

Object-Orientation Applied to VHDL Descriptions

David Cabanis, Prof Sa'ad Medhat
School of Electronics
Bournemouth University
5125BB, POOLE, UK
dcabanis@bmth.ac.uk
Nick Weavers
IBM UK Havant

Abstract

This paper describes a research project, carried out by Bournemouth University in collaboration with IBM Havant UK, for a possible extension to VHDL.

This paper demonstrates how an object-oriented approach applied to VHDL will increase the designer's productivity using the three concepts of objects, classes and methods.

Finally, a proposed object oriented extension to VHDL will be examined, which can handle Encapsulation, Abstraction and Derivation.

Section 1. Introduction

VLSI designs are becoming increasingly complex as technology advances. This design complexity is closely related to factors such as the number of different elements and the interaction between them, together with different user requirements. Therefore the uptake of object-oriented design techniques are envisaged as they represent an efficient solution for the building of complex systems.

Medium and large VHDL designs are implemented using conventional analysis and design techniques such as Functional decomposition[1] and SA/SD (System Analysis and System Design) [2]. The Functional Decomposition approach for VLSI designs enables fast representation of user requirements using a systematic approach, whilst SA/SD

features a coherent means of describing the data and processing parts of a system. However, both techniques fail to address the reusability and maintainability issues in the design process. These analysis methods focus on the functionality of a system at the expense of its data consistency.

The proposed solution outlined in this paper, is an improvement on the existing analysis and design methodology, as it highlights the need for design evolution. It aims to facilitate reusability, maintainability and allows higher description levels of VHDL designs.

Object-orientation concentrates on data rather than functions, since general data is less likely to change during the design process, and is more stable than functions. Object-orientation has recently had a large impact in the software design community. Languages such as C++[3], Eiffel[4], SMALLTALK[5], brought many advantages such as object modelling of the system, modularity, reusability, and extendibility of the code. There is strong evidence that this has led to a better productivity from designers and higher design reliability [6].

This research examines existing analysis and design techniques in an attempt to demonstrate the benefit of an object-oriented design methodology over conventional methods when applied to hardware design.

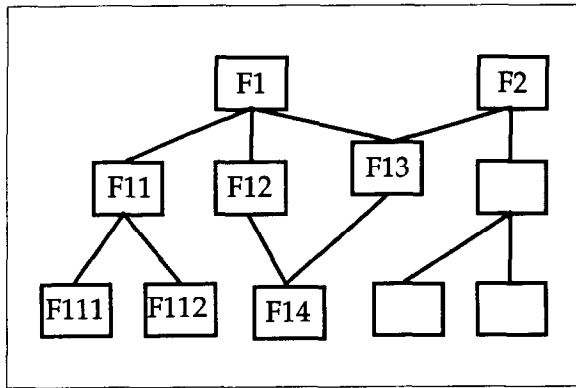


Figure 1. Functional Decomposition.

A study of VHDL's suitability for object oriented descriptions is described and conclusions are drawn on features required in an object-oriented version of VHDL. Examples of the proposed enhanced semantics are given. Some fundamental limitations are also described.

Section 2. Rationale

The use of analysis and design methods is often motivated by the designers need to manage design complexity and to keep the development and maintenance costs down. Numerous analysis and design methodologies have been proposed since the 70's, contributing to a better definition of design processes. This is in addition to an attempt to standardise the system representations used by designers. In the electronic engineering world, two main methods are successfully used: Functional Decomposition and System Based Analysis.

Functional decomposition treats one function at a time, and hierarchically decomposes it into a set of sub-functions. This process continues until the decomposition level is low enough to be implemented by a set of simple functions as shown in figure 1. This method is based on the principle that a problem can be divided into smaller manageable problems to reduce the complexity of the design. Consequently, the functional decomposition is seen as a simple approach towards achieving a sensible solution. Nevertheless, with this approach the analysis effort is mainly based on creating design functions at the expense of neglecting data consistency. Furthermore, function decomposition rules are not clearly stated and

therefore the decomposition hierarchy of a system will vary from one designer to another. Finally, the non-hierarchical interconnections encountered in the design of complex systems (between different taxonomies or within the different levels of the same tree) are difficult to represent with the functional decomposition paradigm.

With system based analysis (inspired from the system theory by [7]) the system is seen as a complex and active object. The structural and functional aims of such an object are described at the System Analysis stage. The modelling of a system is therefore represented by both Data models and Process models. Design rules are also provided for a better data consistency and process description. This approach appears to be an improvement over the functional decomposition method. However, in this paradigm the data and processes description techniques are remotely related.

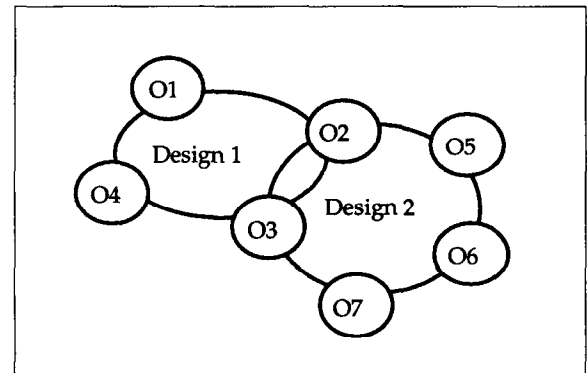


Figure 2. Object Decomposition.

The arguments for the use of an improved analysis and design technique are based upon the fact that it provides an easier development of reusable VHDL units (genericity, tailorability), and a higher level of modelling, together with a better object abstraction. It also provides a mechanism for encapsulation. These improvements aim to overcome the complexity barrier of VLSI designs, thereby, achieving an enhancement in productivity and reliability. The main contribution of the object-orientation is, in its ability to tie each function to a set of operations related to the system's objects. This provides a better understanding of the object semantics and allows modularity and reusability (see figure 2) of components throughout the system design process. When compared to

previous analysis techniques, object-orientation overcomes the identified limitations by providing consistency in data and process descriptions. It also enables identification and construction of stable system elements. Although this method requires a greater effort to obtain the design abstraction compared to the functional decomposition approach, it has many advantages. These include an integrated description of data and processes and the ability to model complex objects and their encapsulation features.

Section 3. Object-Orientation Concepts

Object-orientation addresses common aspects such as *abstraction*, *encapsulation*, *inheritance* [8] and *reuse*. The abstraction is the principle of ignoring those aspects of the subject that are not relevant to the current purpose, in order to concentrate on those that are. Abstraction is applied to procedures and data. With the procedural abstraction, any operation that has a well defined purpose with a mainstream effect can be treated by its users as a single entity. This is despite the fact that the operation might be achieved by a number of lower level operations. The data abstraction is based upon the action of defining a data type in terms of the operations that apply to an object of that type. The value of an object can only be modified through the use of dedicated operations.

Encapsulation is used when developing an overall program structure. Each component of a program should encapsulate or hide a single design decision. The interface to each module is defined in such a way as to reveal as little as possible about its inner workings.

One of the most promising concepts of object-orientation is Inheritance. Inheritance is based on the ability to design new functions from an existing one by reusing similarities amongst classes. Inheritance portrays generalisation and specialisation, making common attributes and services explicit within the class hierarchy. As projects evolve, there is no need to change existing classes. New classes are derived from the older ones and therefore will not disturb those parts of the description that use the parent classes.

Reusability as defined in object-oriented programming languages is targeted toward delta coding. The basic idea is that no two different designs will ever need exactly the same components. Therefore, if the exact abstraction required is not in the program library, inheritance will allow the designer to choose one that is close enough to what the design requires, and tailor it. The designer is only required to modify the behaviour provided by the ancestor abstraction to achieve the one that is needed. This is an efficient means of rapidly developing new components tailored to specific needs from existing ones.

Section 4. Is VHDL Object-Oriented ?

Many benefits of object-oriented technology come from abstraction and encapsulation. VHDL features the 'package', a very powerful encapsulation mechanism. Packages can be used for many purposes, including (but not limited to) the building of abstract data types. A package allows the designer to group types (simple types, records and arrays), functions and procedures that are logically linked. Procedures can have input/output parameters whereas functions will have at least one return parameter. A type can be derived from an other type by adding constraints allowing a very limited inheritance mechanism. However, abstract data types are not sufficient to make object orientation effective. In large projects containing many objects, organisation and factoring of the common property is needed. Hence, packages are not adapted to the particular needs of building classes when using a classification orientation.

Like most of the modern hardware description languages, VHDL is based on a structured programming technique. Structured programming was designed to organise complex programs more efficiently by linking the processing functions to the data structures. It has limitations when the data structure and processes of the program need to be reused in a new description. Object oriented programming differs from the procedural approach by considering objects as active entities that return values in response to messages. In addition, the

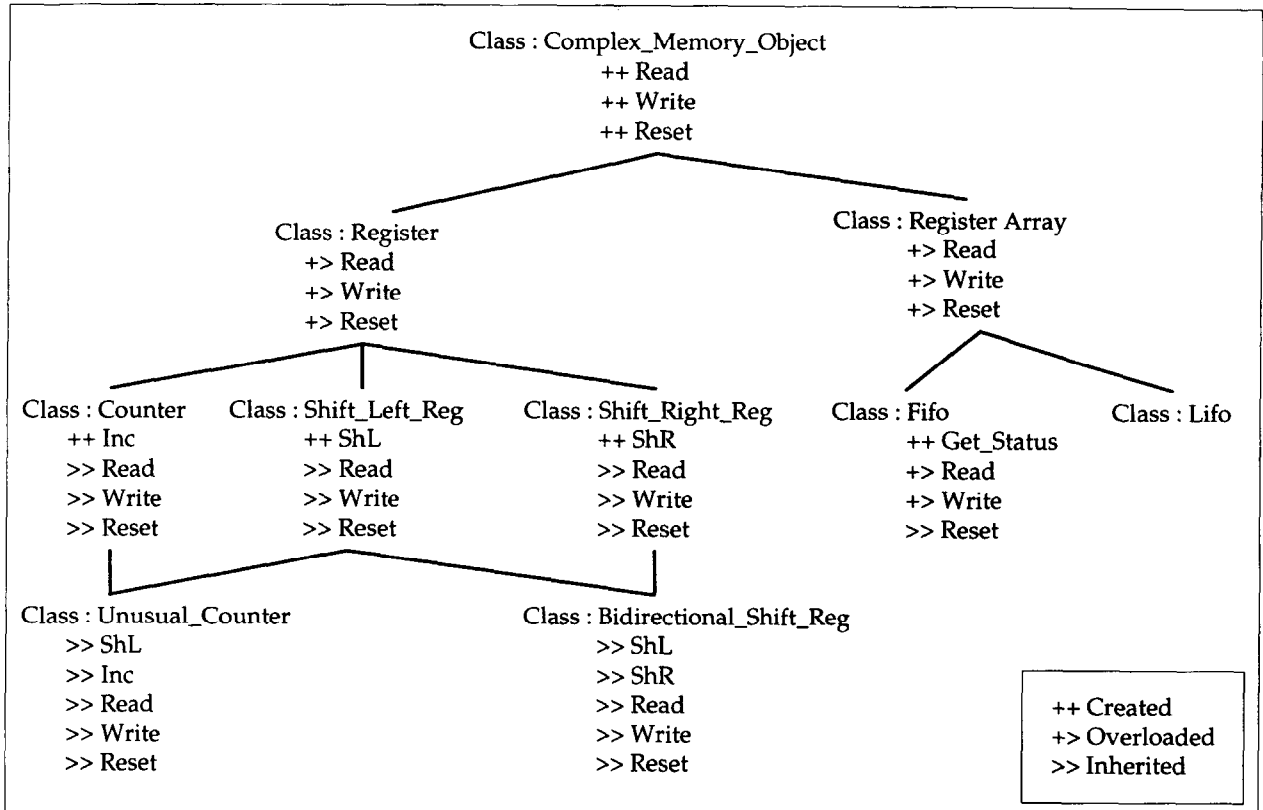


Figure 3. Hardware Classification Example.

data and functions in object-oriented programs are organised within the same entity. This is different from conventional programming methodologies where the data and functions exist in separate structures. The internal data structure of object-oriented entities is hidden by a function access mechanism. This key feature is absent from the VHDL language. VHDL does not have the necessary syntax and mechanisms to implement object-orientation, and as a result it needs to be extended.

Section 5. Object-Oriented Extensions to VHDL

The definition of an object-oriented extension to VHDL is based on the study of existing object-oriented programming languages (OOPL's) and object-oriented extensions to procedural languages. OOPL's (e.g. SMALLTALK[5], MODULA2, SIMULA[9]) have been used in the implementation of Hardware Description Languages such as ODICE[10], LOGLAND[11], OODE[12] and SDL[13].

Decisions regarding encapsulation and inheritance are essential factors in determining the implementation process of an OOPL. A simple and natural approach for implementing the object oriented extension to VHDL is envisaged. This is to ensure that the designer's learning curve is kept to a minimum, and the psychological barrier for learning yet another complex language is reduced. An important issue to raise here is, how could these attractive features be applied to VHDL?

Objects are generally thought of as being instances of classes. An obvious means of organisation is the classification orientation[8]. Using classification, objects are organised into a hierarchy of classes. General classes contain features common to many sub-classes. Specialised sub-classes implement only the behaviour that are common to a subset of the parents' class objects. The classification organisation is akin to the classification of biological species. For example, in figure 3 the use of classification applied to a hardware design is demonstrated. A class Complex_Memory_Object would gather the

properties of a complex memory object, and a class `Shift_Right_Register` derived from `Complex_Memory_Object` will contain only those aspects that are peculiar to a Shift Right Register. Since a shift right register is a complex memory element, it must still have the properties of the `Complex_Memory_Object`.

```

class register is
  private constant BitsNbr : integer := 8;
  signal content: bit_vector(0 to BitsNbr -1);
begin
  public method register() is
  begin
    this.reset();
  end method register;
  public method reset() is
  begin
    this.write(others => '0');
  end method reset;
  public method read(out data : bit_vector) is
  begin
    data <= content;
  end method read;
  private method write(in data : bit_vector) is
  begin
    content <= data;
  end method write;
  public method write(in data : bit_vector;
                      in clk : bit) is
  begin
    wait until (clk'event and clk = '1')
    this.write(data);
  end method write;
end class register;

```

Listing 1. Encapsulation Process.

To implement object-orientation, a few keywords have been added to the VHDL semantics. The encapsulation process is demonstrated in listing 1. The *class* declaration corresponds to the new abstract data type's features. A class encapsulates its own data, types and sub-routines within the declarative part. These are global to all instances of that class. The declaration takes place inside the class statement part.

A hierarchy of classes is supported and is declared as part of the class declaration statement. The metaclass mechanism is not supported since these proposed object-oriented extensions are not intended to make VHDL into

a true object-oriented language (e.g. Smalltalk where all classes are seen as objects).

```

entity example is
port (...);
class register is
-- see Listing 1
end class register;
architecture example1 of example is
  signal reg1 : register();
  signal reg2 : register();
begin
  process(...)
  method reset(in clk : bit) of reg2 is
  begin
    wait until clk'event and clk = '1';
    this.reset();
  end method reset;
begin
  reg1.reset(); -- Normal behaviour
  reg2.reset(master_clock); -- Overloaded behaviour
end process;
end example1;

```

Listing 2. Objects and Methods Usage.

The encapsulation process is controlled via the use of three keywords : *private*, *public* and *restricted*. Private class members are members for which access is guaranteed only to the class member's functions. Public members are accessible by client classes and, restricted members are accessible to children classes. In listing 1, it is shown that methods such as `register`, `reset`, `read`, and `write` are declared as public. Thus they are accessible by any clients of the object `register`. Another method `write`, overloads the public method `write` and, is declared as a private. As a result, the overloaded `write` method will only be accessible by the functions of the class `register`. To favour encapsulation, methods and data within a class are declared by default as restricted.

The use of objects and methods are shown in listing 2. An object is defined as an instance of a given class. Object are instantiated by a constructor function (with same name as the object's class) and can be overloaded. The choice of the constructor is made according to the type of the arguments. As the purpose of an object's constructor is to merely instantiate an object, it is not considered as a message call. A *Method* in the

object oriented extension is an operation specified at the class level. The method characterises an abstract data type and is defined as the action carried out on an object. Syntactically, a method has a signature of the style: Object.Method (Argument1,...). With an object-oriented approach, a distinction is made between the receiving object and the message sent to it. Inherited methods are often required to be redefined in order to implement a different behaviour. This makes use of the local properties of the sub-class. The process of redefining the behaviour of an inherited method is called method overloading. When called to act upon an object, the method will execute the appropriate behaviour according to the object definition. This implements a polymorphic behaviour. The keyword *this* has been added to the language semantics, in order to refer to the object of the currently executing method.

```

class shift_right_register use public register is
  private constant BitsNbr : integer := 8;
  signal content: bit_vector(0 to BitsNbr -1);
begin
  public method shift_right_register() is
  begin
    this.reset();
  end method shift_right_register;
  public method shr(in clk : bit) is
  begin
    wait until (clk'event and clk = '1')
    this.write('0' & content(1 to BitsNbr -1));
  end method shr;
end class shift_right_register;

```

Listing 3. Inheritance Mechanisim.

The classification and inheritance process is featured in the listing 3. The meaning of the use statement is expanded to select one or many classes to be super classes of the new class. The inheritance process allows the designer to define the type of inherited properties to be used when declaring the super class(es): *private*, *restricted* or *public*. Single Inheritance is often argued to have the same functionality as Multiple Inheritance[14], through adding extra properties or creating extra classes. However, Multiple Inheritance has many advantages. It allows a better organisation of objects by avoiding redundancies in

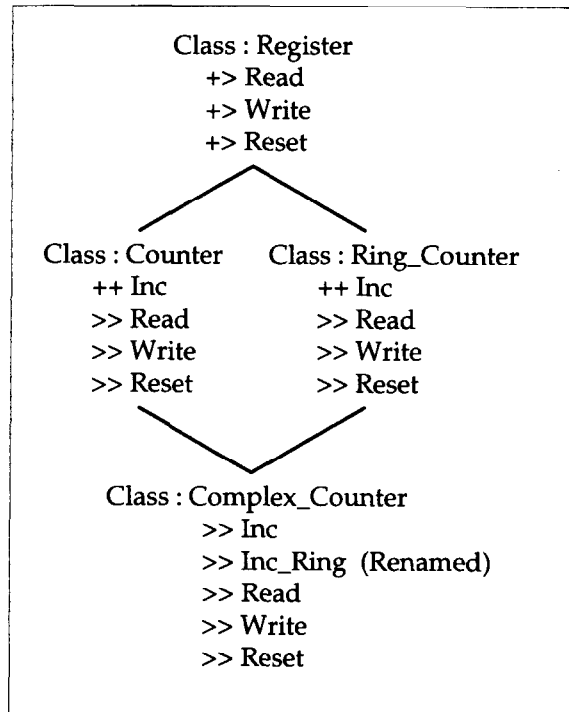


Figure 4. Naming Conflict.

```

class ComplexCnt use Counter,
                  RingCounter is
  method map (Inc_Ring <= Inc of
              RingCounter); -- renaming
begin
  public method ComplexCnt() is
  begin
    (...)
  end method ComplexCnt;
end class ComplexCnt;

```

Listing 4. Method Mapping Construct.

declarations of attributes and methods. However, its use requires prudence due to possible conflicts of method names. In order to avoid potential name clashes, a method mapping construct is provided. As with a port map declaration the method mapping allows the designer to rename methods names to be used within the newly derived classes as shown in figure 4 and listing 4. In figure 3 an example of multiple inheritance is also given. The Bi-directional_Shift_Register class is derived from the Shift_Right_Register class and the Shift_Left_Register class. It can be observed that, the design of the bi-directional shift register, is entirely based on reused code.

Section 6. Conclusions

Whenever a new approach becomes fashionable, the tendency is to prefer it to existing ones. It would not be appropriate to systematically prefer a bottom up data driven method, to a top down function based one. The function decomposition allows designers to rapidly model users' requirements while an object oriented approach enables a better integration of many user needs. In this paper, the benefits of object-orientation applied to hardware design through the use of encapsulation, abstraction, derivation and reuse have been demonstrated. A classification based object-oriented extensions to VHDL has been presented and its use demonstrated. The use of object-oriented extensions to VHDL when combined to object-oriented analysis has been shown to represent a consistent design approach. It provides higher degrees of modularity and extendibility and subsequently could lead to a better design productivity being achieved.

The object oriented extension to VHDL is still at its development stage, further work aims to extend its functionality. Generic classes for container objects and pre/post execution conditions to facilitate development and reliability of the designs, are areas which are currently being examined for further research.

Dynamic allocation and dislocation of object is not currently supported although it was suggested that the use of cached logic FPGAs[] as a target technology would allow dynamic reallocation. The next step in the course of development is the production of a pre-processor for compilation of object-oriented VHDL code to 1076-1993 VHDL.

It is intended to produce a comprehensive hierarchy of classes that are reusable through the inheritance process. The extension of VHDL to cover hardware/software co-design is also under investigation and this may lead to further research subjects.

References

- [1] Dahl O., Dijkstra E. 'Structured programming' Academic Press, London, 1972.
- [2] Yourdon E., 'Modern Structured Analysis' Yourdon Press, Prentice-Hall, 1989.
- [3] Stroustrup B. 'The C++ Programming Language', Addison-Wesley, Reading Massachusetts, 1986.
- [4] Meyer B., 'Eiffel The Language' Prentice-Hall, London, 1992.
- [5] Goldberg A., Robson D., 'Smalltalk-80 : The language and its Implementation' Addison Wesley, Reading, Massachusetts, 1989.
- [6] Anderson B. and Gossain S., 'Hierarchy Evolution and the Software Life Cycle' Object-Oriented Software.
- [7] Bertalanffy, L von 'General System Theory, Foundation, Development, Application' G. Braziller, New York 1968.
- [8] Khoshafian S., 'Abnous R., 'Object Orientation : Concepts, Languages, Datagases, User Interfaces' Wiley, New York, 1989.
- [9] Dahl O. J., Nygaard K., 'Simula - An Algol Based Simulation Language', Communication of the ACM Vol9, n 9, 1966, pp 671-678.
- [10] Wolfgang Muller, 'ODICE : Object oriented hardware descriptions in CAD environment', CHDL and their applications IFIP 1990.
- [11] Adam Pawlak, 'Modern object oriented programming language as a HDL', CHDL and their applications IFIP 1987.
- [12] Akikazu Takeuchi, 'Object Oriented description environment for computer hardware', CHDL and their applications 1981.
- [13] Wolfgang Glunz, 'Integrating SDL and VHDL for system level hardware design', CHDL and their applications IFIP 1993.
- [14] Armstrong A., Mitchell R., 'Uses and abuses of inheritance' Software Engineering Journal January 1994.