

Object-Oriented Extensions to VHDL

Douglas D. Dunlop

Intermetrics, Inc.
7918 Jones Branch Drive, #710
McLean, VA 22102

dunlop@wash.inmet.com

Abstract

The advances in object-oriented techniques in recent years have had a major impact on software development methodologies. It has become clear that these techniques offer significant advantages over more traditional "structured development" approaches throughout the software life cycle. Due largely to these insights, researchers have begun to explore the possible role of object-oriented technology in the development of hardware and mixed hardware/software systems. Hardware description languages, and in particular VHDL, often play a central role in the development of these systems. As such, a specific topic of interest is how VHDL would benefit from object-oriented technology and what corresponding benefits would accrue in the hardware development process. In this paper we explore the idea of "object-oriented extensions" to VHDL, survey the work that has been done in this area, compare and contrast approaches, and make recommendations for further work.

1. Introduction and Motivation

Beginning with the development of the Simula programming language in the late 1960s, there has been a great deal of work in developing object-oriented technology and applying this technology to the software development process. Careful application of the object-oriented paradigm in software development has been shown to increase flexibility, promote re-use, and ease development. Established programming languages have been revised and extended to better support object-oriented development (see, e.g., [2], [5]), while object-oriented capabilities seem to be a "given" these days for newly developed programming languages (see, e.g., [9]). Early-life-cycle software development activities such as requirements analysis and software design also benefit from the application of object-oriented principles (see, e.g., [4]), and indeed, an often-cited advantage of an object-oriented methodology is better integration and improved continuity across different phases of the software life cycle.

What is object-oriented development about? Briefly, the object-oriented paradigm involves viewing a system as a collection of interacting objects. Languages that support this form of development typically provide an *encapsulation* mechanism for defining the interface of an object. The interface of an object usually consists of a set of *operations*

(or methods) for that object. *Inheritance* in object definitions is frequently provided as a way of dealing with the commonality that exists across different kinds of objects. Often related to inheritance is the idea of a *class*, a collection of objects all sharing a certain set of properties. A final important idea is that of *run-time binding*, that is, a means of deferring the determination of what operation (or method) body must be invoked until run time. It is the combination of fundamental capabilities such as these that is dramatically changing the way software engineers look at software and the way they develop their products.

Given this effect on the way software is built, those interested in hardware development have begun to investigate the application of object-oriented technology to the process of developing hardware (see, e.g., [8]). The essential objective here is to leverage off the object-oriented advancements in the software area to develop improved techniques and methods for hardware development. The hope of payoff with this line of study is particularly high for the earlier stages of the hardware development process where the level of modeling that takes place bears more resemblance to the modeling, design, or programming of a software system. Due to the prominent role VHDL plays in many hardware-development methodologies, an area of active interest in this regard is VHDL. One of the first papers to look at object-oriented development in VHDL is [14]. In this paper the author discusses the advantages of an object-oriented approach to modeling in VHDL but concludes that the present language is not well suited for this form of model development and that language extensions would be required in order to apply this approach in a reasonable way.

In the sections that follow we expand on this idea of enhancements to the definition of VHDL that would allow the application of the object-oriented paradigm in VHDL model development. We begin in the following section with a look at some recent changes in Ada to support object-oriented programming by way of type classes. In Section 3, we then consider the general idea of extending VHDL to provide a similar object-oriented form of type classes. VHDL design entities as objects are discussed in Section 4 and in Section 5 we look at object-oriented mechanisms for communication and synchronization between design entities. VHDL packages are briefly discussed in Section 6 and we conclude with a brief discussion of related ongoing work in enhancing VHDL.

2. "Programming by Extension" in Ada 9X

The original designers of VHDL borrowed many features of the Ada programming language and applied them in the definition of VHDL. In particular, many of the "behavioral" aspects of VHDL such as sequential statements, subprograms, and expressions, are based on similar functionality in Ada. VHDL borrows much from Ada for its type system, including its mechanism for numeric types, the concept of overloading subprograms and operators, and language facilities for building complex data structures such as records and access types. Finally, VHDL employs the Ada notion of "elaboration" and includes the Ada encapsulation unit of a "package" with a separate specification and body.

During the last four years, a revised version of the Ada standard has been under development. The revised Ada language definition (see [2]) is presently in Draft International Standard form and is expected to be approved in 1994. The main enhancements incorporated in this revision of Ada consist of *protected objects*, a high-performance mechanism for sharing data between tasks; *hierarchical libraries*, an

improved facility for organizing compiled Ada program units; and full support for *object-oriented programming*.

The Ada enhancements to support object-oriented programming were largely due to successes in other programming languages such as Smalltalk-80 ([6]), C++ ([5]), Eiffel ([10]), and CLOS ([3]) in facilitating the maintenance, re-use, and evolution of software through paradigms similar to what is called *programming by extension* in [1]. The essential idea here is being able to construct extensions to established software components in building new functionality without modifying or disturbing the existing components.

Programming by extension in Ada 9X rests on two fundamental capabilities. One of these is the ability to define a new type in terms of an existing type through a process called "type derivation". When a new type is derived from a second type, the new type inherits the operations (subprograms) and data that are applicable for the second type. The new type may choose to override the inherited operations and "extend" the type by adding new data (i.e., adding "fields" or "components"). The set of types that are derived, either directly or indirectly, from a given type can be thought of as forming a class of types rooted at the given type. The second capability in Ada 9X to support programming by extension is the ability to write subprograms whose parameters are constrained to belong to a class of types rather than a specific type. Operations on the parameters within the subprogram take place by way of run-time binding (also known as, e.g., "late binding", "dynamic binding", and "run-time dispatch") to the appropriate operation based on the actual type of the parameters involved.

As a VHDL-oriented illustration of these ideas, consider a scenario where a CPU and its instruction set are being modeled. One might begin with an abstract "root instruction type" that corresponds to any instruction the CPU executes. This type might have associated with it abstract operations to display and execute the instruction. Derived from this root type might be a type corresponding to a "register-to-register" instruction. This type might include fields that indicate the specific operation to be performed and the two registers involved. Specialized versions of the operations that display and execute instructions would also be provided for this type. Another type derived from the root type might be a type corresponding to a "register-to-memory" instruction. Here fields would be included to indicate the specific operation, the register, as well as whatever data was required for addressing memory. Again, specialized versions of the display and execute routines would be provided for this type. Given this arrangement, subprograms could be written that operate on *any* instruction with run-time binding taking place as required for uses of the display or execute routines within the subprogram. Such subprograms are guaranteed to remain correct and applicable, even as the model is extended to cover new kinds of instructions (e.g., a "memory-to-memory" instruction).

The reaction of the Ada community to these language extensions has been extremely positive. There is strong desire among users to get the new Ada standard in place and for Ada compiler vendors to upgrade their implementations to the new language. This is due in large part due to the features for programming by extension in Ada 9X and the need for support like this in tackling the complex programming tasks of the 1990s.

3. Applying Type Classes to VHDL

It is clearly possible to extend VHDL along the lines described in the previous section, i.e., by adding to VHDL similar support for type derivation with inheritance and run-time binding. This is essentially the approach taken in [12]. Movement in this direction is clearly more of a "stretch" for VHDL than it is for Ada 83 (the current Ada standard, i.e., [15]). This is because Ada 83 already supports a weak version of type derivation and includes the basic notion of the set of "derivable operations" associated with a type. This combination provided the Ada 9X designers with an obvious foundation on which to build the Ada 9X type-class mechanism.

Due, in part, to this difficulty in extending the VHDL type system in this manner, there exists another strategy that adds to VHDL a system of classes with inheritance and run-time binding without integrating this capability into the normal VHDL type system. The proposal in [17] appears to be an example of this approach. It also has more of a C++ than an Ada 9X feel to it. Both approaches offer roughly the same functionality to the user and differ primarily in the philosophy of extending the language. Both approaches realize benefits in terms of ease of re-use (or "recycling" as it is described in [17]), maintenance, and improved support for system-level modeling.

A third proposal for type classes in VHDL can be found elsewhere in these proceedings ([16]). This approach offers a new encapsulation mechanism for explicitly associating a type with a set of operations. The VHDL "subtype" mechanism is extended to provide the inheritance capability and run-time binding is also supported. The reader is referred to [16] for details.

3.1 VHDL "Variant Records"

One of the language needs that was not met in the 1993 revision of the VHDL standard was language support for what are sometimes called "variant records". This terminology comes from Pascal and Ada, where a record type can be defined whose layout (i.e., the number and types of the fields in the record) varies according to the "kind" of the record. The need for a capability like this in VHDL can be apparent in modeling, for example, a message-passing system where the messages being exchanged are of different forms and sizes. One wants a single "message type" that allows for a variety of different kinds of messages with different representations.

Due largely to lack of time in working out the detailed semantics for signals of a variant record type, variant records were not incorporated into VHDL 93. From the point of view of the object-oriented future of VHDL, however, it may be that this is just as well. One of the important lessons from the Pascal and Ada experience with variant records is that variant records tend to be difficult to work with from a re-use and software evolution point of view. For example, the act of "adding a new variant" not only requires changing the definition of the variant record (typically forcing a great deal of re-compilation) but, even worse, everywhere in the software that the code explicitly checks to see what the "current variant" is (typically by way of a case statement), the code usually needs to be modified to take into account the new variant.

Indeed, it is largely maintenance difficulties such as these that motivate the interest in object-oriented programming. A system of type classes, where the individual types correspond to the "record variants", makes the "adding a new variant" scenario considerably more manageable. The most important difference with the type-class

approach to record variants is that, typically, there exists in the code no explicit checks to test the current variant of the data; essentially with the object-oriented approach this is done "behind the scenes" by the run-time binding. It is the run-time binding that allows existing code to continue to function without modification as new types are added to the class in order to implement new data variants.

3.2 Experience with Type Classes in MHDL

The MIMIC Hardware Description Language (MHDL, see [11]) is a recent hardware description language targeted toward analog and microwave CAD. One of the important characteristics of MHDL is the support it provides for type classes. Roughly speaking, the functionality provided by MHDL in this regard is similar to that described above, i.e., it allows for classes of types that may share a common set of operations, it allows functions to be written that accept parameters constrained to belong to a type class rather than a specific type, and it provides run-time binding within these functions to implement operations on the parameters.

Although it is still early in the development of MHDL, the usefulness of its system for type classes is apparent when looking at the MHDL code for the "pre-defined MHDL environment" (see Chapter 10 of [11]). The goal for this MHDL code is to pre-define a set of very general-purpose routines that, in particular, do not have assumptions built into them about the underlying types of the operands involved. As a simple illustration, the pre-defined MHDL environment includes a general routine that performs matrix multiplication and that does not assume any particular index type for the arrays or any particular type for the array elements. It does however assume that the index type and element type each belong to a certain class, thereby allowing the routine to make use of the operations in the classes (implemented by run-time binding) in computing the product of the matrices. In general, the MHDL pre-defined environment is a successful example of using an object-oriented methodology in developing highly-reusable and highly-flexible modeling resources. While not a complete substitute "generic packages" or "parameterized packages" as have occasionally been suggested for VHDL, it is evident from the MHDL experience that a type class system along the lines that have been described in this section would go a considerable way toward meeting this need.

4 VHDL Design Entity Objects and Classes

Due to the critical role of a design entity (i.e., an entity/architecture pair) in VHDL, there is natural interest in basing object-oriented extensions to VHDL on the idea of a design entity as an object belonging to a class. In a sense, this concept is already supported in the existing language by the possibility of multiple architectures for a given interface (entity declaration). From the perspective of the current language then, a class of design entities is simply those that share a common interface.

This is a simple and largely effective scheme for grouping design entities. It is, however, limited in two ways.

First, it assumes that the elements in a design entity class share a common port/generic type profile. When modeling a hardware device at different levels of abstraction, it is not uncommon to want to consider the types of the ports on the device as being different at the differing levels of abstraction. As an example, at a high level it may make sense to view the port on a device as an integer while at a more detailed level it may be best to

consider the same port as a vector of bits. It is worth noting that a scheme of VHDL type classes, similar to that described in Section 3, is a potential resource in addressing this problem. Assuming the existence of type classes, it would be meaningful to allow the ports on an entity interface to be associated with a type class rather than a specific type. Potentially this would allow design entities to be more "generic" in the sense of being instantiatable with a variety of differently-typed signals/ports as actuals. It also might allow various specializations of such a design entity for some set of concrete port types to be grouped together and related to one another in a language-defined way.

Second, the present scheme for classes of design entities lacks an inheritance capability. While there is the ability of an architecture body to "inherit" the declarations of the corresponding entity, this is a very limited case and does not allow an entity declaration to inherit, e.g., the ports of another entity declaration. Nor does it allow an architecture body to be built by extension off another architecture body. As an example application of an inheritance capability across design entities, consider an abstract design entity that captures the power and ground characteristics of some collection of devices. Such a power/ground design entity might include ports for the power and ground, associated generic parameters, defaults for these parameters, and perhaps a corresponding set of assertions. Design entity models of devices that share these power and ground properties could be built by inheriting from the abstract power/ground design entity. Such an approach is useful from the point of view of identifying commonality across design entities and in promoting a partitioning of concerns. Various forms of design entity inheritance are described in both [7] and [12].

5. Message Passing, Higher-Level Synchronization Between Design Entities

As the desire increases to apply VHDL in earlier phases of the product life cycle and to use VHDL at higher levels of abstraction, there has been increased interest in the development of a set of object-oriented VHDL extensions intended to better support system-level and architecture-level modeling. Example applications here include hardware/software co-design and performance modeling. A specific development effort along this line (see [7]) is currently taking place under the ARPA/Tri-Service RASSP program.

Central to the emerging approach in [7] is the notion of an *EntityObject*, a generalized kind of VHDL design entity. The definition of an EntityObject allows for inheritance. When an EntityObject is defined by inheritance from a parent EntityObject, the inheritable features of the parent may be optionally re-defined. EntityObjects include *operations* that are used for communication between EntityObjects. Architecture bodies of EntityObjects include implementations of the operations and may contain instance variables to be used in implementing the operations. EntityObjects are created by a special object declaration that is analogous to a component instantiation statement for a normal VHDL design entity. The concept of an EntityObject *handle* is provided to allow references to be created to EntityObjects (e.g., an array of handles can be used to sequence through a collection of EntityObjects).

EntityObjects communicate by way of their operations. Operations are a synchronization mechanism that is based on a message-passing paradigm, thereby providing more convenience than the existing VHDL inter-process synchronization based on signal sensitivity lists and events. Message processing has sequential semantics. In particular, messages are queued, handled by the receiver sequentially, and

the message sender is blocked until the message sent is processed. Multiple EntityObjects may implement the same operations, thereby implying run-time binding.

More details on this approach may be found in [7]. A benchmark application example that uses the extended language features as well as a front-end tool that implements the extended language on top of standard VHDL are under development.

6. VHDL Packages

We previously discussed what type classes might be like in VHDL and we have also looked at object-oriented VHDL from the perspective of design entity classes. For the sake of completeness, here we briefly consider a third view, namely a package-based perspective on object-oriented VHDL.

In the present language, packages lag considerably behind design entities in terms of abstraction capabilities. Packages may not be instantiated or parameterized in any way, nor are multiple bodies allowed. To make these ideas more concrete, one cannot build a general "bit vector arithmetic" package where the word size for the arithmetic operands could be controlled by a "package generic parameter", where one could select between a ones complement and twos complement implementation by selecting between a pair of "variant package bodies", and where one could tailor the package in these ways on a use-by-use basis, particularizing it each time the package was "instantiated." Enhancements along these lines, possibly together with more convenient means for package inheritance and selected re-definition, would improve the object-oriented nature of VHDL packages.

7. Related DASC VHDL Work

A Study Group on Object-Oriented Extensions to VHDL was recently created within the DASC (the Design Automation Standards Committee within the IEEE Computer Society) and has the objective of investigating the idea of incorporating object-oriented technology into VHDL. (Please contact the author if you have interest in becoming involved with this group.)

The VHDL Shared Variables Working Group (P1076A) of the DASC is examining VHDL language changes that would provide a language-defined way of obtaining mutually-exclusive access to variables that are shared across multiple VHDL processes. The emerging solution from the group is based on the concept of a "monitor" that couples data and associated operations and that has an object-oriented look and feel to it. This concept presently lacks facilities for inheritance or run-time binding but could be viewed as a special case of a more general and widely applicable language mechanism.

The VHDL Analog Extensions Working Group (P1076.1) of the DASC is developing extensions to VHDL to support modeling of analog devices and electrical-level description. This group has briefly explored the idea of classes of related design entities (referred to occasionally as "design entity overloading") where the corresponding interface objects (ports and generics) across these design entities are of distinct but related types. An application scenario is a collection of design entities for the same device that vary with respect to the modeling view taken of the hardware. This idea is similar in several regards to the general concept of design entity classes discussed in Section 4.

8. Concluding Remarks

As the use of VHDL continues to increase and the language is put to more and more challenging uses, it is important to consider ways in which VHDL might grow in order to increase its usefulness to the end user. Although there are no doubt a number of individual and specific improvements that could be made to VHDL, it seems that if there is a general direction for promising language evolution it would be in making VHDL more object oriented. Much of the work accomplished thus far, including that described in [7] and [16], suggests that extending VHDL with object-oriented enhancements is indeed a legitimate line of potential language growth.

It is important that VHDL language extensions towards this end strike the right balance between new capability for the user and cost in terms of increased language complexity and impact on new and existing language implementations. To make progress in finding this balance there is a need for fully-developed and refined proposals, analyses of tradeoffs with respect to integration into the existing language, user-level rationale and motivation with supporting examples, and perhaps a general implementation model. It is hoped that this work will go forward and be reviewed and guided by the IEEE VHDL committees.

Acknowledgments

The author is grateful to Intermetrics for its support with this line of work.

This work was funded in part by USAF/AFMC/ASC contract F33615-94-C-1549.

References

- [1] *Ada 9X Rationale*, Version 5.0. Intermetrics, Inc, 1994.
- [2] *Ada 9X Reference Manual*, Version 5.0. Intermetrics, Inc, 1994.
- [3] Bobrow, D., DeMichiel, L., Gabriel, R., Keene, S., Kiczales, G., and Moon, D. *Common Lisp Object System Specification X3J13 Document 88-002R*. SIGPLAN Notices Vol. 23.
- [4] Booch, G. *Object-Oriented Analysis and Design with Applications*, 1994, Benjamin/Cummings.
- [5] Ellis, M. and Stroustrup, B., *The Annotated C++ Reference Manual*, Addison-Wesley.
- [6] Goldberg, A. and Robson, D., *Smalltalk-80: The Language*, Addison-Wesley, 1989.
- [7] Hurst, D. *OO-VHDL: Object-Oriented Extensions for VHDL*, slides included with DASC Study Group on Object Oriented Extensions minutes of June 11, 1994 (San Diego), distributed by CMS.
- [8] Kumar, S., Aylor, J., Johnson, B. and Wulf, W., *Object-Oriented Techniques in Hardware Design*, IEEE Computer, June 1994.
- [9] Madsen, O., Moller-Pedersen, B. and Nygaard, K. *Object-Oriented Programming in the BETA Programming Language*, Addison-Wesley, June 1993.
- [10] Meyer, B. *Object-Oriented Software Construction*, Prentice Hall, 1988.
- [11] *MHDL Language Reference Manual*, Version 2.0, Intermetrics, Inc, March 1994.
- [12] Mills, M., *Proposed Object Oriented Programming (OOP) Enhancements to the Very High Speed Integrated Circuits (VHSIC) Description Language (VHDL)*. WL-TR-93-5025, August, 1993, Wright Laboratory.

- [13] Oczko, A., *Hardware Design with VHDL at a Very High Level of Abstraction*, Proceedings of the 1st European Conference on VHDL Methods, 1990.
- [14] Perry, D., *Applying Object Oriented Techniques to VHDL*, VIUF Spring 1992 Conference.
- [15] *Reference Manual for the Ada Programming Language*, ANSI/MIL-Std-1815a edition, 1983.
- [16] Willis, J., *A Proposal for Minimally Extended VHDL to Achieve Data Encapsulation, Late Binding, and Multiple Inheritance*, to appear in VIUF Fall 1994 Conference.
- [17] Zippelius, R. and Muller-Glaser, K. D., *An Object-Oriented Extension of VHDL*, VHDL Forum for Europe, Spring 1992.

•