Abstraction
Object Oriented Paradigm (OOP) has changed the perspective how the program must be written. OOP gives better develop, maintain, analysis, coding, and understanding of complex system. It affects also VHDL as a language to describe behaviour of digital circuit in field programmable gate arrays (FPGA) and application specific integration circuit (ASIC). There are many proposals for extension of VHDL to provide OOP way. One of them is Objective VHDL. Objective VHDL is designed to fulfil object orientation in VHDL without worrying about the programming languages will be used in software parts. Objective VHDL has complete solution from generating high level VHDL until synthesis and simulator.

1. Introduction

1.1. What is VHDL

VHDL (VHSIC Hardware Description Language) is usually used to describe behaviour of field programmable gate arrays (FPGA) and application specific integration circuits (ASIC) in electronic design automation of digital circuit. VHSIC is Very High Speed Integrated Circuit. VHDL is originally developed in US Department of Defense to describe the behaviour of the ASICs which supplier companies were including in equipment. It simplified to understand all implementation specific details

The initial version of VHDL, designed to IEEE (Institute of Electrical and Electronics Engineers) standard 1076-1987, included a wide range data types, such as numerical (integer and real), logical (boolean and bit), character and time, bit_vector (arrays of bit), and string (arrays of character). The new IEEE standard 1164 was including multi valued logic which represented a signal’s drive strength (none, weak, or strong) including unknown values. The latest version is VHDL 2005.

The key advantage of using VHDL is the model can be verified before translating the design into real hardware (gates and wires). The other is it can describe real concurrent system. It is not like the other programming languages that implement the concurrent as run sequentially but with schedule.

The translated VHDL is mapped into a programmable logic device such as a Complex Programmable Logic Device (CPLD) and FPGA or ASIC. Another type of hardware description language like VHDL is Verilog.

1.2. What is Object Oriented Programming

Object Oriented Programming (OOP) is a computer programming paradigm. The idea of this paradigm is to implement the functions and instruction in objects or individual units. Each object can communicated to each other by sending and receiving messages and it looks like independent unit with it own role.

This paradigm comes up to improve programming languages. The procedural language is hard to read and understand especially by new programmer or new worker. It makes the project longer to accomplish.

The advantages are greater flexibility, maintainability, and easy to learn. The power of OOP approach is often simpler to develop, maintain, analysis, coding, and understanding of complex situations and procedures than other programming paradigms. It said that OOP is more real world language because you can see the object but not the structure.

The concepts of OOP are
• Class: the abstraction of a model
• Method: subroutine operating on single object
• Object: the instance of a class, it is unique
• Abstraction: class that only implements the skeleton of the class not the
  implementations. The other classes will do the implementations.
• Inheritance: creating subclasses
• Encapsulation: ensuring that class only looks into the function in other class but not
  the implementation.
• Polymorphism: ability to choose the right method implementation for different objects
  with the same method name.

The implementation of OOP in programming language can be seen in Java, Simula, C++,
and Smalltalk.

2. Object Oriented VHDL

2.1. Background of Object Oriented VHDL

Object oriented paradigm attracts many software engineering because it is easy to manage
design complexity and increase software reuse. It affects in VHDL, the object oriented must
include in the standard of VHDL.

The original purpose of VHDL is to describe the behaviour of the system. It encapsulates
the system, more like black box view of piece hardware, and communicates with wire, signal
in term of VHDL. With the same idea behind the making of these models, it is possible to
integrate them.

2.2. Development of Object Oriented VHDL

The development of Object Oriented VHDL (OOVHDL) started in the early 1990. Some
researchers make OOVHDL dialects such as VHDL_OBJ [1], VHDL++ [2], and Vista [3] but
the latest news only three still exist. There are Ashenden’s SUAVE [4], Schumacher’s OO-
VHDL [5], and Objective VHDL [6].

The primary target of SUAVE is simulation. Schumacher makes translation from
OOVHDL to VHDL using record and subprograms which lack of VHDL synthesis tools
support. The main emphasize of the Objective VHDL is designed to ease synthesis with some
hardware-related restriction and hardware specific semantics [7].

2.3. What is Objective VHDL

Objective VHDL is one extension of VHDL. It provides object oriented paradigm to the
language. The use of Objective VHDL facilitates the use of object-orientation as a design
methodology that embraces both the hardware and software parts of an embedded system,
independent of the fact that a different object oriented language may be used for software [7].

The approach of Objective VHDL is based on the definition of a metal model of object
orientation which is suitable for translation into traditional, statically allocated, bound, and
scheduled hardware resources. The model is independent of a source language [7]. But still it
must be limited with restriction in respect to synthesis supported.

Overviews of the features in the Objective VHDL are:
• Object state: Each object has its own state. State is an attribute in term of class. The
  attribute contains the value which calls attribute’s state space. We store all the state
  information in this state space. The information about state space allows us to
calculate the number of bits which must be allocated in a memory or register to store
an object’s state.
• State transitions: Class has method to modify the object’s state. One object must not directly change the other object’s state. We used this method to invoke the other object. Method has input parameters, output parameters, and subroutine.

• Object lifetime: Unlike other object oriented language which support new and delete operator in term of dynamic memory allocation, Objective VHDL only supports static allocation. It avoids the problem of lack of resources.

• Communication between objects: In object oriented, each class communicates with each other by passing the message. The message contains identification of the requested method and values for its input parameters. Channels are used to passing the message. The target object waits message from the client object. Sometimes target object send back the output parameter to the client. All communication channels are static in targeting hardware synthesis. Since all objects are static, the static channels are possible to implement.

• Request arbitration: In a concurrent model, a server object may receive several messages at the same time from several concurrent clients. Because all the clients can access to object’s state in the same time, it can lead to inconsistency during read and write operation. A modelling guideline allowing synchronization has been devised in [5]. It is using guard or monitor to avoid this hazard.

3. Implementation of Objective VHDL

3.1. Class

Class is abstraction of a model. It defines structures (attributes), behaviours (methods), and relationships (associations). It describes what the object is and does. Objective VHDL provides this implementation by extending VHDL grammar. The grammar for class implementation is

| type_definition ::= scalar_type_definition |
|                  | composite_type_definition |
|                  | access_type_definition |
|                  | file_type_definition |
|                  | class_type_definition |

Table 1. Type definition grammar

| class_type_definition ::= class_type_declaration |
|                        | derived_class_type_declaration |
|                        | class_type_body |

Table 2. Class type definition grammar

It defines new type definition which is class type definition. This new definition implements class attributes, methods, and associations. It also supports inheritance, abstraction, and encapsulation which will be explained later.

Class has attributes, in the term VHDL, it is called states. In order to use this property, class must be instantiated. The instantiated class is called object. The object stores the values (states). The class can be instantiated as a signal, variable, and constant. These instantiated type declarations will affect the behaviour of the object especially when invoking the methods. These behaviours are affecting when the signal is changing. Object with signal declaration will be affected after one delta VHDL cycle, object with variable declaration will be affected immediately, and the constant object cannot be changed.
The Objective VHDL separates the implementation of the class into declaration part and body part as we can see in Table 2. The effect of separating implementation will be explained in method, object, and abstraction chapters.

| class_type_declaration ::= [abstract] class [formal_generic_clause] [class_type_declarative_item] end class [class_type_simple_name] |
|---|---|---|---|---|
| Table 3.Class type definition grammar |

In Table 3, the declaration part is almost similar with entity declaration in standard VHDL with additional new keyword such as abstract, class, and end class. The class keyword indicates that anything, declared in the scope between class and end class, is property of the class. The property can be generic variable, class attribute, and public method. Generic variable is used to parameterize the class. The abstract keyword will be explained in abstraction chapter.

<table>
<thead>
<tr>
<th>class_type_declarative_item ::= class_attribute_declaration</th>
<th>class_type_common_declarative_item</th>
<th>class_type_object_configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 4.Class type declarative item grammar</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| class_attribute_declaration ::= class attribute identifier : subtype_indication [:=expression]; |
|---|---|---|
| Table 5.Class attribute declaration grammar |

<table>
<thead>
<tr>
<th>class_type_common_declarative_item ::= type_declaration</th>
<th>subtype_declaration</th>
<th>constant_declaration</th>
<th>subprogram_declaration</th>
<th>alias_declaration</th>
<th>use_clause</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 6.Class type common declarative item grammar</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| type name is class … end class name; |
|---|---|---|
| Table 7.Class declaration keyword |

Table 4, 5, and 6 explain about the extension for standard VHDL. The extension introduces the class attribute declaration grammar. The grammar is used to specify the object’s state. The collection of object’s states is called state space. However class type common declarative item is similar with type common declarative item in standard VHDL. The scope from this declarative is local to the class.

| class_type_body ::= class body {class_body_declarative_item} end class body [class_type_simple_name] |
|---|---|---|---|---|
| Table 8.Class type body grammar |

<table>
<thead>
<tr>
<th>class_body_declarative_item ::= class_attribute_declaration</th>
<th>class_body_common_declarative_item</th>
<th>class_body_object_configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 9.Class body declarative item grammar</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The class type body has also class body declarative item. The class body declarative implements all the methods and derived methods in class type declarative for non abstract class. It declares also the object’s private variable. The differences of the declarations of attribute and method between class type declarative and class type body will be explained in object and method chapter.

Let’s make a class according to Figure 1.

```plaintext
type FIFO is class
  generic
    size: Positive;
    bits: Positive;
  );
subtype Item_t is Integer range 0 to 2**bits-1;
type Buffer_array is array (0 to size -1) of Item_t;
class attribute item: Buffer_array := (others =>0);
class attribute first: Natural range 0 to size -1 := 0;
class attribute nxt: Natural range 0 to size -1 := 0;
class attribute empty: Boolean := true;
constant type: Integer := 1;
function is_full return Boolean;
function is_empty return Boolean;
for variable
  procedure put (val: in Integer);
  procedure get (val: out Integer);
end for;
end class FIFO;

type FIFO is class body
  class attribute index: Natural range 0 to size := 0;
function next_index (index: in Integer) return Integer is
begin
```
... end

function is_full return Boolean is begin ... end

function is_empty return Boolean is begin ... end

for variable procedure put(val: in Integer) is begin ... end

procedure get(val: out Integer) is begin ... end end for

end class body FIFO;

Example 1. Example of class FIFO

We can see in this example 1, class FIFO has two generic values, size, the maximal entries in FIFO, and bits, bit-width of each entry. It has subtype Item_t which is representing the range of the allowed number in the FIFO, and type Buffer_array which is representing maximal number of entries. It declares four class attributes (item, first, nxt, and empty) in class type declaration and one attributes (index) in class body declaration. Attribute item is representing the buffer in FIFO, attribute first is holding the first index of buffer, attribute nxt is holding the next index which is available, and attribute empty indicates if the buffer is empty. The class has four methods in declarative part which are two functions and two procedures and one method in body part. The function is_full is to indicate if the buffer is full or not. The function is_empty is to indicate if the buffer is empty or not. The procedure put implements if new item will be added in buffer. The procedure get implements if the item will be taken from the buffer. The function next_index is to get the next available index. We will explain about the differences between procedure and function later. The repetitions of class name are optional.

This example implements non abstract class. The requirement to be non abstract class is the class must have body part and implements all the methods in class type declaration. The example has body part which has implemented five methods.

3.2. Method

Method is subroutine in a class. It uses to communicate between classes. The primary purpose of the method is to provide mechanism for accessing the private or protected data stored in an object or a class. Method consists of input parameter, and output or return parameter. The good writing method explains about the preconditions are used by this input parameter method, and postconditions after this method has been executed.
Objective VHDL provides two implementations of method as we can see in table 12. These implementations are procedure and function. Procedure consists of input parameters, output parameters, and / or input output parameters. The difference between input and output parameters is input parameter including input output parameter must be initialized, and output must not. Function consists of input parameters, and / or optional return value. The main difference between procedure and function is function does not allow modifying object’s state and procedure does.

```vhdl
type FIFO is class body
  class attribute index: Natural range 0 to size := 0;
  function next_index (index: in Integer) return Integer is
    begin
      if index + 1 < size then return index + 1;
      else return index + 1 - size
    end return
  function is_full return Boolean is
    begin
      return nxt = first and not empty;
    end
  function is_empty return Boolean is
    begin
      return empty;
    end
  for variable
  procedure put(val: in Integer) is
    begin
      assert not is_full report "FIFO overflow" severity failure
      item(nxt) := val;
      nxt := next_index(nxt);
      empty := false;
    end
  procedure get(val: out Integer) is
    begin
      assert not is_empty report "FIFO underflow" severity failure
      val := item(first);
      first := next_index(first);
      empty := first = nxt;
    end
end for
```
In example 2, get and put are declared as procedure because they change the object’s states, and next_index, is_full, and is_empty are declared as function because they do not modify anything in object’s states only read. Function next_index, is_full, and is_empty can be changed to procedure but procedure get and put cannot be changed to function because it is not allowed in VHDL and Objective VHDL. Function must not modify any object’s states.

Objective VHDL divides declaration in two parts. This affects also to the method. If a method is declared in class type declaration and class body declaration, the method is public method, if a method is only declared in class body declaration, the method is private method, and if a method is only declared in class type declaration, the class is abstract. There is no protected method in Objective VHDL.

| class_type_object_configuration  ::=  for object_specification {, object_specification} {class_type_common_declarative_item} end for; |
| object_specification            ::=  signal | variable | constant |

Table 12. Class type object configuration grammar

Objective VHDL provides new mechanism to handle method which accesses to object’s state. It is called class body object configuration. The grammars can be seen in table 4 and 12. This mechanism enables declaration of methods which are specific to class instantiation. The class can be instantiated as signal, variable, or constant. If the class is instantiated as variable, the invoking method will be method with variable as object configuration. Attributes must be accessed like a variable. If it uses signal, attributes must be accessed like a signal. If it uses constant, attributes must be a constant. One method can only have one object configuration but same method can be declared more than one with different object configurations. This feature affects also the semantics for assignment object’s attributes but the attribute’s operators (such as “+”, “*”) are not affected. Signal’s assignment takes one VHDL delta cycle to be effective affected, and variable’s assignment takes immediately. In example 2, all of the procedures uses variable as object configuration. It affects all of the assigned object’s attributes as variable (variable uses := for assignment).

The methods using signal as object configuration need special attention. Signal assignment needs one VHDL delta cycle, the concurrency problem arise when same object are parallel accessed. Objective VHDL provides solution for this problem. We will explain this solution later.

3.3. Objects

Object is instance of a class and is unique. It means object can only be declared once. If other objects want to use it, it must be referenced. Several objects can have the same class. If the objects are equal, it means the objects are similar in class type, and object state. Objective VHDL implements the object as circuit block diagram.

Objects in programming languages usually put in memory (such as stack or heap) or in a file to maintain its states. VHDL including Objective VHDL does not support this memory management, more precise it does not have any stack or heap memory. Because of this limitation, there is no dynamic memory allocation for object’s declaration. All of the objects must be initialized in the first place. As we can see in last example, there are no class constructor and class destructor methods so every object in Objective VHDL is static. The same limitation affects with pointer. Objective VHDL does not support pointer, object reference, and call by reference. To maintain object’s state, Objective VHDL uses the register
to store state space. The problem arises when the same object is run parallel. It needs mechanism to ensure the right states are properly stored. Objective VHDL has proposed concurrent mechanism. It is including guard expression and scheduling policy. We will explain this mechanism later.

Object VHDL has two mechanisms to declare object’s state. First declares object’s state in class type declaration, and the other declares in class body type declaration. The effects of these mechanisms are the former makes object’s state to become protected type, and the latter makes object’s state to become private type. Objective VHDL does not have public object’s state type. These mechanisms affect the inheritance behaviour.

In the example 1, FIFO has five protected object’s state as class attributes and one private object’s state as constant. Object cannot directly change the other object’s state, because there is no public object’s state type. An object uses method to change other object.

```
variable fifo_object: FIFO generic map(size=>8, bits =>3);
```

Or

```
subtype FIFO_8_3 is FIFO generic map(size=>8, bits=>3);
signal fifo_object: FIFO_8_3;
```

---

### Example 3. Example of Declaring Objects

Declaring objects are the same with declaring any data types in VHDL. The first example declares directly the variable object to class FIFO with buffer size is 8 and the value of the item is in the range from 0 to 7. The second uses subtype declaration to make class FIFO with buffer size is 0 and item range is from 0 to 7, then declares the signal object with this new subtype.

When object is declared as variable, all the methods with variable object configuration are available. It cannot access the method with signal or constant as object configuration. The same rules apply with object declared as constant or signal. The methods with no state’s modifications can be put in common declarative. These methods can be accessed by any object independent to object configuration.

#### 3.4. Abstraction

Abstraction is ignoring the details of the class implementation. The purposes of this feature are to hidden the detail implementations from the user, to make generality of the class (template), and to make developer task easier. Objective VHDL support this feature. Class is declared as abstract class if it uses keyword abstract when declaring the class (table 3). If a class is declared as abstract, the implementation in class body is optional. Abstract class may only work with class declarative part.

```
type Buffer_t is abstract class
    generic
        size : Positive; -- Max. no. of entries in buffer
```

Let’s make a abstract class according to Figure 2.
bits : Positive -- No. of bits per entry
);  
subtype Item_t is Integer range 0 to 2**bits – 1;
type Buffer_array is array ( 0 to size -1 ) of Item_t;
class attribute item : Buffer_array := (others => 0);
for variable, signal
    procedure put ( val : in Integer );
    procedure get (val : out Integer );
end for;
end class Buffer_t;

Example 4. Example of abstract class adopted from [8]

In example 4, Buffer_t is declared as abstract. It has two generic values, subtype Item_t, type Buffer_array, class attribute item, and two procedures, put and get, which declared with variable and signal as object configuration. Because this example uses two object configuration types, the derived non abstract class must implement same method with different object configuration type. It will be explained in inheritance chapter.

Buffer_t is an abstract class. Because of this reason, the example does not implement the procedure get and put in class body. Moreover it has no class body part.

3.5. Inheritance

Inheritance means makes new class using existing class. The new class is called derived class or subclass, and the existing class is called base class or superclass. The derived class can be abstract or non abstract class. The reason to make inheritance is create specialization of existing class.

The derived class will have all of the protected variables and public methods in superclass which reside in class declarative part. This condition will give some effect to derived class. If the superclass is abstract class, the derived non abstract class must implement the methods which are not implemented in superclass. If the superclass is non abstract class, the derived non abstract class can optionally implement the methods in superclass. This mechanism is called overriding.

The subclass object can be declared as superclass object but not vice versa. It gives a lot of advantage when doing programming. It can used superclass object when writing some code, and the program will execute the subclass implementation. It gives flexibility to extend the circuit design.

```
derived_class_type_declaration ::= new [abstract] class class_type_name with
[formal_generic_clause]
{class_type_declarative_item}
end class [class_type_simple_name]
```

Table 13. Derived class type definition grammar

```
type D is new class P with ... end class D;
```

Table 14. Derived class declaration keyword

In table 13 and 14, we use keyword new and with to make the derived class. The derived class can be abstract and non abstract. The other parts have the same rule with class type definition including the repetition of derived class name.
We change the previous example to apply inheritance

```pascal
type FIFO is new class Buffer_t with
  class attribute first: Natural range 0 to size -1 := 0;
  class attribute nxt: Natural range 0 to size -1 := 0;
  class attribute empty: Boolean := true;
  constant type: Integer := 1;
  function is_full return Boolean;
  function is_empty return Boolean;
  for variable, signal
    procedure put (val: in Integer);
    procedure get (val: out Integer);
  end for;
end class FIFO;

type FIFO is class body
  class attribute index: Natural range 0 to size := 0;
  function next_index (index: in Integer) return Integer is
  begin
    ...
  end

  function is_full return Boolean is
  begin
    ...
  end

  function is_empty return Boolean is
```
begin
  ...
end

for variable
procedure put(val: in Integer) is
begin
  ...
end

procedure get(val: out Integer) is
begin
  ...
end
end for

for signal
procedure put(val: in Integer) is
begin
  ...
end

procedure get(val: out Integer) is
begin
  ...
end
end for
end class body FIFO;

Example 5. Example of derived class

In Example 5, we modified class FIFO in the example 1 to be derived class. The class FIFO is subclass of class Buffer_t. Because class FIFO is non abstract class, and class Buffer_t is abstract class, class FIFO must implement all of the superclass unimplemented methods. The example has two derived methods, put and get. To implement these methods, they must be declared in derived class declaration part and derived class body part. The superclass has declared two object configurations for put and get methods. The subclass must implement two methods with same name but different object configuration. The generic values (size and bits), subtype Item_t, type Buffer_array, and class attribute item have been erased because they have been declared in superclass Buffer_t.

3.6. Polymorphic Objects

Polymorphism is the ability of objects with different types to respond to method calls of methods with the same name. Object oriented programming encapsulates all methods in a class. Because of this, class can have the same method name, and signature with maybe different implementation. For the inheritance class, it is called override. Polymorphism has a strong relationship with inheritance feature. This ability makes sure that the object calls the right method to be executed. To choose the right method is called binding. The binding can be static or dynamic. Objective VHDL implements this ability with dynamic binding. It means all the class checking and verifying assignment is during running time but checking if the assignment has the same root class is during analysis.
The declaration object with polymorphism ability called class wide type. The polymorphic object can be declared with an abstract class but the assignment to it must with non abstract class with the same root class.

![Diagram of Producer and Consumer](image)

**Figure 4.** Produce and Consume communicate using class wide type

We make two new classes (Produce and Consume). The classes communicate with each other using class wide type with Buffer_t as root class. The implementation for this figure is

```vhdl
Example 6. Using the polymorphic object adopted from [8]

type Producer is class
  port(link: inout Buffer_t'CLASS);
  class attribute token : Integer := 0;
  for variable
    procedure produce;
  end for;
end class Producer;

type Producer is class body
  for variable
    procedure produce is
      begin
        server.put(token);
      end;
    end for;
end class body Producer;

type Consumer is class
  class attribute token : Integer := 0;
  for variable
    procedure consume(signal link: inout Buffer_t'CLASS);
  end for;
end class Consumer;

type Consumer is class body
  for variable
    procedure consume(signal link: inout Buffer_t'CLASS) is
      begin
        server.get(token);
      end;
    end for;
end class body Consumer;

signal fifo_obj: FIFO;
process
  variable producer_obj1: Producer port map(link => fifo_obj);
  variable producer_obj2: Producer port map(link => fifo_obj);
```


variable consumer_obj: Consumer;
begin
...
  producer_obj1.produce;
  producer_obj2.produce;
  consumer_obj.consumer(buffer_obj);
...
end

Example 7. How to do the assignment adopted from [8]

In the example 6, it has two classes (Produce and Consume) which have link signal to communicate as class wide type. Link signal is declared using Buffer_t’CLASS, it means the signal can be assign to any object with Buffer_t as the root class. In the example 7, it declares three objects (producer_obj1, producer_obj2, and consumer_obj). The objects use the link signal to communicate with each other. The link signal assigns to class FIFO. It can be done because the FIFO has Buffer_t as it superclass.

Produce class is called server, and Consume class is called client. Objective VHDL does not allow client to memorize its server by storing a reference. Because of this, each method invocation, all the required server objects for executing method must be passed as parameter as we can see in example 7.

3.7. Concurrency

As we can see in previous chapter, the signal’s assignment arise concurrency problem. To handle this problem, Objective VHDL suggests guard model and scheduling policies.

- Modelling of guard expression
  As we know, the difference between variable and signal are variable can be declared and used in sequential parts of VHDL or Objective VHDL model, and signals can be used concurrently. Because of this, the guard modelling will be integrated only in method implementation for signal. The implementation of guard modelling is

```vhdl
procedure guard(expression: Boolean)

Table 15. Guard implementation model
```

The procedure is called as first statement in a method. If the boolean expression is true, then this method can calling the guarding method, otherwise it quits and requeues the service request.

- Scheduling policy
  Objective VHDL provides scheduling mechanism in library SCHEDULERS. The library implements static priority scheduling, round robin scheduling, enhanced round robin scheduling, equal priority scheduling, and first come first serve scheduling.

```vhdl
for signal
  procedure put(val: in Integer) is
    variable object_copy: FIFO;
  begin
    guard(not is_full);
    object_copy := this;
    object_copy.put(val);
```
this <= object_copy;
end

procedure get(val: out Integer) is
  variable object_copy: FIFO;
begin
  guard(not is_empty);
  object_copy:=this;
  object_copy.get(val);
  this <= object_copy;
end;
end for;

Example 8. Implementation method with signal as object configuration and guard expression

In example 8, it declares the guard procedure inside put and get procedures. First, it checks if the guard expression returns true or not. If it is true then it assigns the variable temporary FIFO to store the current value. Later, this temporary will be assigned to the object with keyword this.

signal fifo_obj: FIFO;
use SCHEDULERS.DECLARATIONS.SCHEDULING;
attribute SCHEDULING of fifo_obj : signal is “Round_Robin”;

signal producer_obj1: Producer port map(link => fifo_obj);
signal producer_obj2: Producer port map(link => fifo_obj);
signal consumer_obj: Consumer;
...
  -- in process 1
  producer_obj1.produce;
  -- in process 2
  producer_obj2.produce;
  -- in process 3
  consumer_obj.consumer(buffer_obj);
...

Example 9. Using scheduling

We modified the fifo_obj into signal with round robin scheduling and each object is running inside different processes. Because objects are running concurrently, it must provide the mechanism which object will be granted to execute the method. It is done by assigning fifo_obj with Round_Robin scheduling

3.8. The tools

The tools are available from LEDA S.A., it is based on LEDA VHDL System. The LVS is VHDL compilation and library management system. It can be used with Objective VHDL. First the Objective VHDL is analyzed by LVS analyser. It performs lexical, syntactic, and semantic analysis then puts the compiled to VHDL library. The library format represents the abstract syntax tree of the Objective VHDL source code [9] is called Object Oriented VHDL Intermediate Format (OO-VIF). Library contents can be accessed through LPI (LEDA Procedural Interface) [10]. The LPI can read, augment, and modify model information. Objective VHDL Translator System is using these capabilities to transform all object oriented construct to VHDL Intermediate Format (VIF).
The compiled code in VHDL library can be transformed back into VHDL source code by LVS. The source code is imported by VHDL simulation environments. In these environments we can find some logical errors and fix it in Objective VHDL source code. After all VHDL code has been generated after synthesis, we proceed with synthesis tools. Below is the architecture from the tools.

![Objective VHDL tool architecture adopter from [8]](image)

**4. Other Object Oriented VHDL**

As we mentioned in earlier chapter, we have three Object Oriented VHDL. There are Ashenden’s SUAVE [4], Schumacher’s OO-VHDL [5], and Objective VHDL [6]. We explain briefly only SUAVE and the comparison with Objective VHDL.

**4.1. What is SUAVE**

Suave is proposed by Peter J. Ashenden. Suave is extension of VHDL based on ADA 95 features. The extended improves support for modelling from system level to gate level. The objectives SUAVE are [11]

- Improving support for high level behavioural modelling by improving encapsulation and information hiding capabilities and providing hierarchies of abstraction.
- Improving support for re-use and incremental development by allowing further delaying of binding through type-genericity and dynamic polymorphism.
- Providing a more abstract form of communication than the existing mechanism of signals and signal assignment
- Providing dynamic process creation and termination
- Preserving capabilities for synthesis and other form of design analysis
- Providing abstraction that are not biased towards hardware or software implementations, allowing subsequent partitioning and refinement (hardware / software co-design)
- Supporting hardware / software co-design through improved integration with programming languages
• Supporting refinement of models through elaboration of components rather than through repartitioning
• Preserving correctness of existing models within the extended language

The design principles of SUAVE are upward compatibility, preserve strong typing, separate declaration and functionality, unification of timing semantics, preserve determinism, generality, scope of VHDL, intermixed abstraction level, and concurrency, preserve and improve consistency and portability, no application specific packages, minimize implementation impact, and maximize implementation efficiency.

4.2. **Comparison with SUAVE**

These are several differences of two languages [12]

1. **Class interface definition**
   • Method definition: The method is declared inside class type definition in Objective VHDL. It approach to group all data and methods in the class construct which can be seen as definition of an abstract data type. It provides methods implicit access to attribute. SUAVE uses declared external operations to the type declaration. It requires the explicit declaration of at least subprogram parameter of the tagged record type in order to identify the subprogram as an operation of that class, and in order to give it access to attribute. In this case, Objective VHDL gives better code organization.
   • Non-invocable (abstract) methods: Abstract method must be explicitly declared abstract in SUAVE. It gives immediately detection of the omission of an implementation of a non-abstract method in an abstract class. Method is implicitly declared as abstract in Objective VHDL if no implementation for this method. With this mechanism, the error can be caught when a non abstract class is derived

2. **Encapsulation and visibility control**
   • Encapsulation of attributes: The attributes in Objective VHDL are protected or private type, but in SUAVE are public or private type.

3. **Instantiation**
   • Instantiation of signal object: Both languages permit signal instantiation but with restriction. SUAVE requires declaring access explicitly, and Objective VHDL uses implicit handling as in VHDL.
   • Parameter passing: SUAVE passes the parameter by reference. It gives limitation for parameter passing by value due to harder view conversions. Objective VHDL does not mention about parameter passing.
   • Initialization: Objective VHDL allows initial values of attributes but SUAVE does not.
   • Assignment and controlling copying: SUAVE can prevent predefinition of assignment and equality operations, and Objective VHDL cannot prevent this predefinition.

4. **Method invocation**
   • Methods designated by identifier: SUAVE uses explicit parameter passing, and Objective VHDL uses prefix notation.
   • Method designated by operator symbols: SUAVE supports infix operator as standard VHDL but Objective VHDL does not.

5. **Polymorphism**
   • Objects of class hierarchy types: Both languages handle signal and constant similarly. The different is how to handle the variable. Objective VHDL allows
variables to be of class wide type and can take on values of different specific
types during their lifetime, and SUAVE provides class wide type via access
value.
• Semantics for signals of class hierarchy types: Objective VHDL uses VHDL
mechanism to resolved class wide signals, and SUAVE does not support this.

5. Summary

Objective VHDL provides object oriented paradigm such as class, object, method,
inheritance, encapsulation, and abstraction. It implements mechanism to deal with object
state, state transition, object lifetime, communication between objects, and request arbitration.

The Objective VHDL supports concurrency. The concurrency is using guard expression
and scheduling policy. There are many implementations of scheduling policy in Objective
VHDL such as static priority, round robin, and enchanted round robin. We can also program
our scheduling policy.

The main advantage and disadvantage of Objective VHDL are the supporting tools. The
tools support ranges are from compiling Objective VHDL source code into the library until
the synthesis of the code and ready to transfer it in hardware level. The tools are also the
limitation of this extension. Because objective VHDL must compatible with the existing
synthesis tools in order to make the code synthesized, it gives to the programmer some
restriction when using this extension.

In the future, we hope for better synthesis tools. Like happened in standard VHDL, the
limitations are arisen from the synthesis side.

6. Reference

VHDL Forum for CAD in Europe (VFE, Spring Conference), 1992.
[2] W. Glunz. Extensions from VHDL to VHDL++. JESSI-AC8 report S2-SP1-T2.4-
Oriented VHDL. Proc. VHDL Int’l Users’Forum (VIUF, Fall Conference), 1997.
the WWW from URL http://eis.informatik.uni-
oldenburg.de/research/request.html.
Design Languages (FDL), 1998.
Extensions to VHDL for High-Level Modeling. Proc. Forum on Design
Languages (FDL), 1998.
DASC Object Oriented Study Grup, 1998