Principles for Language Extension to VHDL to Support High-Level Modeling

Peter J. Ashenden <petera@cs.adelaide.edu.au>

Philip A. Wilsey <phil.wilsey@uc.edu>

Technical Report TR-03/97

Department of Computer Science
The University of Adelaide, SA 5005
Australia

Technical Report TR-204/05/97/ECECS

Department of Electrical and Computer Engineering
and Computer Science
University of Cincinnati,
PO Box 210030
Cincinnati, OH 45221−0030
USA

This work was partially supported by Wright Laboratory
under USAF contract F33615−95−C−1638

Abstract

This report reviews proposals for extensions to VHDL to support high-level modeling and places them within a taxonomy based on 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.

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 space. This has 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 when partitioning of the design is not firm.

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). This avoids the need for changing design languages part-way through the design flow, with consequent interface and equivalence checking problems. VHDL [28], as it currently stands, is a hardware description language that is well suited to modeling small-scale to medium-scale systems at levels of abstraction up to register transfer. 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. Indeed these expectations are expressed as requirements by the IEEE DASC OO-VHDL Study Group [6]. 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 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 ex-

tension. 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.

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, 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 [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. This also makes 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 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.

Proposal	Data Modeling	Structural Modeling	System-Level Modeling
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	