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 : comple
```