SUAVE Proposal for Extensions to VHDL for High-Level Modeling

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

Philip A. Wilsey <phil.wilsey@uc.edu>

Dale E. Martin <dmartin@ececs.uc.edu>

Technical Report TR-97-07

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

Technical Report TR-207/08/97/ECECS

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

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

Abstract

Designers are increasingly using VHDL for high-level modeling. However, their task is hindered by the lack of object-oriented and genericity features in the language. Experience in programming languages shows that these features significantly aid management of complexity in large designs and promote re-use of modules. SUAVE extends VHDL by adapting several object-oriented and genericity features from Ada-95. The extensions improve support for modeling in VHDL from system level down to gate level. This report describes the extensions and illustrates their use with 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 abstraction data types (ADTs) and re-use [4]. 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 [8]. 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 [1].

SUAVE aims to improve support for high-level modeling in VHDL by extending the language with features for object-orientation and genericity. As well as adding specific language features, some existing features are generalized. We have previously argued that extending VHDL in this way has the side-effect of improving its expressiveness at all levels of abstraction [2].

The purpose of this report is to define the language extensions proposed in SUAVE. Most of the features added to VHDL are adapted from features in Ada-95 [7], and are included largely for the same reasons that they are included in Ada-95 [3]. Section 2 of this report 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 further extensions to improve genericity in VHDL. Section 6 describes some minor changes the the existing languages to allow the extended features to integrate cleanly with existing features. The changes to the existing syntax rules for VHDL are summarized in Section 7.

While this report does describe the SUAVE extensions in some detail, it does not present the level of detail that would be required to allow an implementation of the extensions. A complete description in the form of changes the the *VHDL Language Reference Manual* [6] will be developed as part of future work in the SUAVE project.

2 SUAVE Design Objectives

A previous paper [2] 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 [6] were adopted in addition to those listed above. They are [5]:

- upward compatibility
- preserve strong typing
- separate declaration and functionality
- unification of timing semantics
- preserve determinism
- preserve generality
- preserve scope of VHDL (gate to system)
- preserve intermixed abstraction levels
- preserve concurrency
- preserve and improve consistency
- preserve and improve portability
- no application specific packages
- minimize implementation impact
- maximize implementation efficiency

3 Extensions to the Type System

A data type in VHDL is characterized by a set of values and a set of operations. The set of values is specified by a type definition. 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.

SUAVE extends the type system of VHDL to improve facilities for defining ADTs by adopting the object-oriented features of Ada-95. An ADT is defined by declaring a type and its primitive operations in a package. Derived types inherit the primitive operations. If a type is declared as a tagged record, additional record elements can be added on derivation. Class-wide types allow definition of operations that operate on values of any type within a derivation hierarchy. Where the particular type of an object of a class-wide type is not known statically, dynamic dispatching is used to determine the operation to invoke.

3.1 Primitive Operations

An operation on a type is expressed in the form of a subprogram (a procedure or a function) that has one or more parameters of the type or a function result of the type. Such a subprogram can be declared in any part of a model where the type is visible. When a type is declared, a number of predefined operations on the type are implicitly declared. For example, when a numeric type is declared, the arithmetic operations are implicitly declared.

SUAVE defines the notion of *primitive operations* of a type as follows.

- The predefined operations implicitly declared when the type is declared are primitive operations of the type.
- For a derived type (see Section 3.2) the primitive operations of the parent type are inherited as primitive operations of the derived type.
- For a type explicitly declared within a package declaration, any subprograms that are also explicitly declared within the same package declaration and that operate on the type are primitive operations of the type.

Example

The following package defines a type for complex numbers. The predefined operators "=" and "/=" are primitive operations. The explicitly declared operations "&", re, im, "+", "- ", "*" and "/" are also primitive operations on type complex.

```
package complex_numbers is

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

    - - The predefined operators "=" and "/=" are primitive operations.

    function "&" ( L, R : real ) return complex;
    function re ( C : complex ) return real;
    function im ( 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;

    - - The operations "&", re, im, "+", "- ", "*" and "/" are primitive operations of the type complex.
```

end package complex_numbers;

– – –

3.2 Derived Types

A *derived type* is a means of inheriting type information from a parent type to form a new type. The derived type inherits the set of values and the primitive operations of the parent type. In each inherited primitive operation, occurrences of the parent type in the operation's profile are replaced by occurrences of the derived type. The inherited primitive operations may be overridden by defining new primitive operations for the derived type. A derived type is defined using a new form of type definition, expressed by the extended syntax rule:

type_definition ::=
    . . .
  | derived_type_definition

derived_type_definition ::=
  [**abstract**] **new** *parent*_subtype_indication [record_extension_part]

The parent subtype indication specifies the *parent subtype*. The type of the parent subtype is the *parent type*. If the reserved word **abstract** is included, the derived type is abstract (see Section 3.4). A record extension part is only allowed if the parent type is a tagged type (see Section 3.3). The derived type is unconstrained if the parent subtype is unconstrained, otherwise the derived type is constrained with the constraints specified in the parent subtype indication.

Example

type word **is new** bit_vector(0 **to** 31);

The type word inherits the predefined logical, shift, "&", "=" and "/=" operations. It is constrained, since the parent subtype indication is constrained.

– – –

A derived type can also be defined by a private extension declaration in the visible part of a package (See Section 4.3), or a formal derived type definition in a generic interface list (see Section 5.3.1).

3.3 Tagged Types and Type Extension

A record type or a private type may include the reserved word **tagged** in its definition. Such a type is called a *tagged type*. See Section 4.3 for a description of private types. The modified syntax rule for a record type definition is:

record_type_definition ::= [[**abstract**] **tagged**] [**limited**] record_definition

record_definition ::=
    record
      element_declaration
      { element_declaration }
    end record [*record_type*_simple_name]
  | **null record**

If the reserved word **abstract** is included, the record type is abstract (see Section 3.4). If the reserved word **limited** is included, the record type is limited (see Section 4.4). An object of a tagged type includes a run-time tag that identifies the specific type used to create the object. For an object of a class-wide type (see Section 3.5), the tag is used to determine the specific type of the object when dispatching an operation on the object.

Example

Suppose the following type and subprograms are declared 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 );
```

The type instruction represents the root type of a hierarchy of kinds of instructions in a CPU. All instructions have an opcode. Other kinds of instructions will be derived from instruction and will extend it with additional fields. The subprograms are primitive operations of instruction and will be inherited by derived types.

– – –

A tagged type may be extended on derivation either by a *record extension* or a *private extension*. The derived type is also a tagged type. See Section 4.3 for a description of private extensions. A record extension is defined by the syntax rule:

record_extension_part ::= **with** record_definition

A record extension defines additional record elements that are included in the derived type in addition to those of the parent type. The names of the elements in the record extension must be distinct from visible element names in the parent type.

Example

```
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 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.

– – –

3.4 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. A tagged type may be declared as abstract by including the reserved word **abstract** in its defi-

nition. Objects may not be declared to be of an abstract type. An *abstract subprogram* is one that has has not body, because it is intended to be overridden when inherited by a derived type. An abstract subprogram declaration may appear in any declarative part where a subprogram declaration may appear. It is declared using an abstract subprogram declaration:

abstract_subprogram_declaration ::= subprogram_specification **is abstract** ;

Example

The following type declaration defines a kind of instruction that addresses memory using a base register and an offset. It is declared abstract, since it is intended to be the parent type for load and store instruction types. 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.

```
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 ) return address;

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

The definition of the load and store instruction types is 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 );

type store_instruction is new memory_instruction with
    record
        source : reg_number;
    end record store_instruction;

procedure perform_memory_transfer ( instr : store_instruction );
```

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

– – –

3.5 Class-Wide Types

For a tagged type T, there is a *class-wide type* denoted by T'Class that 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 T'Class. An object of type T'Class can have a value of any specific type in T'Class. The tag of a value of T'Class determines the specific type of the value. The only elements of an object of type T'Class that are visible are those that are visible for type T. There are no primitive operations of a class-wide type. However, a subprogram may have a parameter of a class-wide type, in which case the subprogram is called a *class-wide operation*. A class-wide type is considered to be an unconstrained type.

Example

Consider the instruction type shown in previous sections. The class-wide type instruction'Class includes instruction, ALU_instruction, and any other types that may be derived from these. For an object of type instruction'Class, the only element that can be accessed is opcode. A class-wide operation to execute any kind of instruction might be declared as:

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

– – –

3.6 Objects of Tagged Types

A constant, variable or signal may be of a specific tagged type. A constant may be of a class-wide type, but must be initialized with a value of a specific tagged type. A variable must be declared to be of a specific type, and may only be assigned values of that specific type. A signal may be of a class-wide type, and may be assigned values of differing specific types. Thus, a signal may be a polymorphic object. An access type may have a class-wide type as its designated type. An object created by an allocator for an access type is a variable of a specific type, and thus may only be assigned values of that specific type. A file type may not have elements of a specific tagged type or of a class-wide type, since the correspondence between tag values and specific types may vary between models.

Note that a signal may not be of a type that directly or indirectly includes an access type element. In the case of a signal of a class-wide type, it may not be possible to check this during analysis of the unit containing the signal declaration. While the root type of the class might not include an access type element, there may be an access type element in an extension in a descendant type. In general, the hierarchy of classes covered by a class-wide type is globally static. A check can be performed at elaboration-time that, for a signal of a class-wide type, the class does not cover any specific type that includes an access type element.*

Example

The following declares two constants. The first is of the specific type instruction. The second is of the class-wide type instruction'class, constrained by the initialization expression to be a value of the specific type instruction:

```
constant nop_instruction : instruction := instruction'(opcode => op_nop);
constant undef_instruction : instruction'Class := instruction'(opcode => op_undef);
```

The following entity declaration represents an instruction register that has a facility to overwrite the stored instruction with a NOP instruction:

```
entity instruction_reg is
    port ( load_enable : in bit;
           jam_nop : in bit;
           instr_in : in instruction'class;
```

---

* An alternative approach is to allow declaration of a signal of a class-wide type that covers a specific type including an access-type element, provided no value of that specific type is ever assigned to the signal. This would require a run-time check during signal assignment to a signal of class-wide type. Possible implementation approaches are to include a status bit in the tag of a tagged type indicating whether the value contains access values, or to include such a bit in the type descriptor identified by the tag.

```
            instr_out : out instruction'class );
end entity instruction_reg;
```

The ports instr_in and instr_out are signals of a class-wide type. A behavioral architecture body for the register is:

```
architecture behavioral of instruction_reg is
begin
    store : process ( load_enable, jam_nop, instr_in ) is
        type instruction_ptr is access instruction'class;
        variable stored_instruction : instruction_ptr := new undef_instruction;
    begin
        if jam_nop = '1' then
            stored_instruction := new nop_instruction;
        elsif load_enable = '1' then
            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.

– – –

3.7 Dispatching

The primitive operations of a tagged type are called *dispatching operations*. The specific types of the operands and the expected result in a subprogram call determine the *controlling tag*, namely the tag of the type whose operation is called. If the controlling tag is statically determined, the particular subprogram body to be invoked is statically determined. If the controlling tag can only be determined dynamically, the subprogram is dispatched at run-time– the particular operation for the type corresponding to the controlling tag is invoked.

Example

Suppose a model includes the declarations:

```
constant halt_instruction : instruction := instruction'(opcode => op_halt);
signal fetched_instruction : instruction'Class;
```

The first of the following two function calls is statically dispatched, since the specific type of the operand halt_instruction is statically determined to be instruction. The second call, on the other hand, must be dynamically dispatched, since at different times fetched_instruction might have values of different types derived from instruction. The tag of fetched_instruction determines which version of privileged is invoked.

```
privileged ( halt_instruction, user_mode )
privileged ( fetched_instruction, user_mode )
```

– – –

3.8 The 'Tag Attribute

There is a predefined private type named Tag. For a specific subtype S, the predefined attribute S'Tag yields a value of type Tag that represents the tag of the type of S. For a class-wide subtype

S, the predefined attribute S'Tag yields a value that represents the tag of the specific tagged type at the root of the class hierarchy denoted by S. For an object X of a class-wide type, the predefined attribute X'Tag yields a value of type Tag that represents the tag of the specific type of X. For an access object X whose designated type is a class-wide type, the predefined attribute X'Tag yields a value of type tag that represents the tag of the specific type of the designated object.

The predefined relational operators ("=", "/=", ""<, "<=", ">" and ">=") are defined for operands of type Tag. For specific tagged types L and R:

- L'Tag = R'Tag iff L and R are the same type,
- L'Tag /= R'Tag iff L and R are different types,
- L'Tag < R'Tag iff L is derived directly or indirectly from R,
- L'Tag <= R'Tag iff L is derived directly or indirectly from R,
- L'Tag > R'Tag iff R is derived directly or indirectly from L,
- L'Tag >= R'Tag iff R is derived directly or indirectly from L or R is the same type as L.

The relational operators applied to class-wide types and objects of class-wide types similarly determine the relationships between the specific types at the roots of the class hierarchies. Note that The ordering implied by the relational operations is a partial order. Thus, L'Tag < R'Tag yielding false does not imply that L'Tag >= R'Tag yields true. If neither L nor R is derived from the other, both operations yield false.

Example

The relational operators provide a means of performing membership tests. Given the instruction types defined in previous examples, and the signal declaration:

```
signal current_instruction : instruction'class;
```

the following test whether the value of current_instruction is a member of the class hierarchy rooted at memory_instruction:

```
current_instruction'Tag <= memory_instruction'Tag
```

– – –

3.9 Type Conversions

SUAVE extends the notion of a type conversion to include value conversion and view conversions. A value conversion is the same as a type conversion currently in VHDL. It takes an operand of a source type and yields a value of a target type. A view conversion, on the other hand, takes a name of a source type and denotes a name of a target type. The revise syntax rules are:

```
name ::=
        . . .
      | type_conversion

type_conversion ::=
        type_mark ( expression )
      | type_mark ( name )
```

A type conversion whose operand is the name of an object is a view conversion if its target type is tagged, or if it appears as an actual parameter of mode out or inout or as the designated name in an alias declaration; other type conversions are value conversions.

Conversion from a source tagged type to a target tagged type is allowed under the following circumstances:

- Conversion is allowed between two specific tagged types if the target type is an ancestor of the source type.
- Conversion from a specific type to a class-wide type is allowed only if the root of the class-wide type is an ancestor of the source type. Such a conversion need not be explicitly stated.
- Conversion from a class-wide type to a specific type is allowed only if the actual value is of the target type or one of its descendants. A run-time check is required.
- Conversion from a source class-wide type to a target class-wide type is allowed only if the classes have a common ancestor and the source class-wide type is covered by the target class-wide type. A run-time check is required if the source type is not an subclass of the target type.

Type conversion of a value or an object of a tagged type does not change the tag or any elements of the value or object. The converted value or object is treated as being of the target type.

3.10 Aggregates

A value may not be converted from a specific tagged type to a specific descendent type. Instead, an extension aggregate must be used. An extension aggregate is a form of record aggregate that uses a record value of a parent type and adds extension elements for the descendent type. The revised syntax rules for aggregates are:

aggregate ::= record_aggregate | extension_aggregate | array_aggregate

record_aggregate ::= (record_element_association_list)

record_element_association_list ::=
    element_association_list
  | **null record**

extension_aggregate ::= (ancestor_part **with** record_element_association_list)

ancestor_part ::= expression | type_mark

array_aggregate ::= (element_association_list)

element_association_list ::=
  element_association { , element_association }

The syntax rule for a normal record aggregate requires revision to handle the possibility of a null record. The syntax rule for an array aggregate is unchanged. The syntax rule for an extension aggregate allows specification of an expression of an ancestor type, followed by values for elements added to that type to derive the descendent type. Alternatively, if only an ancestor type name is specified, an ancestor record value with default element values is used.

Example

The variable potential_load is of type load_instruction, a descendent of the type instruction. The assignment below uses the value of the constant nop_instruction of type instruction, just containing an opcode element. The aggregate extends this value with elements base and offset required for the type memory_instruction and the element destination required for the type load_instruction.

```
variable potential_load : load_instruction;
. . .
potential_load := load_instruction'(nop_instruction with base => 0, offset => 0, destination => 0);
```

– – –

4 Extensions for Encapsulation

Definition of an ADT requires that the concrete implementation details of the type are hidden from users. The interface visible to users should consist only of the operations for manipulating ADT values. SUAVE extends the features of VHDL packages to improve encapsulation of the implementation of ADTs. It adopts the mechanisms of private types and private parts in packages to provide improved encapsulation.

4.1 Declaration of Packages

In VHDL-93 a package may only be declared at the library level. This prevents use of packages for defining abstract data types that are local to a declarative region. In SUAVE, the rules for declarative parts are modified to allow a package declaration or a package body to occur in:

- **S** an entity declaration
- **S** an architecture body
- **S** a block statement
- **S** a generate statement
- **S** a process statement
- **S** a subprogram body

The revised syntax rules are:

```
entity_declarative_item ::=
        . . .
      | package_declaration
      | package_body_declaration

block_declarative_item ::=
        . . .
      | package_declaration
      | package_body_declaration

process_declarative_item ::=
        . . .
      | package_declaration
      | package_body_declaration
```

subprogram_declarative_item ::=

. . .
| package_declaration
| package_body_declaration

Furthermore, a package declaration may occur within a package declaration and a package body may occur within a package body. The revised syntax rules are:

package_declarative_item ::=

. . .
| package_declaration

package_body_declarative_item ::=

. . .
| package_body_declaration

Since a package declaration and its corresponding package body (if required) form a single declarative region, they must both be declared directly within the same enclosing declarative region. If a package declaration is nested within a package declaration and the inner package requires a body, then the outer package also requires a body and must contain the body of the inner package.

A name declared within a package declaration is visible by selection in the scope of the package name by using the package name as a prefix. A name may be made directly visible in the scope of the package by writing a use clause that names the package as a prefix.

Example

The following architecture body for a FIFO includes a package that defines a queue type. The package declaration defines a number of items, including the type queue and the subprogram empty. Since there is a subprogram specification in the package declaration, a package body is required. The items declared in the package are visible by selection within the architecture body. The use clause makes the type name queue directly visible.

```
architecture behavioral of FIFO is
    package queues is
        . . .
        type queue is . . .
        function empty ( Q : queue ) return boolean;
        . . .
    end package queues;
    package body queues is
        function empty ( Q : queue ) return boolean is . . .
        . . .
    end package body queues;
begin
    FIFO_manager : process is
        use queues.queue;
        variable FIFO_queue : queue;
    begin
        . . .
        if queues.empty ( FIFO_queue ) then . . .
```

```
        . . .
    end process FIFO_manager;
end architecture behavioral;
```

– – –

A package declaration directly or indirectly nested within a subprogram body or a process statement may not contain a signal declaration. This is an extension of the rule that a signal may not be declared within a subprogram body or a process statement.

A variable declaration in a package declaration that is directly or indirectly nested within a subprogram body or a process statement may be an ordinary variable declaration. If the package is nested with a subprogram body, the variable is only accessible to the process that is the parent of the subprogram invocation, so concurrent access is not possible. A similar argument applies if the package is nested within a process statement. A variable declaration in a package declaration that is directly or indirectly nested within any other context must be a shared variable declaration. In such circumstances, the variable may be accessible to more than one process, so concurrent access is possible.

4.2 Visible and Private Parts of Packages

A package declaration may be divided into two parts: a visible part and a private part. The visible part may be used to declare the interface of an ADT so that it is visible to users. The ADT is expressed as a private type and related operations. The private part of the package is then used to declare the concrete implementation details of the private type that should be hidden from the user, but which may not be deferred to the package body. See Section 4.3 for a description of private types and for examples of private parts of packages. The modified syntax rule for a package declaration is:

```
package_declaration ::=
    package identifier is
        [ formal_generic_clause ]
        package_declarative_part
    [ private
        package_private_declarative_part ]
    end [ package ] [ package_simple_name ] ;

package_private_declarative_part ::= { package_private_declarative_item }

package_private_declarative_item ::=
      subprogram_declarative_item
    | type_declaration
    | subtype_declaration
    | constant_declaration
    | variable_declaration
    | file_declaration
    | alias_declaration
    | attribute_declaration
    | attribute_specification
    | disconnection_specification
```

| use_clause
| group_template_declaration
| group_declaration
| package_declaration

See Section 5.1 for a description of a generic clause in a package declaration. The first declarative part in the package declaration is the visible part and the second declarative part is the private part. The visible part and the private part are considered to form a single declarative part for the purpose of rules that relate to items within declarative parts. The names declared within the visible part are visible by selection where the package name is visible. They may be made directly visible by a use clause that names the package. The names declared in the private part are visible within the declarative region of the package and nowhere else.

4.3 Private Types and Private Extensions

A type declared in the visible part of a package declaration may be declared as a *private type* or a *private extension*. These define a *partial view* of the type, in which only some of the details of the type are defined. The *full view* of the type defines all of the details of the type. It must be defined by a full type declaration for the type in the private part of the package. The syntax rules for defining a private type and a private extension are:

type_declaration ::=
. . .
| private_type_declaration
| private_type_extension

private_type_declaration ::=
type identifier **is** [[**abstract**] **tagged**] [**limited**] [**access**] **private** ;

private_extension_declaration ::=
type identifier **is** [**abstract**] **new** *ancestor*_subtype_indication **with** [**access**] **private** ;

If the reserved word **abstract** is included, the type is abstract (see Section 3.4). If the reserved word **tagged** is included, the type is tagged (see Section 3.3) and must be completed in the full view with a tagged type. The full view may be a tagged type even if the partial view is not. If the reserved word **limited** is included, the partial view of the type is limited (see Section 4.4). The full view of the type may also be limited, but usually it is not, since the implementation of operations on the type will need to use assignment. If the reserved word **access** is included, the type may be of an access type or include an element of an access type. The reserved word is required if the full view of the type does include an access type. This allows checking that the type is not used in contexts where inclusion of an access type is prohibited, for example, as the type of a signal.

A private type allows the concrete details of the type to be hidden from users of the type. A private extension allows the fact that a type is derived from a particular ancestor to be known, but hides the details of the extension. The ancestor type specified must be a specific tagged type. In the full view, the type must be derived from the specified ancestor type, although it need not be derived directly. It may be derived via one or more intermediate types.

For a private type or private extension, only the partial view is visible to users of the package. However, within the private part and body of the package, the full view is visible. The full view must be completed in the private part of the package declaration rather than being deferred to the package

body in order to provide the analyzer with enough information to be able to allocate object of the type, determine parameter passing mechanisms, etc.

The full view of a private type must be a constrained type. This ensures that the size of a value of the type is known when an object of the type is to be created. The full view may not be a file type. A file object can not be of a private type or private extension, nor may it have a private type or private extension as the element type. This is required, since a file element may not be of a tagged type, and the full view of a private type may be tagged even if the partial view is not.

Example

An ADT for complex numbers can be defined in a package as follows.

```
package complex_numbers is

    type complex is private;

    constant i : complex;

    function "&" ( L, R : real ) return complex;
    function re ( C : complex ) return real;
    function im ( 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
            re, im : real;
        end record complex;

end package complex_numbers;
```

Since the type complex is not limited, assignment is allowed and equality is predefined. This is appropriate for the Cartesian representation used in the implementation.

– – –

Example

The following package declares an ADT for tokens used in uninterpreted modeling. The private type token is declared tagged so that it can be extended by ADT users. The concrete implementation is as a tagged record type.

```
package tokens is

    type token is tagged private;

    function new_token return token;

    . . .

private

    type token is
        tagged record
            id : natural;
```

```
            creation_time : time;
        end record token;
end package tokens;
```

The token type can use used as the parent of a derived type as shown below. The new_token function for colored tokens creates its result with an extension aggregate based on a value of the parent type.

```
type colored_token is new token with
    record
        color : color_type;
    end record;
function new_token ( new_color : color_type := default_color ) return colored_token is
begin
    return ( token'(new_token) with color => new_color );
end function new_token;
variable next_token : colored_token := new_token;
variable subsequent_token : colored_token := new_token(red);
```

– – –

Example

The following package declares an ADT for traceable tokens that record their recent history of flow around a queuing network. The type is derived from the token type declared in the example above, but the details of the extension are private.

```
package traceable_tokens is
    type traceable_token is new token with private;
    function new_token return traceable_token;
    . . .
private
    type history_queue_type is ...
    type traceable_token is new token with
        record
            history_queue : history_queue_type;
        end record traceable_token;
end package traceable_tokens;
```

– – –

4.4 Limited Types

A limited type is a view of a type for which the assignment operation is not allowed. A type may declared limited by including the reserved word **limited** in a private type declaration, a formal private type definition, or a record type definition. A type is also limited if it is a composite type with a limited element, or a descendent of a limited type. The equality operator is not predefined for a limited type.

Example

The following package declares an ADT for lists of thingies. The concrete implementation is as a linked list of elements. The type is declared limited because deep copy is required, rather than

just copying the pointer to the first element in a list. The private type declaration also includes the reserved word **access** because the concrete implementation uses access types.

```
package thingy_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;  element : thingy );
    . . .
private
    type element_node;
    type element_ptr is access element_node;
    type list is new element_ptr;
end package thingy_lists;

package body thingy_lists is
    use work.thingies.all;
    type element_node is
        record
            next_element : element_ptr;
            element : thingy;
        end record element_node;
    constant empty_list : list := list( element_ptr'(null) );
    procedure copy ( from : in list;  to : out list ) is . . .
    . . .
end package body thingy_lists;
```

– – –

5 Generics

Reuse of a software module can be improved by making it applicable in a wider set of contexts, for example, by making it more generic. VHDL currently includes a mechanism, generic interface constants, that allows components and entities to be parameterized with formal constants. Actual generic constants are specified when components are instantiated and when entities are bound. The generic constant mechanism is widely used to specify timing parameters and array port bounds, amongst other things.

SUAVE extends the generic mechanism of VHDL to improve support for reuse. There are two main aspects to the extension. The first is to allow subprograms and packages to have generic interface clauses. The second is to allow formal type parameters in a generic interface clause, making the generic item reusable for a variety of different types. Formal subprogram and formal package parameters are also allowed as a corollary to allowing formal type parameters.

5.1 Generic units

SUAVE extends the declaration of packages and subprograms to allow inclusion of a formal generic clause. The extended syntax rule for a package declaration is shown in Section 4.2, and is repeated here:

package_declaration ::=
package identifier **is**
[*formal*_generic_clause]
package_declarative_part
[**private**
package_private_declarative_part]
end [**package**] [*package*_simple_name] ;

A package that includes a formal generic clause is a generic package. A generic package is a template for an ordinary package, and does not provide declarations itself. It must be instantiated as described in Section 5.2. Examples of packages with formal generic clauses are shown in Section 5.3.

The extended syntax rule for a subprogram specification is:

subprogram specification ::=
procedure designator
[**generic** (generic_list)] [(formal_parameter_list)]
| [**pure** | **impure**] **function** designator
[**generic** (generic_list)] [(formal_parameter_list)] **return** type_mark

A subprogram declaration or body that includes a formal generic clause in its specification is a generic subprogram. A generic subprogram is a template for an ordinary subprogram. It cannot be called, but must be instantiated as descibed in Section 5.2. Examples of subprograms with formal generic clauses are shown in Section 5.3.

If a generic subprogram is declared as a separate subprogram specification and subprogram body, the subprogram body must include the formal generic clause, which must conform with the formal generic clause in the subprogram specification.

5.2 Instantiating Generic Units

A generic subprogram may be instantiated as a subprogram. The syntax rule is:

generic_subprogram_instantiation ::=
[subprogram_kind] designator **is new** *generic_subprogram*_name
[generic_map_aspect] ;

Similarly, a generic package may be instantiated as a package. The syntax rule is:

generic_package_instantiation ::=
package identifier **is new** *generic_package*_name
[generic_map_aspect] ;

A generic package that directly or indirectly contains a declared signal may not be instantiated within the declarative region of a process statement or a subprogram body. Instantiation of components and entities remains unchanged. Examples of instantiation of generic subprograms and packages are shown in Section 5.3.

5.2.1 Extended Generic Maps

A generic map aspect is used to associate actual generics with formal generics upon instantiation of a generic subprogram, a generic package, a component or an entity. A generic map is also used

to associate actual generics with formal generics in a block statement. The extended syntax rule for an actual designator, allowing specification of actual generics for formal type, subprogram and package generics, is:

actual_designator ::=
. . .
| type_mark
| *subprogram*_name
| *package_instance*_name

For a formal generic type, the associated actual type is designated by a type mark. For a formal generic subprogram, the associated actual subprogram is designated by a subprogram name. For a formal generic package, the associated actual package is designated by the name of a package instance. Examples of extended generic maps are shown in Section 5.3.

5.3 Extended Generic Clauses

SUAVE extends the kinds of formal generics that may be specified in a formal generic clause to include formal types, formal subprograms and formal packages. These can be included in generic clauses of package declarations, subprogram specifications, block statements, entity declarations and component declarations. The revised syntax rule is:

interface_declaration ::=
. . .
| interface_type_declaration
| interface_subprogram_declaration
| interface_package_declaration

Interface type, subprogram and package declarations may only appear in formal generic clauses. A formal generic clause may only include interface constant, type, subprogram and package declarations.

The rule (LRM ¶4.3.2.1) that prohibits use of an item declared in an interface list within the declaration of other items in the interface list is relaxed in the case of generic interface lists. Items declared in a generic interface list may be used in the declaration of items declared subsequently in the interface list.

5.3.1 Formal Generic Types

An interface type declaration defines a formal generic type that can be used to pass a particular type when the generic unit is instantiated. The form of the interface type definition determines the class of type that can be passed as the actual generic type. The syntax rules are:

interface_type_declaration ::=
type identifier **is** interface_type_definition ;

interface_type_definition ::=
interface_private_type_definition
| interface_derived_type_definition
| interface_discrete_type_definition
| interface_integer_type_definition

| interface_physical_type_definition
| interface_floating_type_definition
| interface_array_type_definition
| interface_access_type_definition
| interface_file_type_definition

5.3.2 Formal Private Types

An interface private type definition defines a formal generic type that can denote any type, subject to the restrictions described below. The syntax rule is:

interface_private_type_definition ::= [[**abstract**] **tagged**] [**limited**] [**access**] **private**

Inclusion of the reserved word **tagged** specifies that the formal type denotes a tagged type– the actual type must be tagged. Omission of the reserved word **tagged** specifies that the formal type is not tagged– the actual type may be tagged or untagged. Inclusion of the reserved word **abstract** specifies that the formal type is abstract– the actual type may be abstract or non-abstract. Omission of the reserved word **abstract** specifies that the formal type is non-abstract– the actual type must be non-abstract. Inclusion of the reserved word **limited** specifies that the formal type denotes a limited type– the actual type may be limited or unlimited. Omission of the reserved word **limited** specifies that the formal generic type is unlimited– the actual type must be unlimited. Inclusion of the reserved word **access** specifies that the formal generic type may be of or include an element of an access type– the actual type may or may not be of or include an access type. Omission of the reserved word **access** specifies that the formal generic type does not include an access type– the actual type must not be an access type nor include an element of an access type.

Example

A package defining an ADT for sets of elements can be made reusable by making it generic with respect to element type, as shown below. The formal type generic element_type is unlimited, non-tagged and non-abstract. Hence, any actual type provided on instantiation must be an unlimited, non-abstract type, but may be a tagged type.

```
package sets is
    generic ( type element_type is access private );

    type set is access private;

    constant empty_set : set;

    procedure copy ( from : in set;  to : out set );

    function "+" ( R : element_type ) return set;                    -- singleton set
    impure function "+" ( L : set;  R : element_type ) return set;   -- add to set
    impure function "+" ( L : element_type;  R : set ) return set;   -- add to set

    . . .

private

    type element_node;
    type element_ptr is access element_node;
    type set is new element_ptr;

end package sets;
```

```
package body sets is
    type element_node is
        record
            next_element : element_ptr;
            value : element_type;
        end record element_node;
    . . .
end package body sets;
```

Given a type thingy, an ADT for sets of elements of this type may instantiated as follows:

```
package thingy_sets is new sets generic map ( element_type => thingy );
```

– – –

5.3.3 Formal Derived Types

An interface derived type definition defines a formal generic type that denotes any type in the derivation tree rooted at a specified type. The syntax rule is:

interface_derived_type_definition ::= [**abstract**] **new** type_mark [**with** [**access**] **private**]

The reserved words **with private** must be included if and only if the ancestor type specified is a specific tagged type, in which case the formal generic type denotes a private extension of the ancestor type. If the reserved words **with private** are omitted the ancestor type must not be a tagged type. The reserved word **abstract** may only be included if the ancestor type is a tagged type, and specifies that the formal generic type denotes an abstract type– the actual type may or may not be abstract. Omission of the reserved word **abstract** specifies that the formal type is non-abstract– the actual type must be non-abstract. Inclusion of the reserved word **access** specifies that the private extension may include an element of an access type– the actual type may or may not include an access type. Omission of the reserved word **access** specifies that the private extension does not include an access type– the actual type must not include an element of an access type.

Example

The following package defines a mixin derivation for adding indexed addressing information to a type derived from the instruction type. Use of the reserved words **with private** in the formal type generic specifies that the type instruction is a specific tagged type and that parent_instruction is derived from it. The type indexed_instruction is declared abstract in case the user needs the type resulting from mixing in the indexed addressing properties to be abstract. If the user needs the type to be non-abstract, a non-abstract version can be derived from indexed_instruction.

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

Suppose types load_instruction and store_instruction are derived from type instruction as follows:

```
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 these instructions can be derived through instantiations of the indexed_addressing_mixin package, as follows:

```
package indexed_loads is
    new indexed_addressing_mixin generic map ( parent_instruction => load_instruction );
type indexed_load_instruction is
    new indexed_loads.indexed_instruction with null record;

package indexed_stores is
    new indexed_addressing_mixin generic map ( parent_instruction => store_instruction );
type indexed_store_instruction is
    new indexed_stores.indexed_instruction with null record;
```

– – –

5.3.4 Formal Discrete Types

An interface discrete type definition defines a formal generic type that denotes any discrete type. The syntax rule is:

interface_discrete_type_definition ::= (<>)

Example

The following entity declaration describes a counter that counts through successive values of any discrete type denoted by count_type:

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

An architecture body for the counter is shown below. Since count_type denotes a discrete type, the process can use the attributes low, high and succ.

```
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;
```

Some examples of instantiation of this counter are:

```
type state_type is ( idle, receiving, processing, replying );
. . .
natural_counter : entity work.counter(behavioral)
        generic map ( count_type => natural )
        port map ( clk => master_clk, data => natural_data );
state_counter : entity work.counter(behavioral)
        generic map ( count_type => state_type)
        port map ( clk => master_clk, data => state_data );
```

– – –

5.3.5 Formal Integer Types

An interface integer type definition defines a formal generic type that denotes any integer type. The syntax rule is:

interface_integer_type_definition ::= **range** <>

Example

The counter example can be rewritten to use a formal integer type, allowing use of the addition operator in the implementation.

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

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 + 1;
                else
                        count := count_type low;
                end if;
        end process count_behavior;

end architecture behavioral;
```

– – –

5.3.6 Formal Physical Types

An interface physical type definition defines a formal generic type that denotes any physical type. The syntax rule is:

interface_physical_type_definition ::= **units** <>

Example

The following generic package that defines a physical type that is the dimensional product of two physical types specified as formal physical type generics:

```
package product_measures is
    generic ( type measure1, measure2 is units <> );

    type product_measure is
        units
            product_unit;
            product_unit_E3 = 1E3 product_unit;
            product_unit_E6 = 1E6 product_unit;
            product_unit_E9 = 1E9 product_unit;
            product_unit_E12 = 1E12 product_unit;
        end units product_measure;

    function "*" ( L : measure1;  R : measure2 ) return product_measure;
    function "/" ( L : product_measure;  R : measure1 ) return measure2;
    function "/" ( L : product_measure;  R : measure2 ) return measure1;

end package product_measures;
```

The implementation of the operators is completed in the package body:

```
package body product_measures is

    function "*" ( L : measure1;  R : measure2 ) return product_measure is
    begin
        return product_measure'val( measure1'pos(L) * measure2'pos(R) );
    end function "*";

    function "/" ( L : product_measure;  R : measure1 ) return measure2 is
    begin
        return measure2'val( product_measure'pos(L) / measure1'pos(R) );
    end function "/";

    function "/" ( L : product_measure;  R : measure2 ) return measure1 is . . .

end package body product_measures;
```

An instantiation of the package to define a type for power as the product of voltage and current is shown below. Use of the "*" and "/" functions defined by the package instance performs dimensionally correct arithmetic on voltage, current and power values.

```
type voltage is
    units
        microvolt;
        millivolt = 1000 microvolt;
        volt = 1000 millivolt;
    end units voltage;

type current is
    units
        microamp;
        milliamp = 1000 microamp;
        amp = 1000 milliamp;
    end units current;

package power_measures is
    new product_measure generic map ( measure1 => voltage, measure2 => current );

alias power is power_measures.product_measure;
alias picowatt is power_measures.product_unit;
```

```
alias nanowatt is power_measures.product_unit_E3;
alias microwatt is power_measures.product_unit_E6;
alias milliwatt is power_measures.product_unit_E9;
alias watt is power_measures.product_unit_E12;
```

– – –

5.3.7 Formal Floating Types

An interface floating type definition defines a formal generic type that denotes any floating point type. The syntax rule is:

interface_floating_type_definition ::= **range** <> . <>

See Section 5.3.10 for an example that includes use of a formal type generic.

5.3.8 Formal Array Types

An interface array type definition defines a formal generic type that denotes any array type. The syntax rule is:

interface_array_type_definition ::= array_type_definition

A formal array type and the associated actual array type must both be constrained or both be unconstrained. Both must have the same dimensionality, the same index types in each dimension and the same element types. For a formal constrained array type, the index constraint must be specified in the form of a type mark, and the actual array type must have the same index range as the formal array type.

Example

The following entity declaration describes a shift register that stores and shifts a vector of arbitrary type:

```
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 architecture body 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 entity can be instantiated as follows:

```
    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 );
```

– – –

5.3.9 Formal Access Types

An interface access type definition defines a formal generic type that denotes any access type. The syntax rule is:

interface_access_type_definition ::= access_type_definition

Example

The following generic procedure copies the value of one dynamic vector to another. The index type and element type of the dynamic vectors are specified as formal generic types, and the dynamic vector type is represented as a pointers to an allocated array.

```
    procedure copy_vector
        generic ( type index_type is (<>);  type element_type is private;
                  type vector is array ( index_type range <> ) of element_type;
                  type vector_ptr is access vector )
        ( src : in vector_ptr;  dest : inout vector_ptr ) is
    begin
        if dest /= null then
            deallocate ( dest );
        end if;
        dest := new src.all;
    end procedure copy_vector;
```

Given the following declarations for dynamic vectors of time values:

```
    type time_vector is array ( natural range <> ) of time;
    type time_vector_ptr is access time_vector;

    variable schedule1 : time_vector_ptr := new time_vector'(1 ns, 3 ns, 10 ns);
    variable schedule2 : time_vector_ptr;
```

The procedure may be instantiated and called as follows:

```
    procedure copy_time_vector is
        new copy_vector generic map ( index_type => natural, element_type => time,
                                      vector => time_vector,  vector_ptr => time_vector_ptr );
    . . .

    copy_time_vector ( src => schedule1, dest => schedule2 );
```

– – –

5.3.10 Formal File Types

An interface file type definition defines a formal generic type that denotes any file type. The syntax rule is:

interface_file_type_definition ::= file_type_definition

Example

VHDL does not allow a file to contain elements that are multidimensional arrays. One means of working around this restriction is to use a file of the element type of the multidimensional array, and to read and write array elements in sequence. The following package provides read and write operations using this approach for two-dimensional arrays. The package is generic with respect to the array type, and includes a file type with the same element type as the array type. The package cannot be written with a private type for the element type, since there are restrictions on the kinds of types that can be included as file elements. This example uses a floating-point type as the element type. Similar packages could be written for other kinds of types that are permissible for file elements.

```
package floating_matrix_IO is
    generic ( type row_index_type, col_index_type is (<>);
              type element_type is (<>.<>);
              type matrix is array ( row_index_type, col_index_type ) of element_type;
              type matrix_file is file of element_type );

    procedure read ( file f : matrix_file;  value : out matrix );

    procedure write ( file f : matrix_file;  value : in matrix );

end package floating_matrix_IO;
```

An implementation of the read and write operations is shown in the following package body.

```
package body floating_matrix_IO is

    procedure read ( file f : matrix_file;  value : out matrix ) is
    begin
        for row_index in row_index_type loop
            for col_index in col_index_type loop
                read ( f, value(row, col) );
            end loop;
        end loop;
    end procedure read;

    procedure write ( file f : matrix_file;  value : in matrix ) is
    begin
        for row_index in row_index_type loop
            for col_index in col_index_type loop
                write ( f, value(row, col) );
            end loop;
        end loop;
    end procedure write;

end package body floating_matrix_IO;
```

An example of instantiation and use of this package is:

```
subtype transformation_index is integer range 1 to 3;
type transformation_matrix is array ( transformation_index, transformation_index ) of real;
type real_file is file of real;
```

```
package transformation_matrix_IO is
    new floating_matrix_IO
        generic map ( row_index_type => transformation_index,
                      col_index_type => transformation_index,
                      element_type => real,
                      matrix => transformation_matrix,
                      matrix_file => real_file );

use transformation_matrix_IO.all;

file transformation_file : real_file;
variable next_transformation : transformation_matrix;
. . .

file_open ( transformation_file, "test_transformations.dat", read_mode );
read ( transformation_file, next_transformation );
```

– – –

5.3.11 Formal Subprogram Generics

An interface subprogram declaration defines a formal generic subprogram that can be used to pass a particular subprogram when the generic unit is instantiated. The syntax rule is:

interface_subprogram_declaration ::=
    subprogram_specification [**is** subprogram_default]

subprogram_default ::= name | <>

The subprogram specification may not contain a generic clause. The subprogram default specifies the subprogram to use if no actual generic subprogram is provided on instantiation. If a name is specified as the subprogram default, it must denote a callable subprogram with the same profile as that of the subprogram specification. If a box is specified as the subprogram default, it indicates that the actual generic subprogram should be a subprogram with the same name and profile as those of the subprogram specification that is directly visible at the point of instantiation.

Example

The following package defines an ADT for lookup tables. A table contains elements that are each identified by a key value. The formal function key_of determines the key for a given element. No default function is provided, so the user must supply an actual function on instantiation of the package. The formal function "<" is used to compare key values. The default function is specified using the "<>" notation, so if an appropriate function named "<" is visible at the point of instantiation, no actual need be specified. The generic procedure **traverse** is parameterized by an action procedure. An instance of **traverse** applies the actual action procedure to each element in the table.

```
package lookup_tables is
    generic ( type element_type is access private;
              type key_type is private;
              function key_of ( E : element_type ) return key_type;
              function "<" ( L, R : key_type ) return boolean is <> );

    type lookup_table is limited access private;

    procedure lookup ( table : in lookup_table;  lookup_key : in key_type;
                       element : out element_type;  found : out boolean );
```

```
        procedure search_and_insert ( table : in lookup_table;  element : in element_type;
                                      already_present : out boolean );
        procedure traverse
            generic ( procedure action ( element : in element_type ) )
            ( table : in lookup_table );
    private
        type tree_record;
        type tree_ptr is access tree_record;
        type tree_record is
            record
                left_subtree, right_subtree : tree_ptr;
                element : element_type;
            end record tree_record;
        type lookup_table is new tree_ptr;
    end package lookup_tables;
```

The package body is shown below. The formal functions key_of and "<" are invoked using the formal name.

```
    package body lookup_tables is
        procedure lookup ( table : in lookup_table;  lookup_key : in key_type;
                           element : out element_type;  found : out boolean ) is
            variable current_subtree : tree_ptr := tree_ptr(table);
        begin
            found := false;
            while current_subtree /= null loop
                if lookup_key < key_of( current_subtree.element ) then
                    lookup ( lookup_table(current_subtree.left_subtree),
                             lookup_key, element, found );
                elsif key_of( current_subtree.element ) < lookup_key then
                    lookup ( lookup_table(current_subtree.right_subtree),
                             lookup_key, element, found );
                else
                    found := true;
                    element := current_subtree.element;
                    return;
                end if;
            end loop;
        end procedure lookup;
        procedure search_and_insert ( table : in lookup_table;  element : in element_type;
                                      already_present : out boolean ) is ...
        procedure traverse
            generic ( procedure action ( element : in element_type ) )
            ( table : in lookup_table ) is
            alias tree is tree_ptr(table);
        begin
            if tree = null then
                return;
            end if;
            traverse ( lookup_table(tree.left_subtree) );
            action ( tree.element );
            traverse ( lookup_table(tree.right_subtree) );
        end procedure traverse;
```

```
end package body lookup_tables;
```

Suppose a model requires a lookup table of test patterns that use character strings as keys. Such a table may be instantiated as shown below. Since the predefined function "<" operating on strings is visible at the point of instantiation, it is used as the actual function for the formal function "<".

```
type test_pattern_type is . . .
function test_id_of ( test_pattern : in test_pattern_type ) return string;

package test_pattern_tables is
    new lookup_tables generic map ( element_type => test_pattern_type,
                                    key_type => string,
                                    key_of => test_id_of );
```

The traversal procedure can be used to count the number of elements in the table by instantiating it as follows:

```
variable count : natural := 0;

procedure count_a_test_pattern ( test_pattern : in test_pattern_type ) is
begin
    count := count + 1;
end procedure count_a_test_pattern;

procedure count_test_patterns is
    new test_pattern_tables.traverse generic map ( action => count_a_test_pattern );
```

The instantiated traversal function can be called with a test pattern lookup table as a parameter, as follows:

```
variable patterns_to_apply : test_pattern_tables.lookup_table;
. . .
count_test_patterns ( patterns_to_apply );
```

– – –

5.3.12 Formal Package Generics

An interface package declaration defines a formal generic package that can be used to pass a particular instance of a generic package when the generic unit is instantiated. The syntax rule is:

interface_package_declaration ::=
    package identifier **is new** *generic_package*_name interface_package_actual_part ;

interface_package_actual_part ::=
      generic map (<>)
    | [generic_map_aspect]

The name must denote a generic package. If the interface package actual part is of the form that includes a box, the actual package may be any instance of the named generic package. If the interface package actual part is a generic map aspect, the actual package must be an instance of the named generic package with the same actual generics as those specified in the generic map aspect. If the interface package actual part is empty, the actual package must be an instance of the named generic package with the same actual generics as the defaults for the generic package.

The following example is adapted from the Ada Rationale [3].

Example

Suppose a generic package for complex numbers is defined as follows:

```
package generic_complex_numbers is
    generic ( type float_type is range <>.<> );
    type complex is private;
    function "+" ( L, R : complex ) return complex;
    function "- " ( L, R : complex ) return complex;
    . . .
private
    . . .
end package generic_complex_numbers;
```

The package is generic so that it may be used with different floating point types. A package for generic complex vectors can be defined as shown below. It is generic with respect to the type of complex number used as vector elements. The package could be defined with the complex type as a formal type generic, but then the operators needed to implement the vector functions would also have to be included as formal subprogram generics. A more succinct form is to specify a formal package generic for the package defining the complex number ADT.

```
package generic_complex_vectors is
    generic ( package complex_numbers is
                  new generic_complex_numbers
                      generic map (<>) );
    use complex_numbers.all;
    type complex_vector is array ( natural range <> ) of complex;
    function "+" ( L, R : complex_vector ) return complex_vector;
    function "- " ( L, R : complex_vector ) return complex_vector;
    . . .
end package generic_complex_vectors;
```

As an illustration of how the operations defined in the formal complex number ADT package are used, the package body for the complex vectors package is as follows:

```
package body generic_complex_vectors is
    function "+" ( L, R : complex_vector ) return complex_vector is
        alias L_norm : complex_vector(1 to L'length) is L;
        alias R_norm : complex_vector(1 to R'length) is R;
        variable result : complex_vector(1 to L'length);
    begin
        assert L'length = R'length
            report "Addition of complex vectors of different lengths";
        for index in result'range loop
            result(index) := L_norm(index) + R_norm(index);
        end loop;
        return result;
    end function "+";
    function "- " ( L, R : complex_vector ) return complex_vector is . . .
    . . .
end package body generic_complex_vectors;
```

Suppose now that the complex numbers package is instantiated as follows:

```
type short_float is range - 10.0 to 10.0;
package short_complex_numbers is
    new generic_complex_numbers generic map ( float_type => short_float );
```

```
alias short_complex is short_complex_numbers.complex;
```

The complex vectors package is then instantiated as follows:

```
package short_complex_vectors is
    new generic_complex_vectors generic map ( complex_numbers => short_complex_numbers );

alias short_complex_vector is short_complex_vectors.complex_vector;
```

Suppose now that a generic package for mathematical functions on floating point types is defined as follows:

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

    function sqrt ( x : float_type ) return float_type;
    function log ( x : float_type ) return float_type;
    . . .

end package generic_float_functions;
```

A generic package for mathematical functions on complex numbers can be defined as shown below. The formal package generic complex_numbers is an instance of the generic complex_numbers package defined above. That package is generic with respect to the underlying floating point type used. The complex functions package must use the generic floating point functions package, but instantiated with the same underlying floating point type. This is enforced by the actual part specified for the formal package generic_float_functions.

```
package generic_complex_functions is
    generic ( package complex_numbers is
                    new generic_complex_numbers
                            generic map (<>);
              package float_functions is
                    new generic_float_functions
                            generic map ( float_type => complex_numbers.float_type ) );

    use complex_numbers.all;

    function sqrt ( x : complex ) return complex;
    function log ( x : complex ) return complex;
    . . .

end package generic_complex_functions;
```

This package can be instantiated for the complex number type as follows:

```
package short_float_functions is
    new generic_float_functions generic map ( float_type => short_float );

package short_complex_functions is
    new generic_complex_functions generic map ( complex_numbers => short_complex_numbers,
                                                float_functions => short_float_functions );
```

– – –

6 Other Related Changes

6.1 Access Constants and Constant Parameters

SUAVE removes the VHDL-93 rule prohibiting an object of class constant from being of an access type or a type that includes an access type. This applies to declared constants and to constant inter-

face objects. A constant access-type object is initialized to point to a given allocated object, and cannot be changed to point to any other object. The value of the object designated by the access value can, however, be changed. The removal of the restriction allows a constant to be declared of an abstract data type that is implemented using an access type. It also allows impure functions to have in-mode constant-class parameters of access type. This means an abstract data type that is implemented using an access type can have operations that are functions, and that have the abstract data type as parameter and result types.

Example

An ADT for dynamically created lists of time values is declared below. The function empty has a constant parameter that is of an access type, hence the function is declared to be impure. Currently in VHDL, this operation would have to be declared as a procedure with an in-mode variable parameter.

```
package time_lists is
    type time_list_ptr is access private;
    function new_time_list return time_list_ptr;
    impure function empty ( time_list : time_list_ptr ) return boolean;
    . . .
private
    type time_list_node;
    type time_list_ptr is access time_list_node;
    type time_list_node is
        record
            value : time;
            next : time_list;
        end record time_list_node;
end package time_lists;
```

– – –

Example

The following declarations define a type for an array of dynamic strings. The constant of this type is initialized with an array of pointers to strings forming a multi-line message.

```
type string_ptr is access string;
type string_array is array ( positive range <> ) of string_ptr;
constant help_message : string_array
    := ( new string'( "Enter the port name and driving value." ),
        new string'( "If the port is to be disconnected, enter ""null"" in place of the value." ),
        new string'( "To terminate simulation, enter ""quit""." ) );
```

– – –

7 Summary of Syntax Changes

type_definition ::=
    . . .
| derived_type_definition

derived_type_definition ::=
    [**abstract**] **new** *parent*_subtype_indication [record_extension_part]

record_type_definition ::= [[**abstract**] **tagged**] [**limited**] record_definition

record_definition ::=
      record
          element_declaration
          { element_declaration }
      end record [*record_type*_simple_name]
    | **null record**

record_extension_part ::= **with** record_definition

abstract_subprogram_declaration ::= subprogram_specification **is abstract** ;

name ::=
      . . .
    | type_conversion

type_conversion ::=
      type_mark (expression)
    | type_mark (name)

aggregate ::= record_aggregate | extension_aggregate | array_aggregate

record_aggregate ::= (record_element_association_list)

record_element_association_list ::=
      element_association_list
    | **null record**

extension_aggregate ::= (ancestor_part **with** record_element_association_list)

ancestor_part ::= expression | type_mark

array_aggregate ::= (element_association_list)

element_association_list ::=
    element_association { , element_association }

entity_declarative_item ::=
      . . .
    | package_declaration
    | package_body_declaration

block_declarative_item ::=
      . . .
    | package_declaration
    | package_body_declaration

process_declarative_item ::=
      . . .
    | package_declaration
    | package_body_declaration

subprogram_declarative_item ::=
. . .
| package_declaration
| package_body_declaration

package_declarative_item ::=
. . .
| package_declaration

package_body_declarative_item ::=
. . .
| package_body_declaration

package_declaration ::=
package identifier **is**
[*formal*_generic_clause]
package_declarative_part
[**private**
package_private_declarative_part]
end [**package**] [*package*_simple_name] ;

package_private_declarative_part ::= { package_private_declarative_item }

package_private_declarative_item ::=
subprogram_declarative_item
| type_declaration
| subtype_declaration
| constant_declaration
| variable_declaration
| file_declaration
| alias_declaration
| attribute_declaration
| attribute_specification
| disconnection_specification
| use_clause
| group_template_declaration
| group_declaration
| package_declaration

type_declaration ::=
. . .
| private_type_declaration
| private_type_extension

private_type_declaration ::=
type identifier **is** [[**abstract**] **tagged**] [**limited**] [**access**] **private** ;

private_extension_declaration ::=
type identifier **is** [**abstract**] **new** *ancestor*_subtype_indication **with** [**access**] **private** ;

package_declaration ::=
package identifier **is**

[*formal*_generic_clause]
package_declarative_part
[**private**
package_private_declarative_part]
end [**package**] [*package*_simple_name] ;

subprogram specification ::=
procedure designator
[**generic** (generic_list)] [(formal_parameter_list)]
| [**pure** | **impure**] **function** designator
[**generic** (generic_list)] [(formal_parameter_list)] **return** type_mark

generic_subprogram_instantiation ::=
[subprogram_kind] designator **is new** *generic_subprogram*_name
[generic_map_aspect] ;

generic_package_instantiation ::=
package identifier **is new** *generic_package*_name
[generic_map_aspect] ;

actual_designator ::=
. . .
| type_mark
| *subprogram*_name
| *package_instance*_name

interface_declaration ::=
. . .
| interface_type_declaration
| interface_subprogram_declaration
| interface_package_declaration

interface_type_declaration ::=
type identifier **is** interface_type_definition ;

interface_type_definition ::=
interface_private_type_definition
| interface_derived_type_definition
| interface_discrete_type_definition
| interface_integer_type_definition
| interface_physical_type_definition
| interface_floating_type_definition
| interface_array_type_definition
| interface_access_type_definition
| interface_file_type_definition

interface_private_type_definition ::= [[**abstract**] **tagged**] [**limited**] [**access**] **private**

interface_derived_type_definition ::= [**abstract**] **new** type_mark [**with** [**access**] **private**]

interface_discrete_type_definition ::= (<>)

interface_integer_type_definition ::= **range** <>

interface_physical_type_definition ::= **units** <>

interface_floating_type_definition ::= **range** <> . <>

interface_array_type_definition ::= array_type_definition

interface_access_type_definition ::= access_type_definition

interface_file_type_definition ::= file_type_definition

interface_subprogram_declaration ::=
    subprogram_specification [**is** subprogram_default]

subprogram_default ::= name | <>

interface_package_declaration ::=
    package identifier **is new** *generic_package*_name interface_package_actual_part ;

interface_package_actual_part ::=
      generic map (<>)
    | [generic_map_aspect]

References

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

[2] 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.

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

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

[5] E. Christen, "VHDL'93 Design Guidelines," , 1997.

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

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

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