

An Object-Oriented View of Structural VHDL Description

Wolfgang Ecker

Corporate Research and Development
Siemens AG, ZFE T SE 5, D-81730 Munich, Germany
Wolfgang.Ecker@zfe.siemens.de

This paper presents an analogy between structural and object-oriented properties with the object to show a migration path for the introduction of the software methodology object-orientation in the hardware design process.

First VHDL structural description capabilities and object-oriented principles are described. Afterwards structural VHDL is analyzed under the aspect of object-oriented description methods. It is shown how a configuration can support "static" polymorphism. Incremental development and implementation as included in inheritance mechanisms can be modeled with structural statements. However some overhead is required.

Thus, an inheritance mechanism, already known from real-time object oriented modeling and similar to ADA tagged types, is proposed for VHDL entity, architecture and component declarations. Application examples and an outlook to genericity conclude the paper.

1 Introduction

The motivation for comparing structural VHDL description capabilities with object-oriented properties is many-sided and influenced by a set of computer science domains:

Object-oriented analysis and design methods [Mey88, CoYo90, ShMe90, Hod91, RBPEL91, JCJO92] dominate current software design methodology. Their application to hardware design is one possibility to improve the current hardware design and specification process. Object oriented description capabilities, however, are required for all approaches. Additionally, new trends in software design as well as new reuse approaches are using domain specific architectures [Gab93] and patterns [GHJV94]. These are also based on object-orientation. The study of architectures is an old fashioned but still very important hardware topic.

Modularity and composition of structural hardware design was often mentioned as model of object-oriented software design. Also object orientation is primarily based on communicating concurrent objects, which are closely related to interconnected concurrent hardware units.

A dualism between objects and units, which are required for the implementation of more abstract objects is shown in [GKRM93]. An other object oriented design strategy, a very promising reuse approach as described in [PHSR95] is mainly based on structural descriptions.

Several approaches for object oriented extension of VHDL already have been made [CoHS94, Dun94, Per92, ScNe95, ScNe95a, SWMC95, ZiMG92] and an object oriented extension study group is currently working towards object oriented VHDL. Moreover object orientation as description paradigm of the year 2000 was postulated in [EcMr96] based on the comparison of hardware and software design methodologies.

An approach for object oriented hardware description using C++ like formulations was described in [KAJW94]. A vice versa view, the analogy between structural and object oriented methods would help also to introduce the software methodology object orientation in the hardware design process.

2 Outline

The paper is organized as follows: First, structural capabilities of VHDL and object oriented description paradigms are enumerated. Afterwards a link between VHDL structural statements and object oriented programming paradigms is made. Due to the fact, that incremental design and implementation of inheritance need some overhead when using structural descriptions only, an inheritance mechanism for entity-, architecture-, and component declarations is shown next. A set of application examples and an outlook to genericity conclude the final paper.

3 VHDL Structural Descriptions

An ideal picture of VHDL structural description is a socket on a board into which a chip is putten into [BFMR]. This board again may be part of a subsystem and connected to a backplane via connector. The VHDL component relates in this picture to the socket, the entity/architecture pair to the chip, and the chip, board, subsystem to hierarchy. The component instantiation can be seen as connection of the socket's pins, whereas ports may be left open, assigned with a constant value or adapted to other types.

The (probably implicit) configuration specifies the relation of one entity/architecture pair with the socket and allows for flexible connection also. In relation with object oriented paradigms it is important to note that different architectures as well as different entities (with probably different interfaces) may be assessed to one component.

The binding of the entity/architecture can be made in three ways:

- directly in the unit where the component instantiation occurs (configuration specification),
- in an additional unit, which allows for more flexibility (configuration declaration), or
- implicitly using VHDL default binding rules.

VHDL'93 supports also entity instantiation which relates to a chip which is directly connected to a board. Moreover, VHDL allows for iterative as well as alternative structural description styles by using generate statements.

The comparison with object orientation is restricted in this paper to structural description style where bit values or composite bit values are used mostly. The VHDL concept of unconstrained arrays or the generic specification of vector bounds allows the specification of vectors of bit in a very flexible way.

4 Principles of Object Orientation

An overview of object oriented principles can be found eg. in [KoMG90]. The main principles and features of object orientation are:

- Classes, which allow an object be composed of a set of sub-elements. The elements are called state or attribute of the class and carry the information of the object. The collection of elements is also called abstraction.
- Encapsulation, which does only allow access to, and manipulation of the elements of an object via a well defined interface.

- Methods, which describe how elements of the object can be modified. They are activated by so called messages or invocation.
- Instance, which describes the incarnation or construction of an object. In most cases the destruction of an object can be specified also.
- Inheritance, which allows a class of an object to be derived from an existing class. Here all elements and all methods are part of the derived class. Methods, however, may be modified.
- Polymorphism, which allows unique handling of objects possessing similar methods. In most cases polymorphism is restricted to objects of inherited classes.
- Genericity, which allows a part of an object to be flexibly specified.
- Early respectively late binding, which describe the tasks associating operations to objects at compile or execution time.

Alternative and iterative constructs are also part of most object-oriented languages. They are not part of object-oriented features but they ease implementation of classes by omitting overhead of object-orientation, which is not required in a lot of cases.

5 Structure and Object-Orientation

Structural VHDL description relates with object oriented features under the idea that the structural view of hardware is static and thus staticness must be related in some sense to the dynamic execution of software. In our approach the analysis of VHDL code relates to compilation, and elaboration to execution.

- The VHDL entity/architecture pair relates to a class. Whereas the entity specifies the interface only and thus supports encapsulation. The architecture describes the implementation of the class. It explicitly declares signals, which partially relate to states or attributes. It is important to note however that signals may be used also to carry interim information.

```
entity regN is
  port( d : in bit_vector;
        q : out bit_vector;
        clk: in bit;
        res: in bit );
end RegN;

entity incN is
  port( x : in bit_vector;
        y : out bit_vector );
end incN;
```

The entity declarations shown above allow either direct access to the status of the object by declaring the state in the interface (eg.: q, y) or by allowing the activation of methods via physical lines (clk, res, x).

- Encapsulation is also ensured due to the fact that no internal object and no implementation detail is external, ie. outside the entity, visible.
- A Component is from an object-oriented point of view a declaration of a link respectively the declaration of a container for a set of classes. The component concept of VHDL allows thus “static” polymorphism due to the fact that the entity/architecture pair associated with a component need not be known at analysis time.

```

component storeN
  port( d : in bit_vector;
        q : out bit_vector;
        c : in bit;
        r : in bit );
end component;

component processN
  port( i : in bit_vector;
        o : out bit_vector );
end component;

```

The components storeN/processN may be associated with the entities (or classes in our comparison) regN/incN. But either different architectures of these entities or different entity-architecture pairs with the same or nearly the same interface (eg. latchN/shiftN, see subsequent listing) may be plugged in also.

```

entity latchN is
  port( d : in bit_vector;
        q : out bit_vector;
        c : in bit;
        rs : in bit );
end latchN;

entity shiftN is
  port( x : in bit_vector;
        y : out bit_vector );
end shiftN;

```

A configuration is required in this case to associate shown design entities with the component storageN/processN.

- The instance of a component specifies in this sense a link to a set of classes and the instance of an entity the derivation of methods.
- Structural statements describe the behavior of a class. They derive state and methods of instantiated (and bound) units and thus allow for modeling inheritance. Moreover instantiation of different components supports multiple inheritance.

```

entity pipeStage is
  port( pi : in bit_vector;
        po : out bit_vector;
        clock : in bit;
        reset : in bit );
end pipeStage;

architecture structural of pipeStage is
  signal int : bit_vector( pi'range );
begin
  P: processN port map( pi, int );
  S: storeN port map ( int, po, clock, reset );
end structural;

configuration Incrementer of pipeStage is
  for structural
    for P: processN use WORK.incN( behavior ); end for;
    for S: storeN use WORK.regN( behavior ); end for;
  end for;
end Incrementer;

```

Here, the configuration Incrementer binds the unit regN to storeN and incN to processN. The behavior for storing and initialization (eg. level, edge) of the unit storeN is derived from regN. Similarly, the incremental behavior of processN is derived from incN.

- Generic parameters facilitate among other things the specification of variant bit vector widths. This can also be achieved by using unconstrained arrays. Both relate under the assumption of weak typing to the object-oriented principle of genericity.

Bit chains with variant length may be processed by the element pipeStage and thus allow for genericity. But, this bit chains may represent different behavior (eg. number, bit map, command, character, ...) even if they are

interpreted as number when the incremter is used.

- Methods in structural design are activated or invoked by protocols. They consist of a
 - physical part, which is related to a set of signals and a
 - logical part, which consists of waveforms.

Considering regN,

- the physical part consists of the signals d,q,clk, res and the
- logical part of
 - a rising edge at clk to activate the method “store”
 - a high value at res to activate the method “reset” and
 - an event at q, which is a message to units connected to q activating their methods.
- Due to the fact that analysis is compared with compilation and elaboration with execution hard binding is specified by configuration specification or entity instantiation and soft binding by configuration declaration.

The configuration Incrementer is thus an example for soft binding. Hardbinding is shown in the listing below:

```
architecture hardbinding of pipeStage is
    signal int : bit_vector( pi'range );
begin
    P: entity incN port map( pi, int );
    S: entity regN port map ( int, po, clock, reset );
end structural;
```

VHDL generate statements relate to alternative as well as iterative constructs in software technique.

6 An Inheritance Concept for Structural VHDL

6.1 Motivation

Assume that the pipeline stage described above should be extended by a zero flag. The behavior of the storage element and the processing element can be inherited again. Their composition, however, must be rewritten completely.

```
entity pipeStageZero is
    port( pi: in bit_vector;
          po : buffer bit_vector;
          zero : out bit;
          clock: in bit;
          reset: in bit );
end pipeStageZero;

architecture structural of pipeStageZero is
    signal int : bit_vector( pi'range );
begin
    P: processN port map( pi, int );
    S: storeN port map ( int, po, clock, reset );
    Z: zeroDetect port map ( po, zero );
end structural;
```

This need not to be done if the model is derived by extending the pipestage model by inserting the line

```
clock: in bit;
```

in the port map and

```
Z: zeroDetect port map ( po, zero );
```

in the statement part.

This simple example shows that a huge overhead is required for incremental extensions when structural des-

model inheritance: Either all internal signals respectively states are visible at the interface, which however conflicts with the principle of encapsulation, or already described declarations and instantiations must be repeated, which causes overhead and is error prone.

6.2 Inheritance concept

To overcome this problem, a method for incremental extension of structural descriptions was presented in [SwMC95]. Here an entity and an architecture can be derived from an existing entity by using the VHDL key word “new”.

In [SwMC95] additionally the concept of EntityObjects is proposed. Here a state called instance variable and a set of methods called operations can be declared and inherited. The methods are activated by executing a send command out of a sequential statement outside the object.

We developed a very similar approach influenced by [SeGk94] and by ADA95 [ADA95]. The keyword “tagged” is used in our approach to show that a declaration is still incomplete. The listing below shows this for an entity- and component declaration of a register.

```
entity registerN is tagged
  generic(width : natural )
  port( clk : in bit );
      d : in bit_vector( width - 1 downto 0);
      q : in bit_vector( width - 1 downto 0) );
end entity registerN;

component registerN is tagged
  generic(width : natural )
  port( clk : in bit );
      d : in bit_vector( width - 1 downto 0);
      q : in bit_vector( width - 1 downto 0) );
end component registerN;
```

To incrementally extend a declaration also the keyword “new”, which already exists in VHDL, is used. Either a finalized declaration with extension, as shown below, a finalized declaration without extension or an other tagged declaration can be derived. All declarations, interface clauses, and statements, which are part of the tagged unit are implicitly part of the derived unit and may be extended. An extension however, which contains an identifier in the same naming scope of the tagged unit overwrites the declaration of the tagged unit.

```
entity lr_registerN is new registerN
  port( load : in bit;
        reset: in bit );
end entity lr_registerN;

component lr_registerN is new registerN
  port( load : in bit;
        reset: in bit );
end component lr_registerN;
```

Similarly an architecture can be marked with the keyword “tagged” and can be derived from existing tagged architectures. This is shown below.

```
architecture behavior of lr_registerN is tagged
begin
  p0:process( clk )
  begin
    if clk = '1' then
      if reset = '1' then
        q <= (others => '0');
      end if;
      if load = '1' then
        q <= d;
      end if;
    end if;
  end process;
end behavior;

architecture check of lr_registerN is tagged
begin
```

```

p1:process( clk )
begin
  if clk = '1' then
    assert load = '0' or reset = '0'
    report "concurrent load and reset"
    severity WARNING;
  end if;
end process;
end behavior;

architecture simulation of lr_register is new
behavior, check
begin
  -- empty
end simulation

```

In the final architecture multiple inheritance is used. This is also possible for entities and components. If identical identifiers are used in units, than the declaration occurring in the later enumerated unit (here: check) remains visible for multiple inheritance. This includes also that identical declarations occur only once.

It is important to note that architectures may be derived from tagged architectures of the same (probably tagged) entity or from tagged architectures from tagged entities, where the entity of the architecture is derived from. It shall be mentioned also that only tagged architectures may be associated with tagged entities. However both tagged and not tagged architectures may be assigned with not tagged entities.

7 Application Examples

7.1 Flexible uP-Interface

The uP-interface of an ASIC generally consists of a processor dependent interface to the CPU-bus, a decoder with data multiplexer and a set of ASIC specific registers. One possibility for the flexible modeling of the interface is the encapsulation of the CPU-interface and the decoder in a separate unit. A template eases the final composition of interface and ASIC registers. The problem of this solution is that the user of the template may also modify the signals between CPU-interface and the decoder.

A very elegant way to overcome this problem is to describe the core of the interface inside a tagged architecture. ASIC dependent uP-interfaces with a set of ASIC specific registers can then be derived from that description.

7.2 Shared Timing/Function Implementation

Performance analysis is one of the key points in early system design. Currently either a separate tool or a special VHDL description is used. Afterwards a new model must be developed.

In [ScEc96] a method for separated synchronization and function description is presented. First, synchronization is implemented and a cycle based performance analysis performed. In a second step, a dummy architecture is connected to the synchronization. Again performance analysis is done with this description. Finally, synchronization is putten together with full functionality and a final evaluation step is performed.

All models are structurally composed and thus behavior inherited as described in section 5. But the overhead rewriting and configuring an architecture composed of the controller and the functional description remains (see section 6.1).

The overhead could be omitted by using structural inheritance. First, an architecture, called control is declared, which consists of the description of the controller only. Second, an architecture is derived from control, which contains a unit representing dummy functionality only. Finally, an other unit is derived, which contains full functionality.

7.3 Test Approach

Assume a model consisting of three serial connected processes. The application of a specific pattern to one of the processes and the propagation of the output values for single process test reasons cause a lot of overhead. Multiple inheritance, as proposed above, allows to test all processes for their own.

First, the entity and one tagged architecture per process must be described. Afterwards a test architecture is derived from each tagged architecture. Processes applying stimuli or taking stimuli and putting them to the output are added. These derived and modified architectures are then tested separately. Finally, the complete model is composed by multiple inheritance and a module test is performed.

7.4 Method Activation by Abstract Message Passing

Abstract method activation as presented in [SwMC95] can be modelled using abstract communication mechanisms as described in [BaEc93]. For this, the entity consists of at least one interface signal to pass messages.

```
entity abstractObject is
  port( messageActivation : inout master_slave_communication );
end abstractObject;
```

The state of the abstract object is declared inside its architecture as well as a set of processes, each responsible for one method. The methods can externally be activated by sending a possibly parametrized message to the process. The execution of the process may then modify the state of the object, return information to the caller, send information via other communication channels or modify other port signals.

The state of the object as well as the methods can easily be extended by using structural inheritance as shown above.

8 Genericity

Several ways exist to achieve generic modeling: Instance, polymorphism, inheritance, generics and generate statements. This shall be discussed for different modelling and application cases by the model of a register.

8.1 Instance

Registers sensitive to different edges and reacting to different reset modes shall be used for the component storageN in the pipeStage example of Section 5. The behavior could be derived from different entity/architecture pairs, however the architecture hardbinding must be modified for each register type.

8.2 Polymorphism

A very elegant way to do so is by using the static polymorphism capabilities of a configuration as described in Section 5.

```
configuration Incrementer_Rising_High of pipeStage is
  for structural
    for P: processN use WORK.incN( behavior ); end for;
    for S: storeN use WORK.regN( rising_high ); end for;
  end for;
end Incrementer_Rising_High;

configuration Incrementer_Falling_Low of pipeStage is
  for structural
    for P: processN use WORK.incN( behavior ); end for;
    for S: storeN use WORK.regN_FL( behavior); end for;
  end for;
end Incrementer_Falling_Low;
```

The configurations showed above bind either the same entity with another architecture implementing the required behavior or another entity/architecture pair.

8.3 Inheritance

The overhead using different instances (see 8.1) can be omitted by structural inheritance. This approach however is not as elegant as using configuration based polymorphism.

8.4 Generics

The most compact modelling style is the use of generics for the selection of different register types.

```
type storeModeType is ( NoMemory, High, Low, Falling, Rising );
type resetModeType is ( NoReset, AsyncHigh, AsyncLow, SyncHigh, SyncLow );

component regNgeneric is
  generic ( storeMode : storeModeType;
            resetMode : resetModeType );
  port ( d : in bit_vector;
         q : out bit_vector;
         clk: in bit;
         res: in bit );
end RegNgeneric;
```

The generic parameters can be used in behavioral models directly or to control generate statements for synthesis purposes.

Generic capabilities of current VHDL however can not be used if values of different types must be handled.

An elegant way to overcome this problem is to use inheritance as shown below.

```
entity regN is tagged
  port ( clk: in bit;
         res: in bit );
end RegN;

architecture core of regN is tagged
  signal di,qi : bit_vector( maxsize downto 0 );
begin
  C : regNbit_vector port map( di, qi, clk, res );
end core;

entity regNinteger is new regN
  port ( d : in integer;
         q : out integer );
end regNinteger;

architecture full of regNinteger is new core
begin
  CI: Int2Bv port map( d, di );
  CO: Bv2Int port map( qi, q );
end full;
```

Here, the interface to clock and reset is part of the entity only. A register storing variant length bit chains is instantiated in the architecture. For each type, the interface is extended by d as well as q and the internal register s are connected to type conversion units.

Generic types as defined in ADA would allow for a very compact modelling in this purpose.

9 Conclusion and Outlook

An object-oriented view of structural VHDL descriptions and structural inheritance mechanisms have been presented. A set of different application examples shows additionally the applicability of object oriented features, structural description capabilities and the object-oriented view of structural VHDL. Current work concentrates on the discussion of the usefulness of tagged entities, generic types to overcome the problem of weak typing, the application of extendable units to other VHDL constructs like packages or configuration and the evaluation of the shown inheritance mechanisms towards not configuration based polymorphism. We also work on the application of object oriented ADA95 features in behavioral descriptions.

10 Bibliography

- [Ada95] *Information technology - Programming languages - Ada, Annotated Ada Reference Manual*. International Standard ISO/IEC 8652:1995(E). Intermetrics, 1995.
- [BaEc93] Bauer, M.; Ecker, W.: *Communication Mechanisms for VHDL Specification and Design starting at System Level*, Proceedings of the Spring'93 Meeting of the VHDL-Forum for CAD in Europe, Innsbruck, 1993.
- [Bar91] Barnes, J.: *Programming in Ada plus Language Reference Manual*, Addison-Wesley Publishing Company, Workham, England, 1991.
- [BFMR] Berge, J-M.; Fonkoua, A.; Maginot, S.; J. Rouillard, J.: *VHDL Designer's Reference*, Kluwer Academic Publishers
- [CoHS94] Covnot, B. Hurst, W. Swamy, S.: *OO-VHDL: An Object Oriented VHDL*. Proceedings of the VHDL International User's Forum, 1994.
- [CoYo90] Coad, P.; Yourdon, E.: *Object Oriented Analysis*. Prentice Hall, 1990.
- [Dun94] Dunlop, D.D. *Object-Oriented Extensions to VHDL*. Proceedings of the VHDL International User's Forum, 1994
- [EcMr96] Ecker, W.; Mrva, M.: *Objektorientierung: Modellierungs- und Entwurfparadigma des Jahres 2000*. 2. GI/ITG/GME Workshop: Hardwarebeschreibungssprachen und Modellierungsparadigmen. 1996.
- [Gab93] Gabriel, R.: *The quality without a name*. Journal of Object-Oriented Programming, September 1993, pp. 86-88.
- [GHJV94] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J.: *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.
- [GKRM93] Glunz, W., Kruse, T., Rössel, T., Monjau, D.: *Integrating SDL and VHDL for System-Level Hardware Design*. Proceedings CHDL'93, Ottawa, Canada, 1993.
- [GIPV92] Glunz, W., Pyttel, A., Venzl, G. *System-Level-Synthesis in Design*, Michel, P., Lauter, U., Duzy, P. (eds): *The Synthesis Approach to digital System*. Kluwer Academic Publishers, 1992.
- [GIUm91] Glunz, W.; Umbreit, G.: *VHDL for High-Level Synthesis of Digital Systems*. Proceedings of the EuroVHDL'91, Marseille, September 5-7, 1991.
- [Hod91] Hodgson, R.: *The X-Model: A Process Model for Object-Oriented Software Design*, Fourth International Conference on Software Engineering and Its Applications, Toulouse, France, 1991
- [JCJO92] Jacobsen, I.; Christerson, M.; Jonsson, P.; Overgaard, G.: *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, Reading, Massachusetts, 1992.
- [Jor94] Jorgensen, J.: *A Comparison of the Object Oriented Features of Ada9X and C++*. "from the net". March, 1994.
- [KAJW94] Kumar, S.; Aylor, J.; Johnson, B.; Wulf, W.: *Object-Oriented techniques in Hardware Design*. IEEE Computer, June 1994.
- [KoMG90] Korson, T.; McGregor, J.: *Understanding Object-Oriented: A Unifying Paradigm*, Communications of the ACM, Vol. 33, No. 9, September 1990
- [Mey88] Meyer, B.: *Object oriented Software Construction*. Prentice Hall, 1988.
- [Per92] Perry, D.: *Applying Object-Oriented Techniques to VHDL*. Proceedings of the VHDL International User's Forum, 1992.
- [PHSR95] Preis, V.; Henftling, R.; Schütz, M.; März-Rössel, S.: *A Reuse Scenario for the VHDL-based Hardware Design Flow*, Proceedings of the EURODAC/VHDL'95
- [RBP91] Rumbaugh, J.; Blaha, M.; Premeriani, F.; Eddy, F.; Lorenson, W.: *Object-Oriented Modeling and Design*, Prentice Hall, 1991.
- [ScEc96] Schneider, C.; Ecker, W.: *Stepwise Refinement of Behavioral VHDL Specifications by Separation of Synchronization and Function*. EURODAC/VHDL'96. submitted.
- [ScNe95] Schumacher, G.; Nebel, W.: *Survey on Languages for Object Oriented Hardware Design Methodologies*. Berge, J-M.; Levia, O., Rouillard, J. (eds): *High-Level System Modeling: Specification Languages*. Kluwer Academic Publishers, 1995
- [ScNe95a] Schumacher, G.; Nebel, W.: *Inheritance Concept for Signals in Object-Oriented Extensions to VHDL*, Proceedings EURODAC/VHDL'95
- [SeGW94] Selic, B.; Gullekson, G.; Ward, P-T.: *Real-Time Object-Oriented Modeling*. 1994
- [ShMe90] Shlaer, S.; Mellor, S.: *Recursive Design*. Computer Language 7, 3, 1990
- [Str91] Stroustrup, B.: *The C++ Programming Language*, Second Edition. Addison Wesley PublishThe C++ Programming Language.ing Company, 1991.
- [SwMC95] Swamy, D.; Molin, A.; Burton, M.C.: *OO-VHDL Externsions to VHDL*, IEEE Computer, October 1995.
- [Vhd187] IEEE Standard VHDL Language Reference Manual, IEEE Std 1076-1987.
- [Vhd193] IEEE Standard VHDL Language Reference Manual, IEEE Std 1076-1993.
- [ZiMG92] Zippelius, R.; Müller-Glaser, K.: *An Object-Oriented Extension of VHDL*. VHDL-Forum for CAD in Europe, Spring'92 Meeting, Santander/Spain, 1992