and must) therefore use pure function methods of the class. It should be emphasized that a special modeling style may have to be developed to really make use of class-wide type resolution. E.g., a class type corresponding to `std_logic`'s 'Z' could be declared in order to represent a non-driving value, and the resolution function could check that only one of the drivers has a value different from that type at all times. Alternatively, it would be possible to use guarded signals to turn off drivers.

There are no implicit type conversions done for class type or class-wide types in association elements of port maps, generic maps, and subprogram parameter association lists.

There exist no pre-defined conversion functions for class types and class-wide types. The user can define and call such functions where appropriate according to VHDL rules.

2 Object-oriented structure and behavior modeling (component abstraction)

These capabilities were included in Objective VHDL upon the request of users who are used to modeling based on structural decomposition of entities and architectures, in accordance with the VHDL notion of the design entity being „the primary hardware abstraction“. Another point is the current commercial emphasis on reusable IP blocks which are mostly described as design entities. The addition of OO features to the design entity was believed to allow a more flexible reuse by enabling modifications to an existing IP component through inheritance. Furthermore, class types (or their tagged record equivalents) were identified to be problematic when concurrent modeling comes into play.

Objects of class types can be considered as passive objects whose methods are executed with

the resources (thread of computation) of the caller. If such an object is to be used concurrently, it must be placed in a declarative region visible to all concurrent callers. This would mean to declare it as a shared variable, with (currently) no synchronization of access, or as a signal with low-level conflict resolution.

It would be possible to make an object of a class type an active one by declaring it in a process and making other objects communicate with it via signals or channels. Such active object would be limited to one internal thread of computation and thereby / therefore have to sequentially service incoming messages (operation requests). Objective VHDL does not preclude this possibility. It should be emphasized, however, that the communication code for receiving messages from channels or doing a signal protocol has to be written by the user, and that this task is not straightforward. In particular, writing communication / synchronization code that is reusable with derived classes requires much attention.

The idea behind Objective VHDL's derived design entities is to provide active, component-based objects. Objective VHDL provides the declaration of derived entities and derived architectures which inherit everything from their respective parent (single inheritance). A derived entity may add generics, ports, any kind of item allowed in the declarative part, and (passive) concurrent statements to those inherited. Similarly, a derived architecture can add any kind of item allowed in the declarative part and concurrent statements. It is possible to replace a labeled inherited concurrent statement by overriding it with a new one using the same label.

The notion of methods and attributes is brought into play by interpreting signals and shared variables of the design entity as attributes and procedures and functions which are declared in an entity (class interface) and implemented in an architecture (class implementation) as methods. However, VHDL encapsulation mechanisms do not allow to call these methods from outside the entity, and Objective VHDL either doesn't provide such method calls nor any other invocation mechanism from outside the design entity. Therefore, the user is required to model communication. A modeling proposal based on channels modeled with class types and class-wide types exists and is supported by a tool that creates part of the required code. Objective VHDL has a special construct, the for entity configuration, which intends to support the communication modeling by making the declarations of an entity visible to an object which is instantiated inside of that entity, even though the declaration of the object's class is outside of the entity. There is ongoing research to improve these capabilities and reduce the required amount of code with the help of additional language support.

Even without methods and attributes, one can make use of derived design entities, e.g., to quote a rather low-level example, for adding a scan path to a gate level circuit, which involves the declaration of additional ports, internal connection signals, and overriding the instantiations of D-type flip-flops by scan flip-flops. However, the addition of processes or other concurrent statements to interact with the inherited ones requires the user to analyze and understand the implementation of the inherited behavior.

3 Genericity

3.1 Formal-constant genericity for types

The possibility to declare generic constants was added to the class types in a recent version (1.2)

of the Objective VHDL LRM. The reason is that without generic constants it proved impossible to parameterize the size of data structures, e.g., of a buffer, in a way that does not involve dynamic allocation. A generic class type contains a generic clause as its first declaration, e.g.:

```
type buffer is
  generic( n : positive );
  type item_array is array( 0 to n-1 ) of item_type;
  ...
```

Note: if `item_type` shall be generic, it can be declared as the class-wide type of an (abstract) class type. Subprograms declared in this (abstract) class type can be considered a substitute for formal subprograms. For instance, a hash function for items could be declared as a method of that (abstract) class type.

Like all other declarations, generic class types are inherited to derived class types. These may declare additional generics.

When instantiating a generic class type, a generic map must be provided. Thereby, a constrained subtype of the unconstrained generic class type is declared, either explicitly by the user or implicitly if the user combines subtype and object declaration:

```
subtype buffer_16 is buffer generic map( n => 16 );
variable my_buffer : buffer_16;
-- or, equivalently:
variable my_buffer : buffer generic map( 16 );
```

This corresponds to the declaration of constrained subtypes from array types with unconstrained ranges. Like with any VHDL subtypes, semantic analysis allows assignments between any subtypes which have the same base type. A run-time error occurs due to such assignment if the actual generic values of the assigned value do not match those of the assignment target.

A problem was identified when using a class-wide type of a generic class type `C` or a class-wide type of a class type `C` from which a generic class type is derived. An object of class-wide type can be assigned a value of type `C` or the derived type, regardless of the actual generic values. Due to the value-based semantics and the translation approach of Objective VHDL, this requires sophisticated processing and knowledge of the actual generic values which are not known before elaboration. Currently, such use of generic class types cannot be handled by the translator. Anyway, generic class types were defined regardless of translatability.

3.2 Formal-type genericity

Objective VHDL does not provide formal type genericity, primarily for complexity concerns. Neither formal types nor formal subprograms are currently part of VHDL, and respective semantic analysis would therefore have to be added to an analyzer. Likewise, elaboration will be complicated by having to consider not only the propagation of generic values, but also of actual types and subprograms passed as generic. Another particular concern of the Objective VHDL analyzer supplier is the need to make packages instantiatable. Moreover, the local declaration and instantiation of packages is likely to require additional restrictions, e.g. on the declaration of derived types in local packages, which complicate analysis further.

Instead, many things that can be modelled with type genericity, e.g., container classes, can as well be modelled with an abstract class and its corresponding class-wide type instead of the formal type. From the abstract type the classes to be contained (which would be passed as actual types in the generic approach) are to be derived.

4 Abstract communication and concurrency

When starting the effort on Objective VHDL, an abstract communication mechanism was believed to be beyond the scope of object-oriented extensions. Later it became apparent that in particular the component-based objects would profit from such a mechanism. Yet no such mechanism was included in the language since its definition is far from straight-forward. A major concern is whether a built-in mechanism can provide the flexibility to model all kind of communication that may be needed. This is especially the case if the built-in mechanism claims exclusiveness, i.e. prevents the use of user-defined mechanisms. Furthermore, even a less flexible but easy-to-use communication mechanism that integrates well with inheritance and polymorphism isn't known to us. This refers especially to the problem of reusing synchronization code when deriving classes from which concurrent active objects are instantiated. The problem, known as inheritance anomaly and not even solved in the object-oriented concurrent programming (SW) community, reveals that in a concurrent context inheritance and communication cannot be considered orthogonal.

The addition of concurrency features such as process interfaces and dynamic creation of processes was and is beyond the scope of Objective VHDL not only because they are not core OO, but also because of the hardware focus of Objective VHDL. Dynamic processes may be useful at system level, in particular when describing systems which include software, but they are believed to be unneeded when describing hardware. As far as software is concerned, the question arises whether a modeling methodology based on a language which originates from an HDL will be commercially successful, even if it is possible from a scientific point of view. Regarding HW/SW co-specification using a single language, object-oriented VHDL would be in competition with SLDL, which actually has a much broader scope, including requirements and constraints specification.

5 Tool implications

The main objective of Objective VHDL is to improve reusability and raise the abstraction level in hardware modeling. It is less focused on system-level which is rather covered by groups such as SLDL and SID. Considering a design flow in which a user writes an abstract initial object-oriented model and refines it, using object-oriented techniques, to the degree required for a hardware implementation, we find it desirable to generate hardware automatically from the refined model so as not to have the user replace all object-oriented constructs by equivalent VHDL descriptions. This means that the use of constructs for class type declaration, derivation, polymorphism, and genericity should not generally prevent synthesis. It does not mean that every use of them, e.g., the initial abstract model, must be synthesizable.

Regarding Objective VHDL, we can say that the extensions to VHDL are synthesizable as long

as they are used together with synthesizable VHDL. They may, however, require sophisticated analysis in order to generate efficient hardware. For instance, this is the case with the class-wide types.

In order to be able to use a VHDL synthesis back-end for logic synthesis from RT or behavioral level and for the lower-level optimizations, it is also of importance to be able to translate to VHDL. Regarding simulation, experiences have shown that simulating (and debugging) on the level of translated code can only be a temporary solution for first evaluations. Better is to have a front-end that hides VHDL-based simulation from the user and displays on OO level. Native simulation is still the best option, but may be most expensive to achieve, in particular if commercial considerations apply to the performance of the simulator.

Section III

Comparison of SUAVE and Objective VHDL Language Features

Peter J. Ashenden, The University of Adelaide
Martin Radetzki, University of Oldenburg

1 Object-oriented data modeling

1.1 Class interface definition

1.1.1 Type definition

SUAVE and Objective VHDL are similar in providing type-based class concepts which allow to encapsulate data and method implementations from the user of the class.

1.1.2 Method definition

While methods are defined inside the class type definition of Objective VHDL, they are defined after the tagged record type definition of SUAVE. The Objective VHDL approach thereby groups all data and methods in the class construct which can be seen as the definition of an abstract data type. It gives methods implicit access to the attributes. SUAVE, on the other hand, with operations declared external to the type declaration, requires the explicit declaration of at least one subprogram parameter of the tagged record type in order to identify the subprogram as an operation of that class, and in order to give it access to attributes. Furthermore, SUAVE relies on good programming style of including only the data type and operations of a single class in a package, and not to mix them up with other declarations. While the class construct of Objective VHDL might be seen positive for providing and enforcing better code organization, it requires the introduction of the object configurations („for signal“ etc.). These add considerable complexities to the language. SUAVE lets the user define the storage class of objects in the explicit object parameter declaration(s). Similar considerations apply to the mode (in, out, inout) of the object parameter, which is explicitly declared in SUAVE and has to be determined by code analysis in Objective VHDL.

1.1.3 Non-instantiable (abstract) classes

The semantics of declaring an abstract class are similar in both languages.

1.1.4 Non-invocable (abstract) methods

Both languages permit abstract methods only with an abstract class. Whereas SUAVE requires an abstract method to be explicitly declared abstract, Objective VHDL implicitly regards a method as abstract if no implementation is provided. The difference is that SUAVE can immediately detect the omission of an implementation of a non-abstract method in an abstract class. In Objective VHDL this error will be caught when a non-abstract class is derived. A consolidation is technically feasible in both ways, introducing explicit declaration of abstract methods into VHDL,

or making abstractness an implicit property in SUAVE.

1.2 Encapsulation and visibility control

1.2.1 Encapsulation of attributes

In SUAVE, attributes are either visible without restriction („public“) or visible only in the implementation of the class itself („private“). Objective VHDL, on the other hand, makes attributes visible either to the implementation of the class and all its derived classes („protected“) or only to the class itself („private“), but never to the public. It would be possible to introduce protected-style attributes into SUAVE through child packages similar to Ada. In Objective VHDL, a declaration of attributes as being public could be added.

Objective VHDL does not provide a means to construct a value of a class type by a kind of aggregate. A class type aggregate could, however, be provided at least in the class implementation (where the attributes are visible) without breaking encapsulation. If public attributes should be added, additional considerations might apply.

1.2.2 Encapsulation of method implementation

Both languages encapsulate method implementations, but in different locations. SUAVE uses package bodies for this purpose, whereas Objective VHDL provides class bodies. This is not a fundamental difference. Differences are related rather to the style of method definition.

1.3 Inheritance

Objective VHDL and SUAVE provide single inheritance of classes. Semantics and the intentions behind are similar.

1.3.1 Specifying inheritance

The languages provide similar constructs to specify a new class as derived from a named parent class. A difference is that SUAVE permits the declaration of derived types also for non-class types. However, this is irrelevant regarding object-orientation.

1.3.2 Augmenting inherited attributes

Both languages allow the addition of attributes in derived classes.

1.3.3 Augmenting inherited methods

Both languages allow the addition of methods in derived classes.

1.3.4 Overriding inherited methods

Both languages allow to override an inherited method in a derived class. A minor difference can be seen in that Objective VHDL requires not only the parameter storage classes and modes, but also their names to match between the overriding and overridden methods. This, however, could be subject to change.

1.4 Instantiation

SUAVE and Objective VHDL are similar in allowing the instantiation of classes as constants, signals (also: port signals), variables, and as subprogram parameters of these storage classes.

1.4.1 Instantiation of constant objects

The same in Objective VHDL and SUAVE.

1.4.2 Instantiation of declared variable objects

The same in Objective VHDL and SUAVE.

1.4.3 Instantiation of dynamically allocated variable objects

The same in Objective VHDL and SUAVE.

1.4.4 Instantiation of signal objects

Both languages permit the declaration of signal objects, but do not permit the signal instantiation of a class which has an attribute of access type. SUAVE requires the user to declare this access property explicitly and may add this feature also to the VHDL records as an enhancement to the language design. Objective VHDL, on the other hand, is guided by the implicit handling as currently found in VHDL.

1.4.5 Parameter passing

When passing variable parameters of a class type, Objective VHDL (like VHDL) does not specify whether they are to be passed by reference or by copy, whereas SUAVE specifies a reference mechanism. It would be possible to restrict variable parameter passing to by-reference. Allowing parameter passing by value in SUAVE would be considerably harder due to the view conversions provided by the language.

1.4.6 Initialization

SUAVE does not allow user-specified initial values of attributes, Objective VHDL does. In this regard, SUAVE follows VHDL which does not provide user-defined initial values of record elements. Such initialization could, however, be provided as it is in Ada. Of course it would be technically possible to omit initialization expressions from attributes in Objective VHDL.

1.4.7 Finalization

There is no user-defined and automatically called finalization in both languages.

1.4.8 Assignment and controlling copying

SUAVE enables the user to prevent pre-definition of (flat) assignment and equality operations by declaring a class as limited, and to enforce the use of user-defined copy and equality operations instead. While it is possible to define such operations in Objective VHDL, it is not possible to prevent the use of the pre-defined ones. Languages could be reconciled by adding the explicit

declaration of the limited (and also the access) property to Objective VHDL, or by removing them from SUAVE.

1.5 Aggregation

1.5.1 Class instances as attributes of classes

Both languages provide class instances as attributes of classes by declaring attributes to be of class type.

1.5.2 Class instances as elements of composite data types

Both languages provide class instances as elements of composite data types by using an class type as the element type.

1.6 Method invocation

Significant differences regarding method invocation exist between Objective VHDL and SUAVE. These differences primarily originate from the way classes, and in particular methods, are declared.

1.6.1 Methods designated by identifiers

Objective VHDL uses a prefixed notation similar to C++ to invoke a method with a specified object. SUAVE, on the other hand, uses explicit parameter passing to pass the object to a method. Yet, the desired effect can be achieved in both languages.

1.6.2 Methods designated by operator symbols

SUAVE allows for infix operator methods as per standard VHDL. In Objective VHDL, this is currently not possible, but could to some extent be achieved either by specifying operator methods in the language (still, some restrictions relating to commutativity of operations apply) or by providing something similar to friend functions of C++.

1.7 Polymorphism

1.7.1 Class-hierarchy types

The languages use the same class-wide type approach.

1.7.2 Objects of class-hierarchy types

Signal and constant objects of class-wide types as well as parameters of class-wide types are handled similarly in the two languages. Class-wide signals require an elaboration-time check for attributes of access types in derived classes, since these may not yet be known when analyzing the class-wide signal instantiation. A major difference can be seen when considering variable objects: Objective VHDL allows variables (declared and dynamically allocated) to be of class-wide type and to take on values of different specific types during their lifetime. SUAVE provides class-wide

variable objects through access values which designate a class-wide type, i.e. only with dynamic allocation. Such access value can point to values of different specific types during its lifetime; however, the type of an allocated object cannot change. It is not possible to bring these paradigms together; an exclusive decision must be made for one of them.

1.7.3 Dynamic dispatch

Objective VHDL and SUAVE bind method calls statically if the specific types of the actual parameters (in the case of Objective VHDL, the specific type of the object prefix of the method call) are (is) statically known. Otherwise, dynamic dispatch is applied. Hence, calls of overridden methods cause dynamic dispatch only when required and not per default such as for all virtual methods in C++. A minor difference is in that a SUAVE method may dispatch on more than one parameter. However, rules apply to reduce each dispatching call to a single decision.

1.7.4 Subtype compatibility and type conversion

SUAVE and Objective VHDL define the same rules for implicit conversions to allow assignments involving class-wide types. An assignment between different class types, however, is possible only in SUAVE using pre-defined view conversions or extension aggregates. If the desired effect of a view conversion is to call an overridden method inherited from a parent type without losing the additional data of the derived type, the same can be achieved in Objective VHDL through a call using the parent class name as a prefix. The pre-defined name 'this' can then be used to allow the called parent method to invoke methods of the derived class (since the parent method has actually been called with an object of the derived class). Still, this is limited to the class implementation, whereas in SUAVE any client of an object can invoke a parent's method with the object. It should be determined whether this additional freedom is desirable.

1.7.5 Staticness of types

The languages' approach to polymorphism preserves the staticness of types where possible. Only the specific type of class-wide types is dynamic. Therefore, in some cases of implicit or explicit conversions run-time checking may be involved.

1.7.6 Semantics for signals of class-hierarchy types

Objective VHDL relies on VHDL mechanisms for resolved class-wide signals. Special modeling styles may have to be adopted in order to make reasonable use of this feature. In SUAVE, resolution functions cannot be declared for class-wide types because it is not possible to define an array type of class-wide elements.

2 Object-oriented structure and behaviour modeling (component abstraction)

While Objective VHDL includes entity/architecture structural inheritance, SUAVE doesn't. A question to be addressed is whether this is in scope for the OO-VHDL study group or whether we should focus on the type-based part. If proposed to be included in a standard, the requirements that lead to structural inheritance should be re-evaluated. Furthermore, the existence of advantages of structural inheritance over hierarchical composition should be addressed.

3 Genericity

3.1 Formal-constant genericity for types

While Objective VHDL provides generic constants only with the class types, SUAVE makes them available also to packages and non-method subprograms.

3.2 Formal-type genericity

SUAVE provides formal types, subprograms, and packages in addition to the formal generic constants. It is technically feasible to add these features to Objective VHDL, either by introducing them into packages, or by adding them only to the class types. The question is if this is in the scope of the OO-VHDL study group, and if we want to include it into a standard.

4 Abstract communication and concurrency

Objective VHDL defines entity-based objects to provide active objects. Communication among these objects is to be modelled by the user with little support from the language. SUAVE does not deal with active objects at all (not even in the type domain); the value of active objects in general is questioned. Instead, language features for abstract communication based on typed channels and message passing are provided. It should be addressed whether active objects are in scope of the OO-VHDL study group. The communication features of SUAVE are discussed in the SID group anyway.

5 Tool implications

5.1 Simulation

While SUAVE developers aim at the implementation of a native simulator, Objective VHDL currently has to be (and can be) translated into VHDL for simulation. The latter approach is not feasible for SUAVE, not even for the object-oriented part of it. Implementation of an Objective VHDL simulator is feasible.

5.2 Synthesis

SUAVE as a whole addresses system level specification rather than synthesis. Yet, large portions of the object-oriented constructs of SUAVE are synthesizable. Synthesis is at least difficult for the view conversions and the pointer-based semantics of polymorphism. These might have to be excluded from a synthesis subset of SUAVE. Objective VHDL, on the other hand, defines value-based polymorphism and avoids view conversions since (high-level) synthesis is considered an important application domain of the language.

Section IV

Considerations on Nebel's Proposal for Class Types

Peter J. Ashenden, The University of Adelaide
 Martin Radetzki, University of Oldenburg

Introduction

Wolfgang Nebel proposed a compromise between the class-type features of Objective VHDL and the features used to construct class definitions in SUAVE. Nebel proposed that a class type have methods that explicitly name parameters whose type is the class, rather than the object on which a method is invoked being an implicit parameter of a method. This document summarizes our exploration of this idea. We use the term *element* to refer to a data attribute of a class type, to avoid confusion with the existing notion of attributes in VHDL and with static data members of a class in C++.

First Cut

We first note that a class type definition must include definition of the class elements. If methods are to have explicit parameters of the class type, the elements must be accessible for each parameter. This suggests use of the selected name notation to refer to elements. Thus an element name should be visible in a selected name whose prefix is a name of the class type. However, the scope of a class element should not be the entire declarative region formed by the class declaration and body, since there is no context (object) in which to interpret the element name.

These considerations lead us to a form of class definition comprising an element definition part and a method definition part. A class body is a separate part of a class definition. We also considered the issue of visibility of element and method names outside the definition, and proposed allowing specification of "public", "protected" and "private" elements and methods. An example illustrating this approach follows, using prototype syntax. The class defines an abstract data type for complex numbers.

```

type complex is
  class
    private re, im : real;
  with
    function cons ( re, im : real ) return complex;
    function "+" ( L, R : complex ) return complex;
    ...
  end class complex;

type complex is
  class body
    function cons ( re, im : real ) return complex is
    begin
      return complex'( re, im );
    end function cons;
  
```

```

function "+" ( L, R : complex ) return complex is
begin
  return complex'( L.re + R.re, L.im + R.im );
end function "+";

```

```

...
end class body complex;

```

The class type could be used as follows:

```

constant i : complex := cons(0.0, 1.0);
variable a, b : complex;
signal s : complex;
...
s <= a + i * cons(2.0, 0.0);

```

This example illustrates that such a class type is a form of composite type similar in nature to a record: it contains elements of heterogeneous type. The element names are visible in selected names and in aggregates. In this example, such visibility is limited to the class definition, due to the word **private**. Elements marked **protected** would also be visible in selected names and aggregates in derived classes. Elements marked **public** would be further visible in selected names and aggregates anywhere that the class name was visible.

Second Cut

The proposal shown above works if the elements can be defined directly using a subtype indication. If, on the other hand, intermediate declarations must be included to define the type of an element, the question of where to put the intermediates declarations arises. It would be desirable to hide the intermediate declarations within the class definition, since they are concrete implementation details that should be encapsulated.

This lead us to consider two alternatives:

- Ignore the problem—intermediate declarations would then be declared outside the class. The fact that such declarations were then used in the class could be hidden.
- Divide the class declaration into public, protected and private parts. Within each part, allow declarations of methods, constants, types, subtypes, etc., as well as *element declarations*.

We show two examples, each worked out using the two alternatives. The first example is an abstract data type for a sequence of test vectors. Using the “ignore the problem” approach:

```

type test_seq;
type test_seq_access is access test_seq;
type test_seq is
  class
    private head_test_vector : test_vector;
    private tail_test_seq : test_seq_access;
  with
    function is_empty ( seq : test_seq ) return boolean;
    function head ( seq : test_seq ) return test_vector;
  ...
end class test_seq;

```

Here, the type declarations used for the linked-list implementation of the sequence are exposed, but the class elements using the types are hidden.

The same example using the “visibility parts” approach is:

```
type test_seq is
  class
    function is_empty ( seq : test_seq ) return boolean;
    function head ( seq : test_seq ) return test_vector;
    ...
  private
    type test_seq;
    type test_seq_access is access test_seq;
    element head_test_vector : test_vector;
    element tail_test_seq : test_seq_access;
  end class test_seq;
```

Here, the intermediate type declarations are hidden in the private part of the class definition, and are subsequently used to declare the class elements. Note the need for a new kind of element declaration, to identify head_test_vector and tail_test_seq as being elements of the class type.

The second example is an abstract class for a buffer, from which we derive a class for a buffer with LIFO discipline. A similar derived class with FIFO discipline could also be derived. Using the “ignore the problem” approach:

```
constant buf_size : natural := 100;
type buf_array is array ( 1 to buf_size ) of item;
type buf is
  abstract class
    protected store : buf_array;
  with
    function is_empty ( B : buf ) return boolean is abstract;
    function is_full ( B : buf ) return boolean is abstract;
    procedure add ( B : inout buf; E : item ) is abstract;
    ...
  end class buf;
type LIFO_buf is
  new buf class
    private top : natural := 0;
  with
    function is_empty ( B : LIFO_buf ) return boolean;
    function is_full ( B : LIFO_buf ) return boolean;
    procedure add ( B : inout LIFO_buf; E : item );
    ...
  end class LIFO_buf;
```

In this case, the element store is visible in the derived class LIFO_buf. The constant buf_size and the type buf_array, though used in the class buf, are not re-exported to the derived class. If the derived class were defined in a separate declarative region from the parent and needed visibility of buf_size and buf_array, it would have to import them separately.

The same example using the “visibility parts” approach is:

```
type buf is
  abstract class
```

```

function is_empty ( B : buf ) return boolean is abstract;
function is_full ( B : buf ) return boolean is abstract;
procedure add ( B : inout buf; E : item ) is abstract;
...
protected
  constant buf_size : natural := 100;
  type buf_array is array ( 1 to buf_size ) of item;
  element store : buf_array;
end class test_seq;
type LIFO_buf is
  new buf class
    function is_empty ( B : LIFO_buf ) return boolean;
    function is_full ( B : LIFO_buf ) return boolean;
    procedure add ( B : inout LIFO_buf; E : item );
    ...
  private
    element top : natural := 0;
  end class LIFO_buf;

```

In this case, the declarations of `buf_size` and `buf_array` are visible in the `LIFO_buf` class, as is the store element, but none of the elements are visible to users of the classes.

Coda

An important point to make about the proposed class type definition is that the methods names, while being declared in the declarative region of the class definition, should also be implicitly exported to the surrounding declarative region. They should not be implicitly “used” in the way names in packages can be used, since used names do not become visible if they are homographs of other names in the outer region. Rather, they should be analogous to the implicitly declared operations that correspond to other type declarations. (For example, when a scalar type is declared, the relational operations are implicitly declared.) The difference is that they should not be hidden by explicitly declared homographs in the outer region. It should be illegal to have a subprogram that is a homograph of a method of a class in the same declarative region.

The reason for having the method declared within the class-type region (instead of the outer region directly) is so that, in the class body, its name is declared at the right level and can't be hidden by a homograph declared immediately within the class body.