

Object Orienting VHDL for Component Modeling

C. R. Ramesh
Senior Consultant
ViewLogic Systems, Inc.
47211 Lakeview Blvd.
Fremont, CA 94538.
(510) 659-0901 x217
cramesh@viewlogic.com

Abstract

This paper examines the Object-Oriented Features that are present in VHDL, such as, abstraction, encapsulation, modularity, polymorphism etc., also examined are the features that can be incorporated in VHDL, such as inheritance, that will enhance the development, maintenance and usability of VHDL component models. Examples have been provided to illustrate the object oriented features available in VHDL. Examples of Object-Oriented extensions to VHDL, based on the preliminary work done in this area have been included. The goal of this paper is to leverage existent ideas towards standardized or customized extensions to VHDL, and to educate designers in being systematic about model development.

A view on the current proposals related to Object-Oriented extensions to VHDL, coming from the shared variable group and the initial findings of the working group on Object Oriented extensions to VHDL is also included.

1. Object-oriented Modeling and Design - Rumbaugh, Blaha et al., Prentice Hall.
2. Object-oriented Software Construction - Bertrand Meyer, Prentice Hall.
3. OO-VHDL: Object-Oriented Extensions for VHDL, Presentation by Vista Technologies, DAC 1994, OO VHDL Study Group.

OBJECTIVES OF OBJECT ORIENTED DESIGN

- Reduction of Implementation Complexity
- Maximization of Code Reuse
- Minimizing Maintenance and upgrading Costs
- Increasing Robustness

Reduction of Implementation Complexity

The reduction in complexity in modeling is accomplished using abstraction and encapsulation.

⇒ Behavioral, Data Flow or Algorithmic models along with visibility rules.

Maximization of Code Reuse

Code reuse is promoted by modularizing the design and by sharing of common data

⇒ Promoted by inheritance, packaging, genericity

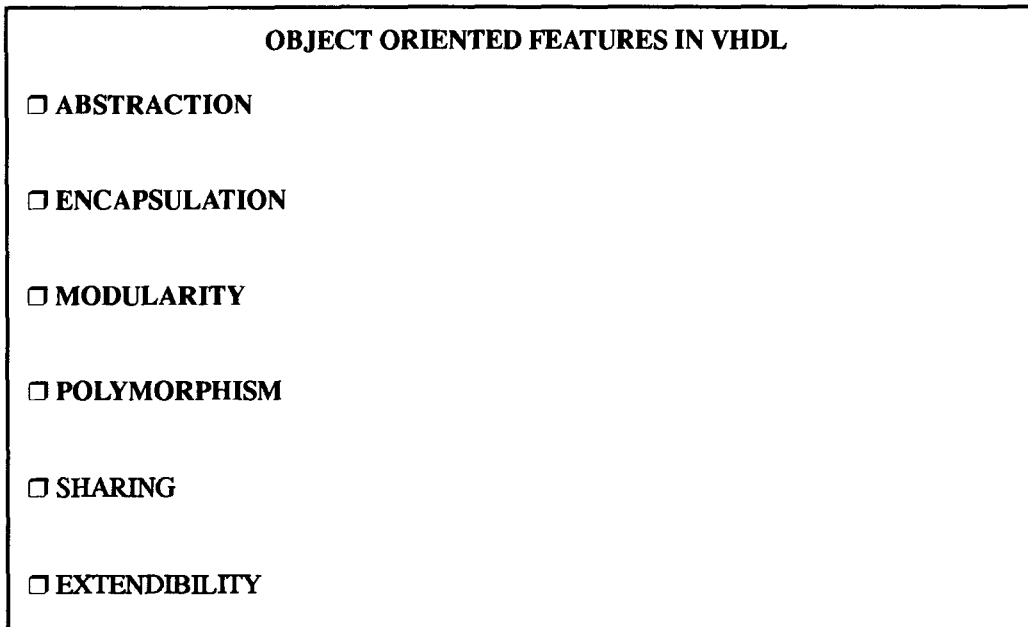
Minimizing Maintenance and upgrading Costs

Polymorphism, extensibility

⇒ Overloading of operators and subprograms along with the usage of generics and generates

Increasing Robustness

Strong typing, increasing code reuse improves quality



Abstraction: Focus on behavior or functionality of the model and not the implementation
Example: Behavioral VHDL models

Encapsulation: Separation of external aspects from internal implementation details

Example:

Subprogram declarations in packages and package bodies

⇒ Visibility rules ensure that internal implementation details (items declared in a package body) are not accessible outside the package body. Only the interface of the subprogram is visible to the external world.

Modularity: The ability to partition a system into smaller subsystems

Example:

subprograms, processes/components in an architecture.

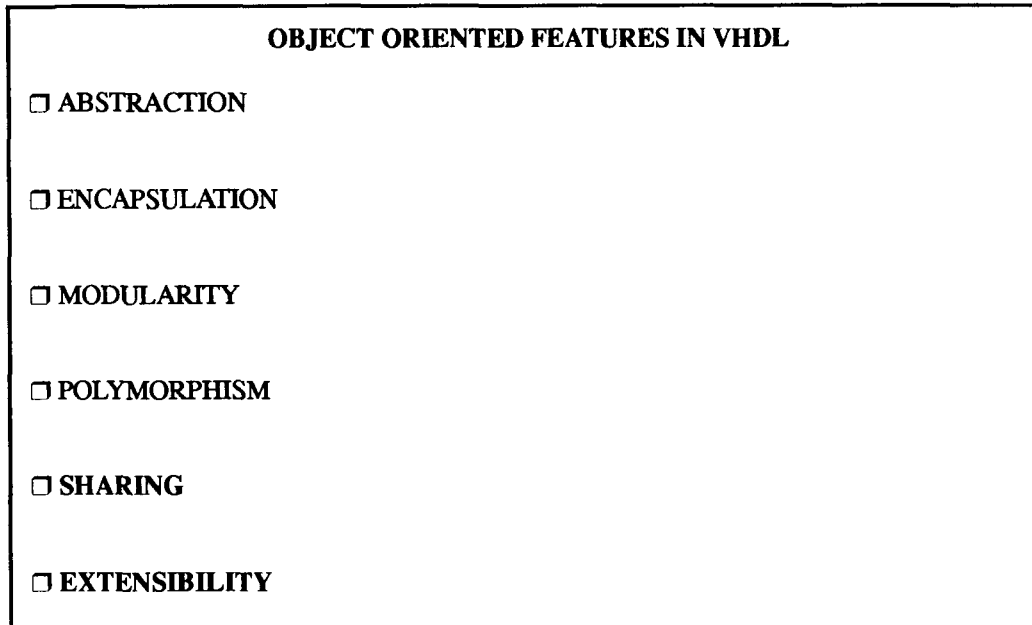
Polymorphism: The ability of an object to take several forms.

⇒ Reduces the semantic complexity of the model and increases the flexibility by automating the decision making process

Example:

Overloading of operators and subprograms and enumeration literals

⇒ Based on parameter and result type profile or on context.



Sharing: The ability to share information within an application or among applications.

1. The sharing of objects initialized at runtime within an application.
 - ⇒ Only signals can be used to share dynamic information among various modules/processes currently.
 - ⇒ VASG group on shared variables. Not currently available in VHDL
2. The sharing of commonly used objects, operators, subprograms, types

Example:

Packages in VHDL promote the reuse of commonly used subprograms, types, operators and global signals.

⇒ Packages in VHDL promote code reuse at the data structure level.

Extensibility: The ability to adapt to changes in the specification of the product.

⇒ Currently this can be accomplished using “generics” and “generates” in VHDL.

Example:

1. Generically constrained ports can be used to model repetitive structures.
2. Usage of unconstrained formal parameters in subprograms allow them to be extended to a range determined at runtime.
3. Generically constrained ports allow one to extend the size of the model
4. Generic timing parameters allow the model to be reconfigured to operate under different conditions.

Example: EXAMPLE ILLUSTRATING ABSTRACTION AND EXTENSIBILITY IN VHDL

```

-----
-- Abstract model of a Read only Memory
-- Parameters than can be changed: Number of addressable loca-
tions, Wordsize
-- and the access time
-----

LIBRARY ieee;
  USE ieee.std_logic_1164.all;
ENTITY ROM is
-----
  --The generic "input_size" determines the number of addressable
locations
  -- The generic "output_size" determines the wordlength
  -- The generic "Taccess" determines the
-----

  GENERIC (INPUT_SIZE: integer := 16;
           OUTPUT_SIZE: INTEGER := 16;
           TAccess: TIME := 3 ns);
  port (
    Address: IN std_ulogic_vector(input_size-1 downto 0);
    Data: OUT std_logic_vector (output_size -1 downto 0);
    Read: IN std_ulogic
  );
END ROM;

ARCHITECTURE behave of ROM is
TYPE mem_array IS array(natural range <>) of std_logic_vector(-
output_size-1 downto 0);
SUBTYPE mem_type is mem_array(2**input_size-1 downto 0);-- 64KX16
memory
begin -- behave
Read: process (Read)
  variable rom: mem_type := (others => (others =>'0'));
begin
  Data <= ROM(To_Integer(Address)) after Taccess;
end process Read;
end behave;

```

THE NEED FOR OBJECT ORIENTED EXTENSIONS TO VHDL

- ① **Sharing information is limited**
- ② **Effect of Modularity on reusability or extensibility is limited**
- ③ **Extensibility of models limited**
- ④ **Polymorphism extended to include classes can increase the flexibility in modeling**

① **Sharing information is limited**

- ⇒ Current features do not exploit commonality among applications (lack of Inheritance)
- ⇒ Limited means to passing global information

Example:

1. Packages do not exploit the commonality among implementations.
2. Dynamic information sharing is limited to Signals

② **Effect of Modularity on reusability or extensibility is limited**

- ⇒ The top-down approach the decomposition of an abstract model does not guarantee reusable modules.

Example:

The decomposition from Behavioral -> RTL -> Gates does not necessarily produce modules that can be reused.

③ **Extensibility of models limited**

- ⇒ Extensibility is limited to generics and unconstrained types

Example:

Shift registers can be modeled using generate statements with their ranges constrained by generic parameters.

THE NEED FOR OBJECT ORIENTED EXTENSIONS TO VHDL

- ① Sharing information is limited
- ② Effect of Modularity on reusability or extensibility is limited
- ③ Extensibility of models limited
- ④ Polymorphism extended to include classes can increase the flexibility in modeling

- ④ Polymorphism extended to include classes can increase the flexibility in modeling
 - ⇒ Polymorphism of operators and subprograms is limited to the data structures.

The equivalent of this in C++ is called the virtual function.

Example: EXAMPLE ILLUSTRATING THE LIMITATIONS OF VHDL

Example2: commonality among implementations not exploited (Sharing information is limited)

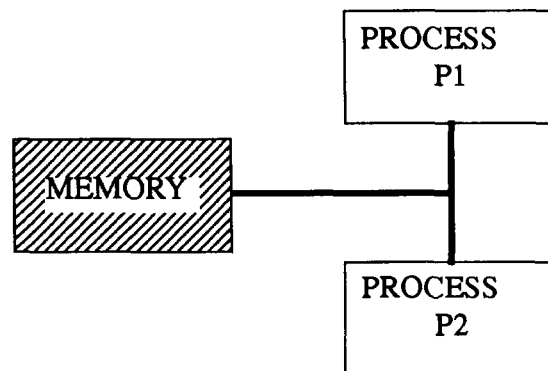
```

ARCHITECTURE behave of ROM is
TYPE mem_array IS array(natural range <>) of
std_logic_vector (output_size-1 downto 0);
SUBTYPE mem_type is mem_array(2**input_size-1 downto 0);
-- 64KX16 memory
begin -- behave
end behave;

```

The type declarations shown above could also be useful when modeling a RAM or a PROM or any other type of memory. The Current restrictions in VHDL do not allow these type definitions to be shared among the various models (information has to be replicated).

Example3: Limited means to passing global information (Sharing information is limited)



- ⇒ Memories are modeled using variables, since they are more efficient than using signals.
- ⇒ Signals can not be an access type.
- ⇒ Modeling a shared memory application is very difficult as all the information have to be passed using signals.

The introduction of shared variables in VHDL can help solve this problem.

OBJECT ORIENTED EXTENSIONS TO VHDL

- **Classification**
- **Inheritance**
- **Polymorphism**

● **Classification**

A collection of design entities with identical behavior and or data structures.

⇒ Increases the level of abstraction in modeling

● **Inheritance**

The ability to share data among implementations (classes) based on a hierarchical relationship.

⇒ Increases focus on the commonality among implementations

⇒ Key in reducing repetition

⇒ Increases the level of abstraction in modeling

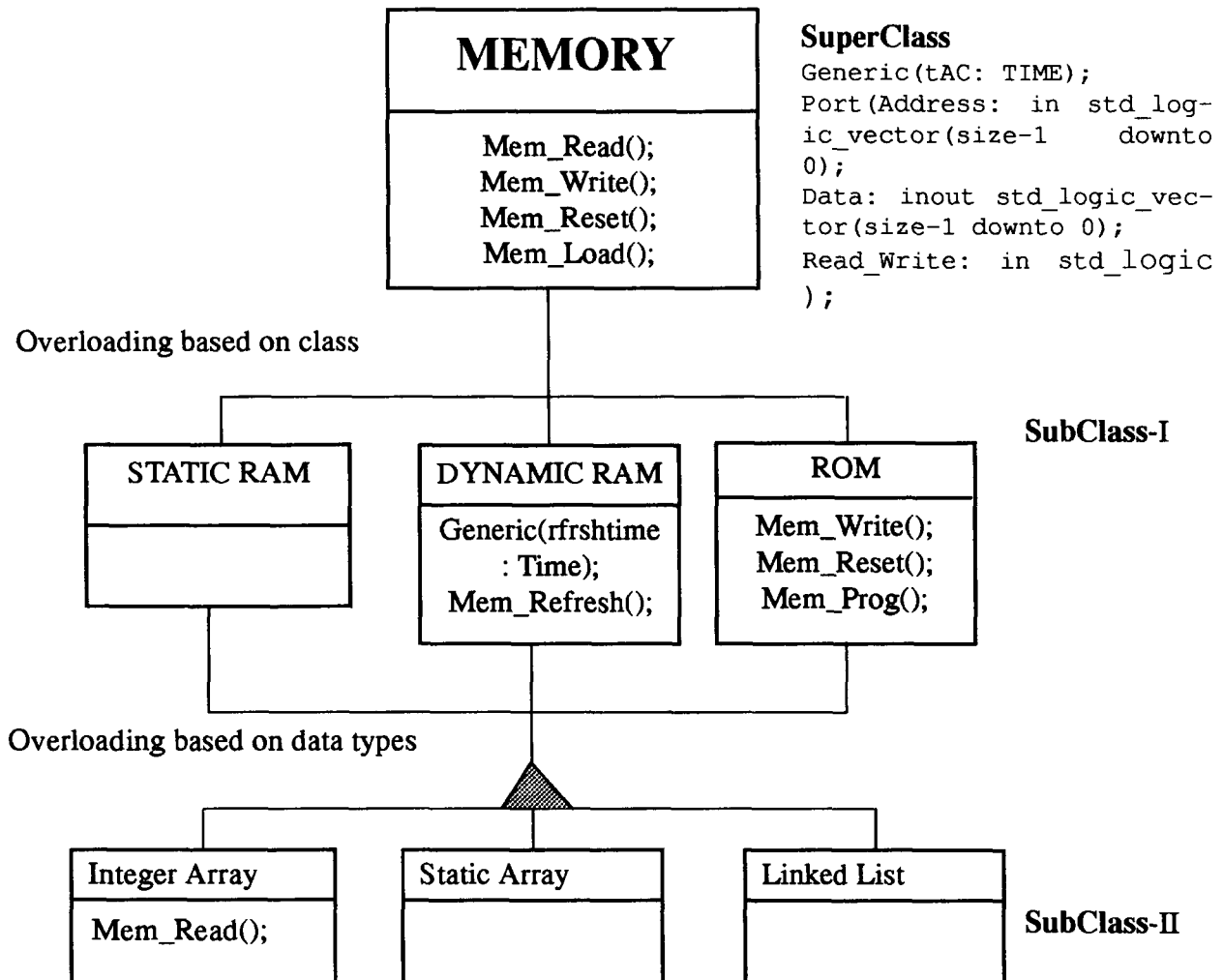
⇒ Reduces the amount of rework to be done to a model

● **Polymorphism**

Polymorphism in VHDL is based on parameter and result type or on the context.

⇒ Polymorphism combined with inheritance, based on classes, parameter and result type or on the context can be a very powerful feature

OBJECT-ORIENTED EXTENSIONS AND COMPONENT MODELING



SuperClass:

⇒ common port information, generics and subprograms are declared as a part of this superclass

SubClass-I:

- ⇒ New Generics and Ports added to the superclass. (extensibility)
- ⇒ Subprograms declared as a part of refinement (extensibility & reuse)
- ⇒ Subprograms overloaded based on class (polymorphism based on class)

SubClass-II: Leaf level

⇒ Overloading based on data-structures (polymorphism based on parameter and result type)

Example 4: EXAMPLE ILLUSTRATING THE OBJECT ORIENTED EXTENSIONS TO VHDL

```

-----
-- Declarations Of a Memory Class (template)
-----
LIBRARY ieee;
  USE ieee.std_logic_1164.all;
ENTITY CLASS MEMORY is
-----
-- The generic "input_size" determines
-- the number of addressable locations
-- The generic "output_size" determines
-- the wordlength
-- The generic "Taccess" determines the
-----
  GENERIC (INPUT_SIZE: integer := 8;
           OUTPUT_SIZE: integer := 16;
           Taccess: time := 3 ns);
  port (
    Address: IN std_logic_vector(input_size-1 downto 0);
    Data: OUT std_logic_vector (output_size -1 downto 0);
    Read_WRITE: IN std_ulogic
  );
  TYPE mem_array IS array(natural range <>) of std_logic_vector(-
output_size-1 downto 0);
  SUBTYPE mem_type is mem_array(2**input_size-1 downto 0);
END CLASS MEMORY;

ARCHITECTURE behave of MEMORY is
  PROCEDURE MEM_READ (ADDRESS: IN std_logic_vector; DATA: OUT std_
logic_vector) IS
  -- BODY OF THE PROCEDURE
  END MEM_READ;
  BEGIN
  -- ARCHITECTURE BODY
  END behave;

```

```

-----
-- ABSTRACT MODEL OF A Of a DRAM
-----
LIBRARY ieee;
  USE ieee.std_logic_1164.all;
ENTITY DRAM IS NEW MEMORY -- Indicates that it's a subclass of
    -- memory
  -- Adding a new generic to the template
  GENERIC (refresh_time: time);
  -- Add a new port to the template
  PORT(REFRESH: IN std_ulogic);
END DRAM;

ARCHITECTURE inherited of DRAM is
  -- Adding a new subprogram to suit the needs of the DRAM
  PROCEDURE MEM_REFRESH(VARIABLE MEMID:INOUT memacctype) IS
  BEGIN
  END MEM_REFRESH;

BEGIN
-- ARCHITECTURE BODY
-- Instantiate the generic implementation of the data structure
u0: generic_mem_ds
END inherited;

CONFIGURATION dram_integer_conf of DRAM is
  FOR inherited
  -- Bind a specific data structure to the dram architecture
  FOR u0: generic_mem_ds
    USE ENTITY INTEGER_ARRAY(INTEGER_ARRAY);
  END FOR;
END dram_integer_conf;

```

CURRENT WORK ON OBJECT-ORIENTED EXTENSIONS TO VHDL

- DASC Study Group on Object-Oriented Extensions to VHDL
- DASC Shared Variable Working Group

Work on Object-Oriented Extensions to VHDL

The group was formed to examine the motivation, requirements, proposals on extensions and terminology, relating to Object-Orienting VHDL. The group is chaired by Doug Dunlop. Some of the needs discussed here have been:

1. To promote the development of re-usable VHDL models,
2. To incorporate higher-level modeling features in VHDL
3. To improve object encapsulation and abstraction

DASC Shared Variable Working Group

This group was formed to unify the various requirements to fulfill dynamic information sharing efficiently and the need to use VHDL for higher level modeling. The group is chaired by Steve Bailey. The group has been involved in identifying and prioritizing the language design requirements. More information about the requirements specification are available by anonymous ftp at “vhdl.org” in the directory “/vi/svwg/doc/require.txt”.