

# **SUAVE Language Description**

7 July 1999

*Peter J. Ashenden*

petera@cs.adelaide.edu.au

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

*Philip A. Wilsey*

phil.wilsey@uc.edu

*Dale E. Martin*

dmartin@ececs.uc.edu

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

Technical Report 99/04  
Department of Computer Science, The University of Adelaide

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 language features for object-oriented modeling, genericity, and abstract communication and concurrency. SUAVE extends VHDL by adapting several object-oriented and genericity features from Ada-95, and by adding more abstract forms of communication and concurrency than those currently in the language. 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 [10]. The term “object-oriented” is used to refer to a language that additionally includes inheritance.

While VHDL can be used for behavioral 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. Furthermore, its signalling features are still closely bound to electrical implementation; no mechanisms are provided for more abstract forms of communication between modules.

SUAVE aims to improve support for high-level modeling in VHDL by extending the language with features for object-orientation, genericity, abstract communication, and dynamic process creation. 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 [1]. We have also presented a discussion of the issues that must be considered in extending VHDL to provide more abstract forms of concurrency and communication [2].

The purpose of this report is to define the language extensions proposed in SUAVE. Many of the features added to VHDL are adapted from features in Ada-95 [9], 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 extensions to improve genericity in VHDL. Section 6 presents the details of the abstract communication language features in SUAVE. Section 7 presents the details of the extensions for concurrency abstraction. Section 8 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 9.

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* [8] will be developed as part of future work in the SUAVE project.

## 2. SUAVE Design Objectives

A previous paper [1] 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 provide a more abstract form of communication than the existing mechanisms of signals and signal assignment,
- to provide dynamic process creation and termination,
- to preserve capabilities for synthesis and other forms of design analysis,
- to provide abstractions that are not biased towards hardware or software implementations, allowing subsequent partitioning and refinement (hardware/software co-design),
- 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. We are guided by Fred Brooks' notion of "conceptual integrity" [5]. As Brooks notes, "Conceptual integrity does require that a system reflect a single philosophy and that the specification as seen by the user flow from a few minds." In designing the SUAVE extensions, the design principals followed during the restandardization of VHDL that lead to the current language [8] were adopted in addition to those listed above. They are [6]:

- 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; L : inout line );
```

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; L : inout line );
```

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 **ab-**

**struct** in its definition. Objects may not be declared to be of an abstract type. An *abstract subprogram* is one that has no 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:

```
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 integer;
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;  
         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  
    begin
```

```

    if jam_nop = '1' then
        instr_out <= nop_instruction;
    elsif load_enable = '1' then
        instr_out <= instr_in;
    end if;
end process store;
end architecture behavioral;

```

---

### 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 specific type, the predefined attribute `X'Tag` yields a value of type `Tag` that represents the tag of the type of `X`. 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`:

- $L'_{\text{Tag}} = R'_{\text{Tag}}$  iff L and R are the same type,
- $L'_{\text{Tag}} \neq R'_{\text{Tag}}$  iff L and R are different types,
- $L'_{\text{Tag}} < R'_{\text{Tag}}$  iff L is derived directly or indirectly from R,
- $L'_{\text{Tag}} \leq R'_{\text{Tag}}$  iff L is derived directly or indirectly from R or L is the same type as R,
- $L'_{\text{Tag}} > R'_{\text{Tag}}$  iff R is derived directly or indirectly from L,
- $L'_{\text{Tag}} \geq R'_{\text{Tag}}$  iff R is derived directly or indirectly from L or R is the same type as L.

The relational operators applied to tags of 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,  $L'_{\text{Tag}} < R'_{\text{Tag}}$  yielding false does not imply that  $L'_{\text{Tag}} \geq R'_{\text{Tag}}$  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 if the target type is a descendent of the root type of the source class-wide type.
- 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:

- an entity declaration
- an architecture body
- a block statement
- a generate statement
- a process statement
- 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 or a package body, 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_declaration
| 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 dynamically elaborated as part of invocation of the subprogram, and is only accessible to the thread of control of the process that is the parent of the subprogram invocation; hence concurrent access to the variable 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 primitive 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_declaration
  | process_declaration
  | 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
  | generic_package_instantiation
  | generic_subprogram_instantiation

```

See Section 5.1 for a description of a generic clause in a package declaration, and see Section 7.1 for a description of process declarations. 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 objects 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 be 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 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, among 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 types in a generic interface clause, making the generic item reusable for a variety of different types. Formal subprograms and formal packages are also allowed as a corollary to allowing formal types.

Note: a further extension yet to be fully designed is the inclusion of formal processes in generic clauses. This parallels formal subprograms, allowing specification of action processes for instances of generic units.

### 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 described 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 declaration and subprogram body, the subprogram body must include the formal generic clause, which must conform with the formal generic clause in the subprogram declaration.

SUAVE also allows the declaration of processes that can be statically or dynamically instantiated, and process declarations can include a generic clause. See Section 7.

## 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 ] ;
```

Generic subprograms may be instantiated in any declarative part in which subprograms may be declared. Generic packages may be instantiated in any declarative parts in which packages may be declared. The extended syntax rules are:

```
entity_declarative_item ::=
    . . .
    | generic_package_instantiation
    | generic_subprogram_instantiation
block_declarative_item ::=
    . . .
    | generic_package_instantiation
    | generic_subprogram_instantiation
process_declarative_item ::=
    . . .
    | generic_package_instantiation
    | generic_subprogram_instantiation
subprogram_declarative_item ::=
    . . .
    | generic_package_instantiation
    | generic_subprogram_instantiation
package_declarative_item ::=
    . . .
    | generic_package_instantiation
    | generic_subprogram_instantiation
package_body_declarative_item ::=
    . . .
    | generic_package_instantiation
    | generic_subprogram_instantiation
primary_unit ::=
    . . .
    | generic_package_instantiation
```

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 process, 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
```

[Note: interface processes declarations to be added here.]

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 be 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 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 floating-point 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 that is directly visible at the point of instantiation and that has the same name and profile as those of the subprogram specification.

#### 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 Process Generics

Note: this language feature to be designed.

### 5.3.13 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. Channels and Communication in SUAVE

At the system level of design, processes representing active objects must interact to communicate data and to synchronize their operation. The simplest form of interaction is message passing, involving the transfer of data from a sender process to a receiver process. The act of message passing can also be used to synchronize processes. SUAVE extends VHDL with message passing for abstract communication as it is a natural abstraction of communication common to both software and hardware. Other forms of interaction, such as rendezvous and remote procedure call are possible, but are oriented specifically toward software implementation. Fortunately, they are easily expressed in terms of message passing.

There are two ways that message passing abstracts away the details of communication in hardware description languages. First, communication events are not tied to specific times, but rather are simply ordered by relative time of sending. This causality-based ordering is weaker and less constraining than clock-time ordering, and is therefore more appropriate at the early stages of design. Second, communication events may be queued (either by queuing messages or processes), rather than relying on the recipient sensing data at the correct time. This allows multiple communication events to form a stream or a transaction without the need for detailed signalling protocols.

### 6.1 Channels

Abstract communication in SUAVE occurs over *channels*, which are of declared *channel types*. Channels can be declared objects or interface objects. A channel is a first-in-first-out buffer of values called *messages*.

#### 6.1.1 Channel Types

SUAVE extends the classes of types that can be defined to include *channel types*. The revised syntax rule for a type definition is:

```
type_definition ::=
    . . .
    | channel_type_definition
```

The syntax rules for a channel type definition are:

```
channel_type_definition ::=
    unbounded_channel_definition
    | unconstrained_bounded_channel_definition
    | constrained_bounded_channel_definition

unbounded_channel_definition ::=
    channel of subtype_indication
    | null channel

unconstrained_bounded_channel_definition ::=
    channel buffer <> of subtype_indication
    | null channel buffer <>

constrained_bounded_channel_definition ::=
    channel buffer_constraint of subtype_indication
    | null channel buffer_constraint

buffer_constraint ::=
    buffer simple_expression
```

The subtype indication specified in a channel type definition is called the *message type* of the channel type. It denotes the subtype of values that may be passed as messages on a channel of the channel type. The base type of this subtype must not be a file type. If it is an access type or is of a type that contains an element that is an access type, the access type must be an access-to-channel type (see Section 6.3.1). The forms of channel type definition that include the reserved word **null** instead of a subtype indication define *null channel types*. Such a channel type is used for a channel on which the messages have no data content.

A channel type may be *unbounded* or *bounded*. A channel of an unbounded channel type has an indefinitely large message buffer. Any buffer capacity limitations are implementation dependent. A channel of a bounded channel type has a message buffer whose size it determined by its channel type. A constrained bounded channel type specifies the buffer size using a buffer constraint. An unconstrained buffered channel type leaves the buffer unspecified. Declaration and definition of constrained and unconstrained channel types is analogous to declaration and definition of constrained and unconstrained arrays. In particular, a constrained channel type is a subtype of an unconstrained channel type with the same message type.

A channel type may only be used to define a channel object, an interface channel, or an access-to-channel type. It may not be used to define any other class of object or type.

The existing VHDL syntax rule for constraints is modified to include buffer constraints:

```
constraint ::=
    range_constraint
  | index_constraint
  | buffer_constraint
```

A buffer constraint may only be used in a subtype indication denoting a subtype of an unconstrained bounded channel type or in a constrained bounded channel definition. The simple expression in a buffer constraint must be a non-negative integer; it determines the buffer size for channels of the channel type or subtype.

The predefined attribute 'Length may be used to determine the buffer size for a constrained bounded channel type or an object of a constrained bounded channel type.

### **Example**

The following declaration defines an unbounded channel type:

```
type acknowledgment_channel is null channel;
```

The following declarations define two constrained bounded channel types:

```
type blocking_request_channel is channel buffer 4 of request_message;
type bigger_request_channel is
    channel buffer 2 * blocking_request_channel'length of request_message;
```

The following declarations define an unconstrained bounded channel type and a constrained bounded channel subtype:

```
type result_channel is channel buffer <> of request_message;
subtype blocking_result_channel is result_channel buffer 2;
```

## **6.1.2 Channel Declarations**

SUAVE extends VHDL to include *channel* objects for abstract message-passing communication. The syntax rule for object declarations is extended to include channel declarations:

```

object_declaration ::=
    . . .
    | channel_declaration

```

One or more channels may be declared using a channel declaration. The syntax rule is:

```

channel_declaration ::=
    channel identifier_list : subtype_indication ;

```

A channel declaration may only appear in a block declarative part, an entity declarative part, or a package declarative part. The subtype indication must denote an unbounded channel type or a constrained bounded channel type.

A channel is analogous to a signal, except that information is transferred using the *send* and *receive* message passing operations (described in Sections 6.2.1 and 6.2.2). There is no notion of resolution of multiple source values, nor of specific times at which values occur on channels. When the channel object is created, the message buffer is initially empty.

### **Example**

The following declarations define two channel objects:

```

channel acknowledgment : acknowledgment_channel;
channel request : blocking_request_channel;
channel result_1 : result_channel buffer 1;
channel result_2 : blocking_result_channel;

```

---

### **6.1.3 Interface Channels**

Interface declarations are extended to include *interface channels*. Interface channels may appear as ports of design entities, components, blocks and processes, and as channel parameters of subprograms. The extended syntax rules for interface declarations are:

```

interface_declaration ::=
    . . .
    | interface_channel_declaration

interface_channel_declaration ::=
    channel identifier_list : [ mode ] subtype_indication

```

The mode, if present, must be one of **in** or **out**. An **in** mode channel may be used to receive messages, and an **out** mode channel may be used to send messages. The subtype indication must denote a channel type. A composite interface channel must be associated in whole. An interface channel of mode **in** must be associated with an actual channel object or be unassociated; it may not be associated with an expression.

A formal interface channel can be declared of any of the three classes of channel types: unbounded, unconstrained bounded or constrained bounded. Where the formal channel is of an unbounded channel type, the associated actual channel must also be of an unbounded channel type. This ensures that, if inferences are made about process behaviour based on the premise that sending to the formal channel does not block, those inferences remain valid independent of characteristics of the actual channel. Where a formal channel is of a constrained bounded channel type or subtype, the associated actual channel must be of the same constrained bounded channel type or subtype, and have the same buffer size as the formal channel. Where a formal channel is of an unconstrained bounded channel type, the actual channel must be of a constrained subtype of the formal's type, and the buffer size of the formal channel is inferred from the buffer size of the actual channel.

Where an interface channel appears as a channel parameter of a subprogram, the actual channel object is passed by reference. Send and receive operations on the formal channel object are performed on the actual channel object.

### **Example**

In the following architecture body, the `image_channel` type represents tokens in an uninterpreted queuing model. The component `image_filter` has unbounded channel ports for receiving and sending tokens. The component instance `filter` has its ports associated with the actual unbounded channel objects `raw_image` and `filtered_image`.

```

architecture performance_modeling of motion_detector is
  type image_channel is channel of image_token;
  component image_filter is
    port ( channel raw_image : in image_channel;
          channel filtered_image : out image_channel );
  end component image_filter;
  channel raw_image, filtered_image : image_channel;
  ...
begin
  filter : component image_filter
    port map ( raw_image => raw_image,
               filtered_image => filtered_image );
  ...
end architecture performance_modeling;

```

---

### **Example**

The following procedure declaration has two channel parameters:

```

procedure process_request ( channel request : in request_channel;
                           channel result : out result_channel );

```

---

### **Example**

In the following model fragment, `pipe_link` is an unconstrained bounded channel type. The process `pipe_stage` has a generic constant `size` that is used to specify the buffer sizes for the formal input and output channel ports. Use of the generic constant in this way ensures that the channels have the same buffer size. The channel objects `link1` and `link2` are declared to be of an anonymous subtype of `pipe_link`, with buffer size `link_buffer_size`. In the process instance `stage1`, the generic constant is given the value `link_buffer_size`, and so the two channel ports assume that value for their buffer sizes. Hence the association with the actual channel objects is legal, since they have the same buffer sizes. This scheme is analogous to the way in which generics and signal ports of unconstrained array types are often used in standard VHDL.

```

type pipe_link is channel buffer <> of link_data;
process pipe_stage is
  generic ( size : natural );

```

```

    port ( channel link_in : in pipe_link buffer size;
          channel link_out : out pipe_link buffer size );
end process pipe_stage;
channel link1, link2 : pipe_link buffer link_buffer_size;
...
stage1 : process pipe_stage
    generic map ( size => link_buffer_size )
    port map ( link_in => link1, link_out => link2 );

```

---

## 6.2 Communication Statements

SUAVE extends the set of sequential statements to include *send statements*, *receive statements* and *select statements*. The extended syntax rule is:

```

sequential_statement ::=
    . . .
    | send_statement
    | receive_statement
    | select_statement

```

### 6.2.1 Send Statement

A message is added to the queue of a channel using a *send statement*. The syntax rule is:

```

send_statement ::=
    [ label : ] send [ expression ] to channel_name ;

```

The expression is disallowed if the channel is of a null channel type. In that case, a data-less message is sent. Otherwise, the expression is required and denotes the value to be sent as a message. The base type of the expression must be the same as the base type of the message type of the channel denoted by the channel name.

Execution of a send statement involves examining the message buffer of the named channel. If the message buffer of the channel is full, the sending process blocks. The channel is full if there is a receiving process associated with the channel for which the number of messages sent on the channel less the number of messages received by that process is equal to the buffer size of the channel. When the channel is no longer full, the sending process resumes and adds the message to the tail of the message buffer of the named channel. The sending process then continues executing. If multiple processes execute send statements to the same channel concurrently, the order in which the messages are added to the message queue is not defined.

An unbounded channel is never full. However, an implementation may run out of storage for the channel buffer, and thus not be able to complete execution of the model. In the case where the buffer size is zero, synchronous communication results, with blocking semantics similar to those of CSP [7].

### Example

The following two statements send to a channel with data and to a null channel respectively:

```

send result_message'( . . . ) to result;

```

---

**send** to acknowledgment;

---

### 6.2.2 *Receive Statement*

A process accepts a message from a channel using a *receive statement*. The syntax rule is:

```
receive_statement ::=
    [ label : ] receive [ target ] from channel_name ;
```

The target is disallowed if the channel is of a null channel type, otherwise it is required. The target must denote a variable name or an aggregate of variable names. Execution of a receive statement involves examining the message buffer of the named channel. If the message buffer is empty, the process suspends until a message arrives. When there is a message available, it is removed from the buffer. If the channel is not of a null channel type, the value of the message is assigned to the target using the same rules as variable assignment.

If multiple processes can read a message channel, all processes receive each message sent to the channel. Furthermore, all processes receive the messages from the channel in the same order. An implementation may achieve this effect either by providing one message queue for the channel, from which each process copies message values, or by replicating the message queue at each process.

#### ***Example***

The following two statements receive from a channel with data and from a null channel respectively:

```
receive next_request from request;
receive from acknowledgment;
```

---

### 6.2.3 *Select Statement*

A process may choose between a number of channels for communication using a *select statement*. The syntax rules are:

```
select_statement ::=
    [ select_label : ]
    select
        [ guard ] select_alternative
    { or
        [ guard ] select_alternative }
    [ or
        timeout_alternative ]
    [ else
        sequence_of_statements ]
    end select [ select_label ] ;
```

```
guard ::= when condition =>
```

```
select_alternative ::=
    receive_alternative
    | send_alternative
```

```

receive_alternative ::=
    receive_statement [ sequence_of_statements ]

send_alternative ::=
    send_statement [ sequence_of_statements ]

timeout_alternative ::=
    timeout_guard sequence_of_statements

timeout_guard ::= after time_expression =>

```

A select statement allows non-deterministic choice between alternative targets for message sending or sources for message reception. It also allows for time-out in the case of communication not being possible within a specified interval. Each select alternative may be guarded by a boolean condition; a guarded alternative may only be chosen if the guard is true.

Execution of the select statement consists firstly of evaluating the guard conditions. An alternative is said to be *open* if it has no guard, or if its guard evaluates to true. If no alternative is open and the select statement has an **else** clause, the statements in the **else** clause are executed, thus completing execution of the select statement. It is an error if no alternative is open and there is no **else** clause.

An open receive alternative can be executed if the channel named in the receive statements has queued messages, or in the case of a channel with a buffer size of zero, if there is a sender waiting to send to the channel. An open send alternative can be executed if the message buffer of the channel named in the send statement is not full, or, in the case of a channel with a buffer size of zero, if all of the receivers of the channel are waiting to receive from the channel.

If there are open alternatives that can be executed, one of the open alternatives is chosen arbitrarily. The send or receive statement is executed, followed by execution of the sequence of statements (if present), completing execution of the select statement.

If there are open alternatives but none of them can be executed, the process executing the select statement blocks until one or more of the open alternatives can be executed. Execution then proceeds as described in the previous paragraph. The guard conditions are not re-evaluated while the process is blocked.

The timeout alternative, if present, specifies the maximum amount of time for which the process will remain blocked waiting for an open select alternative to be executed. If the process remains blocked for the specified time after commencing execution of the select statement, the process resumes and executes the sequence of statements in the timeout alternative, thus completing execution of the select statement. If the timeout clause has a timeout interval of 0 fs, the process will resume on the next simulation cycle. If the timeout alternative is omitted, the process may block indefinitely.

### **Example**

In the following example, the process `access_controller` arbitrates between readers and writers of a shared resource. A reader sends a read-request message to the process, and only proceeds when the process responds with an acknowledgment. When the reader finishes reading, it sends a read-finished message to the process. Writers obey a similar protocol. Multiple readers are allowed concurrent access, provided the number of active writers is zero. Only one writer at a time is permitted, and then only if there are no active readers. The guards in the select statement control the reception of request messages, based on the number of readers or writers currently active.

```

type read_request_channel is channel of . . . ;
type read_finished_channel is null channel;

```

```

type write_request_channel is channel of . . . ;
type write_finished_channel is null channel;
. . .

channel read_request : read_request_channel;
channel read_finished : read_finished_channel;
channel write_request : write_request_channel;
channel write_finished : write_finished_channel;
. . .

access_controller : process is
  variable number_of_readers, number_of_writers : natural := 0;
begin
  select
    when number_of_writers = 0 =>
      receive read_request_info from read_request;
      number_of_readers := number_of_readers + 1;
      . . . - - acknowledge read request
    or
      receive from read_finished;
      number_of_readers := number_of_readers - 1;
    or
      when number_of_readers = 0 and number_of_writers = 0 =>
        receive write_request_info from write_request;
        number_of_writers := number_of_writers + 1;
        . . . - - acknowledge write request
    or
      receive from write_finished;
      number_of_writers := number_of_writers - 1;
  end select;
end process access_controller;

```

---

### **Example**

The following example illustrates inclusion of blocking send and receive alternatives in a select statement. The process models a network interface that accepts packets from a source and forwards them in a stream over a network. Flow control is modeled by the finite buffer size of the channel type (`packet_stream_channel`) representing the network stream. The network is assumed to be unreliable, so the receiver (not shown), periodically sends acknowledgment messages that include the sequence number of the last correctly received packet. The network interface process saves packets until they have been acknowledged. If an acknowledgment message indicates incorrect reception of a packet, the network interface process resets the packet save buffer back to the last correctly received packet and retries sending from the incorrectly received packet.

```

type packet_source_channel is channel buffer 0 of packet_type;
type packet_stream_channel is channel buffer window_size of packet_type;
type ack_stream_channel is channel buffer 1 of ack_type;

process network_interface is
  port ( channel packet_source : in packet_source_channel;
        channel packet_stream : out packet_stream_channel;
        channel ack_stream : in ack_stream_channel );
  . . .

begin
  select

```

```

    not full(save_buffer) =>
        receive incoming from packet_source;
        insert(incoming, save_buffer);
    or
        send next_outgoing(save_buffer) to packet_stream
        advance(save_buffer);
    or
        receive (ok, last_seq_no) from ack_stream
        if ok then
            reclaim(last_seq_no, save_buffer);
        else
            reset(last_seq_no, save_buffer);
        end if;
    end select;
end process network_interface;

```

---

### **Example**

The following statements show how a real-time process might request information from a server. If the server does not respond before the process's deadline, the process proceeds without the response.

```

send request_details to server_request;
select
    receive response_info from server_response;
    act_on(response_info);
or after 10 ms =>
    act_on(default_info);
end select;

```

---

### **Example**

The following example illustrates the use of a timeout alternative in a select statement to choose an alternative action when an output channel is full. The model describes a lossy message source for a network system. The system (not shown here) has an input channel that can accept messages at a given maximum rate. The channel has a bounded buffer to absorb bursts of messages that arrive at a greater rate. However, if the buffer capacity is exceeded, the message source discards messages.

```

type bounded_channel is channel buffer max_size of message_type;
process message_source is
    port ( channel message_stream : out bounded_channel );
    variable next_message : message_type;
begin
    ... -- construct next message in stream
    select
        send next_message to message_stream;
        ... -- log successful send
    or after 0 fs =>
        ... -- log loss of message from stream
    end select;
end process message_source;

```

---

### 6.3 Dynamically Created Channels

SUAVE provides mechanisms for dynamically creating channels in order to communicate with dynamically created processes.

#### 6.3.1 Access-to-Channel Types

An access type may be declared to have a channel type as its designated type. Such an access type is called an *access-to-channel* type.

##### **Example**

The following declarations respectively define a type to be used for message values, a channel type, an access-to-channel type, and a record type containing an element of the access-to-channel type:

```

type result_value is . . . ;
type result_channel is channel result_value;
type result_ref is access result_channel;

type request_info is record
  . . . ; -- info for the transaction
  result_please : result_ref;
end record request_info;

```

---

#### 6.3.2 Dynamic Allocation and Deallocation of Channels

A channel may be dynamically allocated using an allocator with a subtype indication denoting a channel type. The access value returned by the allocator designates the newly allocated channel.

The procedure **deallocate** is implicitly declared for access-to-channel types, just as it is for other access types. Deallocating a channel designated by an access-to-channel value causes loss of any messages in the message queue of the channel. Subsequent use of the access-to-channel value is erroneous, as are subsequent send and receive operations using formal channels directly or indirectly associated with the deallocated channel.

##### **Example**

In the following model fragment, the variable declarations defines a variable of an access-to-channel type initialized with a reference to a dynamically created channel. The send statement send a message containing a reference to the dynamically created channel. The receive statement receives a message from the dynamically created channel. The call to the **deallocate** procedure deallocates the dynamically created channel.

```

variable result : result_ref := new result_channel;
. . .
send ( . . . , result ) to request;
receive . . . from result.all;
. . .
deallocate ( result );

```

---

## 7. Extensions for Abstraction of Concurrency

A system-level design language needs to allow expression of concurrent processes representing the active objects in a system. In some systems, the number of active objects is not statically determined, but may vary during operation of the system. For example, in a client/server system, new service agents may be created as requests arrive from clients, allowing multiple requests to be processed concurrently. In order to describe such systems, a system-level design language must allow expression of process types that may be dynamically instantiated and terminated.

The model of concurrency in VHDL is based on processes which are statically specified in architecture bodies. However, the language does not allow specification of a process type that can be separately instantiated. Instead, the process must be encapsulated in a design entity and instantiated through the component instantiation mechanism. This is cumbersome, and has the disadvantage of implying structural partitioning. Furthermore, it does not allow dynamic instantiation of processes.

These deficiencies can be overcome by extending VHDL to include process declarations, abstracting over the statically specified processes currently provided in the language. A process interacts with its environment using the communication mechanism provided by the language. Therefore, a process declaration includes an interface in which formal communication objects can be specified. A process declaration can be statically instantiated as a concurrent statement in an architecture body, with bindings made between formal and actual communication objects. It can also be dynamically instantiated by the execution of a sequential process instantiation statement. Process declarations and their instantiation and termination are described more fully below.

### 7.1 Process Declarations

SUAVE extends declarative parts to include process declarations and process bodies as follows:

```

process_declaration ::=
    process_specification
    end process [ process_simple_name ] ;

process_body ::=
    process_specification
    process_declarative_part
    begin
    process_statement_part
    end process [ process_simple_name ] ;

process_specification ::=
    process identifier is
    [ generic_clause ]
    [ port_clause ]

```

Process declarations, like subprogram declarations, may be defined with separate specifications and bodies. Process declarations and process bodies may be included in the declarative parts of entity declarations, architecture bodies, block statements, generate statements, process statements, process bodies, and subprogram bodies. The extended syntax rules are:

```

entity_declarative_item ::=
    . . .
    | process_declaration
    | process_body

```

```

block_declarative_item ::=
    . . .
    | process_declaration
    | process_body
process_declarative_item ::=
    . . .
    | process_declaration
    | process_body
subprogram_declarative_item ::=
    . . .
    | process_declaration
    | process_body

```

Process declarations may also be included in package declarations and package bodies, and process bodies may be included in package bodies. The extended syntax rules are:

```

package_declarative_item ::=
    . . .
    | process_declaration
package_body_declarative_item ::=
    . . .
    | process_declaration
    | process_body

```

If a package declaration contains a process declaration, the package must have a package body that contains a process body corresponding to the process declaration. If a process has separate declaration and body, the generic clause and port clause in the body must conform with the generic clause and port clause in the declaration.

## 7.2 Concurrent Process Instantiation Statement

Static instantiation of declared processes is done using a *process instantiation statement*. The syntax rules are:

```

concurrent_statement ::=
    . . .
    | process_instantiation_statement
process_instantiation_statement ::=
    [ instantiation_label : ]
    process process_name
    [ generic_map_aspect ]
    [ port_map_aspect ] ;

```

A process instantiation statement is equivalent to a block statement with the generic clause and port clause taken from the process declaration and the generic map aspect and port map aspect taken from the process instantiation statement. The declarative part of the block statement is empty, and the statement part contains a process whose declarative part and statement part are taken from the process body. The meaning of any identifier within the block statement and the process it contains is that associated with the identifier in the process declaration or body. To illustrate application of these rules, consider the following process body and instantiation statement:

```

process p is
    generic ( g : integer );
    port ( channel c : c_chan );

```

```

    variable v : integer;
begin
    v := x;
end process p;
...
p_inst : process p
    generic map ( g => 5 )
    port map ( c => c1 );

```

The process instantiation statement is semantically equivalent to:

```

p_inst : block is
    generic ( g : integer );
    generic map ( g => 5 );
    port ( channel c : c_chan );
    port map ( c => c1 );
begin
    p : process is
        variable v : integer;
    begin
        v := ... . x;
    end process p;
end block p_inst;

```

The name *x* is prefixed to ensure that it refers to the same item visible in the process declaration rather than any homograph that hides the name.

### 7.3 Sequential Process Instantiation Statement

Dynamic instantiation of a process is performed using a *sequential process instantiation statement*. The syntax rules are:

```

sequential_statement ::=
    ...
    | sequential_process_instantiation_statement
sequential_process_instantiation_statement ::=
    [ label : ]
    process process_name
        [ generic_map_aspect ]
        [ port_map_aspect ];

```

Execution of a sequential process instantiation statement involves the following steps:

- elaboration of the generic list of the process declaration to create the formal generics, and association of the actual generics with the formal generics;
- elaboration of the port list of the process declaration to create the formal ports, and association of the actual signals, channels and values with the formal ports;
- elaboration of the declarations of the process; and
- creation and initialization of the drivers of the process.

The newly instantiated process then commences execution of its statement part concurrently with the instantiating process in the current simulation cycle. The newly instantiated process is said to *depend* on the instance or activation of the declaration or statement that immediately contains the declaration of the process. That instance or activation may not re-

turn or terminate until all of the processes that depend on it have terminated, since such processes may refer to items declared by the declaration or statement.

### 7.3.1 *Dynamic Association with Signal Ports*

The semantics of dynamically associating a signal with a signal port of a process depends on the mode of the port. For an unassociated signal port of mode **in** with a default value expression, the driving and effective values of the port are set to the value of the expression. Similarly, for a signal port of mode **in** associated with an expression, the driving and effective values of the port are set to the value of the expression. For a signal port of mode **in** associated with a signal, the port becomes part of the same net as the signal, and the effective value of the port is determined by the signal update algorithm.

For a signal port of mode **out** or **buffer** associated with a signal, the driver in the process is a source of the port. The port becomes a new source for the associated signal. This is analogous to reconnection of a driver that had previously become disconnected through a null signal assignment.

For a signal port of mode **inout**, the semantics are a combination of the semantics for ports of mode **in** and mode **out**.

### 7.3.2 *Dynamic Association with Channel Ports*

The semantics of dynamically associating a channel with a channel port of a process also depends on the mode of the port. For an unassociated channel port of mode **in**, the port denotes an empty message buffer. For a channel port of mode **in** associated with a channel, the port denotes the message buffer of the channel. Messages sent to the channel prior to association of the port with the channel are not received by the process. Messages sent to the channel after the association is made are received by the process. If a message is sent to the channel in the same simulation cycle that the association is made, it is not defined whether the process receives the message. However, if the process receives one message from a given sender, it receives any message subsequently sent by that sender.

For an unassociated channel port of mode **out**, the port denotes a message buffer to which the process may send messages. However, no process will ever receive the messages. For a channel port of mode **out** associated with a channel, the port denotes the message buffer of the channel. Messages sent by the process to the port are added to the message buffer of the channel.

## 7.4 *Process Termination*

A process may terminate by executing a sequential statement called a *terminate statement*. The syntax rules are:

```
sequential_statement ::=
    . . .
    | terminate_statement
terminate_statement ::=
    [ label : ] terminate ;
```

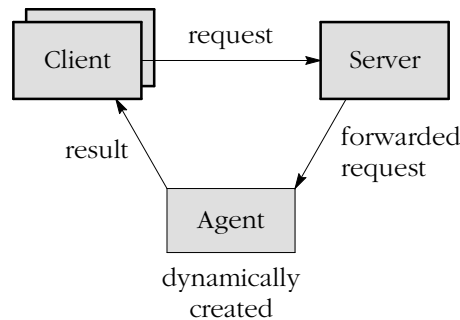
A terminate statement is only allowed within the statement part of a process body. Termination of a process involves the following actions:

- The process waits until all processes that depend on it have terminated.
- The drivers of the process are disconnected from the signals that they drive. It is an error if any of these signals are not guarded signals.

- The formal ports are disassociated from the actual signals and channels.

## 7.5 Example: A Client-Server System

This example is a model of a client-server system in which the server is multi-threaded, allowing it to serve multiple transactions concurrently. Since the number of clients to be served concurrently is not known *a priori*, the server creates agents dynamically to perform the transactions. The organization of the system is illustrated below. The system may ultimately be implemented in software, but it is desirable to model it early in the design flow before hardware/software partitioning is performed.



The type `result_channel` represents a channel for receiving result messages from the server, and the type `result_ref` is a reference to such a channel. The type `request_info` is the message type for requests to the server. It includes a reference to the channel upon which the client expects to receive the result of the request. The type `request_channel` represents a channel for sending requests, and the type `request_ref` is a reference to a request channel.

The client process's port is a channel upon which it sends requests. Part of the client's state is a dynamically created channel for receiving transaction results. When the client makes a request, it includes the reference to its result channel as part of the request.

The server process has a channel port for receiving requests, and encapsulates a process declaration for agents, which also has a channel port for requests. The body of the server receives a request message on its request channel, and saves the request in the variable `info`. It then dynamically creates a new request channel and a new agent process, with the agent's request channel port mapped to the new request channel. The server then forwards the saved request message via the new channel. The newly created agent receives the forwarded message, performs the transaction, and sends the results to the channel referenced in the request message. The agent then terminates. While the agent is processing the transaction, the server may receive further request messages and create agents to process them concurrently.

**architecture system\_level of client\_server\_system is**

```

type result_value is . . . ;
type result_channel is channel of result_value;
type result_ref is access result_channel;
type request_info is record
  . . . ; -- info for the transaction
  result_please : result_ref;
end record request_info;
type request_channel is channel of request_info;
type request_ref is access request_channel;

process client is
  port ( channel request : out request_channel );
  
```

```

    variable result : result_ref := new result_channel;
begin
    ...
    send ( . . . , result ) to request;
    receive . . . from result.all;
    ...
end process client;

process server is
    port ( channel request : in request_channel );
    process agent is
        port ( channel request : in request_channel );
        variable info : request_info;
    begin
        receive info from request;
        . . . ; -- perform transaction
        send . . . to info.result_please.all;
        terminate;
    end process agent;
    variable info : request_info;
    variable new_agent_request : request_ref;
begin
    receive info from request;
    new_agent_request := new request_channel;
    process agent
        port map ( new_agent_request.all );
    send info to new_agent_request.all;
end process server;
channel server_request : request_channel;
begin
    the_server : process server
        port map ( request => server_request );
    client_pool : for client_index in 1 to number_of_clients generate
        a_client : process client
            port map ( request => server_request );
        end generate client_pool;
end architecture system_level;

```

## 8. Other Related Changes

This section described miscellaneous extensions to VHDL that enable more effective use of the extensions described earlier.

### 8.1 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 interface 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""." ) );
```

---

## 9. Summary of Syntax Changes

```

actual_designator ::=
    . . .
    | type_mark
    | subprogram_name
    | package_instance_name

aggregate ::= record_aggregate | extension_aggregate | array_aggregate

ancestor_part ::= expression | type_mark

array_aggregate ::= ( element_association_list )

block_declarative_item ::=
    . . .
    | package_declaration
    | package_body_declaration
    | process_declaration
    | process_body
    | generic_package_instantiation
    | generic_subprogram_instantiation

buffer_constraint ::=
    buffer simple_expression

channel_declaration ::=
    channel identifier_list : subtype_indication ;

channel_type_definition ::=
    unbounded_channel_definition
    | unconstrained_bounded_channel_definition
    | constrained_bounded_channel_definition

concurrent_statement ::=
    . . .
    | process_instantiation_statement

constrained_bounded_channel_definition ::=
    channel buffer_constraint of subtype_indication
    | null channel buffer_constraint

constraint ::=
    range_constraint
    | index_constraint
    | buffer_constraint

derived_type_definition ::=
    [ abstract ] new parent subtype_indication [ record_extension_part ]

element_association_list ::=
    element_association { , element_association }

entity_declarative_item ::=
    . . .
    | package_declaration
    | package_body_declaration
    | process_declaration

```

```

| process_body
| generic_package_instantiation
| generic_subprogram_instantiation

extension_aggregate ::= ( ancestor_part with record_element_association_list )

generic_package_instantiation ::=
    package identifier is new generic_package_name
    [ generic_map_aspect ] ;

generic_subprogram_instantiation ::=
    subprogram_kind designator is new generic_subprogram_name
    [ generic_map_aspect ] ;

guard ::= when condition =>

interface_access_type_definition ::= access_type_definition

interface_array_type_definition ::= array_type_definition

interface_channel_declaration ::=
    channel identifier_list : [ mode ] subtype_indication

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

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

interface_discrete_type_definition ::= ( <> )

interface_file_type_definition ::= file_type_definition

interface_floating_type_definition ::= range <> . <>

interface_integer_type_definition ::= range <>

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

interface_package_declaration ::=
    package identifier is
        new generic_package_name interface_package_actual_part ;

interface_physical_type_definition ::= units <>

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

interface_subprogram_declaration ::=
    subprogram_specification [ is subprogram_default ]

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

name ::=
    . . .
    | type_conversion

object_declaration ::=
    . . .
    | channel_declaration

package_body_declarative_item ::=
    . . .
    | package_declaration
    | package_body_declaration
    | process_declaration
    | process_body
    | generic_package_instantiation
    | generic_subprogram_instantiation

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

package_declarative_item ::=
    . . .
    | package_declaration
    | process_declaration
    | generic_package_instantiation
    | generic_subprogram_instantiation

package_private_declarative_item ::=
    subprogram_declaration
    | process_declaration
    | 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
| generic_package_instantiation
| generic_subprogram_instantiation
package_private_declarative_part ::= { package_private_declarative_item }
primary_unit ::=
    . . .
    | generic_package_instantiation
private_extension_declaration ::=
    type identifier is
        [ abstract ] new ancestor_subtype_indication with [ access ] private ;
private_type_declaration ::=
    type identifier is [ [ abstract ] tagged ] [ limited ] [ access ] private ;
process_body ::=
    process_specification
    process_declarative_part
    begin
        process_statement_part
    end process [ process_simple_name ] ;
process_declaration ::=
    process_specification
    end process [ process_simple_name ] ;
process_declarative_item ::=
    . . .
    | package_declaration
    | package_body_declaration
    | process_declaration
    | process_body
    | generic_package_instantiation
    | generic_subprogram_instantiation
process_instantiation_statement ::=
    [ instantiation_label : ]
    process process_name
        [ generic_map_aspect ]
        [ port_map_aspect ] ;
process_specification ::=
    process identifier is
        [ generic_clause ]
        [ port_clause ]
record_aggregate ::= ( record_element_association_list )
record_definition ::=
    record
        element_declaration
        { element_declaration }
    end record [ record_type_simple_name ]
    | null record
record_element_association_list ::=
    element_association_list
    | null record

```

```

record_extension_part ::= with record_definition

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

receive_alternative ::=
    receive_statement [ sequence_of_statements ]

receive_statement ::=
    [ label : ] receive [ target ] from channel_name ;

select_alternative ::=
    receive_alternative
  | send_alternative

select_statement ::=
    [ select_label : ]
    select
        [ guard ] select_alternative
    { or
        [ guard ] select_alternative }
    [ or
        timeout_alternative ]
    [ else
        sequence_of_statements ]
    end select [ select_label ] ;

send_alternative ::=
    send_statement [ sequence_of_statements ]

send_statement ::=
    [ label : ] send [ expression ] to channel_name ;

sequential_process_instantiation_statement ::=
    [ label : ]
    process process_name
        [ generic_map_aspect ]
        [ port_map_aspect ] ;

sequential_statement ::=
    . . .
    | send_statement
    | receive_statement
    | select_statement
    | sequential_process_instantiation_statement
    | terminate_statement

subprogram_declaration ::= subprogram_specification [ is abstract ] ;

subprogram_declarative_item ::=
    . . .
    | package_declaration
    | package_body_declaration
    | process_declaration
    | process_body
    | generic_package_instantiation
    | generic_subprogram_instantiation

subprogram_default ::= name | <>

```

```

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

terminate_statement ::=
    [ label : ] terminate ;

timeout_alternative ::=
    timeout_guard sequence_of_statements

timeout_guard ::= after time_expression =>

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

type_declaration ::=
    . . .
    | private_type_declaration
    | private_type_extension

type_definition ::=
    . . .
    | derived_type_definition
    | channel_type_definition

unbounded_channel_definition ::=
    channel of subtype_indication
    | null channel

unconstrained_bounded_channel_definition ::=
    channel buffer <> of subtype_indication
    | null channel buffer <>

```

## 10. References

- [1] 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.
- [2] P. J. Ashenden and P. A. Wilsey, “Considerations on System-Level Behavioural and Structural Modeling Extensions to VHDL,” *Proceedings of VHDL International Users Forum Spring 1998 Conference*, Santa Clara, CA, pp. 42–50, 1998.
- [3] J. Barnes, Ed. *Ada 95 Rationale, Lecture Notes in Computer Science*, vol. 1247. Berlin, Germany: Springer-Verlag, 1997.
- [4] G. Booch, *Object-Oriented Analysis and Design with Applications*. Redwood City, CA: Benjamin/Cummins, 1994.
- [5] F. P. Brooks, Jr., *The Mythical Man-Month*, Anniversary ed. Reading, MA: Addison-Wesley, 1995.
- [6] E. Christen, “VHDL’93 Design Guidelines,” personal communication, 1997.
- [7] C. A. R. Hoare, *Communicating Sequential Processes*. London: Prentice Hall, 1985.
- [8] IEEE, *Standard VHDL Language Reference Manual*. Standard 1076-1993, New York, NY: IEEE, 1993.
- [9] ISO/IEC, *Ada 95 Reference Manual*. International Standard ISO/IEC 8652:1995 (E), Berlin, Germany: Springer-Verlag, 1995.
- [10] P. Wegner, “Dimensions of Object-Based Language Design,” *ACM SIGPLAN Notices*, vol. 22, no. 12, *Proceedings of OOPSLA ’87*, pp. 168–182, 1987.