

OO-VHDL: An Object Oriented VHDL

Burton M. Covnot
David W. Hurst
Sowmitri Swamy

Vista Technologies, Inc.
1100 Woodfield Rd.
Schaumburg, IL 60173
email: {burt, hurst, swamy}@vistatech.com

Abstract

As part of the Rapid Prototyping of Application-Specific Signal Processors (RASSP) program, Vista, along with Martin Marietta Labs (Moorestown, NJ), has identified new object-oriented (O-O) constructs which can be implemented on top of VHDL with the aid of a language pre-processor. These object-oriented VHDL enhancements will be incorporated into the RASSP methodology's goals of using object-oriented analysis/design, rapid prototyping, and performance modeling to achieve major productivity gains. The preprocessor approach allows models defined in the new O-O extension language, OO-VHDL, to be used in conjunction with existing VHDL simulators and other tools.

1.0 Introduction

The RASSP program is developing an environment for the rapid design of signal processing architectures. One of the key goals of RASSP is to use VHDL as the central design and specification language for DSP systems. To accommodate the RASSP top-down design goals, Vista Technologies is developing high level modeling constructs appropriate for abstract system, board, and component level modeling.

Several hardware and software languages were studied to identify useful features/constructs which may be

applied to VHDL. Features were chosen to simplify the creation, use, and reuse of behavioral and architectural level models. The object-oriented model, therefore, was a natural choice, since it meets all of these requirements

It is important to note that we are *not* defining a new language based on VHDL, rather we are identifying new constructs which can be implemented on top of VHDL with the aid of a language pre-processor, as shown in Figure 1. A pre-processor implementation will generate simulatable VHDL code enabling existing VHDL tools to be leveraged. OO-VHDL supports system-level design and behavioral modeling. Applications include rapid prototyping, hardware/software co-design, and performance modeling. High-level models can be used as executable specification for lower levels.

OO-VHDL has been used to model the IEEE 802.4 Token Passing Bus Access Method for standard LANs and an example signal processing system, ESPS, developed by Martin Marietta Labs. The ESPS and its components are similar to the type of components that have been used in signal processing systems that RASSP is interested in developing. These models demonstrate the effectiveness of OO-VHDL for rapid prototyping of complex models, for modeling hardware/software interactions, and for synchronizing access of concurrently executing objects.

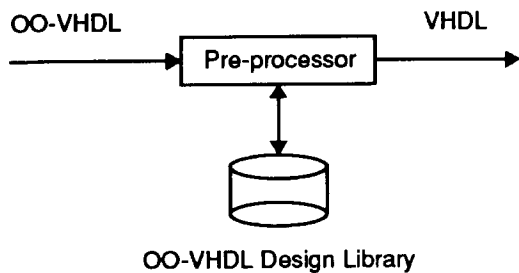


Figure 1: The OO-VHDL Pre-processor

2.0 Elements of the O-O Approach

Object-oriented languages contain capabilities useful in designing, building, and maintaining large systems. These capabilities include encapsulation, reusability, inheritance, and message passing. An object-oriented design is one based on objects, not algorithms.

Objects are typically described in two parts: an interface part and an implementation part, roughly corresponding to a VHDL entity declaration and architecture body. The object interfaces, known as *class description* in O-O terminology, documents the operations performed by an object. By reading a class description, it is easy to determine the functionality of a component, assuming, of course, that descriptive function names are used. Although one could inspect the implementation in an entity's architecture body, the design summary provided by the class description would be a quicker read, and more readily understandable.

An object-oriented approach increases the potential for component reuse. For example, suppose a simple behavioral description of a highway/farmroad traffic light controller exists, and left turn signals need to be added. The traditional approach to this problem would be to copy the old behavioral description and start modifying it. There are two major problems with this approach:

- VHDL code is replicated, making it difficult to propagate improvements made to the base design, and
- the resulting component description becomes unnecessarily complex, obfuscating the differences between similar components.

An object-oriented design process overcomes these problems by factoring out the common functionality of similar components in an *inheritance* hierarchy. Without inheritance, reuse can only occur at the component level, that is, either you must use the component “as is”, or you must design a new one.

Thus, the object-oriented paradigm economically expresses reusable components through the combination of inheritance and data encapsulation. In O-O terms, an *inheritance hierarchy* is composed of *classes*, classes define *objects*, and objects are *instantiated* before or when they are used. Furthermore, objects may contain *instance variables*, *class variables* and internal (private) and external (public) functions. These functions are commonly referred to as *methods*, and the phrase “sending a message” is O-O terminology for invoking a method. Instance variables are only visible within an object, while class variables are visible to all objects of a particular class.

Object-oriented language features can improve the productivity of VHDL in several ways:

- *Inheritance* increases the potential for component reuse.
- *Class descriptions* increase component comprehension.
- *Message passing* increases the designer's behavioral expressive power.

What does VHDL need to be object-oriented? According to Grady Booch, an object oriented language must provide for the following [1]:

- Abstraction -- the ability to define abstract behavior or “black boxes”,
- Modularity -- the ability to group abstractions,
- Encapsulation -- enables data/code hiding,
- Hierarchy -- allows is-a relationships (subclassing), and
- Messages -- used to communicate with an object.

3.0 Abstraction, Modularity, & Encapsulation

VHDL already has, to some degree, abstraction, modularity, and encapsulation. VHDL entities, procedures, and functions support abstraction, while packages and files provide modularity. The encapsulation units provided by VHDL, packages and entities, are, by themselves, not general enough to represent objects because it is not possible to encapsulate (hide) the values of vari-

ables/signals from the users of a package; and while an entity permits encapsulation, its abstraction capabilities are limited to port declarations.

Packages allow functions, type definitions, and (global) signal declarations to be grouped as a unit or module. You cannot, for example, share the value of a variable or signal between the procedures and functions of package without declaring the data item to be global. That is, packages do not provide module variables¹; hence, since data hiding is not possible, packages do not support the encapsulation of data. For entities performing more than one function, it is not possible to identify, using language constructs, which ports are used by which function. This prevents the definition of abstract *interfaces* to entities.

Nevertheless, an *object* must allow *some or all* of its procedures, functions, and variables to be visible from the outside. Hence VHDL packages cannot be objects, since all of its data must be global to persist between procedure/function calls, and neither can entities be objects, since procedures and functions defined inside of the architecture body cannot be accessed from outside. A new type of abstraction/encapsulation unit is required: EntityObjects. EntityObject descriptions are converted to standard, simulatable VHDL through a pre-processor.

As its name suggests, the EntityObject is based on entities. Entities were chosen over packages as the means of encapsulation because:

1. Entities are the primary abstraction used to model hardware components. The VHDL entity has ports representing pins, signals representing wires, and an independent execution thread to hold the state, or current values, of the component.
2. Entities maintain their own state without relying on global signals/variables. An architecture body may define signals to hold values between executions. Moreover, architecture bodies may have *process* statements, which in turn can define variables to hold values between executions.
3. It is not possible to “hide”, or encapsulate, implementation specific data in a package. All of the signals/variables defined in a package are accessible to anyone using that package.

1. In C, for example, module variables are implemented using *external static* variables. See [3] for more information on the properties of static variables.

To demonstrate the EntityObject, consider a simple component, the Counter, as described by Douglas Perry in [5]. The Counter has the following capabilities: increment, reset, and return the current value. Since each operation corresponds to a function, the class description documents the operations of the object, and hence its behavior. Contrast this with pure VHDL, where the user of an entity must know the set of ports to excite to obtain the desired action. That is, using an EntityObject, component behavior can be described abstractly, in terms of functions, not just signal, or port interfaces and low-level handshaking protocols. The OO-VHDL version of the Counter is shown below. Note that the abstract interface to the EntityObject (*the operations*) are similar in appearance to subprogram specifications.

```
EntityObject Counter is
  -- "Entity" part
  port (clock : in Bit );
  generic (maximum_value : integer );

  -- "Object" part
  operation Increment;
  operation Reset;
  operation GetVal (out_val : out integer);
end EntityObject Counter;

architecture pure_oo_behavior of Counter is

  -- Define a variable to hold our current state.
  -- "Instance" variables are visible to all
  -- operations defined in this class.

  instance variable current_val : integer;

  operation Increment is
  begin
    if ( current_val >= maximum_val ) then
      current_val := 0;
    else
      current_val := current_val + 1;
    end if;
  end;

  operation Reset is
  begin
    current_val := 0;
  end;

  operation GetVal (out_val: out integer) is
  begin
    out_val := current_val;
  end;

begin
  -- null architecture body
  -- Any concurrent VHDL statement may appear
  -- here, however.
end pure_oo_behavior;
```

Note that ports and generics, if required, can be used in an EntityObject. This is necessary, when, for example, an EntityObject must interact with a pure entity at the same design level. The clock port defined in the Counter object is for illustrative purposes only, i.e., it is not used in the Counter architecture body.

4.0 Inheritance

All elements of an EntityObject are inherited when subclassed. These elements include: declarations, all defined operations, any user specified ports or generics, and statements appearing in the architecture body. A subclass can change the definition of most of these elements simply by reusing the element's name, or in the case of a concurrent statement, the statement's label. As an example, consider the LoadCounter, a subclass of Counter.

```
EntityObject LoadCounter is new Counter
-- Note: "is new" denotes refinement
-- or subclassing.

-- "Entity" part is inherited from Counter

-- "Object" part
-- override Counter's Reset operation
-- and define a new operation
operation Reset;
operation LoadVal (in_val : in integer);
end EntityObject counter;
```

Since the LoadCounter subclasses the Counter, it inherits the GetVal and the Increment operations, the clock port, and the maximum value generic. Since inherited elements are visible in the subclass, the LoadVal operation can reference the Counter generic maximum_value. For example, LoadVal may be specified as:

```
operation LoadVal (in_val : in integer) is
begin
  if ( in_val > maximum_value ) then
    current_val := maximum_value;
  else
    current_val := in_val;
  end if;
end;
```

As mentioned previously, concurrent statements and declarations found in the architecture body are also inherited. If a concurrent statement is labeled in the superclass, a subclass can replace it simply by defining a new statement using the same label. Ports and declarations, however, cannot be overridden since they are, pre-

sumably, required for proper operation of the defining superclass.

Inheritance also applies to regular entity declarations. Entities follow the same inheritance rules as EntityObjects, except, of course, that inheritance of operations does not apply. Consider the following example. Suppose we have a nand gate component and we want to count the number of outputs from the component. Using inheritance, we can simply extend the behavior of the nand gate. This can be done in one of two ways: create a new component by subclassing the nand gate or extend the nand gate's architecture body. Both approaches are shown here. Below is the basic nand gate component.

```
-- The nand gate definition.
entity nand_gate is
  port (in1, in2 : in Bit; output : out Bit);
end nand_gate;

architecture nand_behavior of nand_gate is
begin
  ll:
    process(in1, in2)
    begin
      out <= not (in1 and in2);
    end process;
end nand_behavior;
```

A new component, nand_gate_acct (derived from [4]) as a subclass of nand_gate appears below.

```
-- The nand gate acct definition.
entity nand_gate_acct is new nand_gate
-- All ports inherited from nand_gate.
-- We could have defined new ones
-- specific to this entity, but it was not
-- necessary for this component.
end nand_gate_acct;

architecture acct_behavior of nand_gate_acct
  is new nand_behavior
begin
  -- The process block Ll, defined in the
  -- nand_behavior body, is inherited from
  -- nand_behavior (as indicated by the is new
  -- clause).

  -- Count the number of times we're executed.
  account:
    process
      variable count: integer := 0;
    begin
      wait on output'transaction;
      count := count + 1;
    end process;
end acct_behavior;
```

If a new component is not desired, just the architecture body of `nand_gate` can be extended. The architecture body in this case is the same as above except that the entity specified in the “of” clause is now `nand_gate`, e.g.,

```
architecture acct_behavior of nand_gate
    is new nand_behavior
begin
    same account process as in previous example
end acct_behavior;
```

Inheritance facilitates reuse. Without it, the extended `nand` component would have to be written by copying the behavior of the original component into a new component, increasing maintenance effort of both since improvements added to one may apply to the other.

5.0 Message Passing

Operations define an abstract procedural interface to an entity. Messages are used to invoke the operations. Like object-oriented software languages, when an OO-VHDL message is sent, the invoker is blocked until the corresponding action (the operation) completes. Thus, from the sender’s point of view, sending a message has the semantics of calling a procedure. This differs significantly from the hardware model which requires specialized protocols to synchronize behaviors and to exchange data between components. The `EntityObject`, by providing a procedural interface to an entity, is well suited for the rapid construction of abstract models found in many top-down VHDL design methodologies as well as for performance modeling.

However, unlike procedure calls, a message only *requests* that a particular operation be performed, it does not cause the operation to execute immediately. An `EntityObject` services one message at a time, i.e., operations execute sequentially. To enforce sequentiality, OO-VHDL queues all message requests sent if an `EntityObject` is actively servicing another operation request. When the current operation completes, the next request will be removed from the queue and serviced. For reliability, OO-VHDL uses an internal message-send protocol to ensure that multiple messages sent to the same `EntityObject` at the same time are not lost.

Messages can be sent from an architecture body to a component `EntityObject` or from one `EntityObject` to another at the same design level. Since messages are not broadcast to all `EntityObjects`, a mechanism is required

to distinguish one EO from another; this mechanism is “naming.” In object-oriented terminology, object names are referred to as *handles*. When an `EntityObject` is instantiated, the designer must specify a name for it. The name is used to send messages to that particular `EntityObject`. Below shows an example of a test bench entity using a `clock_gen` and a `Counter` component.

```
entity test_bench is end test_bench;

architecture test1 of test_bench is
    component clock_gen -- declare an entity
        (clock : out Bit)
    end component;

    ObjectComponent counter -- declare an EO
        (clock : in Bit)
    end ObjectComponent;

    signal out_val_1 : integer;
    signal clock_val : Bit
begin
    -- instantiate an entity
    cl: clock_gen
        port map (clock_val);

    -- Instantiate a counter object; a_counter
    -- is its handle. Port mappings are
    -- specified in the same manner as
    -- entity ports.
    object a_counter: counter
        port map (clock_val);

    process
    begin
        -- Send messages to the counter
        -- identified by the handle a_counter
        send a_counter :Reset;
        send a_counter :Increment;
        send a_counter :GetVal(out_val_1);
    end process;

end test1;
```

`EntityObject` handles may also be stored in variables. This permits, for example, an algorithm to send messages to a dynamic set of objects without having to specify each object handle directly. The for-loop in the architecture body below depicts this.

```
architecture test2 of test_bench is
    :
    :
begin
    object a_counter: counter
        port map (clock_val);
    object b_counter: counter
        port map (clock_val);
    object c_counter: counter
```

```

    port map (clock_val);

process
    variable object_list is array (0 to 2)
        of EO_Handle;
begin

    object_list(0) := a_counter;
    object_list(1) := b_counter;
    object_list(2) := c_counter;

    for i in 0 to 2 loop
        send object_list(i) Reset;
        send object_list(i) Increment;
        send object_list(i) GetVal(out_val_1);
    end loop;

end process;

end test2;

```

OO-VHDL, like many other object-oriented languages, supports *polymorphism*. That is, if two or more Entity-Object have operations with identical *signatures* (same name and same parameter type profile), operation requests may be sent to instances of these EO's using the same message statement. As an example suppose the EO's Counter, LoadCounter, and Timer all have a parameterless reset operation defined. The following procedure, *ResetAll*, can then be defined and used as follows:

```

architecture test2 of test_bench is

    -- EntityObject declarations not shown --

    procedure ResetAll (object_list :
        array ( integer range <> )
            of EO_Handle) is
    begin
        for i in object_list'range loop
            send object_list(i) Reset;
        end loop;
    end ResetAll;
begin -- body

    object a_counter: counter
        port map (clock_val);
    object a_load_counter: load_counter
        port map (clock_val);
    object a_timer: counter
        port map (clock_val);

    process
    begin
        variable object_list is array (0 to 2) of
            EO_Handle;

        -- Store three objects, each from a
        -- different EntityObject class in the

```

```

        -- object array.
        object_list(0) := a_counter;
        object_list(1) := a_load_counter;
        object_list(2) := a_timer;

        -- Call procedure to reset all of objects
        -- in the array.

        ResetAll(object_list);

    end process;

end test2;

```

The procedure *ResetAll* exhibits polymorphism since it does not know, at analysis time, what kind of objects it will be sending *Reset* message to. OO-VHDL routes the message to the appropriate object dynamically during a simulation run. If an object passed into *ResetAll* does not have a *Reset* operation defined for it, a run-time error will result.

6.0 Concurrency Control

Each VHDL process statement represents an execution thread, which, conceptually, runs in parallel with other process statements. To exploit this concurrency at the behavioral level, high-level communication and synchronization language constructs are required. Concurrent programming constructs allow designers to focus on the design, not the details of data mutual-exclusion or process synchronization of their algorithmic models, especially important when developing rapid prototypes or executable specifications.

VHDL provides two types of low-level concurrency control constructs: those that suspend execution until some particular expression becomes true and bus-resolution functions. Since the former relies only on changing signal values, the designer must introduce new signals to control the execution order of independently running processes. The latter construct, bus-resolution function, provides no support for process synchronization, since they only handle signal assignment clashes; if two assignments to the same signal occur at the same time, the signal's bus-resolution function examines the proposed values and returns the "strongest" value or, perhaps, an entirely different value altogether. Both of these construct types operate at the *signal* level, not at the execution-thread or *process* level, necessary to model high-level behavior in VHDL; for this reason, concurrent programming constructs are required to support concurrency at the *object* level.

Of the existing concurrency control approaches, the Distributed Processing, or DP, model (similar to the remote procedure call model) and the Ada rendezvous are the most desirable for behavioral modeling. These approaches were chosen, because of their ease of use and generality. (See [2] for a survey of concurrent programming models.) Thus, OO-VHDL contains a combination of the DP and the Ada approach; this combination is referred to as DP/A in OO-VHDL.

To overcome the deficiencies of Distributed Processing, as identified in [2], Ada rendezvous constructs were added to the base DP model. These constructs allow:

- invoked operations to release the caller at will, instead of always waiting until the operation completes (as in DP),
- explicit ordering of operation invocation (unlike DP) via Ada-like **select** and **accept** statements.

6.1 DP in OO-VHDL

Distributing Processing, like many concurrent programming models, relies on the ability of processes, or tasks, to be suspended and queued. The OO-VHDL preprocessor generated code performs this function. In addition, it is also desirable to set priorities on incoming messages. In the event that two messages are sent at the same time, one may be selected to be executed over the other in a deterministic fashion. Below is an example of the Counter object declaration with DP. Notice that, besides the optional setting of operation priorities, the Counter class (including its architecture bodies) remains unchanged.

```
EntityObject Counter is
  -- "Entity" part
  port (clock : in Bit );
  generic (maximum_value : integer );

  -- "Object" part
  operation Increment <priority 2>;
  operation Reset <priority 1>;
  operation GetVal(out_val : out integer);
end EntityObject Counter;
```

6.2 DP/A in OO-VHDL

To support the Ada-like concurrency model, an additional language concepts is required: message-flow primitives, in the form of Ada-like **accept** and **select** statements. The **accept** statement allows a process to only receive messages of a specified type, while the **select** statement permits more than one **accept** statement to be active at a time. For example:

```
loop
  select
    accept GetVal
    begin
      statements
    end;
    :
  or
    accept Increment
    begin
      statements
    end;
  end select;
end loop;
```

In this example, either a GetVal or an Increment message will be accepted and executed; other messages, such as Reset, if sent, will remain queued and their senders will be suspended. If the code was instead written as:

```
loop
  accept GetVal
  begin
    statements
  end;

  accept Increment
  begin
    statements
  end;

end loop;
```

then every GetVal message would *have* to be followed by an Increment message.

Unlike the Ada **accept** statement, no arguments can appear in an OO-VHDL **accept** statement. If a begin-end block appears after an **accept** statement, it will be executed when the **accept** statement completes. The begin-end block may contain another **accept** statement.

Using the **accept** statement, an EntityObject can suspend message execution until the *arrival* of a particular message and the *completion* of its corresponding EO operation. Thus, all incoming message requests are queued until the specified operation completes. This is the semantics of DP/A. Upon completion of the **accept** statement, including the begin-end block, the EO will revert to using DP semantics, i.e, DP/A semantics apply only while an **accept** statement is either waiting or executing its optional begin-end block.

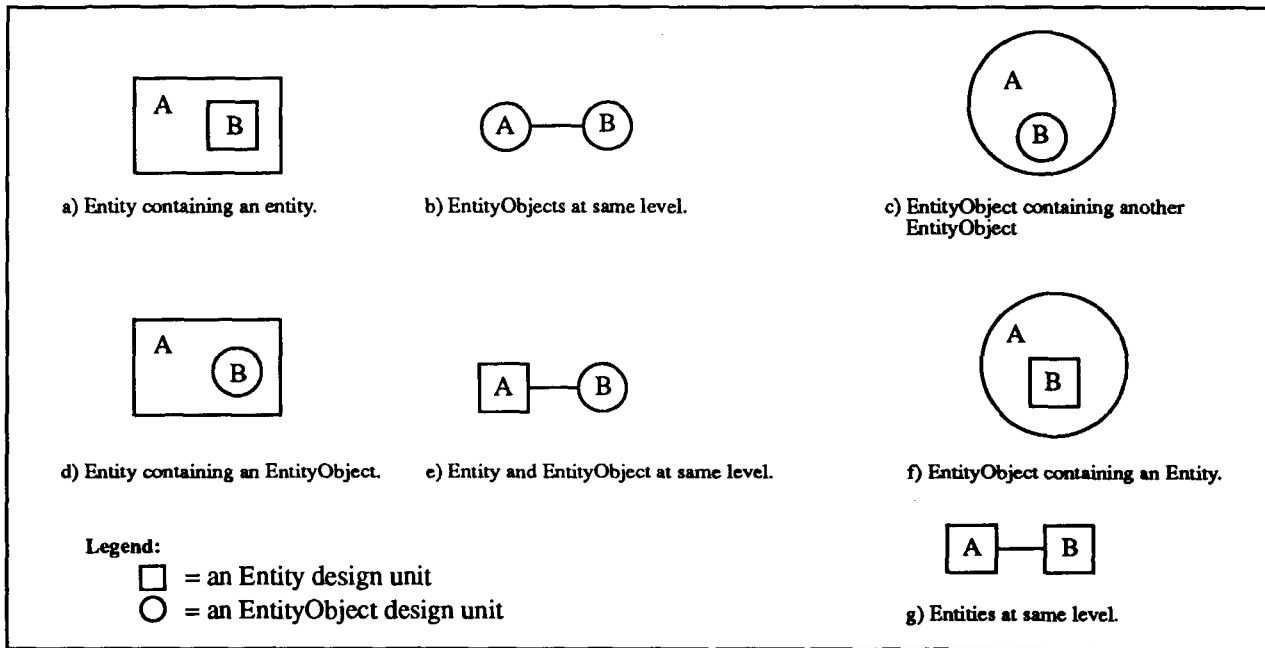


Figure 2: Possible Entity and EntityObject Communication Patterns

7.0 Combining Entities and EntityObjects

It is our intent that OO-VHDL EntityObjects be allowed to mix freely with normal VHDL entities. That is, an entity, or more precisely, an architecture body, may contain or use an EntityObject and vice versa. Figure 2 shows the various possible, and allowable, combinations.

8.0 OO-VHDL in Use

8.1 Token Bus

Our first example modeled the IEEE 802.4 Local Area Network Token-passing Bus Access Method. The token bus protocol is a standard for managing communication on a network configured as a physical bus. This configuration usually operates as a broadcast media, that is, each station can receive all signals transmitted on the bus. Contention on the network is resolved by passing around a logical token. The station which currently holds the token controls the bus and has the right to transmit data. Each station is only allowed to hold the token for a limited period of time. When this period expires, the station must pass the token on to its successor. Sequencing of the stations is in order of descending network addresses and then from the lowest address to the highest address, so the token circulates around all of the stations on the network in a logical ring. In the

steady state, operation alternates between transferring data and transferring the token.

We developed a model of the token bus protocol in stages, with each successive stage showing increasing detail. Each level of detail was implemented as a separate model. This approach allowed us to use the interfaces defined in each stage as the specification for the design in the next stage. A common network model application program interface for sending and receiving data frames from one station to another was defined, allowing the simple producer/consumer application to be used without change as we moved from one stage to the next. This simplistic application program runs on each station and simply generates and receives data frames, sending to and receiving from the other stations on the network at random. When producing data, each application program selects the destination station at random and determines a random number of data frames to be sent to that station. Since we were modeling the behavior of the communications protocol, we were able to neglect the actual content of the data. When all of the data frames have been successfully transmitted, the application selects a new station at random. When receiving data frames, the application program keeps track of the origin of the data and counts the total number of frames received from each station.

The staged approach allowed us to focus on a different set of problems at each stage. The four stages are described in the following paragraphs.

Concurrent Programming Model. In the first stage, the token bus protocol is neglected completely and network communication is modeled entirely using concurrent sequential processes. We used this stage to focus on building the application program which was used to drive the token bus model in the subsequent stages.

Sequential Bus Control Model. The second stage implements a simple centralized scheme for resolving bus contention by passing a token from station to station. In this model, the Medium Access Control (MAC) service specification was implemented, but the MAC protocol itself was neglected.

Full Token Bus Protocol Model. The third stage implements the complete MAC layer protocol for the token bus access method.

RTL-Level Behavior Model. The final stage consists of an RTL-level simulation of the MAC hardware and software drivers. This implements the lower part of the MAC layer as a hardware simulation. This level of detail was not modeled in the project.

OO-VHDL was an excellent tool for modeling this problem. In handling the staged approach, we created an abstract superclass which defined the standard interfaces to the MAC component which was then subclassed for each of the four stages. This allowed us to share the common parts of the interface and concentrate on the details of the model. The use of OO-VHDL messages tremendously simplified the task of modelling communications, in that we could ignore bit-level protocols for transferring data and focus on the higher-level issues.

8.2 Example Signal Processing System

A model of an Example Signal Processing System (ESPS) was developed under the RASSP project as an aid to exploring enhancements to VHDL and design tools. The ESPS consists of numerous processing elements (PE's) and a matrix of switching elements (SE's) connecting the processors to each other and to system I/O. Being modular, the system may be configured in many different ways. It is also scalable and can accommodate anywhere from two to over 1,000 processing elements.

In the ESPS, processor elements send messages to other PE's via the switches. The switches decode routing information embedded in the messages and route them from one port to another. Data is relayed from one switch to another as soon as a connection is available. If there are no contentions, the data transfer will resemble a point-to-point transfer because every connection in the path is actively transmitting data. If there is a contention, the transfer is suspended and data is stored in the switches prior to the contention point and will be forwarded when the contending transfer completes.

One of the design goals was to compare and evaluate different topological configurations as they apply to specific applications. To accomplish this, the system topology, the software application, and the component behavior all had to be modelled orthogonally. The system topology is the structural interconnection of the basic processor and switching elements. The software application describes the sequence of mapping and communication operations carried out by each of the processor elements in the system. The component behavior is the description of what a component does given specific inputs or program operations. Orthogonality allows us to mix and match components independently.

The OO-VHDL model simulates the ESPS at the packet level. This model neglects the low-level details of transmitting words between switching elements and deals entirely at the level of exchanging packets. Packet transfers are performed using OO-VHDL messages rather than signals and complicated protocols. By abstracting out the details required to model word-level data transmission, the model becomes much simpler and easier to understand. In addition, we were able to completely avoid all of the problems inherent in using VHDL bus-resolution functions for signals carrying complex data types.

There are three basic components in the OO-VHDL ESPS model. An abstract class, `NetElement`, defines the common protocol for transmitting packets. A single operation, `PacketTransfer()`, is used to send packets from one `NetElement` to another. Two subclasses inherit from `NetElement`. These are `SwitchElement` and `ProcessorElement`. The `SwitchElement` entity object implements the behavior required to route packets from each of its four ports to the other three. The `ProcessorElement` implements the behavior necessary to interpret a program, to originate packet transmission, and to receive packet transmissions.

Since the ESPS simulation does not perform any actual computation or interpret any actual data, no data is actually transmitted between processor and switching elements. Each packet consists of only the necessary header and control information, including its priority, routing list, packet id, packet size, message id, message length, source pid, and destination pid. Transmission delays are simulated by waiting during the transmission operation for an amount of time proportional to the purported size of the packet. Packets are transmitted atomically. Each transmission either succeeds or it fails in toto. No pre-emption is supported. If it fails, there is no partial transmission. The sender must try again later to send the entire packet. Transmissions are throttled on a full packet basis. If a receiver cannot accept a packet because its send buffers are full, it will deny the transmission, causing the transmission to fail.

The resulting O-O model is significantly smaller than the full bit-level model would be and allows the dynamics of the entire array of processor and switching elements to be simulated without the overhead of the bit-level transactions. We felt that this model would still be able to provide valuable information about the performance of the system under different configurations. Congested areas in the switching array could be identified and routes adjusted to compensate. The model itself was built very quickly and is little more than six hundred lines of code in length. On the downside, OO-VHDL would not be very effective for constructing an accurate bit-level models. The message medium is best utilized for requesting operations to be performed and, therefore, is not appropriate for representing input and output values of a component. In addition, timing accuracy cannot be guaranteed due to queuing. In retrospect, OO-VHDL was seen as a very powerful tool for doing system-level behavioral modeling and for rapid prototyping.

9.0 Conclusion

Various programming and modeling constructs were examined to enhance the modeling capability of VHDL. The new language incorporating these constructs, OO-VHDL, is object-oriented and includes support for high-level behavioral modeling and support for concurrent programming. Hardware and/or software modeled in OO-VHDL can benefit from the abstraction, encapsulation, inheritance, and message passing of an object-oriented design. These benefits include improved model readability, increased potential for reuse, and lower maintenance costs.

OO-VHDL will be converted to plain VHDL via a pre-processor, invoked just before the model is analyzed. Thus, models written in this extension language can run on current VHDL simulators and may also be operated on by other VHDL tools.

Acknowledgments

This work was performed under RASSP Contract No. DAAL01-93-R-3616, under contract from Martin Marietta. We would like to thank Martin Marietta Corporation for their support and Carl Hein for providing us with the ESPS model and for many helpful suggestions.

REFERENCES

- [1] Booch, G. *Object Oriented Design*, Benjamin/Cummings Publishing, Redwood City, CA. 1991.
- [2] Gehani, N. *Ada: Concurrent Programming*, Prentice-Hall Inc., Englewood Cliffs, NJ. 1984.
- [3] Kernighan, B. W. and D. M. Ritchie. *The C Programming Language*, Prentice-Hall, Englewood Cliffs, NJ. 1988. (2ndEdition)
- [4] Lipsett, R., C. Schaefer, and C. Ussery. *VHDL: Hardware Description and Design*, Kluwer Academic, Norwell, MA. 1989.
- [5] Perry, D. "Applying Object Oriented Techniques to VHDL", *VIUF Conference Proceedings*, VHDL International, Spring 1992.
- [6] Vista Technologies. "OO-VHDL Language Reference", RASSP Technical Report, TR-1.2.11.1.3-01.