27

Object Oriented Extensions to VHDL, The LaMI proposal

Judith BENZAKKI, *Bachir* DJAFRI
LaMI, Université d'Evry
Cours Monseigneur Roméro 91025 Evry , FRANCE,
{benzakki, djafri}@lami.univ-evry.fr

Abstract

VHDL[14] is a language for describing digital hardware. It provides many attractive features and supports an efficient design methodology. However, there are certain limitations to the power of the language that can be addressed, at least partially, by Object Oriented Extensions. Several proposals were made to extend VHDL and add mechanisms from the object oriented domain. In this paper we describe the study, carried out by the LaMI laboratory, of another approach to Object Oriented extensions to VHDL in view of a future revision of the language. Related works will be reviewed and the motivations for this proposal will be introduced and illustrated through examples.

Keywords

VHDL, object, inheritance, encapsulation, message-passing, polymorphism, reusability

1 INTRODUCTION

Although design automation tools have sped up the design process, the steady increase in digital system complexity keeps up the challenge faced by the designer. Consequently, alternative design methods must be found to further reduce development time[2][3]. In the software domain, the introduction of object oriented methods has revolutionized the process of developing software, and greatly improved re-usability and maintainability[7][8]. Abstraction and management of complexity are the main drivers behind current efforts to add object oriented extensions to VHDL.

A language is defined as being an *object oriented language* if it supports *objects, classes* and *inheritance.* These concepts will be defined in this paper and we will show through examples how these extensions can improve process development and endow the system designer with more abstraction capabilities.

In VHDL, one of the most important concepts is the *component*, which

encapsulates a "black box" view of a piece of hardware. This makes VHDL suitable to develop detailed low-level models, but not to write abstract high-level models, which specify what a piece of hardware should do, and not how it is supposed to do it. Within the object oriented paradigm, a system is viewed as a collection of communicating objects. Some object oriented features, such as abstraction, encapsulation, and modularity are already part of VHDL. This proposal tries to incorporate other object oriented features, such as inheritance and polymorphism.

Our approach considers objects as **active** entities (*object* = *entity*) which can return values in response to messages. Then, in the same *entity*, data and functions (called *operations*) are grouped to favor encapsulation and data abstraction.

2 OVERVIEW OF THE PROBLEM

Like most structured programming languages, VHDL has some limitations when the data structures, processes, entities, and architectures need to be reused. As an example, consider the ROM design shown below:

```
-----------------------------------------------------------------
-- Abstract model of Read Only Memory
-- Parameters that can be changed: Number of addressable locations,
-- Wordsize and Access time
-----------------------------------------------------------------
Library IEEE;
 Use IEEE.std_logic_1164.all;

Entity ROM is
-----------------------------------------------------------------
-- The generic "input_size" determines the number of addressable
-- locations
-- The generic "output_size" determines the wordlength
-- The generic "TAccess" determines the access time
-----------------------------------------------------------------
GENERIC ( INPUT_SIZE : integer := 16;
          OUTPUT_SIZE : integer := 16;
          TAccess : TIME := 3 ns );
PORT ( Address : IN std_logic_vector (input_size-1 downto 0);
       Data : OUT std_logic_vector (output_size-1 downto 0);
       Read : IN std_logic );
END ROM;

ARCHITECTURE ROM_architecture of ROM is

TYPE mem_array is array(natural range <>) of std_logic_vector(output_size-1 downto 0);
SUBTYPE mem_type is mem_array(2**input_size-1 downto 0);

begin -- ROM_architecture

Read_proc : process (Read)
 variable rom : mem_type := (others => (others =>'0'));
begin
     Data <= ROM(To_Integer(Address)) after TAccess;
end process Read;

end ROM_architecture;
```

The **Read** operation done by the ROM is described using a process statement. This model could also be useful to design a RAM or any other type of memory. The current restrictions in VHDL do not allow addition or redefinition of component functionalities. Designers must either use the component 'as is' or design a new one. Object-oriented extensions can overcome this limitation and increase the potential for reuse by adding inheritance to entities and architectures.

3 OBJECT ORIENTED CONCEPTS

Procedural programming, whose most famous representative languages are probably Pascal and C, is characterized by the decomposition of the application into independent procedures working on distinct data. However, in case part of the data structure is changed, whole or part of the procedures concerned by this modification must be rewritten. Therefore, the idea is to group the data (stored in variables called **attributes**) and the procedures (called **operations** or **methods**) into one entity: the **object**. Objects are the key concept of object oriented technology. They all have both *state* (**attributes**) and *behaviour* (**operations**).

The following illustration is a common visual representation of an object:

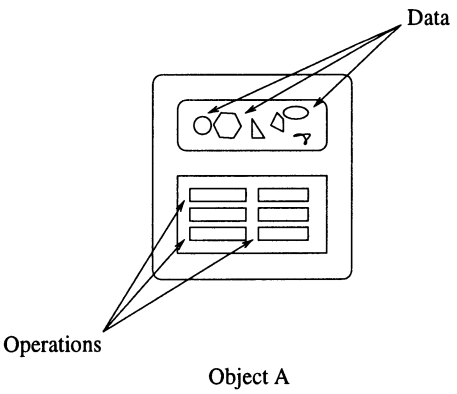
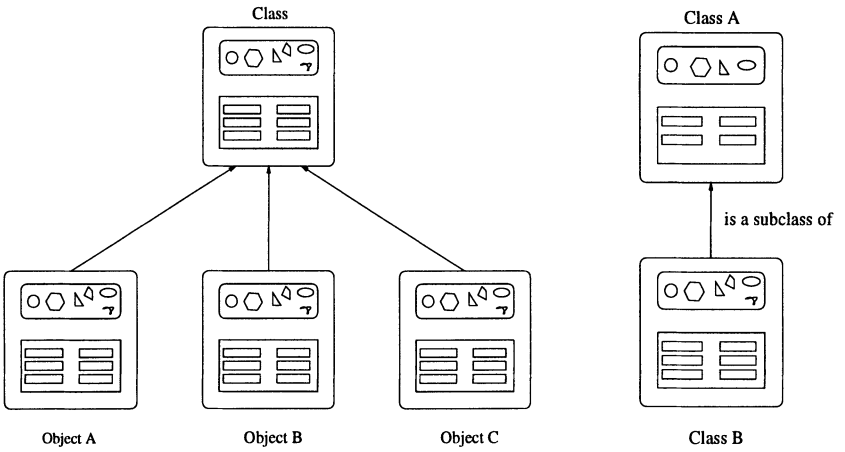


Figure 1 Visual representation of an object

Everything that the object knows (state) and can do (behaviour) is expressed by the variables and operations within that object. Packaging an object's variables within the protective custody of its operations is called *encapsulation.* Typically, encapsulation is used to hide unimportant implementation details from other objects. Thus, the implementation can change at any time without changing other parts of the program. In addition to this *information hiding* mechanism, it provides the benefit of *modularity.* In object oriented terminology, we often say that an object is an *instance* of a certain

class of objects that share the same characteristics. A *class* defines the variables and the operations common to all objects of a certain kind. However, classes and objects are sometimes difficult to differentiate. This is partially because objects and classes look very similar. In the real world it is obvious that classes are not themselves the objects that they describe. They are just a template or prototype that defines the objects (see Figure 2 (a)).



(a) Classes and objects

(b) SuperClass and SubClass

Figure 2 Relations Class/Object and SuperClass/SubClass

Objects are generally defined in terms of classes. Object oriented systems take this a step further and allow new classes to be defined in terms of other classes, thus, providing the benefit of reusability. These new classes are called *subclasses* (also known as child classes or derived classes) of the *superclass* (also known as parent class or base class) as shown in Figure 2 (b). Each subclass *inherits* both the state (in the form of variable declarations) and operations from the existing superclass. However, subclasses are not limited to the state and behaviours provided to them by their superclass. They may also include other variables and operations than the ones they inherited from the superclass. They can also *override* inherited operations and provide specialized implementations for those operations.

Object oriented technology involves viewing a system as a collection of objects that interact and communicate with each other via *messages*. When object A wants object B to perform one of its operations, object A sends a message to object B and object B will respond by performing the operation and perhaps modifying the values of its variables (see Figure 3).

Messages are made of three parts:

- the `object` to whom the message is addressed

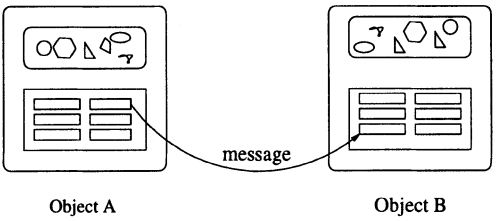


Figure 3 Communication between objects

- the name of the `operation` to perform
- the `parameters` needed by the operation.

These three components are enough information for the receiving object to perform the desired operation.

We will see in the next paragraphs how these concepts are applied in the different approaches for object oriented extensions to VHDL.

4 THE LANGUAGE EXTENSIONS

Three different proposals for object oriented language extensions to VHDL have been made. The first proposal comes from the VISTA Technologies Company[11][13] and is based on a new design unit, which is an extension of entities and their corresponding architectures. The second approach is based on type extension and has been developed at the University of Oldenburg and the OFFIS Institute[12]. The third approach, developed by the University of Bournemouth in collaboration with IBM[9][10], proposes classes as an alternative to packages to implement abstract data types.

The main features of these three proposals are given below.

4.1 The VISTA proposal

The VISTA proposal is mainly based on a new design unit called *Entity-Object*, which is an extension of VHDL entities and their corresponding architectures. An EntityObject represents an extra abstract data type which may have ports, generics and operation declarations. Operations are similar to procedures and their body is specified in the corresponding architectures. The operations are visible outside the EntityObject and can be invoked from other EntityObjects. Instance variables are introduced to define attributes of EntityObjects. In contrast with signals, there is no fixed interconnection between EntityObjects to invoke operations. Instead, so-called *handles*, which can be signals or variables, are used to call operations. Messages received by an EntityObject are queued and then executed in a sequential order. Enti-

ties, architectures, and EntityObjects can inherit several elements of existing entities, architectures, and EntityObjects. For communications, two new instructions are introduced, an *accept* and a *send* command. They are similar to those in Ada. An EntityObject corresponds to a task, an operation to an entry and the call of a send command is similar to the call of a task.

4.2 The Oldenburg proposal

The second approach to the OO extension to VHDL is inspired by the language Ada 95 (OO Extension of Ada)[1]. 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. Data structures, i.e. records, (tagged for inheritance) are used for modelling objects, and procedures are used to implement the behavioural part of the abstract data type (methods of classes). Generalization is introduced by an attribute *'Class*, which can be attached to a tagged record. Tagged records and the associated procedures are declared in the same package. The key for reuse is an inheritance concept together with a concept for generalization and abstraction. A new record can be derived from existing tagged records with the key words *new* and *with*. Behaviour can be incrementally specialized by the declaration of additional procedures or by the redefinition of inherited procedures. Communication between objects which are declared in the same process can be realized by a simple procedure call.

4.3 The Bournemouth/IBM proposal

In this third proposed extension, a 'class' is an abstract data type that defines the type and the behaviour of common objects. Generic clauses and method mappings are defined in the class header. The class can be defined at three levels: package, entity and architecture. Signals and instance variables (attributes) can be declared as objects of a defined class, and control over the accessibility of attributes and behaviour of a class is ruled by a three-level encapsulation mechanism: private, public and restricted. This proposal provides two types of inheritance: simple inheritance (single parent) and multiple inheritance (multiple parents). The *use* statement is expanded to select the inherited classes while the naming problem caused by multiple inheritance is avoided by the *method map* construct. The proposed extension enables multiple concurrent accesses to the object. Therefore, two or more operations can be carried out concurrently on the same object. Exclusion mechanisms are then used to avoid undeterministic behaviour.

The three proposals try to enhance the possibilities for modeling at a higher

level of abstraction. They use different constructs for modelling classes and objects and provide two different paradigms of object extension of VHDL. The LaMI proposal is based on physical components and suggests that VHDL entities be considered as objects. In contrast with the VISTA proposal, object operations are concurrent and have the semantic of VHDL processes.

4.4 The LaMI proposal

The main goal of object oriented extensions to VHDL is to provide more abstraction to the system designers and make designs more easily maintainable and reusable. VHDL already incorporates, to some degree, abstraction, modularity, and encapsulation, but it has some limitations when data structures, processes, entities, and architectures need to be reused in a new description. This proposal considers the different aspects described in the **Design Objectives Document** discussed through the IEEE DASC group on Object Oriented Extension to VHDL[5].

(a) Objects

Due to the critical role of a design entity (entity/architecture pair) in VHDL, there is a natural interest in basing object oriented extensions to VHDL on the idea of a design entity as an object belonging to a class. VHDL entities offer a good encapsulation mechanism while their abstraction capabilities are limited to port declaration. An object, in this proposal, is used to model a hardware component. It is an extention of a VHDL entity with methods called **operations**. Operations and their parameters are declared in the entity. Operation bodies, which define the implementation of these operations, are defined in the corresponding architectures of the entity. They are analogous to process bodies. The sequential statements in the operation statement part are analogous to those found in processes.

To demonstrate the extentions of the VHDL entity, consider the simple component seen above. The ROM provides only one operation, Read. This operation corresponds to a process. The object oriented version of the ROM is shown below:

```
-----------------------------------------------------------------------
-- Abstract OO_model of Read Only Memory
-- Parameters that can be changed: Number of addressable locations,
-- Wordsize and Access time
-----------------------------------------------------------------------
Library IEEE;
 Use IEEE.std_logic_1164.all;

Entity ROM is
-----------------------------------------------------------------------
-- The generic "input_size" determines the number of addressable
-- locations
-- The generic "output_size" determines the wordlength
-- The generic "TAccess" determines the access time
```

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

Operation interfaces are, like ports, visible outside the entity. They are similar to Ada task Entries[4]. Each operation may have in, out, or inout parameters which are signals only. These operations may be used together with generics but not with ports. Mixing ports and operations is not allowed.

Each operation has a body in the corresponding architectures of entity as shown below.

```
ARCHITECTURE OO_ROM_architecture of ROM is

TYPE mem_array is array(natural range <>) of std_logic_vector(output_size-1 downto 0);
SUBTYPE mem_type is mem_array(2**input_size-1 downto 0);

-- Internal data structure : internal state
Instance variable rom : mem_type := (others => (others =>'0'));

begin -- OO_ROM_architecture

Operation Read ( Address : IN std_logic_vector (input_size-1 downto 0);
                 Data : OUT std_logic_vector (output_size-1 downto 0)) do
begin
     Data <= ROM(To_Integer(Address)) after TAccess;
end operation Read;

end OO_ROM_architecture;
```

The internal state of the object is defined by **instance variables** which are used like shared variables[15][6]. These instance variables are not "public". They can be accessed only by operations of the object itself (strong encapsulation).

The internal state can also be defined by component declarations (objects) and the operations may declare local variables which retain their values between successive calls. These variables are similar to those declared in VHDL processes.

(b) Inheritance

In order to increase the potential for component reuse, entities are extended to support the *inheritance* mechanism. In this proposal, we add inheritance to entities and architectures by adding the keyword **is new** to the entity (or architecture) declaration. Each entity (or architecture) after the reserved words *is new* refers to an existing entity (or architecture) that is inherited. This allows new VHDL Entities and Architectures to inherit characteristics and functionalities of existing ones and thus to extend them. Inheritance is simple inheritance and all elements of an entity or architecture are inherited when subclassed. These elements include declarations, instance variables, all defined

operations and generics. New operations or instance variables may be defined, and operations may be redefined (overridden). Declarations, however, may not be overridden since they are required for proper operation of the defining superclass. For example, consider the component RAM which is similar to the ROM with an additional operation `Write`. Note that in this example, the RAM inherits all the caracteristics of the ROM such as the generic part, the operation `Read` and the variable `rom`.

```
-------------------------------------------------------------------
-- Abstract OO_model of Random Access Memory
-------------------------------------------------------------------
Library IEEE;
 Use IEEE.std_logic_1164.all;

Entity RAM is new ROM
-------------------------------------------------------------------
-- The generic part is inherited
-- Operation Read is inherited too
-------------------------------------------------------------------
-- defining new operation Write

Operation Write ( Address : IN std_logic_vector (input_size-1 downto 0);
                  Data : IN std_logic_vector (output_size-1 downto 0));

END RAM;

ARCHITECTURE OO_RAM_architecture of RAM is new ROM_architecture

-- TYPE mem_array, SUBTYPE mem_type and variable rom are inherited

begin -- OO_RAM_architecture

Operation Write ( Address : IN std_logic_vector (input_size-1 downto 0);
                  Data : IN std_logic_vector (output_size-1 downto 0)) do
begin
     ROM(To_Integer(Address)) <= Data after TAccess;
end operation Write;

end OO_RAM_architecture;
```

(c) Communication

While inheritance increases the potential for component reuse, message passing increases the designer's behavioural expressive power. Messages are used to invoke operations within an entity. The syntax of a message is: Object.Operation(parameters). A message is the only means of accessing the internal data representation. It is sent from an architecture body to an object (entity with object oriented declaration) at the same design level. This differs from a procedure call. The message *requests* that a particular operation be performed. It causes the operation to execute immediately, and the sender is blocked until the operation is performed (blocking call) unless the operation does not contain any return argument (non blocking call). The illustration of the two kinds of messages is given by Figure 4. From the sender's point of view, sending a message has the semantics of changing a signal's value in the sensitivity list of a process.

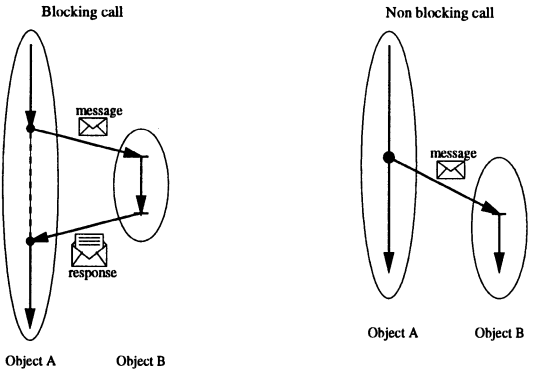


Figure 4 Communications

Messages sent to the same entity operation (object) at the same time are not lost. In case they invoke the same operation, messages are queued until this operation is performed. When the operation finishes, the next request is removed from the queue and served as shown in Figure 5. A message-send protocol is used to ensure that multiple messages sent to the same operation of an object (entity) at the same time are not lost.

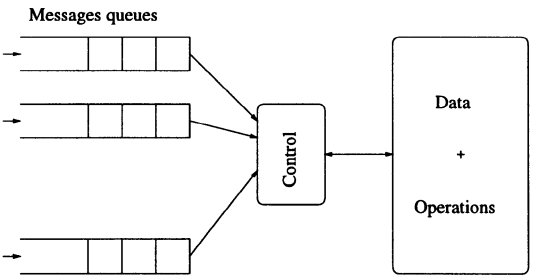


Figure 5 Messages protocol

The execution of operations of the same entity is concurrent (same as process) so that instance variables may be accessed by more than one operation. If two or more operations access an instance variable at the same time, the conflict is resolved by means of shared variables with a "protect" mechanism[15].

As expected from object oriented languages, an operation of an entity (object) O may call an operation of O. Then *self* is used in place of the object name.

(d) Polymorphism

Polymorphism is a mechanism related to inheritance that allows the designer to invoke operations without knowing the object. VHDL does not support

polymorphism. This proposal supports a broadcasting mechanism through a new attribute: `'Component`, which allows the same message to be broadcast to all objects of the same super class.

For example, suppose that the Counter entity has been defined with `GetVal`, `Reset`, and `Increment` operations (see Figure 6). By subclassing, LoadCounter inherits these operations and defines a new one: `Load`. All derived classes from the Counter class have the `Reset` operation. The operation `ResetAll` can then be defined as follows.

```
entity test_bench is end test_bench;

architecture Polymorphism of test_bench is

-- Object declarations

component counter_1 : Counter generic map(max_value => 8);
component counter_2 : LoadCounter generic map(max_value => 10);
component counter_3 : LoadCounter generic map(max_value => 16);

begin -- Polymorphism

Operation ResetAll do

begin

  Counter'component.Reset;

-- this statement calls the reset operation of all
-- components derived from the Counter class (Entity)

end Operation ResetAll;

end Polymorphism;
```

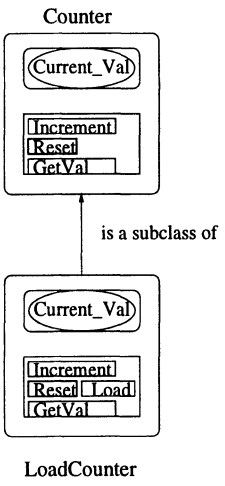


Figure 6 Subclassing

There will be no run-time error because we are sure that all objects derived from the Counter entity have at least the invoked `Reset` operation.

Note that there is no mapping mechanism for operations and that the internal state of the object can also be defined by component declarations (objects).

5 CONCLUSION

Table 1 illustrates how the different published proposals define the main object oriented concepts.

	Class/ Object	Inheritance mechanism	Communi- -cation	Polymorphism
VISTA	extension of entity concept EntityObject	entity/archi entityObject	Ada tasks like (send, accept,...)	handle: new predefined type
Olden- -burg	extension of type concept with tagged	record elements tagged types	protocols	'class attribute
Bourne- -mouth	type extension (class)	class	signals model	
LaMI	extension of std-entity concept	entity/archi anything inside	new protocol	new attribute: 'component

Table 1 Comparison of proposals for VHDL object-oriented extensions

The four proposals represent attractive alternatives to the Object Oriented extension for VHDL. The LaMI proposal has chosen to extend the VHDL entity in a simple manner by adding only two keywords **-is new** and **operation-** to model an object. This allows new objects to be inherited from existing ones, and the behaviour of an entity to be abstracted by describing the operations performed. These extensions do not force the designer to learn new syntax. They provide a new methodology where components are described as active objects (entities) which interact by sending messages to operations.

The dynamic polymorphism demonstrates the power of this approach by providing a broadcasting mechanism. All the examples presented in this paper have been rewritten in VHDL'93 to show that these extensions can be easily added to VHDL compilers. These extensions may be handled by a preprocessing tool which translates OOVHDL into standard VHDL. We are currently developing an OOVHDL $\rightarrow$ VHDL'93 preprocessor.

ACKNOWLEDGEMENTS

The authors would like to thank all the LaMI researchers for their collaboration and helpful discussions, and they would also like to acknowledge the insightful suggestions and comments of the reviewers on earlier drafts of the paper.

REFERENCES

[1] Changes to Ada – 1987 to 1995. International Standard ISO/IEC 8652:1995(E), 1992.

[2] M. Aiguier, J. Benzakki, G. Bernot, and M. Israël. ECOS: From Formal Specification to Hardware/Software Partitioning. *VHDL Forum*, September 1994.

[3] M. Aiguier, S. Beroff, L. Freund, G. Bernot, and M. Israël. Using Axiomatic Specifications for Hardware System Design. *CEEDA'96*, January 1996.

[4] J. Barnes. *Programmer en Ada.* InterEditions, 1988.

[5] J. M. Bergé, W. Nebel, and W. Putzke. Requirements and Design Objectives for an Object Oriented Extension to VHDL, August 1996. Preliminary Draft.

[6] J.M. Bergé, A. Fonkoua, S. Maginot, and J. Rouillard. *VHDL'92, The New Features of the VHDL Hardware Description Language.* Kluwer Academic Publishers, 1993.

[7] G. Booch. Object-Oriented Development. *IEEE Transactions on Software Engineering*, SE-12(2), February 1986.

[8] G. Booch. *Object Oriented Design.* Benjamin/Cummings Publishing, Redwood City, CA, 1991.

[9] D. Cabanis. Proposed Object Oriented Extensions to VHDL. *Report Version 1.0, Bournemouth University*, September 1995.

[10] D. Cabanis and S. Medhat. Classification-Orientation for VHDL: A Specification. *VHDL Forum for CAD in Europe, SIG-VHDL Spring'96 Working Conference, Dresden*, May 1996.

[11] B. M. Covnot, D. W. Hurst, and S. Swamy. OO-VHDL: An Object Oriented VHDL. *Proceedings of the VHDL International User's Forum*, 1994.

[12] G. Schumacher and W. Nebel. Inheritance Concept for Signals in Object-Oriented Extensions to VHDL. *Proceedings of the EURO-DAC'95 with EURO-VHDL'95, IEEE Computer Society Press*, 1995.

[13] S. Swamy, A. Molin, and B. Covnot. OO-VHDL: Object-Oriented Extensions to VHDL. *IEEE Computer*, October 1995.

[14] IEEE Standard VHDL Language Reference Manual. IEEE Std 1076-1993, Revision of IEEE Std 1076-1987, 1994.

[15] J. C. Willis. Preface to *shared variable language change specification*,

March 1996. Draft language change specification (LCS) proposing a revision of the VHDL-93 language reference manual (LRM) with respect to shared variables.