This work has been accepted for publication in the *Proceedings of FDL '98*.

SUAVE: Object-Oriented and Genericity Extensions to VHDL for High-Level Modeling

Peter J. Ashenden
Dept. Computer Science
The University of Adelaide, SA 5005
Australia
petera@cs.adelaide.edu.au

Philip A. Wilsey and Dale E. Martin
Dept. ECECS, PO Box 210030
University of Cincinnati
Cincinnati, OH 45221-0030, USA
phil.wilsey@uc.edu, dmartin@ececs.uc.edu

Abstract

The SUAVE project aims to introduce object-oriented and genericity extensions into VHDL in a way that does not disturb the existing language or its use. Designers regularly define abstract data types by using aspects of VHDL's type system, subprograms, and packages. They also use VHDL's genericity mechanism to parameterize component and entity declarations with formal generic constants. The SUAVE approach builds on these basic mechanisms by strengthening the facilities for encapsulation and adding an inheritance mechanism. It also extends the genericity mechanism by allowing formal generics types and by allowing generics to be specified in the interfaces of subprograms and packages. The SUAVE extensions are based on the features of Ada-95. They allow units to be re-used in a much wider variety of contexts without modifying the original code. By choosing an incremental and evolutionary approach to extensions, SUAVE avoids major additions to the language that would complicate choice of mechanisms for expressing a design. This paper outlines the SUAVE extensions and illustrates their use through some examples.

1. Introduction

VHDL is widely used by designers of digital systems for specification, simulation and synthesis. Increasingly, designers are using VHDL at high levels of abstraction as part of the system-level design process. At this level of abstraction, the aggregate behavior of a system is described in a style that is similar to that of software. Data is modeled in abstract form, rather than using any particular binary representation, and functionality is expressed in terms of interacting processes that perform algorithms of varying complexity. A subsequent partitioning step in the design process may determine which aspects of the modeled behavior are to be implemented as hardware subsystems, and which are to be implemented as software.

Experience in the software engineering community has lead to adoption of object-oriented design and programming techniques for managing complexity through abstract data types (ADTs) and re-use [10]. Features included in programming languages to support these techniques are abstraction and encapsulation mechanisms, inheritance, and genericity. The term "object-based" is widely used to refer to a language that included abstraction and encapsulation mechanisms [24]. The term "object-oriented" is used to refer to a language that additionally includes inheritance.

While VHDL can be used for modeling at the system level, it has some deficiencies that make the task more difficult than it would otherwise be. These difficulties center around language features (or lack of some features) for supporting complexity management. VHDL is currently somewhat less than object-based, as its encapsulation mechanism are weak. It is certainly not object-oriented, as it does not include any form of inheritance. While it does include a mechanism for genericity, that mechanism is severely limited, allowing only parameterization of units by constant values. We have discussed these issues in a previous paper [3].

SUAVE aims to improve support for high-level modeling in VHDL by extending the language with features for object-orientation and genericity in a way that does not disturb the existing language or its use. As well as adding specific language features, some existing features are generalized, the facilities for encapsulation are strengthened, and an inheritance mechanism is added. Private types and private parts in packages support improved encapsulation. Type derivation, record type extension, and class-wide types with dynamic dispatching support inheritance.

SUAVE also extends the genericity mechanism of VHDL by allowing types to specified as formal generics and by allowing generics to be specified in the interfaces of subprograms and packages. Use of formal type generics allows units to be reused in contexts where data of different types is to be manipulated. For example, a multiplexer can be specified with a formal type generic for the type of the input and output data. This allows the multiplexer model to be reused as a bit, bit_vector, std_logic, std_logic_vector, integer, or user-defined-type multiplexer, without modifying the original model code. Use of generics in the interfaces of subprograms and packages allows definition of container abstract data types that can be reused to contain data of different types. For example, a generic package can be defined to represent and manipulate sequences of integer, time values, or test vectors for different devices under test, again without modifying the original package code.

We have previously argued [4] that, in addition to supporting object-orientation, these extensions improve the expressiveness of VHDL and enhance reuse across the modeling spectrum from high-level to gate-level. Furthermore, the genericity extensions interact with the extensions for object-oriented data modeling to significantly improve support for high-level behavioral modeling and for developing test-benches. By choosing an incremental and evolutionary approach to extensions, SUAVE avoids major additions to the language that would complicate choice of mechanisms for expressing a design. In addition, the implementation burden is not large, and there is no performance penalty in simulation or synthesis if the mechanism are not used.

The SUAVE approach is similar to that proposed by Mills [17] and by Schumacher and Nebel [20]. It is contrasted with others that have been proposed [9, 12, 19, 22, 25], that add new, separate mechanisms for combining abstraction, encapsulation, and inheritance for object-orientation. Such mechanisms replicate aspects of the existing features of VHDL, making design choices for expressing a model more complex.

* This work was partially supported by Wright Laboratory under USAF contract F33615-95-C-1638.

This paper outlines the SUAVE extensions for object-orientation and genericity, and illustrates their use through some examples. More complete presentation of the extensions can be found in the SUAVE report [6].) Most of the features added to VHDL are adapted from features in Ada-95 [16], and are included largely for the same reasons that they are included in Ada-95 [8]. Section 2 of this paper outlines the design principles and objectives that were followed in deciding how to extend VHDL. Subsequent sections describe the extensions in detail and illustrate them with examples. Section 3 describes the extensions to the type system of VHDL to support type derivation, extension and class-wide programming. Section 4 describes the extensions that improve the encapsulation features of VHDL. In combination, the extensions in these two sections turn VHDL into an object-oriented language. Section 5 describes the SUAVE type genericity extensions, and Section 6 describes and illustrates further extensions for subprogram and package genericity. Our conclusions are presented in Section 7.

2. SUAVE Design Objectives

A previous paper [4] reviews the issues to be addressed in extending VHDL for high-level modeling and discusses principles that should govern the design of language extensions. As a result of that analysis, a number of design objectives were formulated for SUAVE:

- to improve support for high-level behavioural modeling by improving encapsulation and information hiding capabilities and providing for hierarchies of abstraction,
- to improve support for re-use and incremental development by allowing further delaying of bindings through type-genericity and dynamic polymorphism,
- to preserve capabilities for synthesis and other forms of design analysis,
- to support hardware/software co-design through improved integration with programming languages (e.g., Ada),
- to support refinement of models through elaboration of components rather than through repartitioning, and
- to preserve correctness of existing models within the extended language.

Since SUAVE is an extension of the existing VHDL language, it is important that the extensions integrate well with all aspects of the existing language. In designing the SUAVE extensions, the design principals followed during the restandardization of VHDL that lead to the current language [15] were adopted in addition to those listed above. The goal was to preserve what Brooks refers to as the "conceptual integrity" of the language [11].

3. Extensions to the Type System

Object-oriented languages support re-use and incremental development through the mechanism of inheritance. SUAVE extends the type system of VHDL by adopting the object-oriented features of Ada-95, including inheritance through type derivation and tagged types with extension on derivation.

3.1 Derived Types and Inheritance

For a type defined in a package, the operations (procedures and functions) defined in the package are called the *primitive operations* of the type. A new type can be defined as being *derived* from a parent type. In that case, the derived type inherits the set of values and the primitive operations of the parent type. An inherited operation can be overridden by defining a new operation with the same name but with operands of the derived type. Furthermore, additional primitive operations can be defined for the derived type. SUAVE adopts the Ada notation for defining a derived type, for example:

```
type event_count is new natural;
```

The derived type is distinct from, but related to, the parent type. Use of derived types helps avoid inadvertent mixing of conceptually different values, and thus improves the expressiveness of the language.

3.2 Tagged Types and Type Extension

As in Ada-95, a record type in SUAVE may include the reserved word **tagged** in its definition. Such a type is called a *tagged type*. An object of a tagged type includes a run-time tag that identifies the specific type used to create the object. The tag is used for dynamic dispatching, which is described below. A tagged record type may be *extended* by deriving a new type with a *record extension* containing additional record elements. This is the origin of the term "programming by extension," sometimes used to describe the Ada-95 approach. The derived type is also a tagged type that can be further extended. Since all elements in the parent type are also in the derived type, inherited operations of the parent type can be applied to objects of the extended type. However, any overriding or newly defined operations for the extended type can only be applied to the extended type (or its derivatives), since they may refer to the elements in the extension.

As an example, consider a type and operations representing an instruction set for a RISC CPU. All instructions have an opcode. ALU instructions additionally have fields for the source and destination register numbers. Thus an ALU instruction can be considered as an extension of a base instruction with just an opcode. This can be expressed in SUAVE by defining the following in a package:

```
type instruction is
    tagged record
        opcode : opcode_type;
    end record instruction;
function privileged ( instr : instruction;
                      mode : protection_mode ) return boolean;
procedure disassemble ( instr : instruction;  file output : text );

type ALU_instruction is
    new instruction with record
        destination,
        source_1, source_2 : register_number;
    end record ALU_instruction;
procedure disassemble ( instr : ALU_instruction;
                        file output : text );
```

The subprograms privileged and disassemble are primitive operations of instruction and are inherited by derived types. The type ALU_instruction is derived from instruction and has four elements: the opcode element inherited from instruction, and the three register number elements defined in the extension. A version of the function privileged is inherited from instruction with the instr parameter being of type ALU_instruction. The disassemble instruction defined for ALU_instruction overrides that inherited from instruction.

3.3 Abstract Types and Subprograms

An *abstract type* is a tagged type that is intended for use solely as the parent of some other derived type. Objects may not be declared to be of an abstract type. An *abstract subprogram* is one that has no body (and requires none), because it is intended to be overridden when inherited by a derived type. Abstract types and subprograms allow definition of types that include common properties and operations, but

which must be refined by derivation of types that represent concrete objects.

As an illustration, consider refinement of the instruction type to represent memory reference instructions using displacement addressing mode. Such instructions include a base register number and an offset. The type for these instructions is declared abstract, since it is intended to be the parent type for load and store instruction types. More precisely,

```
type memory_instruction is
    abstract new instruction with record
        base : register_number
        offset : integer:
    end record memory_instruction;
function effective_address_of ( instr : memory_instruction );
procedure perform_memory_transfer
            ( instr : memory_instruction ) is abstract;
```

The function effective_address_of is not abstract, since it can calculate the result using the data in a memory_instruction record. The function can be inherited "as is" by derived types. The procedure perform_memory_transfer, on the other hand, is declared abstract since the direction of transfer depends on whether a memory instruction is a load or a store. The derived types must provide overriding non-abstract implementations of this procedure. Examples are derived types for load and store instructions, as follows:

```
type load_instruction is
    new memory_instruction with record
        destination : reg_number;
    end record load_instruction;
procedure perform_memory_transfer
            ( instr : load_instruction );
procedure disassemble ( instr : load_instruction;
                        file output : text );
type store_instruction is
    new memory_instruction with record
        source : reg_number;
    end record store_instruction;
procedure perform_memory_transfer
            ( instr : store_instruction );
procedure disassemble ( instr : store_instruction;
                        file output : text );
```

Objects cannot be declared to be of type memory_instruction, but they can be declared to be of type load_instruction or store_instruction.

3.4 Class-Wide Types and Operations

One of the most important aspects of object-oriented programming is the use of *classes*. SUAVE adopts the Ada-95 mechanism of *class-wide types* to deal with classes. This contrasts with languages such as Simula [13], C++ [21] and Java [14] that introduce a special construct for classes. (See our paper that compares the two approaches [2].)

Class-wide types are denoted using the 'Class attribute. For a tagged type T, the *class-wide type* denoted T'Class is the union of T and all types derived directly or indirectly from T. The type T is called the *root* of the class-wide type. For example, the class-wide type instruction'class denotes the hierarchy of types rooted at instruction, and including ALU_instruction, memory_instruction, load_instruction and store_instruction.

An object of a class-wide type can have a value of any specific type in T'Class. Such an object is called *polymorphic*, meaning that it can take on values of different types during its lifetime. SUAVE allows constants, dynamically allocated variables and signals to be of a class-wide type. When an operation is applied to an object of a class-wide type, the tag of the value is used to determine the specific type, and thus to determine which primitive operation to invoke. This is called *dynamic dispatching*, or *late binding*, and is an essential aspect of object-oriented languages. As an example, consider the following signal declaration and application of an operation:

```
signal fetched_instruction : instruction'class;
disassemble ( fetched_instruction );
```

If the value of the signal is of type instruction, the version of disassemble for that type is invoked. However, if the value of the signal is of one of type load_instruction, the overriding version defined for load_instruction values is invoked. The choice is made dynamically at the time of the call.

While there are no primitive operations of a class-wide type, a subprogram may have a parameter of a class-wide type. Such a subprogram is called a *class-wide operation*. For example:

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

Since the parameter is polymorphic, dynamic dispatching may be required for operations on the parameter within the subprogram.

As a final example in this section, consider an instruction register that can jam a TRAP instruction in place of the store instruction. First, two constants are declared for the TRAP instruction and an undefined instruction:

```
constant halt_instruction : instruction
            := instruction'(opcode => op_halt);
constant undef_instruction : instruction
            := instruction'(opcode => op_undef);
```

Next, the entity is declared:

```
entity instruction_reg is
    port ( load_enable : in bit;
           jam_halt : in bit;
           instr_in : in instruction'class;
           instr_out : out instruction'class );
end entity instruction_reg;
```

The ports instr_in and instr_out are signals of a class-wide type and so may take on values of any of the types in the instruction hierarchy. A behavioral architecture body for the register is:

```
architecture behavioral of instruction_reg is
begin
    store : process ( load_enable, jam_halt, instr_in ) is
        type instruction_ptr is access instruction'class;
        variable stored_instruction : instruction_ptr
            := new undef_instruction;
    begin
        if jam_halt = '1' then
            deallocate ( stored_instruction );
            stored_instruction := new halt_instruction;
        elsif load_enable = '1' then
            deallocate ( stored_instruction );
            stored_instruction := new instr_in;
        end if;
        instr_out <= stored_instr.all;
    end process store;
end architecture behavioral;
```

The process implements the register storage using the local variable stored_instruction. Since a variable cannot be of a class-wide type, stored_instruction is defined as an access value, pointing to a dynamically allocated object of type instruction'class. It is initialized to the undefined instruction. When a HALT instruction is to be jammed, a new instruction object initialized to the halt instruction value is allocated. Similarly, when an input instruction is to be stored, a new instruction object of the corresponding specific type is allocated and

initialized to the input instruction. The designated instruction object is assigned as the output of the register.

4. Extensions for Encapsulation

A data type in VHDL is characterized by a set of values, specified by a type definition, and a set of operations. An *abstract data type* (ADT) is one in which the concrete details of the type definition are hidden from the user of the ADT. The user may only use the operations of the ADT to manipulate values. ADTs are important tools for managing complexity in a large design.

VHDL currently includes the *package* feature, which can be used to define an ADT. The concrete type and associated operations are declared in the package declaration, and the implementations of the operations are declared in the package body. While this approach allows the implementation details of the operations to be hidden from the ADT user, it exposes the details of the concrete type. A user may inadvertently (or deliberately) modify values of the concrete type directly, rather than by using the provided operations. This can potentially place the ADT value in an inconsistent state. It also reduces the maintainability of the design.

SUAVE extends the type system and package feature of VHDL to provide secure encapsulation of information in an ADT. It adopts the mechanisms of private types and private parts in packages from Ada-95. This meets one of the design objectives for SUAVE: to improve encapsulation and information hiding.

As a first step, the use of packages is generalized by allowing them to be declared as part of most declarative regions in a model, not just as library units. SUAVE allows a package declaration and body to be declared in an entity declaration, an architecture body, a block statement, a generate statement, a process statement, and a subprogram body. Thus, the concept of a package is changed from that of a "heavy-weight" library-level unit to that of a "light-weight" declarative item. This is important, since packages are used to declare types and operations defining classes, as well as instances of generic packages (see Ashenden *et al* [5]).

4.1 Private Parts and Private Types

The second extension of the package feature is to allow a package declaration to be divided into a *visible part* and a *private part*, as follows:

```
package name is
    . . .      – – visible part
private
    . . .      – – private part
end package name;
```

Items declared in the visible part are exported and may be referred to by users of the package. Items declared in the private part, on the other hand, are not visible outside the package. When using a package to define an ADT, the type is declared as a *private type* in the visible part of the package, along with the specifications of the primitive operations of the type. A private type declaration only provides the name of the type. The concrete details of the type are declared separately in the private part of the package.

As an example, the following package defines an ADT for complex numbers:

```
package complex_numbers is
    type complex is private;
    constant i : complex;
    function cartesian_complex ( re, im : real ) return complex;
    function re ( C : complex ) return real;
    function im ( C : complex ) return real;
    function polar_complex ( r, theta : real ) return complex;
    function "abs" ( C : complex ) return real;
    function arg ( C : complex ) return real;
    function "+" ( L, R : complex ) return complex;
    function "–" ( L, R : complex ) return complex;
    function "*" ( L, R : complex ) return complex;
    function "/" ( L, R : complex ) return complex;
private
    type complex is
        record
            r, theta : real;
        end record complex;
end package complex_numbers;
```

A user of this package can declare objects of type complex and invoke operations on complex numbers, for example:

```
signal x, y, z : complex := cartesian_complex(0.0, 0.0);
. . .
z <= x * y after 20 ns;
```

However, the fact that complex numbers are represented in polar form is hidden. Indeed, the representation may be changed without requiring changes to the user's code.

4.2 Private Extensions

SUAVE adopts the Ada-95 mechanisms for integrating encapsulation with inheritance. A private type can be declared to be tagged, indicating that it can be used as the parent of a derived type. The concrete details remain hidden in the private part of the package. A tagged private type can also be declared abstract if it should not be directly instantiated. For example, a network packet at the media-access level of a protocol suite might be declared as follows:

```
package MAC_level is
    type MAC_packet is abstract tagged private;
    . . .
private
    type MAC_packet is tagged record
            . . .
        end record MAC_packet;
end package MAC_level;
```

A tagged private type can be extended using type derivation, as described in Section 3. However, for the derived type to take on the form of a secure ADT, it should be declared as a *private extension*. This allows the details of the extension to be encapsulated. For example, the network packet type defined above may be extended with payload information to form a network-level packet:

```
package network_level is
    type network_packet is
        new MAC_packet with private;
    . . .
private
    type network_packet is
        new MAC_packet with record
            . . .
        end record network_packet;
end package network_level;
```

A user of this package knows that a network-level packet is derived from a MAC-level packet, and thus inherits all of the operation applic-

able to a MAC-level packet. The concrete details of both types, however, remain hidden.

4.3 Contractual Details

In adopting the Ada-95 features for private types into VHDL, some minor changes were required to take account of interactions with VHDL-specific features. In particular, VHDL prohibits signals from being of a type that includes access values. The reason for the restriction is that signals are the communication medium between processes, which execute concurrently. If processes were to pass access values between one another, the designated variable would be shared and thus liable to uncontrolled concurrent access. Furthermore, in a parallel implementation of a simulator, different processes may execute in different address spaces or on different processors. An access value created in one process may be meaningless in the addressing context of another.

SUAVE requires that a private type whose concrete implementation includes an access value to indicate the fact in the private type declaration with the keywords **access private**. The same requirement applies to a private extension that includes an access value. Such types cannot be used for signals. Indication of the existence of an access type in the concrete type can be viewed as a form of contract between the type provider and users. Absence of the indication is contract that the concrete type does not include access values. In the case of a signal of a class-wide type, there may be a derived type in the class that includes an access value. While this cannot be checked during analysis, it can be determined at elaboration time, since the complete hierarchy covered by the class is known at that time.

Another form of contract that can be specified relates to assignment. If a private type includes the keyword **limited**, assignment is not allowed by the user of the type, and the equality operator is not predefined. This feature is adopted from Ada, and is useful for types denoting linked data structures. Assignment normally involves element-wise copying of values, and equality involves element-wise comparison. For linked structures, deep copy and deep comparison may be more appropriate. The type is declared limited in the visible part of the package, and copy and equality operations are provided. The implementations of the operations have full view of the type, and so can implement the required deep copy and comparison.

As an example of the two forms of contractual detail described in this section, consider the following ADT for a set of test vectors:

```
package test_vector_lists is
    type list is limited access private;
    constant empty_list : list;
    procedure copy ( from : in list;  to : out list );
    impure function "=" ( L, R : list ) return boolean;
    procedure add ( L : inout list;  test : in test_vector );
    . . .
private
    type element_node;
    type element_ptr is access element_node;
    type list is new element_ptr;
end package test_vector_lists;
```

The list type is represented by the private type list, whose concrete representation is a singly-linked list of elements. Since the type includes access values, the keyword **access** is included in the private type declaration. Further, since the intended semantics of list assignment is to copy the elements to the target, the private type is made limited. Hence the package provides a copy operation and an equality operation. The body of the package is outlined as follows.

```
package body test_vector_lists is
    type element_node is record
            next_element : element_ptr;
            element : test_vector;
        end record element_node;
    constant empty_list : list := list ( element_ptr'(null) );
    procedure copy ( from : in list;  to : out list ) is . . .
    . . .
end package body test_vector_lists;
```

This illustrates a further extension to VHDL made by SUAVE: constants and constant parameters may include access values. This improves the expressiveness of the language by allowing constants to be of an ADT whose implementation happens to include access values. It also allows operations of such an ADT to be written functions with constant in-mode parameters of the type and a result of the type.

5. Extension of Generics

VHDL currently allows an entity declaration, a component declaration or a block statement to include a *generic clause*, which defines formal generic constants for the unit. Generic constants are typically used to specify timing and other operational parameters and to specify index bounds for array ports. These uses are illustrated by the following entity declaration for a multiplexer:

```
entity mux is
    generic ( Tpd : time;
              width : positive;
              trace : boolean := false );
    port ( sel : in bit;
           d0, d1 : in bit_vector(0 to width – 1);
           d_out : out bit_vector(0 to width – 1) );
end entity mux;
```

The generic constant Tpd is used to parameterize the multiplexer with respect to propagation-delay; width is used in the index constraints for the data ports; and trace is used to control whether the multiplexer traces values passed to the output. The generic constants are visible in any architecture corresponding to this entity, and can be used in the implementation of the structure or behavior of the design entity.

When a unit with a generic clause is instantiated, a *generic map aspect* is used to associate actual values with the formal generic constants. For example, the entity shown above might be instantiated as follows:

```
data_mux : entity work.mux(behavioral)
    generic map ( Tpd => 1.6 ns, width => 16, trace => true )
    port map ( . . . );
```

The actuals are constants whose values are used in place of the formal generic constants for this instance. Association of actuals with formal generics occurs when the instance is elaborated prior to simulation or synthesis.

5.1 Overview of Extensions of Generics

One of the main aspects that constrains re-use of the multiplexer entity described above is that it can only be instantiated to deal with bit-vector values. A more re-usable multiplexer entity would be instantiable for arbitrary types. Thus, it is desirable to be able to specify the data type as a formal generic. In many cases, this is feasible, since the implementation of a unit does not depend on the details of any particular type. For example, a behavioral implementation of a multiplexer simply involves assigning values from input to output, irrespective of the type of the values. A given multiplexer instance, however, should only be allowed to deal with values of one particular type, namely the type of the signals connected to its data ports. This restriction is in conformance with the strong-typing philosophy of VHDL.

SUAVE extends the generic clause feature of VHDL by allowing specification of formal generic types. A unit may use a formal type to

define ports and other objects in its implementation. When the unit is instantiated, an actual type is associated with the formal type for that instance. The association occurs when the instance is elaborated.

The particular mechanism for specifying formal types is modeled on the corresponding mechanism in Ada-95 [16], but is adapted to integrate cleanly with the existing generic mechanism in VHDL. A formal generic type is specified in the following form in a generic clause:

type *identifier* **is** *interface_type_definition*

For example, the multiplexer entity might be revised as follows to include a formal generic type for the data to be handled. (The propagation delay and tracing generic constants are omitted for clarity of illustration.)

```
entity mux is
    generic ( type data_type is private );
    port ( sel : in bit;  d0, d1 : in data_type;
           d_out : out data_type );
    end entity mux;
```

SUAVE allows a number of different classes of type definition, each restricting the actual type that can be associated when the unit is instantiated, as shown in Table 1. (Further refinements to the first three classes are described in the SUAVE report [6].) The implementation of a unit can make use of the knowledge about the formal type afforded by the definition. For example, it may use arithmetic operations on a formal integer type, or indexing on a formal array type. Section 5.2 includes further examples of use of formal types.

Formal type	**Restrictions on actual type**
private	actual can be any type that allows assignment
new *type_mark*	actual must be derived from the specific type (see Ashenden, 1997, #94)
new *type_mark* **with private**	actual must be derived from the specific tagged type (see Ashenden, 1997, #94)
(<>)	actual must be a discrete type
range <>	actual must be an integer type
units <>	actual must be a physical type
range <>.<>	actual must be a floating-point type
array (*index_type*) **of** *element_type*	actual must be an array type with the specified index and element types
access *subtype*	actual must be an access type with the specified designated type
file of *type_mark*	actual must be a file type with the specified element type

Table 1. Classes of formal generic types in SUAVE.

The actual value to be associated with a formal type is specified as a type name in the generic map aspect. For example, the generic multiplexer described above might be instantiated for integer data types as follows:

```
int_mux : entity work.mux(behavioral)
    generic map ( data_type => integer )
    port map ( . . . );
```

SUAVE further extends VHDL by allowing package declarations and subprogram specifications to include generic clauses, enabling definition of template packages and subprograms that can be re-used with different type bindings. This feature combines with the object-oriented extensions in SUAVE [7] to provide means of defining generic abstract data types in a type-secure way.

A generic package includes a generic clause before the declarations in the package, for example:

```
package float_ops is
    generic ( type float_type is range <>.<> );
    . . .
end package float_ops;
```

A generic package such as this cannot be used directly. Instead, it must be instantiated and actual generics associated with the formal generics, for example:

```
type amplitude is range –10.0 to +10.0;
package amplitude_ops is new float_ops
    generic map ( float_type => amplitude );
```

Note that the instance is a package declared within an enclosing declarative region. SUAVE generalizes the use of packages by allowing them to be declared in inner regions, rather than just as library units. This generalization is also related to the use of packages in the object-oriented extensions, and is discussed further in that context [7].

A generic subprogram includes a generic clause before the parameter list, analogous to the way in which an entity includes the generic clause before the port list, for example:

```
procedure swap
    generic ( type data_type is private )
    ( a, b : inout data_type ) is
    variable temp : data_type;
begin
    temp := a;  a := b;  b := temp;
end procedure swap;
```

A generic subprogram cannot be called directly, but must be instantiated first, for example:

```
procedure swap_times is new swap
    generic map ( data_type => time );
```

This declares a procedure with two parameters of type time. A call to the procedure includes a normal actual parameter list, for example:

```
swap ( old_time, new_time );
```

5.2 Examples Using Generic Types

5.2.1 Generic Multiplexer

An entity declaration for a generic multiplexer is shown in Section 5. A corresponding architecture body is:

```
architecture data_flow of mux is
begin
    with sel select
        d_out <= d0 when '0', d1 when '1';
end architecture data_flow;
```

This illustrates that the implementation is independent of the details of the data type. It simply uses the value of the select input to choose which of the two inputs to assign to the output.

5.2.2 Generic Queue ADT Package

One common use of generic packages in Ada is to define re-usable abstract data types (ADTs) for container structures, such as list, queues and sets. SUAVE enables such ADTs to be defined in VHDL. As an example, the following generic package declaration defines an ADT interface for queues of homogeneous elements:

```
package queues is
    generic ( type element_type is private );
```

```
    type queue is access private;

    impure function new_queue return queue;
    impure function is_empty ( Q : in queue ) return boolean;
    procedure append ( Q : inout queue;
                       E : in element_type );
    procedure extract_head ( Q : inout queue;
                             E : out element_type );
private
    type element_node;
    type element_ptr is access element_node;
    type element_node is record
            next_element : element_ptr;
            value : element_type;
        end record element_node;
    type queue is record
            head, tail : element_ptr;
        end record queue;
end package queues;
```

The type of elements to be included in a queue is represented by the formal type element_type. The queue type is a private type [7], whose concrete implementation is a linked list of nodes, each containing a value of the element type. The details of the concrete implementation are in the private part of the package (between the keywords **private** and **end packge**), and are thus not hidden from a package user. The ADT operations have parameters of the queue and element types. The queue package might be instantiated and used to deal with queues of test vectors, for example, as follows:

```
type test_vector is . . .;
package test_queues is new queues
    generic map ( element_type => test_vector );

variable tests_pending : test_queues.queue
            := test_queues.new_queue;
. . .
test_queues.append ( tests_pending, generated_test );
```

5.2.3 Generic Counter

A counter is a device that increments a value of some discrete type, starting at the smallest value and returning to the smallest value after reaching the largest value. A generic counter that deals with any discrete type can be described as follows. First, the entity declaration is:

```
entity counter is
    generic ( type count_type is (<>) );
    port ( clk : in bit;  data : out count_type );
end entity counter;
```

The notation used for the formal generic type specifies that the actual type must be a discrete type. A behavioral architecture body corresponding to the entity is:

```
architecture behavioral of counter is
begin
    count_behavior : process is
        variable count : count_type := count_type'low;
    begin
        data <= count;
        wait until clk = '1';
        if count = count_type'high then
            count := count_type'succ(count);
        else
            count := count_type'low;
        end if;
    end process count_behavior;
end architecture behavioral;
```

The state of the counter is represented by the variable, whose type is the formal generic type. Since the type must be discrete, the implementation is free to use the 'low attribute to initialize the state. Similarly, the process statement uses the 'succ attribute to increment the count value, and the 'high attribute to determine when the value has reached its maximum. Some examples of instantiating this counter design entity are:

```
subtype short_natural is natural range 0 to 255;
type state_type is ( idle, receiving, processing, replying );
. . .
short_natural_counter : entity work.counter(behavioral)
    generic map ( count_type => short_natural )
    port map ( clk => master_clk, data => short_data );
state_counter : entity work.counter(behavioral)
    generic map ( count_type => state_type)
    port map ( clk => master_clk, data => state_data );
```

5.2.4 Generic Shift Register

A shift register stores and shifts elements of a one-dimensional array. The way in which the shift register operates is independent of the particular index and element types of the array. Hence, a generic shift register can be described as follows. First, the entity declaration is:

```
entity shift_register is
    generic ( type index_type is (<>);
            type element_type is private;
            type vector is
                array ( index_type range <> )
                    of element_type );
    port ( clk : in bit;
            data_in : element_type;  data_out : vector );
end entity shift_register
```

The index type is discrete, and the element type can be any type that allows assignment. The vector type illustrates use of preceding formal types in the generic clause to specify the index and element types of the array. This is a minor change to VHDL adopted from Ada as part of adopting the generic mechanisms. A behavioral architecture body for the shift register is:

```
architecture behavioral of shift_register is
begin
    shift_behavior : process is
        constant data_low : index_type
                    := data_out'low;
        constant data_high : index_type
                    := data_out'high;
        type ascending_vector is
            array ( data_low to data_high )
            of element_type;
        variable stored_data : ascending_vector;
    begin
        data_out <= stored_data;
        wait until clk = '1';
        stored_data(data_low
                        to index_type'pred(data_high))
            := stored_data(index_type'succ(data_low)
                                to data_high);
        stored_data(data_high) := data_in;
    end process shift_behavior;
end architecture behavioral;
```

The state of the shift register is represented by the variable stored_data, whose type is an array of the same size and element type as the formal array type. The behavior in the process statement is ex-

pressed using only the knowledge that the index type is discrete, that the stored data can be indexed, and that the data output port can be assigned the stored array value. An example of instantiation of the shift register is:

```
signal master_clk, carry_in : bit;
signal result : bit_vector(15 downto 8);

bit_vector_shifter : entity work.shift_register(behavioral)
    generic map ( index_type => natural,
                  element_type => bit,
                  vector => bit_vector )
    port map ( clk => master_clk,
               data_in => carry_in,  data_out => result );
```

5.2.5 Mixin Inheritance

In the companion paper [7], we describe the SUAVE features for object-oriented inheritance based on derived tagged types. These features can be combined with formal generic derived types to provide a form of *mixin inheritance* [23]. Languages such as C++ use multiple inheritance for this purpose, but with the SUAVE mechanisms, multiple inheritance is not needed.

To illustrate mixin inheritance, consider description of an instruction set for a RISC CPU. The basic type of instruction, including only an opcode, can be described as:

```
type instruction is tagged record
        opcode : opcode_type;
    end record instruction;
```

This is a tagged record type, and thus can be extended with additional elements when new types are derived from it. The derived types inherit the operations applicable to the parent type and can override inherited operations and define additional ones.

The basic instruction type might be extended to define memory reference instructions that use indexed addressing mode, requiring base and offset register numbers. Rather than replicating the description of register numbers and operations in each kind of memory reference instruction, the description is encapsulated so that it can be re-used for any extension derived from the instruction type. The package declaration encapsulating the description is:

```
package indexed_addressing_mixin is
    generic ( type parent_instruction is
                    new instruction with private );

    type indexed_instruction is
        new parent_instruction with record
            index_base, index_offset : register_number;
        end record indexed_instruction;

    function effective_address
                ( instr : indexed_instruction ) return address;
end package indexed_addressing_mixin;
```

The package has a formal generic type that represents a parent instruction type to be extended. The derived type indexed_instruction extends the parent instruction with base and index register numbers, and has effective_address as an applicable operation. To see how this package might be used, consider descriptions of load and store instruction types, derived from the basic instruction type:

```
type load_instruction is
    abstract new instruction with record
        destination : register_number;
    end record load_instruction;

type store_instruction is
    abstract new instruction with record
        source : register_number;
    end record store_instruction;
```

Indexed versions of each of these instruction types can be derived through instantiations of the indexed_addressing_mixin package:

```
package indexed_loads is
    new indexed_addressing_mixin
        generic map ( parent_instruction => load_instruction );
alias indexed_load_instruction is
        indexed_loads.indexed_instruction;

package indexed_stores is
    new indexed_addressing_mixin
        generic map ( parent_instruction
                        => store_instruction );
alias indexed_store_instruction is
        indexed_stores.indexed_instruction;
```

6. Formal Subprograms and Packages

In the previous sections, formal generic types were described and illustrated. SUAVE also adopts formal generic subprograms and packages from Ada-95. These features significantly aid re-use of generic units. Use of formal generic subprograms is described in this section. Space considerations preclude description of formal generic packages; they are described in the SUAVE report [6].

A formal generic subprogram is defined by including a subprogram specification in a generic clause. When the unit is instantiated, an actual subprogram with the same parameter and result type profile must be supplied. There are two idiomatic uses of this feature. The first arises when a generic unit includes a formal generic type and needs the instantiator to supply an operation on values of that type. The second arises when a unit needs the instantiator to supply an action procedure or a call-back procedure that will be invoked as part of an operation. Both of these uses are illustrated by the following package declaration for an ordered collection ADT, adapted from Ashenden [1]:

```
package ordered_collection_adt is
    generic ( type element_type is private;
              type key_type is private;
              function key_of ( E : element_type )
                            return key_type;
              function "<" ( L, R : key_type )
                            return boolean is <> );
    type ordered_collection is limited access private;
    function new_ordered_collection
                return ordered_collection;
    procedure insert ( c : inout ordered_collection;
                       e : in element_type );
    procedure traverse
        generic ( procedure action
                            ( element : in element_type ) )
        ( c : in ordered_collection );
private
    type ordered_collection_object;
    type ordered_collection_ptr is
        access ordered_collection_object;
    type ordered_collection_object is record
            next_element,
            prev_element : ordered_collection_ptr;
            element : element_type;
        end record tree_record;
    type ordered_collection is new ordered_collection_ptr;
end package ordered_collection_adt;
```

The position of each element in a collection is determined by its key. Since the package has know knowledge of the actual element type, the formal generic functions key_of and "<" are used. An instantiator of this package will supply actual functions that are appropriate for the actual element and key types. The notation "<>" indicates that the default actual function will be whichever conforming "<" function is visible at the point of instantiation. The generic procedure traverse has a formal generic subprogram for the action to be applied to each element in the collection. The package body for this ADT is:

```
package body ordered_collection_adt is

    function new_ordered_collection
                return ordered_collection is
        variable result : ordered_collection_ptr
                    := new ordered_collection_object;
    begin
        result.next_element := result;
        result.prev_element := result;
        return ordered_collection(result);
    end function new_ordered_collection;

    procedure insert ( c : inout ordered_collection;
                        e : in element_type ) is
        variable current_element : ordered_collection _ptr
                    := ordered_collection_ptr(c).next_element;
        variable new_element : ordered_collection_ptr;
    begin
        while current_element /= ordered_collection_ptr(c)
            and key_of(current_element.element)
                    < key_of(e) loop
            current_element := current_element.next_element;
        end loop;
        – – insert new element before current_element
        new_element
            := new ordered_collection_object'(
                        element => e,
                        next_element => current_element,
                        prev_element
                            => current_element.prev_element
                        );
        new_element.next_element.prev_element
            := new_element;
        new_element.prev_element.next_element
            := new_element;
    end procedure insert;

    procedure traverse
        generic ( procedure action
                            ( element : in element_type ) )
        ( c : in ordered_collection ) is
        variable current_element : ordered_collection _ptr
                    := ordered_collection_ptr(c).next_element;
    begin
        while current_element
                    /= ordered_collection_ptr(c) loop
            action ( current_element.element );
            current_element := current_element.next_element;
        end loop;
    end procedure traverse;

end package body ordered_collection_adt;
```

The body of the insert procedure simply calls the formal generic functions to determine the key of an element and the compare keys. Similarly, the body of the traverse procedure calls its formal generic procedure to invoke the action on each element.

An illustration of the use of the ADT is also adapted from Ashenden [1]. Suppose test-bench requires a collection of stimulus vectors ordered by time of application to the device under test. The declarations for the stimulus vectors are:

```
type stimulus_element is record
        application_time : delay_length;
        pattern : std_logic_vector
                    (0 to stimulus_vector_length – 1);
    end record stimulus_element;

function stimulus_key ( stimulus : stimulus_element )
                    return delay_length is
begin
    return stimulus.application_time;
end function stimulus_key;
```

The ordered collection ADT package can be instantiated to deal with simulus vectors:

```
package ordered_stimulus_collection_adt is
    new ordered_collection_adt
        generic map ( element_type => stimulus_element,
                    key_type => delay_length,
                    key_of => stimulus_key,
                    "<" => std.standard."<" );
```

The traverse procedure can be instantiated to apply each stimulus vector to the device under test:

```
use ordered_stimulus_collection_adt.all;
variable dut_stimuli : ordered_collection
            := new_ordered_collection;
signal dut_inputs : std_logic_vector
                    (0 to stimulus_vector_length – 1);
procedure apply_stimulus ( stimulus : stimulus_element ) is
begin
    dut_inputs <= stimulus.pattern;
    wait for 100 ns;
end procedure apply_stimulus;

procedure apply_all_stimuli is new traverse
    generic map ( action => apply_stimulus );

. . .

apply_all_stimuli ( dut_stimuli );
```

7. Conclusion

In this paper we have described the SUAVE extensions to VHDL to improve its support for modeling at all levels of abstraction. We have presented the features that provide object-orientation as a combination of improved abstraction, encapsulation and inheritance mechanisms, and the genericity features that improve support for re-use. Most of the features are drawn from Ada-95 and are adapted to integrate with modeling features that are specific to VHDL. Drawing on Ada is appropriate, since VHDL was originally strongly influenced by Ada. In a sense, SUAVE is an evolution of VHDL that parallels the evolution from Ada-83 to Ada-95.

SUAVE improves modeling support by generalizing and extending existing mechanisms, rather than by adding whole new features. In particular, SUAVE avoids replication of the abstraction & encapsulation mechanisms already provided by the package feature. Adding a separate class feature, as proposed in Objective VHDL [19], for example, replicates many aspects of packages and so complicates a designer's choice of expression of design intent.

Space considerations preclude a more detailed definition of the features added in SUAVE. The interested reader can find a more complete description in the SUAVE report [6]. Work is now in progress to implement the extensions within the framework of the SAVANT project [18].

References

[1] P. J. Ashenden, *The Designer's Guide to VHDL*. San Francisco, CA: Morgan Kaufmann, 1996.

[2] P. J. Ashenden and P. A. Wilsey, *A Comparison of Alternative Extensions for Data Modeling in VHDL*, Dept. Computer Science, University of Adelaide, Technical Report TR-02/97, ftp://ftp.cs.adelaide.edu.au/pub/VHDL/TR-data-modeling.ps, 1997.

[3] P. J. Ashenden and P. A. Wilsey, "Considerations on Object-Oriented Extensions to VHDL," *Proceedings of VHDL International Users Forum Spring 1997 Conference*, Santa Clara, CA, pp. 109–118, 1997.

[4] P. J. Ashenden and P. A. Wilsey, *Principles for Language Extension to VHDL to Support High-Level Modeling*, Dept. Computer Science, University of Adelaide, Technical Report TR-03/97, ftp://ftp.cs.adelaide.edu.au/pub/VHDL/TR-principles.ps, 1997.

[5] P. J. Ashenden, P. A. Wilsey, and D. E. Martin, "Reuse Through Genericity in SUAVE," *Proceedings of VHDL International Users Forum Fall 1997 Conference*, Arlington, VA, pp. 170–177, 1997.

[6] P. J. Ashenden, P. A. Wilsey, and D. E. Martin, *SUAVE: A Proposal for Extensions to VHDL for High-Level Modeling*, Dept. Computer Science, University of Adelaide, Technical Report TR-97-07, ftp://ftp.cs.adelaide.edu.au/pub/VHDL/TR-extensions.pdf, 1997.

[7] P. J. Ashenden, P. A. Wilsey, and D. E. Martin, "SUAVE: Painless Extension for an Object-Oriented VHDL," *Proceedings of VHDL International Users Forum Fall 1997 Conference*, Arlington, VA, pp. 60–67, 1997.

[8] J. Barnes, Ed. *Ada 95 Rationale*, vol. 1247. Berlin, Germany: Springer-Verlag, 1997.

[9] J. Benzakki and B. Djaffri, "Object Oriented Extensions to VHDL: the LaMI Proposal," *Proceedings of Conference on Hardware Description Languages '97*, Toledo, Spain, pp. 334–347, 1997.

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

[11] F. P. Brooks, Jr., *The Mythical Man-Month*, Anniversary ed. Reading, MA: Addison-Wesley, 1995.

[12] D. Cabanis and S. Medhat, "Classification-Orientation for VHDL: A Specification," *Proceedings of VHDL International Users Forum Spring '96 Conference*, Santa Clara, CA, pp. 265–274, 1996.

[13] O. J. Dahl and K. Nygaard, "Simula: An Algol Based Simulation Language," *Communications of the ACM*, vol. 9, no. 9, pp. 671–678, 1966.

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

[15] IEEE, *Standard VHDL Language Reference Manual*. Standard 1076-1993, New York, NY: IEEE, 1993.

[16] ISO/IEC, *Ada 95 Reference Manual*. International Standard ISO/IEC 8652:1995 (E), Berlin, Germany: Springer-Verlag, 1995.

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

[18] MTL Systems Inc., *Standard Analyzer of VHDL Applications for Next-generation Technology (SAVANT)*. MTL Systems, Inc, http://www.mtl.com/projects/savant/, 1996.

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

[20] G. Schumacher and W. Nebel, "Inheritance Concept for Signals in Object-Oriented Extensions to VHDL," *Proceedings of Euro-DAC '95 with Euro-VHDL '95*, Brighton, UK, pp. 428–435, 1995.

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

[22] S. Swamy, A. Molin, and B. Covnot, "OO-VHDL: Object-Oriented Extensions to VHDL," *IEEE Computer*, vol. 28, no. 10, pp. 18–26, 1995.

[23] A. Taivalsaari, "On the Notion of Inheritance," *ACM Computing Surveys*, vol. 28, no. 3, pp. 438–479, 1996.

[24] P. Wegner, "Dimensions of Object-Based Language Design," *ACM SIGPLAN Notices*, vol. 22, no. 12, *Proceedings of OOPSLA '87*, pp. 168–182, 1987.

[25] J. C. Willis, S. A. Bailey, and R. Newschutz, "A Proposal for Minimally Extending VHDL to Achieve Data Encapsulation Late Binding and Multiple Inheritance," *Proceedings of VHDL International Users Forum Fall '94 Conference*, McLean, VA, pp. 5.31–5.38, 1994.