# PRINCIPLES FOR LANGUAGE EXTENSIONS TO VHDL TO SUPPORT HIGH-LEVEL MODELING*

*Peter J. Ashenden*
Lecturer
Dept. of Computer Science
The University of Adelaide
Adelaide, SA 5005
Australia

*Philip A. Wilsey*
Assistant Professor
Dept. of ECECS, PO Box 210030
University of Cincinnati
Cincinnati, OH 45221-0030
USA

## ABSTRACT

This paper reviews proposals for extensions to VHDL to support high-level modeling and places them within a taxonomy that describes the modeling requirements they address. Many of the proposals focus on object-oriented extensions, whereas this paper argues that extension of VHDL to support high-level modeling requires a broader review. The paper presents a detailed discussion of issues to be considered in adding high-level modeling extensions to VHDL, including concurrency and communication, abstraction using entity interfaces, object-oriented data modeling, encapsulation, signal assignment semantics, shared variables, multiple inheritance, genericity and synthesis. Emphasis is placed on the importance of designing simple orthogonal semantic mechanisms that interact in well defined ways, and that integrate cleanly with exisiting language features.

Keywords: high-level modeling, VHDL, object-orientation, genericity, concurrency, communication.

## 1. INTRODUCTION

In recent years, as the complexity of hardware systems has increased, designers have been forced to include high-level modeling as a stage in the design flow. Specifying and simulating systems at a high level of abstraction allows more reliable capture of requirements and more extensive exploration of the design

* This work was partially supported by Wright Laboratory under USAF contract F33615-95-C-1638.

space. High-level modeling and related activities have the potential to reduce development time and cost. However, in order to support high-level modeling, a design language must allow specification of data and behavior in an abstract manner [41]. It should not force the designer to make design decisions that are better deferred to later in the design flow. For example, it should not force the designer to choose between a hardware and a software implementation too early, or to specify a detailed communications protocol before partitioning the design.

Ideally, a design language should allow the designer to span the spectrum of abstraction from high-level down to implementation (invoking compilers or synthesis tools to realize the final implementation). It is desirable to a single design languages throughout the design flow, avoiding interface- and equivalence-checking problems that otherwise arise. VHDL [28], as it currently stands, is a hardware description language that is well-suited to modeling small- to medium-scale systems at levels of abstraction up to register-transfer level. However, it has some serious shortcomings when used for modeling large-scale systems or systems at a high level of abstraction. In order to remedy these shortcomings, complexity-management and abstraction techniques that have been used successfully in programming languages should be reviewed as candidates for extensions to VHDL.

An important development in the software engineering community is the movement towards object-orientation as a means of managing the complexity inherent in large software systems. Object-orientation allows a design space to be partitioned into manageable pieces with well-constrained interactions, and facilitates the reuse and evolution of modules. According to Booch, "object orientation involves the elements of *data abstraction, encapsulation,* and *inheritance with polymorphism*" in a language [8, page 181]. The ideas denoted by these terms are clearly defined and illustrated by Booch [7] and have be reviewed extensively by other authors, so we do not include any detailed discussion here.

Given the success of object-oriented techniques in the software domain, it is appropriate to consider their inclusion in a hardware description language such as VHDL. We expect that the use of object-oriented techniques in hardware description languages will help designers manage complexity, improve their productivity, and improve the reliability of the design process. However, object-oriented extensions should not be viewed as a panacea. Object-orientation focusses mainly on abstraction over data and the related operations, and does not address issues such as concurrency and communication. Successful extension of VHDL to support high-level modeling requires a broader review of these and other issues.

Our aim in this paper is not to present or support any particular extension to VHDL. Instead, we review previous proposals for extensions and attempt to categorize them to enable detailed analysis and discussion. Our categorization is based on the semantics of each proposal, insofar as the semantics are elabo-

rated in each proposal. We attempt to identify some pitfalls that can trap unwary players and show that many of the previously proposed extensions encounter the pitfalls and violate principles of good language design.

When considering extensions to a hardware description language, it is necessary to identify the requirements of the language users, and to ensure that proposed extensions meet these requirements. Bergé *et al* [6] report on a survey of several European telecommunication companies and system houses to discover their requirements of a design language. In particular, the respondents were asked to identify their current problems using VHDL and their expectations for object-oriented VHDL. The responses reported were:

- to target a higher level of abstraction for modeling,
- to simplify and speed up the process of specification,
- to ease the addition of new functionality,
- to improve the functional validation process,
- to improve the degree of reusability,
- to improve capability for documentation,
- to increase automation in the design flow,
- to improve consistency with object-oriented notations or languages used in hardware/software codesign, and
- to improve ease of learning and application.

The proposals for extension to VHDL surveyed in this paper meet these requirements to varying degrees.

The remainder of this paper is organized as follows. Section 2 presents some general principles for language design that we argue should be adhered to when developing proposals for language extension. Section 3 presents a taxonomy of approaches to high-level modeling extensions to VHDL and places

previously published proposals within the taxonomy. Section 4 discusses a range of issues that must be considered when designing high-level modeling extensions to VHDL and discusses the way in which previous proposals address the issues (or, in some cases, fail to address them). Finally, Section 5 concludes with a discussion of our plans to develop high-level modeling extensions to VHDL—hopefully avoiding the pitfalls along the way.

## 2. LANGUAGE DESIGN PRINCIPLES

The design of a programming language or a hardware description language is a difficult task. Since the language is the vehicle for expression of design intent, a good language can greatly help the design process, whereas a poor language can significantly hinder it. A language should conform to a set of ideals or philosophies to make it coherent, easy to learn, and easy to read and understand. This is what Brooks refers to as "conceptual integrity" [9]. We present here some views on language design principles that lead to high-quality languages. While many of these principles may appear to be common sense or general "motherhood and apple pie" statements, it is important to bear them in mind throughout the language design process. They are all too often overlooked, particularly when language design is conducted by a committee of diverse interests. As Brooks notes, "Conceptual integrity does require that a system reflect a single philosophy and that the specification as seen by the user flow from a few minds" [9, page 49].

### 2.1 Design of Semantics

The foremost principle is that language design should focus on semantics first and syntax second. The semantics of language features embody the meaning of the features, and determine what design intent can be expressed in the language. The benefits of a semantics-based language design methodology are illustrated by Tennent [46]. He comments that a methodological approach based on semantics is "intended to help a designer cope with the detailed problems of achieving consistency, completenes, and regularity in the design of specific language features," and "has the effect of drawing [the designer's] attention to deeper structural issues" in a language. Syntax, on the other hand, is the concrete expression of semantic features. While poor syntax may obfuscate the design intent, it does not prohibit expression of the intent. Good syntax design allows the designer to think about and communicate design intent clearly.

### 2.2 Simplicity of Mechanism

In determining semantic features to be included in a language, sufficient simple semantic mechanisms should be preferred over more complicated general solutions; the simple mechanisms can then be used to build application-specific solutions. The semantic mechanisms should, as much as possible, be ortho-

gonal to each other. As Hoare suggests [26], "concentrate on one feature at a time," and "reject any that are mutually inconsistent." By choosing simple orthogonal semantic mechanisms, interaction between mechanisms is reduced and easier to understand. Simplicity of mechanism and reduced interaction make it easier for tool builders to optimize their implementation of language features.

### 2.3 Design of Extensions

When extending an existing language, the preceding principles should be applied to the extensions. Simple semantic mechanisms should be chosen to augment the existing mechanisms, not to replace them. The new features should conform to the same design philosophies that were followed in the original language design so as to maintain architectural coherence. Careful consideration must be given to interactions between new features and existing features. While the semantics of new features are of primary concern, integration of new syntax is also important. Extensions should aim for stylistic consistency with the existing language. New features that are just syntactic rewrites of existing features ("syntactic sugar") should only be included if they significantly enhance the expressiveness of the language. As Wirth puts it [49], "distinguish . . . between what is essential and what ephemeral."

## 3. TAXONOMY OF PREVIOUSLY PROPOSED EXTENSIONS

Previous proposals for extending VHDL for high-level modeling have been couched in terms of object-oriented extensions, and have focussed on three areas of language usage: data modeling, structural modeling, and concurrency and communication (sometimes referred to as "system-level modeling"). These areas are also reviewed by Dunlop [16]. Table 1 summarizes the approaches adopted by each of the previously proposed extensions. We discuss the concepts in more detail in the following sections. Note that the examples shown in this and subsequent sections are intended only to illustrate the concepts. No concrete language proposal is implied.

**Table 1.** Summary of proposals for object-oriented extensions to VHDL.

| Proposal | Data Modeling | Structural Modeling | Concurrency and Communication |
|---|---|---|---|
| Mills [36] | Ada-95 approach (tagged type with single inheritance) | tagged entity/architecture with multiple inheritance | |
| Schumacher and Nebel [42] | Ada-95 approach (tagged type with single inheritance) | | |
| Dunlop [17] | illustrates general concepts using Ada-95 approach | | |
| Willis *et al* [48] | class-based with multiple inheritance | | implicit: class types for shared variables have monitor semantics |
| Objective VHDL: Radetzki *et al* [39] | class-based with single inheritance | entity classes with single inheritance | |
| Ecker [18] | | tagged entity/architecture with multiple inheritance | |
| Ramesh [40] | | entity classes with inheritance (only single inheritance illustrated) | |
| Mills [37] | | inheritance via configuration | |
| Vista OO-VHDL: Swamy *et al* [45] | | entity classes with inheritance (only single inheritance illustrated) | entity classes with operations (modified monitor semantics); inheritance (only single inheritance illustrated) |
| Benzakki and Djafri [5] | | entity classes with multiple inheritance | entity classes with operations (modified monitor semantics); multiple inheritance |
| Cabanis *et al* [11] | | | class-based with operations (concurrent invocation with ad hoc concurrency control); multiple inheritance |

### 3.1 Extensions for Data Modeling

Object-oriented extensions for data modeling address the way in which data values are described in a model. Currently, VHDL provides a type system similar to that of Ada, but with some simplifications. Proposals for extending VHDL suggest that this simplified type system is insufficient for modeling data with complex structure. They argue that object-oriented techniques for expressing data should be incorporated into VHDL to support modeling at a high level of abstraction. In particular, language features to support inheritance and polymorphism should be added, since they are key features to support object-

oriented programming. Two main approaches have been canvassed for object-oriented data modeling in VHDL: *programming by extension* and *class-based*.

***Programming by Extension*** The programming by extension approach involves adopting features of Ada-95 [32], and is the basis of proposals by Mills [36], and Schumacher and Nebel [42]. Dunlop [17] also illustrates the general concepts of object-oriented data modeling using this approach. It involves firstly defining a parent type as a tagged record, with primitive operations on the parent type defined as subprograms that include a parameter of the parent type. For example, the following code defines a parent type that represents a general CPU instruction and two operations on instructions:

```
type instruction is tagged record
      opcode : opcode_type;
   end record;
procedure check_opcode ( instr : in instruction;  . . . );
procedure perform_op ( instr : in instruction );
```

Next, the parent type is refined by deriving a new type with additional record elements. While the derived type inherits the primitive operations of the parent type, the operations can be replaced with alternative implementations. In addition, new primitive operations may be defined for the derived type. For example, a register-mode ALU instruction may be defined as a refinement of a general instruction, as follows:

```
type register_alu_instruction is new instruction
   with record
      dest, src1, src2 : reg_number;
   end record;
procedure perform_op ( instr : in register_alu_instruction );
```

The operation check_opcode is inherited and can be applied to values of the derived type. The operation perform_op is replaced with a new version that is specific to values of the derived type.

In the Ada-95 approach, the hierarchy of derived types forms the inheritance hierarchy required in object-oriented programming. The term class is used to refer to the tree of types derived from a given parent type. Polymorphic typing comes from declaration of objects of unconstrained class types, denoted using the attribute 'class. For example:

```
procedure execute ( instr : in instruction'class; . . . );
```

The formal parameter of this procedure is polymorphic, and may take a value of type instruction or of any type derived from instruction. Thus, within the procedure, the call

```
perform_op ( instr );
```

requires a dynamic check of the tag associated with the actual parameter value to determine the type of the actual parameter. The type of the actual parameter determines which procedure is called. This process is called "dispatching" or "late binding," as the type of the actual parameter is bound at run-time rather than being statically determined.

***Classes*** The class-based approach to object-oriented data modeling extensions in VHDL is influenced by Java [21], C++ [44] and their predecessors (notably, Simula [13]). This approach is followed by Willis *et al* [48], Objective VHDL [39], Vista OO-VHDL [45], Benzakki and Djafri [5], and Cabanis *et al* [11]. The approach involves the definition of classes that encapsulate the definitions of data and operations of objects. A class may inherit encapsulated data and operations from another class. The inheriting class is called a subclass and the parent is called the superclass. As an example, the instruction type shown above might be defined as follows (using a VHDL-like style of expression, approximately mirroring C++ semantics):

```
type instruction is class
    opcode : opcode_type;
    procedure check_opcode ( . . . );
    procedure perform_op;
  end class;
```

A value of this type contains an encapsulated opcode value that can be operated upon only by the procedures defined in the class. These procedures are often called "methods" or "operations." A subclass representing register-mode ALU instructions might then be defined as:

```
type register_alu_instruction is instruction class
    dest, src1, src2 : reg_number;
    procedure perform_op;
  end class;
```

The opcode value and the check_opcode operation are inherited from the superclass instruction, whereas the perform_op operation is overridden by a new version in the subclass. Objects are created as instances of these class types, and operations are invoked by naming a particular instance. For example:

```
variable instr_reg : instruction;
. . .
instr_reg.perform_op;
```

The encapsulated value for the instance of the class is implicitly available for use in the perform_op operation. The object denoted by instr_reg may be a member of the instruction class or one of its subclasses.

Thus, the invocation of the perform_op operation may require dynamic dispatching depending on the particular class of the object.

While a number of proposals are based on this approach, only Willis *et al* [48] and Radetzki *et al* [39] limit their discussion to its use for data modeling. Other proposals (cited later in this section) extend its use to system-level modeling. The driving motivation for incorporating classes in a language definition is to provide direct language support for the principles of object-orientation in a single language feature. Hence, a class is a unit of abstraction, encapsulation, modularity, hierarchy (through inheritance) and typing in this approach.

### 3.2 Extensions for Structural Modeling

Object-oriented extensions for structural modeling address the issue of reuse of hardware designs to form new designs. Design entities are viewed analogously to classes, with component instances being objects. The generic constants and ports defined in a design entity are properties of objects. The proposed extensions for structural modeling identified in Table 1 suggest that new design entities can be derived by inheriting generics and ports from a parent entity and adding new generics and ports. In addition, the process statements and component instances from the parent architecture body are inherited, and new processes and component instances are added to the derived architecture body.

There appear to be two approaches to object-oriented structural modeling, paralleling the two approaches to data modeling. However, the differences, insofar as they are described in the proposals, are syntactic rather than semantics-based. One approach, proposed by both Mills [36] and Ecker [18], involves using the keyword "tagged" to identify an entity or architecture that can be inherited. The approach proposed for Objective VHDL [39] is similar, but omits the keyword "tagged" and allows any entity to be inherited. Although the finer details of syntax vary between the proposals, an illustrative example is:

```
entity counter is tagged
   port ( clk, out_en : in bit;  q : out std_logic_vector );
end entity counter;
architecture behavioral of counter is tagged
   signal count : natural;
begin
   increment : process ( clk ) is
   begin
      if clk'event and clk = '1' then
         count := (count + 1) mod 2**q'length;
      end if;
   end process increment;
   drive :
      q <= To_vector(count) when out_en = '1' else
              (others => 'Z');
end architecture behavioral;
```

Here, the ports clk and q are properties of the entity that can be inherited, and the statements increment and drive are the properties of the architecture that can be inherited.

The proposals use the keyword "new" to specify inheritance into a derived entity or architecture. Ecker's proposal does not clearly indicate whether an inheriting architecture inherits from a parent architecture of the parent entity, or from alternative architectures of the derived entity. Examples of both cases appear in the proposal without comment on the distinction. The Mills' proposals and Objective VHDL, on the other hand, clearly indicate that a derived architecture inherits from a nominated architecture of the parent entity. For example (not using syntax of either proposal):

```
entity resettable_counter is new counter with
   port ( reset : in bit );
end entity resettable_counter;

architecture resettable_behavioral of resettable_counter is new behavioral with
begin
   increment : process ( clk, reset ) is
   begin
      . . .
   end process increment;
end architecture resettable_behavioral;
```

The derived entity adds the port reset to those inherited from the parent entity. The derived architecture inherits the internal signal count and the statement drive from the parent architecture, and overrides the statement increment with a modified version.

The other approach to structural modeling, proposed by Ramesh [40], basically uses the keyword "class" in place of "tagged" to indicate inheritance, but is otherwise the same. The underlying semantics are inheritance of generics and ports in the entity declarations and inheritance of concurrent statements in the architecture bodies.

Mills [37] also proposes a semantically similar alternative, where inheritance is specified in the binding indication of a configuration declaration or configuration specification, rather than in an entity or architecture declaration. The difficulty with this approach is that the binding is performed during elaboration rather than during analysis. Hence, an inheriting design entity is unable to refer to ports, signals and other items declared in a parent design entity. This lack of visibility significantly limits the way in which an inheriting design entity can extend or refine the implemention of the parent. In order to avoid such limitations, it might be possible to defer analysis of an inheriting design entity until elaboration time. However, to do so would be significantly at variance with the existing philosophy of early, separate analysis and error detection.

### 3.3 Concurrency and Communication Extensions

Object-oriented extensions for system-level modeling address the fact that the communication model implied by signals and ports in VHDL is inappropriate for abstract designs in which the inter-module communication protocols are not yet defined. In the early design stages, a system can be modeled as a collection of communicating concurrent processes that request operations of one another and transfer data (often represented by abstract tokens) between one another. The detailed representation of data, the partitioning into hardware or software modules, and the sequencing and timing of communication over concrete interconnections are design decisions deferred to a later stage in the design flow.

Proposed extensions to VHDL for system-level modeling are based on an object-oriented approach, and seek to represent the system as a set of concurrent objects that communicate by invoking operations in other objects. The intention is to use object-oriented techniques to improve the development process at this early stage in the lifecycle. We identify two distinct approaches that have been proposed for extending VHDL in this area. The first approach involves extending the notion of an entity, viewing it as a form of class and adding operations that can be invoked by processes. For example (using no particular proposer's syntax):

```
entity class elevator is
   operation call ( floor : floor_number );
   operation where_are_you return floor_number;
end entity class;

entity class elevator_with_fire_service is
   new elevator with
   operation set_emergency_mode;
   operation clear_emergency_mode;
end entity class;
```

The first entity class defines operations to call an elevator to a specified floor and to query the location of an elevator. The second entity class inherits these operations, and defines new operations to set and clear and emergency operating mode. A model might instantiate elevator objects as follows:

```
object normal_elevator : elevator;
object emergency_elevator : elevator_with_fire_service;
```

A process within the model might invoke operations as follows:

```
emergency_elevator_location := emergency_elevator.where_are_you;
emergency_elevator.set_emergency_mode;
emergency_elevator.call ( ground_level );
```

The Vista OO-VHDL language described by Swamy *et al* [45], and the proposal by Benzakki and Djafri [5] follow this approach. These proposals also address structural modeling, but motivate their extensions

by system-level modeling needs. We discuss a number of problems with this approach and with the particular proposals in Sections 4.1 and 4.2.

The second approach to extensions for system-level modeling involves adding a class concept to the language, as described above for data modeling, and addressing the issue of concurrency control for multiple processes accessing an object. The proposal by Willis *et al* [48] implicitly addresses this issue by making classes take on the characteristics of monitors when instantiated as shared objects. In such cases, mutual exclusion is enforced for concurrent access to a shared object. The proposal by Cabanis *et al* [11], on the other hand, permits concurrent access to a shared object. It addresses concurrency control by providing some predicates that the designer can use to determine whether concurrent access is occurring and thus control program flow (for example, by busy-waiting). Objective VHDL [39] does not include explicit language extensions for concurrency. Rather, it suggests a way of using classes to encapsulate communication protocols implemented using signals and ports. We discuss the use of object-oriented techniques for system-level modeling further in Section 4.

# 4. ISSUES FOR HIGH-LEVEL MODELING EXTENSIONS TO VHDL

One of our guiding principles for language extension mentioned earlier concerns integration of new features with existing features. Thus, if we are to consider new features for VHDL to support high-level modeling, we need to identify existing features that relate to high-level modeling. With this in mind, we can clearly see that VHDL already includes many features that relate to the principles cited by Booch as necessary for object-orientation. Subprograms, entities, and packages support abstraction and encapsulation (albeit weak encapsulation in the case of packages); and overloading provides a limited, ad-hoc form of polymorphism. In the terminology of Wegner [47], these features are sufficient for VHDL to be called "object-based."

The main issues that are not addressed by the existing language for high-level modeling are a more dynamic concurrent process model; a more abstract form of communication between concurrent processes; inheritance-based hierarchy (for data types and hardware structures); the form of dynamic polymorphism that goes with inheritance (namely, dynamic binding); a stronger form of abstraction and encapsulation for abstract data types (ADTs); and a more flexible form of static polymorphism, such as that represented by generics in Ada. We maintain that language extensions to support high-level modeling should address these issues *without subverting or replacing existing language features*. Furthermore, there should be a clear separation of concerns between language features, so that a given feature does not attempt to serve multiple underlying modeling requirements. The interactions between language features should be well defined and understandable to language users.

## 4.1 Concurrency and Object-Oriented Extensions

One central issue that is not adequately addressed by previous proposals is the relationship between object-oriented extensions and the concurrency and communication features in the language. There is a long history of concurrent language design [2] and, more recently, concurrent object-oriented language design [1]. In considering the relationship between object-oriented features and concurrency, Lim and Johnson suggest that

> "Designing features for concurrency in OOP languages is not much different from that of other kinds of languages—concurrency is orthogonal to OOP at the lowest level of abstraction. OOP or not, all the traditional problems in concurrent programming still remain. However, at the highest levels of abstraction, OOP can alleviate the concurrency problem . . . by hiding concurrency inside reusable abstractions." [35]

We concur with this view, and believe that it applies equally to adding object-oriented features to the concurrent language VHDL. We suggest that the conceptual model for concurrency and communication should be considered first, then features for object-orientation relating to concurrency and communication should be designed to integrate with the chosen conceptual model and with existing language features.

***Concurrency Models***

VHDL already has a concurrency model, based on statically instantiated processes communicating and synchronizing via statically declared signals. Processes express the behaviour of a module, and the external signals sensed and driven by the processes embody the interface to the module. An entity declaration expresses an abstraction of a module's behavior by presenting the interface in terms of input and output ports.

Two main problems with the existing language features arise when attempting high-level modeling. First, the static process structure is inflexible, making it difficult to model such subsystems as multi-threaded servers in a client/server system. Such servers, which may ultimately be implemented in software, dynamically create threads of control to concurrently handle multiple incoming requests for transactions. Without the ability to dynamically create threads, interleaving the concurrent transactions becomes cumbersome. The second problem, discussed in Section 3.3, is that communication using VHDL signals is at too low a level of abstraction. A signal in VHDL is intended to model a physical connection between hardware modules, and represents the trajectory of values on the connection over time. When modeling

at a higher level, communication should simply model interaction of processes, possibly with data transfer and possibly with synchronization.

Schumacher and Nebel [43] present a survey of langauges in widespread use for high-level modeling. A number of them, including Statecharts [23], Estelle [10, 31] and SDL [33, 38], are based on a conceptual model of process behavior described using an extended finite-state machine notation. (The hierarchical state machines used in Statecharts are also adopted in the more recently proposed Uniform Modeling Language (UML) [8].) Processes are statically instantiated in Statecharts, whereas in Estelle and SDL they can be dynamically instantiated as part of an action associated with a state transition. In all three languages, the communication structure is statically specified. In Statecharts, communication takes place through actions in one state machine triggering events in other state machines. In Estelle and SDL, communication takes the form of buffered asynchronous message passing. Estelle also allows a form of communication via shared variables, and SDL also allows synchronous remote procedure call (much like the rendezvous of Ada). Schumacher and Nebel also identify CSP [25] and its derivative OCCAM [30] as languages used for high-level modeling. In these languages, the conceptual model of process behavior is a sequentual thread of program execution. Processes are statically instantiated, and communicate using unbuffered synchronous message passing on statically instantiated, typed communication channels.

From this brief review of concurrency and communication features in other languages, it can be seen that there are a range of alternative models. Processes can be statically instantiated, or may be dynamically instantiated and terminated. Communication can take the form of message passing, remote procedure call, or through shared data. For the message passing alternative, messages may be sent via statically instantiated channels or to named destination processes. Further, message passing may be buffered and asynchronous, or may be unbuffered and synchronous.

Given that VHDL processes currently express behavior using sequential statements like those of programming languages, it would seem most appropriate to keep this form, rather than adopting some form of extended finite-state machine model. In those cases where a state machine formulation of behavior is clearest, the states and transitions can be readily expressed using sequential code. However, it is not immediately clear what combination of concurrency and communication features are most appropriate for high-level modeling, so we leave this as an area for further research. An important consideration in choosing among the alternatives is to maintain compatibility and conceptual integrity with the existing language.

As an illustration of a possible extension to the concurrency and communication model of VHDL, we consider a conceptual model in which processes may be dynamically instantiated and communicate using

remote procedure call. This is the approach taken in Ada, so we borrow some Ada features for this illustration. Consider a client/server system in which a client requests transaction of the server, and the server handles multiple transactions concurrently. In order to allow transaction to proceed concurrently, each transaction is handled by an agent process created dynamically by the server. The server might be described as follows:

```
server : process is
  entry request_transaction ( . . . );
  type server_agent is process
       entry request_transaction ( . . . );
       . . .
    end process server_agent;
  type server_agent_ref is access server_agent;
  . . .
begin
  loop
    accept request_transaction ( . . .) do
      transaction_agent := new server_agent;
      requeue transaction_agent.request_transaction;
    end accept;
  end loop;
end process server;
```

The server process repeatedly accepts requests to perform a transaction. For each request, it creates a new instance of the server_agent process type, and forwards the client's request to the new instance. The server is then free to accept the next request while the agent processes the previous request.

***Object-Orientation***

A number of the proposals for extensions to VHDL [5, 6, 11, 45] suggest that object-oriented classes are the most appropriate mechanism for abstract system-level modeling. While it is true that classes can be used to model hardware systems, as demonstrated by Kumar *et al* [34], the class-based approach gives rise to significant problems. Indeed, Kumar *et al* state that they "use C++ to demonstrate the usefulness of object-oriented techniques, not to provide arguments for or against its use in hardware modeling and design." It is unfortunate that the term "message passing" is often used to denote method invocation, since that causes confusion with true message passing between active concurrent objects; thus leading to a confusion between object-oriented features and concurrency features.

The chief problem with using classes as the focus of modeling concurrent systems is that classes are data-centric. To use them in this context forces a monitor-based approach to concurrency. Monitors were first proposed as a concurrency mechanism by Hoare [24], and many of the subsequent concurrent language proposals arose out of the difficulties inherent in the monitor approach [2], in particular, the difficulties

arising from nested monitor calls. It may be that the monitors paradigm does not match the way system-level designers view systems at an abstract level. The fact that many of the system-level description languages mentioned in Section NO TAG are process-based and use message passing suggests that the message-passing paradigm may be more appropriate.

We believe that it is inappropriate to prejudice the language extension process by assuming a class-based solution for system-level modeling at the outset. Classes may be appropriate for data modeling, but the abstract concurrency issues should be dealt with orthogonally. Classes may then be used to provide encapsulation and inheritance for whatever concurrency model is chosen.

### 4.2 Entities and Object-Oriented Extensions

Two of the proposals for object-oriented extensions to VHDL suggest extending the concept of a design entity to include aspects of classes. Vista OO-VHDL [45] provides *EntityObjects*, which extend entities by allowing inclusion of publicly visible procedures called *operations*. Benzakki and Djaffri [5] also propose addition of operations, but to ordinary entities as an alternative to ports. Both proposals allow derived entities to inherit from parent entities.

We have a number of criticisms of these proposals. Both proposals suffer from the problems of using classes to model concurrent objects (as discussed above) and subvert the concept of design entities by using them for this purpose. Design entities, as a language construct, are intended to model instantiable modules, and to abstract over and encapsulate structure (expressed in terms of component instances) and/or behaviour (expressed in terms of processes that are sensitive to and assign to signals). Benzakki and Djaffri at least preserve the view of an entity as a statically instantiable module with a declared interface and an encapsulated implementation. Our main criticism of that proposal is its poorly conceived concurrency control.

Vista OO-VHDL, on the other hand, significantly complicates the semantics of design entities and component instances by the way in which it allows dynamic use of the name of an EntityObject instance. It provides a type called EO_Handle that denotes a name of an EntityObject instance. Values of the type may be passed as parameters and transmitted using signals. The main problem is that EO_Handle values are not typed with the signature of the designated EntityObject. Thus, when analyzing an operation call, it is not possible to check statically that the EntityObject has the required operation. This violates the strong type-checking philosophy of VHDL, and allows more design errors to pass through the development process to run-time. It also imposes run-time overhead in checking for correct use of an EntityObject. Furthermore, it violates the encapsulation of an object's implementation. Through a design error

or poor coding practice, an object might export an EntityObject that provides operations that expose implementation details that are supposed to be hidden. These characteristics of the extension violate the principles of object-oriented design described by Booch and others, and violate the language design principle of coherence with the base language. Swamy *et al* note that they were "guided by one goal: providing the language to modelers as quickly as possible" ( [45, page 19]. Perhaps undue haste may have compromised their language design process.

Our view is that the existing semantics of entities, architectures, components, component instantiation, port interfaces, signal assignment, and signal sensitivity are central to VHDL as a hardware description language and are what distinguish it from conventional programming languages. The entity declaration serves to define an abstract interface for the communication mechanism implemented by a module. If class features are added to the language and monitor calls used for interprocess communication, then the monitor interface should be seen as a new aspect of an entity interface. The encapsulation of the implementation should remain strong. Alternatively, if some other form of concurrency and communication is added, an abstraction for its communication mechanism should be added to the entity interface with strong encapsulation. This is an orthogonal issue to adding inheritance to design entities for structural modeling, as discussed in Section 3.

### 4.3 Object-Oriented Extensions for Data Modeling

In Section 3, we identified two approaches for object-oriented extensions for data modeling: the programming by extension approach as seen in Ada-95, and the class-based approach. In a conventional programming language, the choice between the two might be seen as a matter of taste. However, in VHDL, there are some stronger considerations. In both approaches, a declared type represents a set of objects; the type is either a tagged record type or a class. Objects of the type are then instantiated. In a conventional programming language, the only kinds of objects that can be created are constants (immutable storage locations) or variables (mutable storage locations).[1] Assignment to a variable is relatively straightforward. In the Ada-95 model, it involves computing a value of the type and invoking the assignment operator to modify the content of the storage location. In the class-based model, the name of the location is encapsulated by the class definition, so assignment involves invoking a method that has access to the name. The method then computes values and modifies the storage location. (Note that this is different from assignment of references to an object, where the value assigned is the name of the storage location and the type of the value is a reference type.)

1. We view a file, in the abstract sense, as a variable. It is a mutable location in secondary storage. Its type is a sequence of values, and it can be read and updated using atomic read and write operations.

While both of these approaches can translate directly into VHDL for constants and variables, it is not clear how they translate for signals. One of the main reasons for considering object-orientation is to allow specification of abstract data types (ADTs), and it seems reasonable to expect to be able to define signals of an ADT. The difficulty is that signal assignment semantics in VHDL are considerably more involved than just updating storage locations. Any object oriented-extension for data modeling must address this issue.

In an Ada-95 approach, the a signal name used on the left-hand side of a signal assignment statement denotes the trajectory stored in a driver for the signal. Values of the correct subtype can be assigned directly using the signal assignment operator. The mechanism for constructing ADTs under an Ada-95 approach involves passing objects of the type to and from operation subprograms. Different kinds of parameters are used for variable and signal objects, so the procedure can determine whether to use variable or signal assignment. For example, using the instruction type discussed previously:

```
signal current_instr : instruction;
procedure force_nop (signal instr : out instruction) is
begin
   . . .
   instr <= nop_instruction;
end procedure;
```

Dunlop [17] illustrates an Ada-95 approach, and proposes a solution to the problem of a signal's subtype changing while part of the signal is being waited on.

In the class-based approach, the state of an ADT is encapsulated with the operations and is only accessible within the implementation of the operations. In conventional programming languages, the state is usually represented by variables, and the operations use variable assignment to modify the state. In an extended VHDL, it might at first seem appropriate to allow an ADT to encapsulate a signal. In that case, signal assignment should be used to update the state. A corollary is that there should be two kinds of ADTs: one for variable objects, containing state in the form of variables; and one for signal objects, containing state in the form of signals. This would require substantial duplication, since, in many cases, objects of both kinds would be required.

To illustrate this problem, consider a digital signal processing (DSP) system which manipulates values of a complex-number ADT. Function units within the DSP system communicate complex-number values between one another over signals, and store complex-number values internally in variables. To deal with these two cases, a model must provide an ADT that encapsulates a complex-number signal, and a separate ADT that encapsulates a complex-number variable. Assignment to the encapsulated state within the ADT implementations must be does with signal assignment in the former ADT, and with variable assignment

in the latter ADT. Operations such as addition, conjugation, etc., would be essentially duplicated in the two implementations. Without duplication of the ADTs in this manner, the strong encapsulation of the internal state in the ADT would be broken. Such duplication is evident in Objective VHDL [39], although that language does provide a mechanism for factoring out parts of a class that are common to variable and signal instances.

A better way of incorporating classes is to constrain the encapsulated state to take the form of variables. The signal assignment operation would then use the variable assignment and equality operations defined for the ADT to update and compare values in transactions on signal drivers. Similarly, the signal update algorithm would use the variable assignment and equality operations to compute driving and effective values and to determine the occurrence of events on signals.

While both an Ada-95 approach and a class-based approach are feasible, an Ada-95 approach may work out more neatly within the existing framework of VHDL. In particular, since VHDL already includes features for defining types and operations in packages, an Ada-95 approach would simply involve an extension of these features. A class-based approach, on the other hand, would involve duplication of these features, unnecessarily increasing the complexity of the language semantics. Pursuing the programming by extension approach will add to the language complexity, but to a far lesser extent.

### 4.4 Encapsulation

Abstract data types (ADTs) are a central language feature for managing language complexity on large systems. An ADT is defined by a set of values and a set of operations for manipulating the value. Importantly, the concrete implementation details of the ADT are hidden. A user of the ADT may only manipulate values through the provided operations.

VHDL provides a partial facility for defining ADTs, based on the package features of Ada. In Ada, an ADT is defined by declaring a private type in a package, along with subprograms that perform the operations on the ADT. The internal structure of the type is visible only within the package and can only be manipulated by operations in the package body; it is not visible to users of the type outside of the package. Unlike Ada, however, VHDL does not provide a means of hiding the concrete details of a type declared in a package. Thus, an ADT defined in VHDL is only encapsulated by convention—it is assumed that the user will only manipulate the values through the operations provided in the package.

An early draft of the VHDL-93 standard included proposed features for defining private types in packages [27]. The issue is more complicated than in Ada, since there are more restrictions on the ways in which

different types may be used. For example, an access type may not be used as the type of a signal in VHDL. Thus, if a private type happens to have a concrete realization as an access type, a problem arises when a user attempts to use the type for a signal. If the user is prevented from doing so, information is effectively "leaked" about the concrete implementation, thus violating the supposed encapsulation. There are analogous difficulties in Ada, which are handled by specifying limitations on the use of a private type. The draft VHDL-93 proposal followed this approach, but the features were dropped from the final standard. We believe that it is important to revisit this issue, as strong encapsulation is necessary for successful implementation of ADTs, which, in turn, are necessary for high-level modeling.

Since ADTs form the basis for object-oriented modeling, any language features included must also support inheritance. Of issue here is the tension between information hiding from users of an ADT, and making information visible to derived ADTs. In C++, this is addressed by the notion of "friend" classes and "restricted" methods in classes. Information that is restricted is hidden from normal users, but is visible to sub-classes that refine the ADT. In Ada-95, the issue is addressed by hierarchical packages, in which private parts are visible to child packages. Thus, whichever approach is followed in extensions to VHDL, there are models to follow in other languages.

### 4.5 Shared Variables

VHDL-93 includes shared variables, which are accessible to multiple processes. The current language definition does not specify concurrency control semantics for concurrent access. However, the 1076a Working Group has proposed a monitor-based solution to concurrency control [29]. This proposal forms the basis for the class-based extension suggested by Willis *et al* [48]. They suggest that concurrency control be implicit, involving mutual exclusion in the case of multiple processes concurrently calling monitor operations. In the case of a class instance being nested within a process, no concurrency control is needed.

The use of classes for data modeling need not, however, imply their use as monitors for shared variables. It may be more appropriate to distinguish between the language features used for object-oriented data modeling and those used for concurrency control. This is the approach taken in Ada-95, in which tagged and derived types are used for data modeling and protected types (a form of monitor) are used for concurrency control. For example:

```
type shared_instruction is protected
    type instr_ptr is access instruction'class;
    variable instr : instr_ptr;
    function get_instr return instruction'class;
    procedure put_instr
                ( new_instr : instruction'class );
  end protected shared_instruction;
```

In this example, the tagged type instruction and its derivatives and the class-wide type instruction'class are used to model the instruction set as described in previous sections. The protected type shared_instruction provides a means of creating a shared variable representing an instruction that can be accessed by several processes via the subprograms get_instr and put_instr. For a given shared variable of this type, only one process at a time can activate either of the subprograms. Hence mutually exclusive access to the encapsulated variable is ensured. The example illustrates the separation of concerns into data modeling aspects (based on tagged types and derivation), and concurrency control aspects (protected types).

An alternative approach may be to adopt classes for data modeling and to allow monitors to encapsulate instances of classes or any other data types. This is another case where concurrency issues and object-orientation should be dealt with orthogonally.

### 4.6 Multiple Inheritance

There appears to be little agreement whether object-oriented extensions to VHDL should allow multiple inheritance or only single inheritance. This parallels the debate in the programming language community. According to Booch, "multiple inheritance [is] like a parachute: you don't always need it, but when you do, you're really happy to have it on hand" [7, page 124]. The decision between single and multiple inheritance may ultimately be a secondary consideration. The Ada-95 "programming by extension" style of data modeling does not support multiple inheritance, so if it is adopted without modification into VHDL, single inheritance would result. However, the effects of multiple inheritance can be achieved in Ada-95 by using the programming by extension features in combination with other features. The techniques are outlined in [4]. If a class-based approach is adopted, the C++ or Java model for multiple inheritance may prove an appropriate model to follow. It is not clear how strong the case is for multiple inheritance in a hardware description language such as VHDL. Implementation costs, added language complexity, integration with other language features, and complexity of use may be important factors.

### 4.7 Genericity

There is another aspect of object-oriented extensions to VHDL that is orthogonal to the issues addressed previously, namely genericity. This is an aspect of polymorphic typing. The inheritance mechanisms

included in the proposals cited allow expression of "is-a" classification hierarchies, but do not adequately deal with "is-part-of" hierarchies. For that, an additional mechanism, such as generics in Ada or template classes in C++, is needed. Ashenden and Wilsey [3] suggest an extension to VHDL package declarations, based on generic packages in Ada, to allow expression of generic ADTs. The idea of generic types and generic subprograms could also be used in combination with the proposed data modeling and structure modeling extensions to allow generic classes or generic entities. This would be useful to describe container classes that are not bound to a particular contained type. For example, a list of objects might be defined as:

```
type list is class
    generic ( type element_type is private );
    . . .
    procedure add ( element : element_type );
    . . .
  end class;
```

Genericity would also be useful to describe functional units which can operate on a variety of related types of data. For example, a shift register that shifts an array of objects might be defined as:

```
entity shift_reg is
  generic ( type item is private;  type index is (<>);
            type vector is array (index) of item );
  port ( shift_clk : in bit;  data_in : in item;
         data_out : out vector );
end entity;
```

### 4.8 Synthesis

VHDL was originally conceived as a hardware design language, without being specifically oriented toward either simulation or synthesis. However, synthesis is an increasingly important part of the design flow. Early synthesis tools were not able to deal with many of the language constructs that were at a level of abstraction much above basic hardware devices. Behavioral synthesis tools developed more recently are able to deal with larger subsets of language features, allowing synthesis to be used earlier in the design flow. If new high-level modeling features are to be added to VHDL, the synthesizability of the features must be considered.

The rationale for adding new features for data modeling and system-level modeling is to aid modeling at a high level of abstraction, above the level at which synthesis would be used. Indeed, there is interest in using new features for system-level design even before hardware/software partitioning has been performed. Hence, it can be argued that high-level modeling features need not be synthesizable. However, this view ignores the ongoing development of behavioral synthesis technology [12, 14, 20] and hardware/software co-synthesis studies [15, 19, 22]. Furthermore, new features added to support high-level model-

ing are likely to be useful for modeling at lower levels of abstraction as well. Hence, their synthesizability using current synthesis technology is an important issue. Ignoring the issue of synthesizability when considering language extensions may ultimately make synthesizability much more difficult. For example, allowing dynamic communication of entity instance names (as in Vista OO-VHDL [45]) may prohibit synthesis of method invocation, whereas constraining method invocation to statically determined entity instances may make synthesis tractable.

Synthesis of proposed object-oriented extensions for structural modeling is less problematic, provided all binding can be performed when the model is elaborated. For example, if a design entity inherits ports, processes and component instances, elaboration of the design entity would involve successive elaboration of the ancestors in the inheritance lattice. This would create a static collection of nets and processes (no different from the current situation) that would then be synthesized using existing techniques.

## 5. CONCLUSION

This paper contains an in-depth survey of the previous proposals for introducing object-oriented and high-level modeling extensions into VHDL. These previous proposals are categorized into three areas based on the modeling requirements they address: data modeling, structural modeling, and abstract system-level modeling. Our analysis shows that, while there is much agreement between the proposals, many of them lack depth of consideration of semantic issues. In particular, they lack generality of applicability to a wide range of modeling problems, and do not integrate consistently with existing language mechanisms.

In this paper we have also identified a number of issues to be addressed when considering extension to VHDL. We emphasize the importance of a semantics-based approach to extensions and present our perspective on what issues are most important when studying semantic issues for possible extension. Extensions for VHDL (or for that matter, any language) should manifest themselves in semantic and syntactic structures that are consistent with the existing language structure. Merely bolting "your favorite language construct" onto the side of an existing language is not only foolish, but likely to destroy the language semantics. Furthermore, expecting a single language construct to solve the vast array of system-level modeling problems also leads to disappointment. We believe that a collective of several, carefully crafted language extensions can be made to migrate VHDL from its current object-based structure to an object-oriented structure suitable for high-level modeling. We further believe that these structures will enhance the existing encapsulation and abstraction facilities of VHDL in ways that will expand VHDL's existing strengths throughout the entire range of its modeling use. The value of this approach is demonstrated in the extension of Ada from Ada-83 to Ada-95. As stated in the Ada-9X Rationale [4]:

> "Rather than providing a number of new language features to directly solve each identified application problem, the extra capability of Ada 9X is provided by a few primitive language building blocks. In combination, these building blocks enable programmers to solve more application problems efficiently and productively."

In performing our analysis of proposals and issues, it has become clear that object-orientation can arise from a single language feature, such as classes, or from the interaction of a number of features. (The latter approach is illustrated by Ada-95.) Whichever approach is chosen, language extensions must be designed to integrate cleanly with the semantic mechanisms and the syntax of the existing language. Furthermore, it is not sufficient to simply adopt features from programming languages. Rather, one must carefully consider the interaction of a proposed extension with the existing hardware modeling constructs in VHDL, such as signals and signal assignment. Consideration must also be given to designing a coherent set of extensions that are not motivated solely by requirements for system-level modeling. For example, while the introduction of private types and generic types would strengthen the encapsulation and polymorphism features required for object-orientation, they would be of general value in the language for other modeling tasks.

Above all, it must be borne in mind that VHDL is a design automation language. As the scope of design automation advances, the language must advance to keep track. However, those advances must not be at the expense of existing uses of the language. The expressiveness of the language for specifying hardware systems must be maintained. Due consideration must also be given to the impact of language extensions on analysis, simulation, and synthesis. The language semantics are already complex. By designing extensions that cleanly integrate with the existing language, we reduce the additional complexity for semantic analysis. Likewise, simulation capacity and performance is already an issue for tool designers and users. Language extensions that impose significant and pervasive run-time burden are unacceptable. Synthesis technology is now extending to the level of behavioral synthesis. Language extensions should not preclude synthesis of high-level modeling constructs by relying exclusively on run-time binding mechanisms.

Lastly, we believe that the ongoing dialogue in the literature focussing on object-oriented extensions to VHDL is too narrow. The focus should be on the broader issues of language extensions to better support a wide range of modeling requirements. The language should become object-oriented only insofar as such extensions might include features for object-oriented programming and modeling.

## REFERENCES


[1] G. Agha, "Concurrent Object-Oriented Programming," *Communications of the ACM*, vol. 33, no. 9, pp. 125–141, 1990.

[2] G. R. Andrews and F. B. Schneider, "Concepts and Notations for Concurrent Programming," *ACM Computing Surveys*, vol. 15, no. 1, pp. 1–43, 1983.

[3] P. J. Ashenden and P. A. Wilsey, "Polymorphic Abstract Data Types in VHDL," *Proceedings of International Conference on Electronic Hardware Description Languages*, Las Vegas, NV, pp. 35–39, 1995.

[4] J. Barnes, Ed. *Ada 95 Rationale*, vol. 1247. Berlin, Germany: Springer-Verlag, 1997.

[5] J. Benzakki and B. Djaffri, "Object Oriented Extensions to VHDL: the LaMI Proposal," *Proceedings of Conference on Hardware Description Languages '97*, Toledo, Spain, pp. 334–347, 1997.

[6] J.-M. Bergé, W. Nebel, and W. Putzke, *Requirements and Design Objectives for an Object-Oriented Extension of VHDL (OO-VHDL)*. IEEE DASC OO-VHDL Study Group, working paper, ftp://vhdl.org/vi/oovhdl/papers/dod_aug96.rtf, 1996.

[7] G. Booch, *Object-Oriented Analysis and Design with Applications*. Redwood City, CA: Benjamin/Cummins, 1994.

[8] G. Booch, *The Best of Booch*. New York, NY: SIGS Books and Multimedia, 1996.

[9] F. P. Brooks, Jr., *The Mythical Man-Month*, Anniversary ed. Reading, MA: Addison-Wesley, 1995.

[10] S. Budkowski and P. Dembinski, "An Introduction to Estelle: A Specification Language for Distributed Systems," *Computer Networks and ISDN Systems*, vol. 14, no. 1, pp. 3–23, 1987.

[11] D. Cabanis and S. Medhat, "Classification-Orientation for VHDL: A Specification," *Proceedings of VHDL International Users Forum Spring '96 Conference*, Santa Clara, CA, pp. 265–274, 1996.

[12] R. Camposano and W. H. Wolf, Eds., *High-Level VLSI Synthesis*. Boston, MA: Kluwer Academic Publishers, 1991.

[13] O. J. Dahl and K. Nygaard, "Simula: An Algol Based Simulation Language," *Communications of the ACM*, vol. 9, no. 9, pp. 671–678, 1966.

[14] G. de Micheli, *Synthesis and Optimization of Digital Circuits*. New York, NY: McGraw-Hill, 1994.

[15] G. de Micheli and M. Sami, Eds., *Hardware/Software Co-design*. Dordrecht, Netherlands: Kluwer Academic Publishers, 1996.

[16] D. D. Dunlop, "Object-Oriented Extensions to VHDL," *Proceedings of VHDL International Users Forum Fall '94 Conference*, McLean, VA, pp. 5.1–5.9, 1994.

[17] D. D. Dunlop, *VHDL Structure Varying Signals and OO Extensions to the VHDL Type System*. IEEE DASC OO-VHDL Study Group, working paper, ftp://vhdl.org/vi/oovhdl/papers/structure-varying-signals.txt, 1995.

[18] W. Ecker, "An Object-Oriented View of Structural VHDL Description," *Proceedings of VHDL International Users Forum Spring '96 Conference*, Santa Clara, CA, pp. pp. 255–264, 1996.

[19] R. Ernst, J. Henkel, and T. Benner, "Hardware-Software Co-synthesis for Microcontrollers," *IEEE Design and Test of Computers*, vol. 10, no. 4, pp. 64–75, 1993.

[20] D. D. Gajski, N. D. Dutt, A. C.-H. Wu, and S. Y.-L. Lin, *High-Level Synthesis : Introduction to Chip and System Design*. Boston, MA: Kluwer Academic Publishers, 1992.

[21] J. Gosling, B. Joy, and G. L. Steele, *The Java Language Specification*. Reading, MA: Addison-Wesley, 1996.

[22] R. Gupta and G. de Micheli, "System Co-synthesis for Digital Systems," *IEEE Design and Test of Computers*, vol. 10, no. 3, pp. 29–41, 1993.

[23] D. Harel, "Statecharts: A Visual Formalism for Computer Systems," *Science of Computer Programming*, vol. 8, no. 3, pp. 231–274, 1987.

[24] C. A. R. Hoare, "Monitors: An Operating System Structuring Concept," *Communications of the ACM*, vol. 17, no. 10, pp. 549–557, 1974.

[25] C. A. R. Hoare, "Communicating Sequential Processes," *Communications of the ACM*, vol. 21, no. 11, pp. 934–941, 1978.

[26] C. A. R. Hoare, "Hints on Programming Language Design," in *Essays in Computing Science*, C. A. R. Hoare and C. B. Jones, Eds. Herts, UK: Prentice Hall, 1989, pp. 193–216.

[27] IEEE, *Draft Standard VHDL Language Reference Manual*. Draft Standard P1076-1992/A, New York, NY: IEEE, 1992.

[28] IEEE, *Standard VHDL Language Reference Manual*. Standard 1076-1993, New York, NY: IEEE, 1993.

[29] IEEE, *Standard VHDL Language Reference Manual*. Draft Standard P1076a-1997, New York, NY: IEEE, 1997.

[30] INMOS Ltd., *OCCAM Programming Manual*. London, UK: Prentice-Hall, 1984.

[31] ISO, *Estelle: A Formal Description Technique Based on an Extended State Transition Model*. Draft International Standard 9074, 1987.

[32] ISO/IEC, *Ada 95 Reference Manual*. International Standard ISO/IEC 8652:1995 (E), Berlin, Germany: Springer-Verlag, 1995.

[33] ITU, *Specification and Description Language (SDL)*. Revised Recommendation Z.100, 1992.

[34] S. Kumar, J. H. Aylor, B. W. Johnson, and W. A. Wulf, "Object-Oriented Techniques in Hardware Design," *IEEE Computer*, vol. 9, no. 6, pp. 64–70, 1994.

[35] J. Lim and R. E. Johnson, "The Heart of Object-Oriented Concurrent Programming," *ACM SIGPLAN Notices*, vol. 24, no. 4, *Proceedings of ACM SIGPLAN Workshop on Object-Based Concurrent Programming*, pp. 165–167, 1989.

[36] M. T. Mills, *Proposed Object Oriented Programming (OOP) Enhancements to the Very High Speed Integrated Circuits (VHSIC) Hardware Description Language (VHDL)*, Wright Laboratory, Dayton, OH, Tech. Report WL-TR-5025, 1993.

[37] M. T. Mills, *A Minor Syntax Change to VHDL Yields Major Object Oriented Benefits*. IEEE DASC OO-VHDL Study Group, ftp://vhdl.org/vi/oovhdl/papers/mills.oct.95.ps, 1995.

[38] A. Olsen, O. Færgemand, B. Møller-Pedersen, R. Reed, and J. R. W. Smith, *Systems Engineering using SDL-92*. Amsterdam: Elsevier, 1994.

[39] M. Radetzki, W. Putzke, W. Nebel, S. Maginot, J.-M. Bergé, and A.-M. Tagant, "VHDL Language Extensions to Support Abstraction and Re-Use," *Proceedings of Workshop on Libraries, Component Modeling, and Quality Assurance*, Toledo, Spain, 1997.

[40] C. R. Ramesh, "Object Orienting VHDL for Component Modeling," *Proceedings of VHDL International Users Forum Fall '94 Conference*, McLean, VA, pp. 5.17–5.28, 1994.

[41] A. Sarkar, R. Waxman, and J. P. Cohoon, "Specification-Modeling Methodologies for Reactive-System Design," in *High-Level System Modeling: Specification Languages*, vol. 3, *Current Issues in Electronic Modeling*, J.-M. Bergé, O. Levia, and J. Rouillard, Eds., 1995, pp. 1–34.

[42] G. Schumacher and W. Nebel, "Inheritance Concept for Signals in Object-Oriented Extensions to VHDL," *Proceedings of Euro-DAC '95 with Euro-VHDL '95*, Brighton, UK, pp. 428–435, 1995.

[43] G. Schumacher and W. Nebel, "Survey on Languages for Object Oriented Hardware Design Methodologies," in *High-Level System Modeling: Specification Languages*, vol. 3, *Current Issues in Electronic Modeling*, J.-M. Bergé, O. Levia, and J. Rouillard, Eds., 1995, pp. 35–50.

[44] B. Stroustrup, *The C++ Programming Language*. Reading, MA: Addison-Wesley, 1986.

[45] S. Swamy, A. Molin, and B. Covnot, "OO-VHDL: Object-Oriented Extensions to VHDL," *IEEE Computer*, vol. 28, no. 10, pp. 18–26, 1995.

[46] R. D. Tennent, "Language Design Methods Based on Semantic Principles," *Acta Informatica*, vol. 8, pp. 97–112, 1977.

[47] P. Wegner, "Dimensions of Object-Based Language Design," *ACM SIGPLAN Notices*, vol. 22, no. 12, *Proceedings of OOPSLA '87*, pp. 168–182, 1987.

[48] J. C. Willis, S. A. Bailey, and R. Newschutz, "A Proposal for Minimally Extending VHDL to Achieve Data Encapsulation Late Binding and Multiple Inheritance," *Proceedings of VHDL International Users Forum Fall '94 Conference*, McLean, VA, pp. 5.31–5.38, 1994.

[49] N. Wirth, "From Programming Language Design to Computer Construction," *Communications of the ACM*, vol. 28, no. 2, pp. 160–164, 1985.