Applying Object Oriented Techniques to VHDL

Douglas L. Perry
Redwood Design Automation
97 S. Second St.
San Jose, CA 95113

Abstract

Object Oriented programming languages contain a number of capabilities that are very useful in designing and building large systems. Some of these include encapsulation, reusability, inheritance, and message passing. This paper will describe some of these techniques, give a brief description of their relevance to large system design, and show how they can be applied to VHDL descriptions.

Section 1. Introduction

Object Oriented Programming(OOP) has become quite popular recently. However, implementations of object oriented languages have been around for a number of years with Smalltalk, C++, and others. The reason for the sudden popularity is that object oriented languages are found to be very good at maintaining order when complex tasks are broken into manageable chunks. As more powerful computers become available programming tasks are getting much more complicated. Programmers are looking to object oriented approaches to keep their complexity under control.

This paper will discuss some of the most interesting concepts in object oriented programming. These are Objects and Messages, Encapsulation, and Inheritance. These topics will be defined and examples discussed. VHDL examples of the same functionality will be presented if possible.

Section 2. Objects and Messages

The basic unit of description in an object oriented language is the object. An object consists of a collection of code and data that describes the behavior of the device being modelled. Messages are used to communicate data and information between objects. Each object has a set of messages that have been defined for that object. The code for the object will implement the behavior of each defined message.

In Fig 1 is an object for a simple counter. It accepts messages for Increment and Reset operations and outputs an Outval message. When the counter receives an Increment message the internal count value stored in the object is incremented. When a Reset message is received the internal count value is set to zero. When the object receives an Out-

val message, the current counter value is returned.

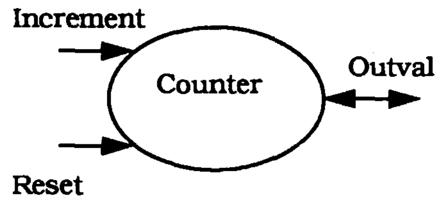

Fig 1

Notice that the only way to change the internal value of the counter is through a message. The only way to read the internal value of the counter is also through a message. The data inside the counter is effectively protected. This allows the modeler the freedom to use any data representation necessary for the operation. Other objects that communicate with this object do not care what the internal representation of the counter value is nor does it matter if the implementor at a later date changes the data representation as long as the message behavior is the same. This principle is called Encapsulation[1].

Section 3. Encapsulation

Encapsulation in an object oriented environment allows the code for the object and the data to be held together in a single class structure. Let's look at a C++ implementation of the counter example from above.

```
const int maxval = 15;
class counter {
   public:
      void increment();
      void reset();
      int outval();

   protected:

      int val_;
};

void counter::increment() {
   if (val_ >= maxval) {

       val_ = 0;

     } else {

       val_++;
     };
 };
```

```
void counter::reset() {
  val_ = 0;
 };

counter::outval() {
   return val_;
 };
```

This description contains one class called counter, that recognizes three messages, increment(), reset() and outval(). By making the message declarations public these messages can be sent by another object. The protected area contains the data for the class and the actual code for the object. By isolating the code and the data for an object in the private region of an object, access to this code and data is restricted. The advantage is that the developer of the object can use any data type or algorithm deemed necessary. Other objects will continue to communicate with the object through messages.

This protection will delineate exactly which code has access to a particular piece of data, and therefore what must change if data types change. This is in contrast to typical environments where access to data is restricted only by scoping rules.

Messages increment() and reset() are messages that only cause an action to occur within the object and no data is transferred from or to the object. However message outval() causes an action to occur in the object and transfers the current value of the counter to whichever object has sent the message. Not only can messages be used to control objects but data can be passed as well.

Let's now update the counter example by adding another message to load the counter with a particular value. One method to accomplish this would be to change the original example as shown below:

```
const int maxval = 15;
class ldcounter {

   public:
      void increment();
      void reset();
      void load(int startval);
      int outval();

   protected:
      int val_;

};

void ldcounter::increment() {
   if (val_ >= maxval) {
        val_ = 0;

   } else {

        val_++;
   };
};
```

```
void ldcounter::reset() {
  val_ = 0;
};

ldcounter::outval() {
   return val_;
};

void ldcounter::load (int startval) {
    val_ = startval;
};
```

We have added a new message called load() that allows the counter to be preset to a particular value. Otherwise the description is exactly the same as before.

This will work but what if the original functionality was needed as well. Two different versions of the description will be needed. A better solution is to leverage off of the first version of the description to create the second. This process is known as subclassing in C++ and the mechanics of the process are determined by the inheritance[3] properties of a language.

Section 4. Inheritance

Inheritance allows a particular class to inherit or receive its properties from another class or classes. The new class is then free to use or override properties to accomplish the chosen task. We can use the inheritance mechanism to create the new counter with the load message, but inherit all of the other messages and functionality from the original counter. An example is shown in Fig 3.

```
class ldcounter : public counter {
   public:
      void load( int startval);

};

void ldcounter::load(int startval) {
   val_ = startval;
};
```

A new class ldcounter is created which is a subclass of base class counter. Class ldcounter inherits all of the data members, and functions of class counter, and defines and implements a new message called load(). Objects of the new class ldcounter will understand reset(), increment(), outval(), and load() messages, while objects of class counter will only understand reset(), increment(), and outval() messages.

The advantage to this approach is that if changes are needed in class counter to fix a problem or update behavior, class ldcounter will inherit those changes automatically. This would not occur in the case where two separate classes were created. Each class would have to be individually updated.

Let's now try to apply these object oriented techniques to VHDL and see how well they will or will not work in the current VHDL environment.

Section 5. VHDL Objects and Messages

In C++ and SmallTalk the primitive design unit is the object. In VHDL the primitive design unit is the Entity. In object oriented languages objects contain code and data to implement the behavior of the device. While in VHDL an entity has one or more architectures that contain the code and data to implement the behavior of a device.

In Object Oriented languages objects communicate with messages. VHDL does not contain the concept of messages explicitly. Entities communicate with signals. Control and data can be passed from entity to entity, but this is a function of the style of the VHDL code and not inherent in the language. In an object oriented approach a message is sent that may cause an action to occur in another object, or other messages to occur. In VHDL a signal has one of its drivers updated to a new value which may or may not change the signal value. If the signal value is changed, an entity monitoring the signal may perform an action in response to the value change. If this all sounds very vague it is because as stated earlier there is no built in message capability in VHDL. Let's look at an example of the counter shown earlier written with a message oriented communication scheme in VHDL[2]. This is shown in Fig 4.

```
package oppack is
   type optype is (reset, increment, outval);
   constant maxval : integer := 15;
end oppack;

use work.oppack.all;
entity counter is
   port (operation : in optype;
         ret_val : out integer);
end counter;

architecture message of counter is
begin
    process(operation'transaction)
        variable value : integer := 0;
    begin
       case operation is

           when reset =>
              value := 0;

           when increment =>
              if (value >= maxval) then
                  value := 0;
              else
                  value := value + 1;
              end if;

           when outval =>
              ret_val <= value;

       end case;
    end process;
end;
```

This example shows one of a number of possible methods to implement a message-like

scheme in VHDL. Package oppack is used to define the message values that other entities can use to communicate with this entity. These message values are defined by type declaration optype.

The counter entity contains a port clause with two ports declared in it. Port operation is the message port that will be used to determine what operation is performed. Port ret_val is used to return the value of counter when asked for by the outval message.

Only one architecture exists for entity counter. The architecture contains a single process statement that is sensitive to transactions on port operation. It is necessary to be sensitive to transactions because this port could easily have more than one increment operations sent in succession. These successive increment operations would not cause an event to occur on port operation, only a transaction. Since a process statement needs an event to trigger its execution, successive increment operations would be ignored. Inside the Process statement a single Case statement decodes the message and performs the appropriate action.

This example shows one way to implement a message scheme but has a few drawbacks. In the C++ example the message and any parameters are considered as a single item, but in the VHDL example the message and the message parameters are split. One way to fix this problem would be to use a record type for the type of message and the data passed in and out. The problem with this approach is that the single record type port would then need a portmode of inout creating much more inefficient code and the evaluation of a lot more resolution functions.

Another drawback of this scheme is that this description is not synthesizeable. Constructs such as 'transaction are currently not supported by synthesis tools and not likely to be in the near future.

Section 6. VHDL Encapsulation

From an object oriented point of view VHDL does have a good concept of encapsulation. Object oriented techniques encourage very well defined interfaces between objects to promote reusability and data protection. In VHDL each entity can only communicate with another entity by a well defined interface via signals. Any local data contained in an entity such as local variables declared in process statements can only be seen within the process. Any data passed between entities is passed through signals.

Section 7. VHDL Inheritance

Inheritance in an object oriented environment allowed a new object to be derived from another. The new object would inherit all of the functionality of the original object and yet be able to add new functionality. True object oriented languages contain inheritance, while object based languages[4] such as VHDL, ADA, and Object Pascal do not. In VHDL an entity can have multiple architectures but each new architecture is separate from another. Each new architecture cannot build on a previous one.

To create the ldcounter example in VHDL while retaining the other entity would require an entirely new architecture as shown in Figure 5.

```
package oppack is
   type optype is (reset, increment, outval, load);
```

```
   constant maxval : integer := 15;
end oppack;

use work.oppack.all;
entity ldcounter is
   port (operation : in optype;
         ld_val : in integer;
         ret_val : out integer);
end ldcounter;

architecture message of ldcounter is
begin
    process(operation'transaction)
        variable value : integer := 0;
    begin
       case operation is

           when reset =>
              value := 0;

           when increment =>
              if (value >= maxval) then
                  value := 0;
              else
                  value := value + 1;
              end if;

           when outval =>
              ret_val <= value;

           when load =>
              value := ld_val;

       end case;
    end process;
end;
```

To add the new functionality required the modification of optype to add the value load, modification of the case statement to add the new functionality, and the addition of a new port to receive the load value.

Another area where VHDL does have a notion of inheritance is entity passive process statements. Passive process statements contained in an entity are used for timing checking, error detection, etc. These processes can reference any entity ports, signals and local process data but cannot assign values to any ports or signals. Every architecture of the entity will inherit the behavior of these passive processes. This allows the user to write one version of a setup or hold detection passive process and share it among a number of architectures of an entity.

VHDL 92

In 1992 VHDL will go through another revision process. There are a number of proposed

changes, but only one that seems to relate to moving the language to a more object oriented point of view. This proposal relates to packages in VHDL. The change will allow packages to contain private types. Private types will allow type information to be local to a particular package. To access the data will require a function or procedure access effectively encapsulating the data.

Conclusions

VHDL can be written in an object oriented manner by writing it in a particular style. It is not a wrong way of writing VHDL, but probably not too useful. To really take advantage of the object oriented paradigm, VHDL needs to have some new semantics added to the language especially in the area of inheritance.

References

[1] S. B. Lippman. *C++ Primer Second Edition.* Addison Wesley, Reading , Mass, 1991

[2] D. L. Perry. *VHDL.* McGraw Hill, New York, NY, 1991

[3] B. Stroustrup. *The C++ Programming Language.* Addison Wesley, Reading, Mass, 1986

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