Proc. VIUF Spring '97 Conference,
Santa Clara, CA (April 1997), pp. 109–118.

Considerations on Object-Oriented Extensions to VHDL*

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

Philip A. Wilsey
Dept. of ECECS, PO Box 210030
University of Cincinnati
Cincinnati, OH 45221–0030
USA

Abstract

This paper reviews proposals for object-oriented extensions to VHDL and places them within a taxonomy based on the modeling requirements they address. The paper also presents a detailed discussion of issues to be considered in adding object-oriented extensions to VHDL, including concurrency, abstraction using entity interfaces, 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.

1. Introduction

In recent years, the software engineering community has moved 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 ([9], page 181). The ideas denoted by these terms are clearly defined and illustrated by Booch [8] and have be reviewed extensively by other authors, so we do not include any detailed discussion here.

We strongly believe that hardware design using a hardware design language is the same intellectual process as software design; consequently the complexity management techniques that are used in software design should also be used in hardware design. Object-oriented design is one such technique. Hence, it is appropriate to consider object-oriented techniques in a hardware description language such as VHDL. Hardware systems are complex, being composed of many interacting components and incurring lifecycles and generations. 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. Indeed these expectations are expressed as requirements by the IEEE DASC OO-VHDL Study Group [7].

Our aim in this paper is not to present or support any particular object-oriented extension to VHDL. Instead, we review previous proposals for extensions and attempt to categorize them so that they are better understood. Our categorization is based on the semantics of each proposal, insofar as the semantics are elaborated 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.

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 object-oriented extensions to VHDL and places previously published proposals within the taxonomy. Section 4 discusses a range of issues that must be considered when designing object-oriented 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 object-oriented extensions to VHDL—hopefully avoiding the pitfalls along the way.

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

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 understand. 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.

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. Syntax is the concrete denotation. While poor syntax may obfuscate the design intent, it does not prohibit expression of the intent.

Simplicity of Mechanism

In determining semantic features to be included in a language, simple semantic mechanisms should be chosen in preference to more complicated general solutions. The semantic mechanisms should, as much as possible, be orthogonal to each other. As Hoare suggests [17], "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. This also makes it easier for tool builders to optimize their implementation of language features.

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, or, as Brooks [10] calls it, "conceptual integrity." 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 [28], "distinguish ... between what is essential and what ephemeral."

3. Taxonomy of Extensions and Previous Proposals

Previous proposals for object-oriented extensions to VHDL have focussed on three areas of language usage: data modeling, structural modeling, and system-level modeling. These areas are also reviewed by Dunlop [12]. 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.

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. The suggestion is that this is insufficient for modeling data with complex structure in hardware models at a high-level of abstraction, and that object-oriented techniques for expressing data should be incorporated into VHDL.

Two main approaches have been canvassed for object-oriented data modeling in VHDL. The first approach involves adopting the "programming by extension" features of Ada-95 [4], and is the basis of proposals by Mills [21], and Schumacher and Nebel [24]. Dunlop [13] also illustrates the general concepts of object-oriented data modeling using this approach. The approach involves defining a base type as a tagged record, and deriving subtypes by extending tagged record types with new record elements. For example:

```
type instruction is tagged record
    opcode : opcode_type;
  end record;
type register_alu_instruction is new instruction
  with record
    dest, src1, src2 : reg_number;
  end record;
```

Proposal	Data Modeling	Structural Modeling	System-Level Modeling
Mills [21]	Ada-95 approach (tagged type with single inheritance)	tagged entity/architecture with multiple inheritance	
Schumacher and Nebel [24]	Ada-95 approach (tagged type with single inheritance)		
Dunlop [13]	illustrates general concepts using Ada-95 approach		
Willis *et al* [27]	class-based with multiple inheritance		implicit: class types for shared variables have monitor semantics
Ecker [14]		tagged entity/architecture with multiple inheritance	
Ramesh [23]		entity classes with inheritance (only single inheritance illustrated)	
Mills [22]		inheritance via configuration	
Swamy *et al* [25]		entity classes with inheritance (only single inheritance illustrated)	entity classes with operations (modified monitor semantics); inheritance (only single inheritance illustrated)
Benzakki and Djafri [6]		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

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

The hierarchy of derived types forms the inheritance hierarchy required in object-oriented programming. Other aspects, including abstraction and modularity, come from existing language features, notably package declarations. Polymorphic typing comes from declaration of procedures with parameters of unconstrained class types. For example:

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

The type of an actual parameter may be bound at run-time rather than being statically determined. The main weakness of this approach stems from the poor encapsulation features of VHDL packages. This could be improved by adopting the stronger encapsulation features of Ada, based on private types within packages.

The second approach to object-oriented data modeling is influenced by C++ and its predecessors, and involves definition of classes, which encapsulate the definitions of data and operations of objects. As an example, the instruction types might be defined as:

```
type instruction is class
    opcode : opcode_type;
    procedure decode ( ... );
  end class;
type register_alu_instruction is instruction class
    dest, src1, src2 : reg_number;
    procedure decode ( ... );
  end class;
```

While a number of proposals are based on this approach, only Willis *et al* [27] 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.

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 [21] and Ecker [14], involves using the keyword "tagged" to identify an entity or architecture that can be inherited, and the keyword "new" to specify inheritance into a derived entity or architecture. For example:

```
entity counter is tagged
   port ( clk : in bit;  q : out bit_vector );
end entity counter;

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

The other approach, proposed by Ramesh [23], 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 [22] proposes a semantically similar alternative, in which 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 significantly limits the way in which an inheriting design entity can extend or refine the implemention of the parent.

Extensions for System-Level Modeling

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 may 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 of communication over concrete interconnections are design decisions deferred to a later stage in the design flow.

Proposed object-oriented extensions to VHDL for system-level modeling seek to represent the system as a set of 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 proposed approaches to 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:

```
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 Vista OO-VHDL language described by Swamy *et al* [25], and the proposal by Benzakki and Djafri [6] follow this approach. These proposals also address structural modeling, but motivate their extensions by system-level modeling needs.

The second approach 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* [27] 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). We discuss the use of object-oriented techniques for system-level modeling further in Section 4.

4. Issues for OO Extensions to VHDL

One of our guiding principles for language extension mentioned earlier is integration of new features with existing features. Thus, if we are to consider new features to support object-oriented techniques, we need to identify existing features that relate to object-oriented techniques. VHDL already includes many features that we can 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. These features are sufficient for VHDL to be called "object-based" in the terminology of Wegner [26].

The main issues that are not addressed by the existing language are a stronger form of abstraction and encapsulation for abstract data types; inheritance-based hierarchy (for data types and hardware structures); and the form of polymorphism that goes with inheritance (namely, dynamic binding of parameter types). We maintain that language extensions to support object-orientation should address these issues without subverting or replacing existing language features.

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 [3] and, more recently, concurrent object-oriented language design [2]. VHDL already has a concurrency model, based on statically instantiated processes communicating and synchronizing via signals. A number of the proposals for extensions [6, 7, 11, 25] suggest that object-oriented classes are more appropriate for abstract system-level modeling. While it is true that classes can be used to model hardware systems, as demonstrated by Kumar *et al* [19], it is not necessarily the best way. (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.

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." [20]

We concur with this view, and believe that it applies equally to adding object-oriented features to VHDL, which is a concurrent language. The 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 [15], and many of the subsequent concurrent language proposals arose out of the difficulties inherent in the monitor approach [3]. It may be that the monitors paradigm does not match the way system-level designers view systems at an abstract level. For example, a paradigm based on CSP [16] may be closer to the way many system-level designers think. An extension for CSP-based concurrency might allow definition of message channels and provide operations to send and receive messages on channels. For example:

```
channel elevator_call : floor_number;
channel elevator_location : floor_number;
elevator : process is
begin
  ...
  receive calling_floor from elevator_call;
  send current_floor to elevator_location;
  ...
end process;
```

Here, elevator_call and elevator_location are message passing channels through which processes communicate. The receive and send operations in the elevator process are synchronous message-passing events, in which a value of the designated type is passed from sender to receiver. CSP message passing, as illustrated in this example, abstracts over the details of protocols for transferring data using VHDL signals.

Another alternative paradigm might be communicating hierarchical state machines, such as that used in the

Uniform Modeling Language (UML) [9]. 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.

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. Swamy *et al* [25] propose *EntityObjects*, which extend entities by allowing inclusion of publicly visible procedures called *operations*. Benzakki and Djaffri [6] 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. Swamy *et al*, on the other hand, significantly complicate the semantics of design entities and component instances by the way in which they allow dynamic use of the name of an EntityObject instance. Their scheme violates the encapsulation of the implementation of an entity, and is type-unsafe. 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.

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.

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, we declare a type to represent a set of objects; the type is either a tagged record type or a class. We then instantiate the type to create objects. In a conventional programming language, the only kinds of objects we can create are constants (immutable storage locations) or variables (mutable storage locations). 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.)

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, and it seems reasonable to expect to be able to define signals of an abstract data type. 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 the Ada-95 approach, the name of a signal represents its stored trajectory, and values of the correct subtype can be assigned directly using the signal assignment operator. The mechanism for constructing abstract data types under the 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 deter-

mine whether to use variable or signal assignment. For example, using the instruction type from above:

```
signal current_instr : instruction;

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

Dunlop [13] illustrates the 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 abstract data type 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, the state of a signal of an abstract data type should be the signal's trajectory, and signal assignment should be used to update the state. This implies that there should be two kinds of abstract data types, one for variable objects and one for signal objects. It is not clear how the specification of state within a class and assignment within operations can be constrained to be consistent with the instantiated use of the class as a variable or a signal. Consequently, the Ada-95 approach may work out more neatly within the existing language framework.

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 [18]. This proposal forms the basis for the class-based extension suggested by Willis *et al* [27]. They suggest that concurrency control be implicit, involving mutual exclusion in the case of multiple processes calling monitor operations concurrently. 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;
```

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.

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" ([8], page 124). The decision between single and multiple inheritance may ultimately be a secondary consideration. The Ada-95 style of data modeling does not support multiple inheritance, so if the Ada-95 style is adopted without modification into VHDL, single inheritance would result. If a class-based approach is adopted, the C++ 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 may be an important factor.

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 [5] suggest an extension to VHDL package declarations, based on generic packages in Ada, to allow expression of generic abstract data types. 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 enti-

ties. 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;
```

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 object-oriented features are to be added to VHDL, the synthesizability of the features must be considered.

The rationale for adding object-oriented 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 object-oriented features for system-level design even before hardware/software partitioning has been performed. Hence, it can be argued that object-oriented features need not be synthesizable. However, this view ignores the ongoing development of behavioral synthesis technology. 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 the proposal by Swami *et al* [25]) 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 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 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 object-oriented 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. 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 [1]:

> "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 pro-

grammers 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. It is not sufficient simply to adopt features from programming languages. It is necessary to consider their interaction with the existing hardware modeling constructs in VHDL, such as signals and signal assignment. Furthermore, consideration should be given to designing a coherent set of extensions that are not motivated solely by requirements for system-level modeling. For example, introduction of private types and generic types would strengthen the encapsulation and polymorphism features required for object-orientation, and 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 object-oriented 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] Ada 9X Mapping/Revision Team, *Ada 9X Rationale*, Intermetrics, Cambridge, MA (1994).

[2] Agha, G., "Concurrent Object-Oriented Programming," *CACM*, Vol. 33, No. 9 (September 1990), pp. 125−141.

[3] Andrews, G. R., and Schneider, F. B., "Concepts and Notations for Concurrent Programming," *ACM Computing Surveys*, Vol. 15, No. 1 (March 1983), pp. 1−43.

[4] ANSI/ISO, *Ada 9X Reference Manual*, ISO/IEC Standard 8652, Intermetrics, Cambridge, MA (1994).

[5] Ashenden, P. J., and Wilsey, P. A., "Polymorphic Abstract Data Types in VHDL," *Proc. ICEHDL '95*, Las Vegas, NV (1995), pp. 35−39.

[6] Benzakki, J., and Djaffri, B., "Object Oriented Extensions to VHDL, the LaMI Proposal," *Proc. CHDL '97*, Toledo, Spain (forthcoming, 1997).

[7] Bergé, J. M., Nebel, W. and Putzke, W., *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).

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

[9] Booch, G., *The Best of Booch*, SIGS Books and Multimedia, New York, NY (1996).

[10] Brooks, F. *The Mythical Man-Month*, Addison-Wesley, Reading, MA (1975).

[11] Cabanis, D., and Medhat, S., "Classification-Orientation for VHDL: A Specification," *Proc. VIUF Spring '96 Conference*, Santa Clara, CA (February 1996), pp. 265−274.

[12] Dunlop, D. D., "Object-Oriented Extensions to VHDL," *Proc. VIUF Fall '94 Conference*, McLean, VA (1994), pp. 5.1−5.9.

[13] Dunlop, D. D., *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).

[14] Ecker, W., "An Object-Oriented View of Structural VHDL Description," *Proc. VIUF Spring '96 Conference*, Santa Clara, CA (February 1996), pp. 255–264.

[15] Hoare, C. A. R., "Monitors: An Operating System Structuring Concept," *CACM*, Vol. 17, No. 10 (October 1974), pp. 549–557.

[16] Hoare, C. A. R., "Communicating Sequential Processes," *CACM*, Vol. 21, No. 11 (November 1978), pp. 934–941.

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

[18] IEEE DASC P1076a Working Group, *Shared Variable Language Change Specification (PAR 1076A)*, http://vhdl.org/vi/svwg/lcs/lcs.htm (1996).

[19] Kumar, S., Aylor, J. H., Johnson, B. W., and Wulf, W. A., "Object-Oriented Techniques in Hardware Design," *IEEE Computer*, Vol. 9, No. 6 (June 1994), pp. 64–70.

[20] Lim, J. and Johnson, R. E., "The Heart of Object-Oriented Concurrent Programming," *Proc. ACM SIGPLAN Workshop on Object-Based Concurrent Programming*, *ACM SIGPLAN Notices*, Vol. 24, No. 4 (April 1989), pp. 165–167.

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

[22] Mills, M. T., *A Minor Syntax Change to VHDL Yields Major Object Oriented Benefits*, unpublished paper, ftp://vhdl.org/vi/oovhdl/papers/mills.oct.95.ps (1995).

[23] Ramesh, C. R., "Object Orienting VHDL for Component Modeling," *Proc. VIUF Fall '94 Conference*, McLean, VA (1994), pp. 5.17–5.28.

[24] Schumacher, G., and Nebel, W., "Inheritance Concept for Signals in Object-Oriented Extensions to VHDL," *Proc. Euro-DAC '95 with Euro-VHDL '95*, Brighton, UK (1995).

[25] Swamy, S., Molin, A., and Covnot, B., "OO-VHDL: Object-Oriented Extensions to VHDL," *IEEE Computer*, Vol. 28, No. 10 (October 1995), pp. 18–26.

[26] Wegner, P., "Dimensions of Object-Based Language Design," *Proc. OOPSLA '87* in *ACM SIGPLAN Notices*, Vol. 22, No. 12 (December 1987), pp. 168–182.

[27] Willis, J. C., Bailey, S. A., and Newschutz, R., "A Proposal for Minimally Extending VHDL to Achieve Data Encapsulation, Late Binding and Multiple Inheritance," *Proc. VIUF Fall '94 Conference*, McLean, VA (1994), pp. 5.31–5.38.

[28] Wirth, N., "From Programming Language Design to Computer Construction," *CACM*, Vol. 28, No. 2 (February 1985), pp. 160–164.