

# Classification-Orientation for VHDL : A Specification

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

## Abstract

As hardware design increases in complexity, so does the development process. To this effect, it is envisaged that the use of analysis techniques focusing on design abstraction and reusability can be advantageous in tackling the issues that conventional analysis and design techniques failed to address. Object-Oriented design methodology has recently had a considerable impact in the software design community as it is tightly coupled with the handling of complex systems. Object-Oriented concentrates on data rather than functions since, throughout the design process, data are more stable than functions. Methodologies for both hardware and software have been introduced through the application of HDLs to hardware design. Common design constructs and principles that have proved successful in software language development should therefore be considered in order to assess their suitability for HDLs based designs.

In this paper we will propose the specifications by Bournemouth University and IBM, UK for an alternative Object-Oriented extension to VHDL: The Classification-Oriented. Related work will be reviewed, and the motivations for the proposed implementation will be introduced. A detailed Backus-Naur form of the proposed new grammar will be presented. An example of the design of an FIR filter will be used to illustrate the modelling capability achieved through the use of the Object-Oriented extensions to VHDL. The concept of the object model will be discussed and conclusions will be drawn with regards to simulation and synthesis issues.

## Introduction

The idea of an Object-Oriented version of VHDL is attracting a growing interest within the design community[4][5][6][7][9]. Object orientation is not a set paradigm but rather a particular approach to the design methodology. To some extent, Object-Oriented only defines critical issues found in the process of building large designs and suggest means for tackling them[3]. Three main techniques have been identified to comply with the requirements of the Object-Oriented approach: Encapsulation, polymorphism and inheritance. When used together on an object based design they provide an increased level of maintainability, reusability and abstraction in comparison with previous design methodologies. Nevertheless, although the main guide lines for the definition of an Object-Oriented system are drawn, actual implementations of such a paradigm into a programming language is left undefined. As the appeal for an Object-Oriented VHDL evolves, different implementation of possible Object-Oriented extensions have been proposed[10][15][16][17][19][20].

Classification-Oriented is presented as an alternative to composition orientation or entity object orientation. Classification-Oriented provides a natural way of decomposing systems and therefore is easy to relate to by hardware designers. Within the design, objects are organised around a new abstract data type that favours encapsulation and therefore abstraction of design parts. Definitions of polymorphic functions are supported within the proposed enhanced abstract data type, bringing extra capabilities to VHDL descriptions. Design reuse is achieved by a controlled inheritance mechanism by selecting appropriate attributes of existing designs to implement new ones.

The proposed paper will be organised around five main sections. The first section will briefly address onto the principal features of Object-Orientation concept and highlight the two tendencies for implementing an Object-Oriented extension to VHDL. The second section, will identify the capabilities of abstract data type of VHDL and will propose an enhanced abstract data type model. The third section will deal with the implementation's of mechanisms (inheritance and encapsulation) to provide better design reuse. The concurrent and sequential behaviour of objects will be presented in the fourth section along with our proposed object communication model. The final section of the paper will examine issues regarding the implementation of the Object-Oriented model for simulation and synthesis.

## Section 1. Rationale

Hardware design closely relates to Object-Oriented systems. A set of self contained objects exchanging messages and executing in a concurrent manner can be seen as both the physical implementation of hardware designs and an Object-Oriented system. Although the terminology: Object-Orientation is well accepted, its actual implementation is not part of a set definition, only the main principles are defined. In the proposed extension three basic capabilities for implementing Object-Orientation in an hardware design are identified. Those three capabilities are: **encapsulation**, **polymorphism** and **derivation**. Those techniques when used, provide design robustness, reusability and abstraction.

To that extent, Object Oriented extensions to VHDL have recently generated an significant interest. Two main approaches have resulted from this primary study.

The first approach is based on the composition orientation paradigm [20][10]. Composition orientation is a technique inspired from the language ADA 95 (OO Extension of ADA) [11]. In this extension, inheritance is seen as a sub-typing operation through the use of tagged types. The abstract data type is defined through the use of package header and package body. Instance variables in the package are implemented through the use of records (tagged for inheritance) and procedures are used to implement the behavioural part of the abstract data type. The second approach to the OO extension to VHDL is based on the concept of the component object[19]. The component object represents an extra abstract data type added to the language in order to overcome the limitations of packages. Packages in the standard VHDL represent the only means of encapsulation and abstract data typing. Nevertheless the use of packages is somewhat restricted for creating abstract data types as the data is accessible without having to refer to access functions contained in the package. With the component object paradigm, the designer can build abstract data types and instantiate objects of that type within the VHDL code. The abstract data type is defined as an entity object and architecture pair. The entity object will declare the interface of the abstract data type, i.e. the operations, while the architecture will declare the behaviour and attributes.

Both solutions represent attractive alternatives to the Object-Oriented extension for VHDL and they can be seen as complementary. Large and complex VHDL descriptions will be simplified as new designs inherit the characteristics and functionality of pre-defined available descriptions. Higher levels of abstraction are reached through the implementation of better encapsulation mechanisms. However, the proposed extension here is based on the classification orientation paradigm. Classification orientation is found in languages such as C++, Smalltalk and Eiffel [1][2]. It provides a natural process of decomposition for hardware designers to relate to. Classes also allow encapsulation controls over data structure and behaviours to be expressed. This insures a better code maintainability together with a better data abstraction. In addition to the classification orientation, powerful encapsulation mechanism can also be used. this encapsulation mechanism will take action during the inheritance process as well as during the object access.

## Section 2. Abstract Data Type Issues

As in ADA, abstract data types are implemented in VHDL via the use of packages. Packages allow the designer to group types (simple types , records or arrays) and allow subprograms to be logically linked. A simple encapsulation scheme is enabled through the use of the package separation into a header and its corresponding body. However, as is illustrated in the composition-orientation and the entity object approach, abstract data types of that kind are not sufficient to make Object-Orientation effective. In large

projects containing many objects, organisation and factoring of the common properties is needed. Hence, packages are not adapted to the particular needs of building objects (instances of data types) nor in the organisation of designs when using classification orientation.

A 'class', in the proposed extension, is an abstract data type that defines the type and behaviour of common objects. Syntactically, a class is defined by a class name and two main divisions. The first division contains the 'class header' and the 'class declarative part', the second division contains the 'class statement part'. These two sub-divisions are implemented to introduce a new abstract data type with respect to the VHDL philosophy: Declarations are kept separated from the actual implementation.

```
class_declaration ::=
    class identifier [ use class_list ] is
        class_header
        class_declarative_part
    begin
        method_specification
    end [ class ] [ class_simple_name ];
```

```
class_list ::= [ encapsulation_kind ] class_name { , [ encapsulation_kind ] class_name }
```

Example: A class declaration with a declarative part and a method specification.

```
class decoder is
    type opcode is (add, sub, mult, fetch);
    data : opcode;
begin
    method read_inst return opcode;

end decoder;
```

A class might also be split into a class header and a class body. An obvious application of this feature is when a class is declared within a package. The designer is given the freedom to declare the declarative part of a class in the package header. The body of the class will be defined in the package body. The package will then represent an abstraction of the class. In allowing the definition of the class behaviour in the package body, we are shortening any re-analysis process in the eventuality of alterations to the class behaviour.

Generic clauses and methods mapping are defined in the class header. Generics are used to parameterise classes, whereas method mapping constructs are declared to avoid method names clashes when using multiple inheritance.

```
class_header ::=
    [generic_clause]
    [method_map_aspect]
```

Example: A class declaration with a generic declaration and a method specification.

```
class reg is
    generic (size: integer := 8);
begin
    method write(data_in : bit_vector((size-1) downto 0));
end reg;
```

Example: A class declaration with multiple inheritance and method map.

```
class UpDownCounter use UpCounter, DownCounter is
    generic (delay: time : 3 ns);
    method map (Count of DownCounter => Countdown);
begin
end UpDownCounter;
```

The class can encapsulate its own data, types and procedures within the class declarative part. These are global to all instances of that class and accessible by derived classes. The declarative part of a class can also host object instantiations, enabling complex objects to be built. The declaration of methods takes place inside the class statement part.

Finally, a class will be defined at three levels: package, entity and architecture, providing a maximum scope of usability throughout the different design units.

The use of 'class' as a base abstract data type will enable VHDL designs to reach higher levels of description. The structure of a class will provide strong foundations for the inheritance and encapsulation process to take place.

### Section 3. Inheritance and Data Encapsulation Mechanism

The encapsulation of data and their associated operations, applies the abstraction principle : An object is only accessible through visible operations and its implementation is hidden. Therefore, the data modifications remain local within the objects and have no effect on the functions using them. Control over the accessibility of attributes and behaviour of a class in our proposed extension is ruled by a three level encapsulation mechanism. The three levels are **private**, **public** and **restricted**. Private class members are members for which access is guaranteed only to the class members functions. Public members are accessible by client classes and restricted members are private members accessible to children classes. By default, class objects, data and methods are restricted to favour full encapsulation (See Figure 1).

This control over accessibility will also be used throughout the inheritance process allowing a selective derivation process. Derivation also called 'inheritance', is defined as: The process of using one class as the basis of an other class. There are two types of inheritance that can be identified: simple inheritance (single parent) and multiple inheritance (multiple parents). The proposed extension supports those two types of inheritance and the 'use' statement is expanded to select the inherited classes. The inheritance mechanism allows the designer to define the type of inherited properties : private, restricted and public in addition to define the ordering in between classes in the lattice.

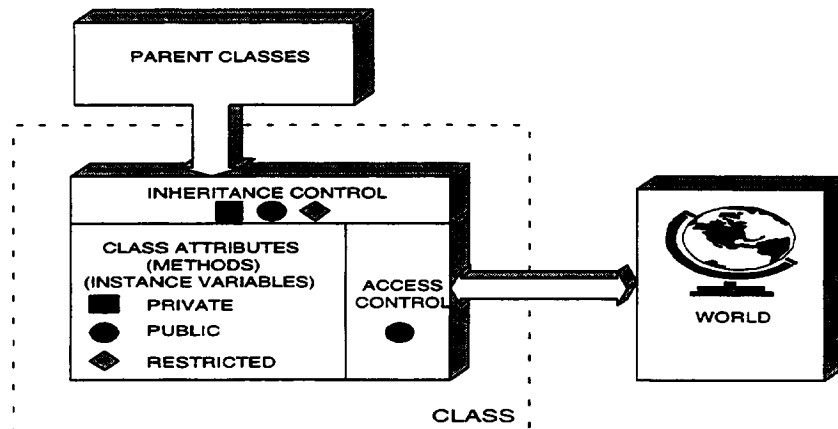


Figure 1 : Encapsulation control.

The need for multiple inheritance is often argued [8] considering that identical functionality can be achieved using single inheritance by adding extra properties and/or creating extra classes. However, multiple inheritance presents advantages over single inheritance. Multiple inheritance allows a better organisation of objects by avoiding redundancies in attribute and method declarations. Besides, its use requires prudence due to possible method name conflicts. The naming problem caused by multiple inheritance is avoided by the **method map** construct. Method mapping enables the renaming of ancestors' method names in order to avoid conflicts of similar methods in the newly defined child class.

```

method_map_aspect ::=
    method map ( method_association_list );

    method_association_list ::=
        method_association_element { , method_association_element }

    method_association_element ::=
        formal_method_name => actual_method_name

    formal_method_name ::=
        formal_method_designator [ (formal_parameter_list) ] [ return
        type_mark ] of formal_class_name

```

Moreover, it is often necessary to redefine inherited methods to provide a different implementation that make use of the local properties of the child class. This is called 'method overloading'. The redefined method has the same name as the inherited one but will act differently depending on the object used. This operation is known as 'polymorphism'. Polymorphic methods in the proposed extension are seen as an extent to the procedures and functions of the 1076 VHDL standard. Once abstract data types are developed, instances of those data types (objects) will be created.

#### Section 4. Concurrent Objects and Message Passing

The object concept and class are interdependent. An object belongs to a class and classes gather logically similar objects. An object can be seen as a unit that comprises data and methods that act upon the data.

In this object-oriented extension the object can be initialised through the use of a *constructor* method. The constructor has the same name as the object class and can be overloaded. The choice of the constructor method will then be made according to its argument types.

In a hardware description language, an object will eventually represent hardware modules. As a result, the conventional OO concepts of dynamically creating and deleting objects (with destructor functions) are not applicable from a synthesis point of view. Creation of an object is implicit and not caused by a message call to a creator function. Consequently, it will only take place when the object is declared.

In standard VHDL, signals are used to communicate between concurrent processes. In an object oriented environment, communication protocols between objects use the concept of message passing. Objects receiving a message will activate an operation (method) in response to a call.

```

method_call ::=
    object_name.method_name[ (argument_list) ]

```

In the basic object model, an object is able to receive more than one message at a time. The type of message is listed in three categories : Event, Command and Request. An Event is a message sent on an object with no input arguments nor return argument an event message will act internally to an object but will not expect any responses. The Command message type will contain input arguments as part of the message but no return argument. The request message differs from the first two types in that he has one or more input arguments and a return argument. A call to an object using a request type for the message will cause

the calling object to wait for the answer from the called object. Allowing multiple calls to occur concurrently has a significant advantage over a message queue as, for instance, an object could be read and written to at the same time (dual port RAMs). However the ability to access two methods within the same delta time implies possible multiple drivers on instances variables and therefore possible non deterministic behaviour (See Figure 2). As a result, a set of attributes are used to perform checks on the state of a particular method. Through the use of those attributes, the designer is able to set some pre-execution conditions on methods.

- (1) 'event : Will return a true when a method call occurs on an object.
- (2) 'last\_event : Will return the time since a previous method call occurred on an object.
- (3) 'active : Will return true when an object as at least one or more methods currently processing.
- (4) 'last\_active: Will return the time since the last method processing activity occurred on an object.
- (5) 'stable [(time)] : Will return true whenever the referenced object as had no method call for the time specified by the optional time expression.
- (6) 'quiet [(time)] : Will return true whenever the referenced object as not been processing any methods for the time specified by the optional time expression.
- (7) 'transaction : Which create a signal of type bit that toggles its value for every method call that occurs on the referenced object.

Example : A class with two methods with a priority for execution.

```

class reg is
    value : bit_vector(7 downto 0);
begin
    public method reset is
    begin
        value <= (others => '0');
    end method reset;

    public method write(data_in : bit_vector(7 downto 0)) is
    begin
        if not(reset'active or reset'event) then
            value <= data_in;
        end if;
    end method write;
end class reg;

```

Base classes will be developed to implement more complex message handling protocols such as method queues with fixed or rotating priorities.

## Section 5. Pre-Compiler: Simulation and Synthesis issues

To implement the Object-Oriented extension to VHDL the pre-compiler solution has been retained. When generating VHDL code from an higher form of description, a number of issues have to be considered.

It is generally understood that the quality of the VHDL code determines the performance simulation and synthesis tools. Along with coding style, an understanding of how the tool is implemented is necessary for generating efficient code. In simulation, three critical points can be identified: Use of variables versus signals to implement objects, problems in using subprograms and impact of intensive use of processes for modelling methods.

The implementation of instance variables can be carried out by either variables or signals. If the signal solution is chosen, an important flexibility is gained over the use of variables as the scope of signals is wider than the scope of variables visibility. Nevertheless, the use of signals give rise to disadvantages, such as the need for scheduling or greater memory usage (attributes). On the other hand, variables can be seen as more efficient for simulation. However, the use of variables to implement instance variables implies that all methods referring to this instance variable will have to be within the same concurrent statement (process, subprogram). This represent a limiting behaviour. The use of constructor function for initialising objects and consequently instance variables, represents an other issue. If the simulation process is divided into three stages, elaboration, initialisation and execution, only variables contained within processes will be initialised before the execution phase. As a result, the use of signals is not suitable to implement instance variables when considering creator functions. Alternatively, shared variables should represent a better solution as it allows a wider scope of accessibility and does not have the disadvantages of signals [13][14]. The issue in this case is due to the tool support. Simulation tools can handle shared variable but synthesis tools cannot.

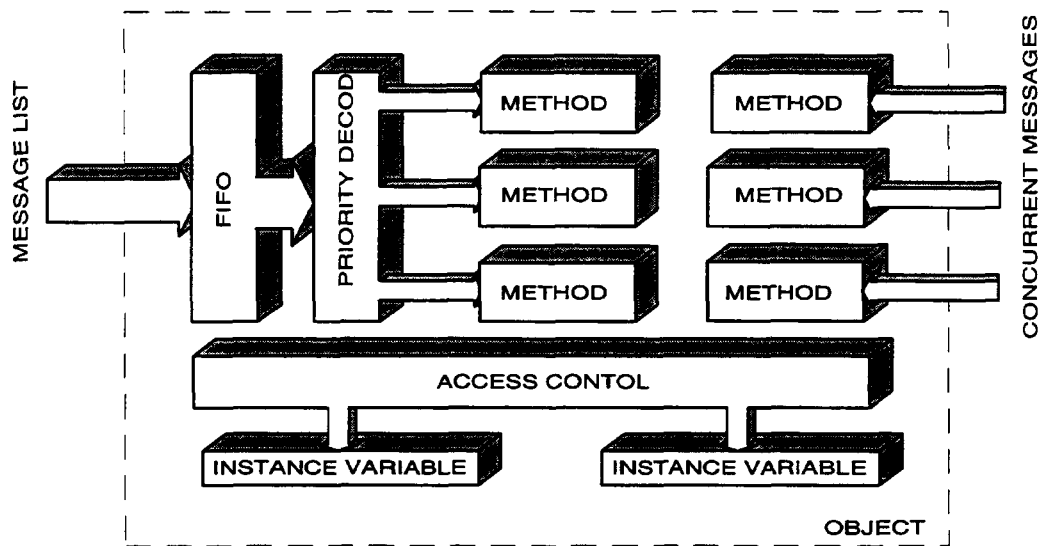


Figure 2 : Object Message Handling (Queued VS Concurrent)

Providing that instance variables are either implemented using signals or shared variables, two alternatives for implementing methods are evaluated processes and subprograms. Processes represent the most obvious implementation because, as methods, they can access with no restrictions to shared variables and resolved signals defined within the architecture scope. Incidentally, the use of overloading is not permitted with a process based solution. Furthermore, the use of processes implies the creation of process lists during the execution phase of the simulation, therefore degrading the simulation speed efficiency.

Alternatively, procedures support overloading and provide most of the capability of the processes, although procedures are elaborated at run time as opposed to elaboration time. This implies slower simulation runs when using procedures.

When considering synthesis, two main issues can be identified: Design size and speed efficiency; the use of OO modelling for hardware design can influence those two factors.

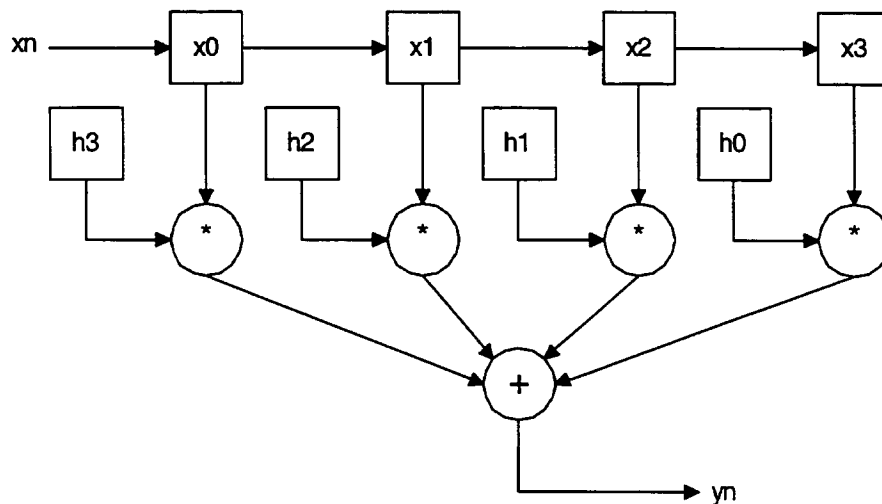
The inheritance is a paradigm commonly used in the OO modelling as it provides fast design cycles. However, from a synthesis point of view, inherited attributes (instance variables or methods) will correspond to an hardware equivalent. Consequently, the classes organisation will have to be designed so that it minimises redundant functionality inherited during the sub-classing process [8]. Alternatively an enhanced pre-compiler should be able to identify the necessary functionality of an object through its use and optimise its implementation. In addition to potential pitfalls introduced by the inheritance, common design techniques used in OO modelling do not find corresponding in the synthesis world. The concept of dynamically creating and removing objects raises interest for simulating large behavioural models. Nevertheless, the action of removing objects has no direct equivalencies in hardware (ASICs synthesis).

Regarding the speed issue, in an object based design, the nature of the message handling is determining. Three main approaches are identified: Message list, concurrent messages and a mixing of both approaches (See Figure 2). The message list implements a queue (FIFO) with possible priority decoding mechanism. The current message implies a data and a command bus for every messages. The last solution represents a compromise in between the two approaches. The choice of an approach versus the others will correspond to a trade off between area or speed efficiency.

Clearly, when developing a pre-processor, numerous issues have to be considered and trade-offs can only be made between simulation/synthesis runs and code conciseness. As a result the proposed processor will implement two different code generations depending on the target (synthesis or simulation) from the same Object-Oriented VHDL base.

## Section 6. FIR Study

This example emphasise the use of the semantic rather than the Object-Oriented design methodology as we concentrate on a small design. The goal of this example is to design an FIR filter. This filter is characterised by two modes of operation: The loading of the coefficient 'h' and the processing operation. For that purpose, a class with a method 'set\_h' and a method 'calculate' is created. The set\_h method will load into h (private instance variable) on a positive edge of the clock, the value contained in its formal argument 'value'. The method calculate performs the shifting operation for the  $x_n$  values and will calculate the value of  $y_n$  on a positive edge of the clock.



```

entity fir_16 is
    port (xn : in integer;
          yn : out integer;
          clk : in bit;
          load : in bit);
end fir_16;

architecture oo of fir_16 is
type vector is array (integer range <>) of integer;

class fir is
    generic (n : integer :=16);
    private h, x : vector(0 to n-1);
begin
method set_h(value : vector(0 to n-1); clk : bit) is
begin
    if clk'event and clk = '1' then
        h <= value;
    end if;
end method set_h;
method calculate(xn : integer; clk : bit) is
    variable tmp : integer;
    variable tmp_x : vector;
begin
    if clk'event and clk = '1' then
        tmp := 0;
        tmp_x := (xn, x(0 to (n-1-1)));
        for i in 0 to n-1 loop
            tmp := tmp + (tmp_x(i)*h(n-i-1));
        end loop;
        x <= tmp_x;
        return tmp;
    end if;
end method calculate;
end class fir;

signal filter_1 : fir generic map (n => 4);

begin
    process(clk)
        constant coef_h : vector := (2, 4, 8, 16);
    begin
        if (load = '1') then
            filter_1.set_h(coef_h, clk);
        else
            yn <= filter_1.calculate(xn, clk);
        end if;
    end process;
end oo;

```

## Conclusions

In this paper we described the study carried out by Bournemouth University in collaboration with IBM, of an extension to VHDL. This proposed extension is based on what is known as the Object-Oriented paradigm. Throughout this paper we demonstrated the high degree of re-usability and expressive powers that is achieved in using the Object-Oriented paradigm in conjunction with VHDL. The concept of Object-Oriented, when applied to VHDL, brings a new set of capabilities and also allow designs to be addressed at a higher level of description. In addition, the resulting environment allows VHDL to be used more effectively with Object-Oriented analysis and design techniques at the system description level. This achieves an integrated and consistent design flow from system level to register transfer level. We provided an understanding of VHDL limitations and Backus-Naur forms of the proposed new constructs were given. A more complete description of the proposed language changes is available in [21]. The pre-compiler is still in development and will be available by the end of August 96. A semantic has already been defined using LEX and YACC and is already available. Further work will concentrate on integrating the proposed extension to VHDL within existing design environments.

## References

- [1] Meyer B., 'Eiffel The Language' Prentice-Hall, London, 1992.
- [2] Goldberg A., Robson D., 'Smalltalk-80 : The language and its Implementation' Addison Wesley, Reading, Massachusetts, 1989.
- [3] Anderson B. and Gossain S., 'Hierarchy Evolution and the Software Life Cycle' Object-Oriented Software.
- [4] Wolfgang Muller, 'ODICE : Object oriented hardware descriptions in CAD environment', CHDL and their applications IFIP 1990.
- [5] Adam Pawlak, 'Modern object oriented programming language as a HDL', CHDL and their applications IFIP 1987.
- [6] Akikazu Takeuchi, 'Object Oriented description environment for computer hardware', CHDL and their applications 1981.
- [7] Wolfgang Glunz, 'Integrating SDL and VHDL for system level hardware design', CHDL and their applications IFIP 1993.
- [8] Armstrong A., Mitchell R., 'Uses and abuses of inheritance' Software Engineering Journal January 1994.
- [9] Perry, 'D Applying Object Oriented Techniques to VHDL.' Proceedings of the VIUF Spring Conference, 217-224, 1992.
- [10] Mills, M. T. Tt. Col.: 'Proposed Object Oriented Programming (OOP) Enhancements to the Very High Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL).' WL-TR-93-5025 Write Laboratory
- [11] John Barnes., 'Programming in Ada95' Addison-Wesley 1995 ISBN 0-201-87700-7.
- [12] Kumar, S.; Aylor, J H.; Johnson B. W.; Wulf, Wm. A.: 'Object-Oriented Techniques in Hardware Design.' Computer, June 1994, p. 64-70, 1994.
- [13] Berge, JM.; Finfoua, A.; Maginot S.; Rouillard, J. 'VHDL'92 The New Features of the VHDL Hardware Description Language.' 1993.
- [14] Bailey, S.a., Willis J.C., 'Shared Variables in VHDL 1993: A Peek into the Past and a Preview of The Future' VIUF Fall 1994 Conference, pp. 8.17-8.26.
- [15] Cabanis D., Medhat S., Weavers N., 'Object-Oriented Applied to VHDL Descriptions' Proceedings VIUF Spring 1995 pp. 3.9-3.15.
- [16] Covnot, B.M, Hurst, D.W., Swamy, S., 'OO-VHDL: An Object-Oriented VHDL', VIUF Fall 1994 Conference pp. 6.1-6.10.
- [17] Dunlop, D., 'Object-Oriented Extensions to VHDL' Proceedings VIUF Fall 1994 pp. 5.1-5.9.
- [18] Verschueren, A.C., 'An Object-Oriented Modelling Technique for Analysis and Design of Complex (Real Time) Systems', PhD Thesis, Technische Universiteit Eindhoven, May 1992, ISBN 90-9005046-9.
- [19] Sowmitri Swamy, Arthur Molin, Burt Covnot, 'OO-VHDL, Object-Oriented Extensions to VHDL' Computer October 1995.
- [20] Guido Schumacher, Wolfgang Nebel, 'Inheritance Concept for Signals in Object-Oriented Extensions to VHDL'. Proceedings EURO-VHDL/EURO-DAC Brighton 1995, pp. 428-435.
- [21] Cabanis D., 'Proposed Object Oriented Extensions to VHDL' Report Version 1.0, Bournemouth University available by email at [dcabanis@bmth.ac.uk](mailto:dcabanis@bmth.ac.uk).