Reuse Through Genericity in SUAVE

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

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

Abstract

VHDL currently has a limited form of genericity in which component and entity declarations can be parameterized with formal generic constants. SUAVE extends the genericity mechanism by allowing formal generics types and by allowing generics to be specified in the interfaces of subprograms and packages. The approach is based on the features of Ada-95. It allows units to be re-used in a much wider variety of contexts without modifying the original code. In this paper, we show that the genericity added by SUAVE enhances reuse across the spectrum of modeling, from high-level to gate level. In particular, the genericity extensions interact with the SUAVE extensions for object-oriented data modeling to significantly improve support for high-level behavioral modeling and for developing test-benches. We show that the genericity extensions integrate seamlessly with the existing language. Furthermore, the implementation burden is not large, and, since generic instantiation is performed at elaboration time, there is no performance penalty in simulation or synthesis.

1. Introduction

One important objective of object-oriented extensions to VHDL is to improve support for re-use of models through the mechanism of inheritance. However, focussing on this approach ignores the important, orthogonal mechanism of genericity, which supports re-use in a much wider variety of contexts. SUAVE extends VHDL by adding both object-oriented and genericity features. The object-oriented extensions are described in a companion paper [3]; this paper describes the genericity extensions.

VHDL currently has a limited form of genericity in component and entity declarations. A component or entity can have formal generic constants, with actual constant values being supplied at elaboration time when the unit is instantiated. Generic constants are typically used in two ways. First, they are used to constrain the index bounds of array ports of a unit. This allows the unit to be re-used in contexts where the bounds of actual signals associated with the ports vary. Second, they are used to specify timing and operational parameters of a unit. This allows the unit to be re-used in contexts where the timing or operational requirements vary.

SUAVE extends the genericity mechanism of VHDL by allowing types to specified as formal generics and by allowing generics to be specified in the interfaces of subprograms and packages. The approach is based on the features of Ada-95. Use of formal type generics allows units to be reused in contexts where data of different types is to be manipulated. For example, a multiplexer can be specified with a formal type generic for the type of the input and output data. This allows the multiplexer model to be reused as a bit, bit_vector, std_logic, std_logic_vector, integer, or user-defined-type multiplexer, without modifying the original model code. Use of generics in the interfaces of subprograms and packages allows definition of container abstract data types that can be reused to contain data of different types. For example, a generic package can be defined to represent and manipulate sequences of integer, time values, or test vectors for different devices under test, again without modifying the original package code.

This paper outlines the SUAVE extensions for genericity and illustrates their use through some examples. A more complete description can be found in the SUAVE report [4]. Most of the features added to VHDL are adapted from features in Ada-95 [7], and are included largely for the same reasons that they are included in Ada-95 [5]. We

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

show that the genericity added by SUAVE enhances reuse across the spectrum of modeling, from high-level to gate level. In particular, the genericity extensions interact with the SUAVE extensions for object-oriented data modeling to significantly improve support for high-level behavioral modeling and for developing test-benches. We show that the genericity extensions integrate seamlessly with the existing language. Furthermore, the implementation burden is not large, and, since generic instantiation is performed at elaboration time, there is no performance penalty in simulation or synthesis.

Section 2 of this paper presents the design objectives that motivated the extensions for genericity in SUAVE. Section 3 is an overview of the existing genericity mechanism in VHDL, with which the SUAVE extensions integrate. Section 4 describes the SUAVE type genericity extensions, and Section 5 illustrates them with a number of examples. Section 6 describes and illustrates further extensions for subprogram and package genericity. Our conclusions are presented in Section 7.

2. SUAVE Design Objectives

A previous paper [2] reviews the issues to be addressed in extending VHDL for high-level modeling and discusses principles that should govern the design of language extensions. As a result of that analysis, a number of design objectives were formulated for SUAVE. Among them, the following motivated our approach to incorporating genericity features into VHDL:

- to improve support for re-use and incremental development by allowing further delaying of bindings through type-genericity and dynamic polymorphism,
- to preserve capabilities for synthesis and other forms of design analysis,
- to support hardware/software co-design through improved integration with programming languages (e.g., Ada),
- to 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, to preserve what Brooks refers to as the “conceptual integrity” of the language [6]. In particular, the extensions for genericity must integrate well with the existing mechanism for generics in VHDL.

3. Overview of Existing Generics

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

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

The generic constant Tpd is used to parameterize the multiplexer with respect to propagation-delay; width is used in the index constraints for the data ports; and trace is used to control whether the multiplexer traces values passed to the output. The generic constants are visible in any architecture corresponding to this entity, and can be used in the implementation of the structure or behavior of the design entity. Note that the generics are in fact constant objects, since that class is the default for interface objects in a generic clause. Thus, the generic clause could be rewritten stating the class explicitly:

```
    generic ( constant Tpd : time; . . . );
```

Note also that the generic constant trace has a default value that is used if no actual value is supplied when the entity is instantiated.

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

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

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

4. Extension of Generics

One of the main aspects that constrains re-use of the multiplexer entity described in Section 3 is that it can only be instantiated to deal with bit-vector values. A more re-usable multiplexer entity would be instantiable for arbitrary types. Thus, it is desirable to be able to specify the data type as a formal generic. In many cases, this is feasible, since the implementation of a unit does not depend on the details of any particular type. For example, a behavioral implementation of a multiplexer simply involves assigning values from input to output, irrespective of the type of the values. A given multiplexer instance, however, should only be allowed to deal with values of one particular type, namely the type of the signals connected to its data ports. This restriction is in conformance with the strong-typing philosophy of VHDL.

SUAVE extends the generic clause feature of VHDL by allowing specification of formal generic types. A unit may use a formal type to define ports and other objects in its implementation. When the unit is instantiated, an actual type is associated with the formal type for that instance. The association occurs when the instance is elaborated.

The particular mechanism for specifying formal types is modeled on the corresponding mechanism in Ada-95 [7], but is adapted to integrate cleanly with the existing generic mechanism in VHDL. A formal generic type is specified in the following form in a generic clause:

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

For example, the multiplexer entity in Section 3 might be revised as follows to include a formal generic type for the data to be handled. (The propagation delay and tracing generic constants are omitted for clarity of illustration.)

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

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

Table 1. Classes of formal generic types in SUAVE.

SUAVE allows a number of different classes of type definition, each restricting the actual type that can be associated when the unit is instantiated, as shown in Table 1. (Further refinements to the first three classes are described in the SUAVE report [4].) The implementation of a unit can make use of the knowledge about the formal type afforded by the definition. For example, it may use arithmetic operations on a formal integer type, or indexing on a formal array type. Section 5 includes further examples of use of formal types.

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

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

SUAVE further extends VHDL by allowing package declarations and subprogram specifications to include generic clauses, enabling definition of template packages and subprograms that can be re-used with different type bindings. This feature combines with the object-oriented extensions in SUAVE [3] to provide means of defining generic abstract data types in a type-secure way.

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

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

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

```
type amplitude is range - 10.0 to +10.0;

package amplitude_ops is new float_ops
    generic map ( float_type => amplitude );
```

Note that the instance is a package declared within an enclosing declarative region. SUAVE generalizes the use of packages by allowing them to be declared in inner regions, rather than just as library units. This generalization is also related to the use of packages in the object-oriented extensions, and is discussed further in that context [3].

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

```
procedure swap
    generic ( type data_type is private )
    ( a, b : inout data_type ) is

    variable temp : data_type;

begin
    temp := a;  a := b;  b := temp;
end procedure swap;
```

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

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

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

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

5. Examples Using Generic Types

5.1 Generic Multiplexer

An entity declaration for a generic multiplexer is shown in Section 4. A corresponding architecture body is:

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

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

5.2 Generic Queue ADT Package

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

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

    type queue is access private;

    impure function new_queue return queue;
    impure function is_empty ( Q : in queue )
                        return boolean;
    procedure append ( Q : inout queue;
                       E : in element_type );
    procedure extract_head ( Q : inout queue;
                             E : out element_type );

private

    type element_node;
    type element_ptr is access element_node;
    type element_node is record
            next_element : element_ptr;
            value : element_type;
        end record element_node;
    type queue is record
            head, tail : element_ptr;
        end record queue;

end package queues;
```

The type of elements to be included in a queue is represented by the formal type element_type. The queue type is a private type [3], whose concrete implementation is a linked list of nodes, each containing a value of the element type. The details of the concrete implementation are in the private part of the package (between the keywords **private** and **end packge**), and are thus not hidden from a package user. The ADT operations have parameters of the queue and element types. The queue package might be instantiated and used to deal with queues of test vectors, for example, as follows:

```
type test_vector is . . .;

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

variable tests_pending : test_queues.queue
            := test_queues.new_queue;
. . .

test_queues.append ( tests_pending, generated_test );
```

5.3 Generic Counter

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

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

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

```
architecture behavioral of counter is
begin

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

end architecture behavioral;
```

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

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

state_counter : entity work.counter(behavioral)
    generic map ( count_type => state_type)
    port map ( clk => master_clk, data => state_data );
```

5.4 Generic Shift Register

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

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

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

```
architecture behavioral of shift_register is
begin

    shift_behavior : process is

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

end architecture behavioral;
```

The state of the shift register is represented by the variable stored_data, whose type is an array of the same size and element type as the formal array type. The behavior in the process statement is expressed using only the knowledge that the index type is discrete, that the stored data can be indexed, and that the data output port can be assigned the stored array value. An example of instantiation of the shift register is:

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

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

5.5 Mixin Inheritance

In the companion paper [3], we describe the SUAVE features for object-oriented inheritance based on derived tagged types. These features can be combined with formal generic derived types to provide a form of *mixin inheritance* [9]. Languages such as C++ use multiple inheritance for this purpose, but with the SUAVE mechanisms, multiple inheritance is not needed.

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

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

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

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

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

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

    function effective_address
                ( instr : indexed_instruction )
                return address;

end package indexed_addressing_mixin;
```

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

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

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

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

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

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

6. Formal Subprograms and Packages

In the previous sections, formal generic types were described and illustrated. SUAVE also adopts formal generic subprograms and packages from Ada-95. These features significantly aid re-use of generic units. Use of formal generic subprograms is described in this section. Space considerations preclude description of formal generic packages; they are described in the SUAVE report [4].

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

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

    type ordered_collection is limited access private;

    function new_ordered_collection
                return ordered_collection;

    procedure insert ( c : inout ordered_collection;
                       e : in element_type );

    procedure traverse
        generic ( procedure action
                            ( element : in element_type ) )
        ( c : in ordered_collection );

private

    type ordered_collection_object;
    type ordered_collection_ptr is
        access ordered_collection_object;
    type ordered_collection_object is record
            next_element,
            prev_element : ordered_collection_ptr;
            element : element_type;
        end record tree_record;

    type ordered_collection is
        new ordered_collection_ptr;

end package ordered_collection_adt;
```

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

```
package body ordered_collection_adt is

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

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

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

end package body ordered_collection_adt;
```

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

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

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

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

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

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

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

```
use ordered_stimulus_collection_adt.all;
variable dut_stimuli : ordered_collection
            := new_ordered_collection;
signal dut_inputs : std_logic_vector
                        (0 to stimulus_vector_length -  1);

procedure apply_stimulus
            ( stimulus : stimulus_element ) is
begin
    dut_inputs <= stimulus.pattern;
    wait for 100 ns;
end procedure apply_stimulus;

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

. . .

apply_all_stimuli ( dut_stimuli );
```

7. Conclusion

In this paper we have described the SUAVE extensions to VHDL to improve its support for re-use through genericity. These features combine with SUAVE's extensions for object-oriented modeling [3] to significantly improve the expressiveness of VHDL at all levels of modeling abstraction. The genericity features described here are drawn from Ada-95 and are adapted to integrate with VHDL's existing genericity mechanism. Drawing on Ada is appropriate, since VHDL was originally strongly influenced by Ada.

Space considerations preclude a more detailed definition of the features added in SUAVE. The interested reader can find a more complete description in the SUAVE report [4]. Work is now in progress to implement the extensions within the framework of the SAVANT project [8].

References

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

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

[3] P. J. Ashenden and P. A. Wilsey, "SUAVE: Painless Extension for an Object-Oriented VHDL," *Proceedings of VHDL International Users Forum Fall 1997 Conference*, Washington, DC, 1997.

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

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

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

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

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

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