Entity Overloading for Mixed-Signal Abstraction in VHDL[*]

C.-J. Richard Shi
Department of Electrical and Computer Engineering
University of Iowa
Iowa City, Iowa 52242, U.S.A.
cjshi@eng.uiowa.edu

Abstract

In this paper we propose to extend VHDL with entity overloading. With a minimal change to existing VHDL, entity overloading provides a strong support for mixed-signal, mixed-level, and mixed-domain abstractions. It is particularly promising in resolving some issues in VHDL-A language design. Furthermore, we illustrate that entity overloading can be combined with certain modeling rules to achieve polymorphic netlist.

1. Introduction

With the rapid development and convergence of computer, consumer electronics and communication technologies, mixed-signal (analog and digital) circuit design is becoming increasingly important. While tools exist to help designers to design digital circuits that work right the first time, mixed-signal design still remains an art. Among many issues, a hardware description language (HDL) capable of describing mixed-signal circuits spanning through the whole design cycle has been identified as a key demand by IEEE, semiconductor industry, both European ESPRIT and U.S. Department of Defense. As noted by Rob Rutenbar, the availability of such a hardware description language will help the development of analog synthesis [6].

The design of mixed-signal hardware description languages, however, presents a great challenge. The root of the difficulty comes mainly from the fundamental difference in how the behavior of analog and digital circuits are described—denotational vs operational. Analog HDLs only specify a set of constraints (usually ordinary nonlinear differential equations) to be satisfied, whereas digital HDLs are similar to traditional programming languages. This difficulty is embodied in several aspects of HDL design. In this paper, we consider mainly the structure aspect. The particular issue we address here is related to VHDL-A, a major effort undergoing to extend VHDL to analog and mixed-signal circuits [8, 12].

Consider a mixed-signal circuit. The same real-world component may have several abstractions that differ not only in their architectures, but also in their interfaces. In particular, the corresponding interface signals for different models of the same design entity cannot be characterized by the same data type. Current VHDL requires the user to define different named entities. Further, VHDL does not provide a direct mechanism to indicate that those entities correspond to the same real-world component. This restriction not only contradicts the naming convention of designers, but also complicates design management.

A complete solution to this problem may need the concept of object-oriented modeling or the use of more sophisticated typing mechanisms. Both require significant changes to VHDL. In this paper, we describe entity overloading—an approach that requires only a minimal change to existing VHDL. We also discuss how entity overloading may be used together with certain modeling rules to achieve polymorphic netlist; i.e., a netlist description that has multiple meanings dependent upon the types and natures of its interface signals.

We note that MHDL have taken a different approach to language design in order to support mixed-signal, mixed-domain abstraction [7]. Although our research is targeted towards VHDL, we expect certain concepts developed in this paper can be useful to other hardware description languages as well.

This paper is organized as follows. In Section 2, we introduce some concepts of hardware description languages. Our definitions are by no means precise nor

[*]This work is sponsored by U.S. Defense Advanced Research Projects Agency (DARPA) under grant number F33615-96-1-5601 from the United States Air Force, Wright Laboratory, Manufacturing Technology Directorate.

rigorous from a language lawyer's perspective, but our intention is to give a unified and intuitive treatment of basic concepts related to both analog and digital HDLs. In Section 3, we analyze the difficulty with the current VHDL in supporting mixed-signal abstraction. In Section 4, we propose entity overloading. Its implications to language design are described intuitively in Section 5. Section 6 shows how entity overloading combined with certain modeling rules can be used to achieve polymorphic netlist. Section 7 concludes the paper.

2. Hardware Description Language Concepts and VHDL

There are two approaches to hardware modeling: *behavioral and structural.* Behavioral modeling refers to modeling the behavior of a hardware system. The structure of behavioral modeling, i.e., the way a behavioral description is organized is irrelevant to an actual hardware. Structural modeling is to model the real structure of a hardware design, which usually corresponds to a specific hardware implementation. In order to model the structural composition of individual components to form a complete hardware system, the composition/structure semantics of the description must mimic or simulate the effect of real hardware connection. Therefore, the notion of *connection semantics* is often used for hardware description languages. In general, a *connection node* in a hardware system has two roles: as a carrier of information (structural aspect), and the actual information it carries (behavioral aspect).

The basic built-in connection semantics in VHDL is the notion of *nets.* The net semantics is achieved by several language constructs. A fundamental assumption used by VHDL for digital systems is that the dual role of nets as the information carrier and information itself can be unified. Therefore, a net is implemented as a SIGNAL in VHDL. Connecting two nets are equivalent to binding two signals.

Perhaps the most useful mechanism for a HDL to describe a complex system is design encapsulation. Basically, each physical component only relates to its environment (or say other components) through its interface nets. From the outside perspective, such interface nets are defined as PORTs in VHDL. Formally, a port is the *formal* of a net, whereas a net is the *actual.* The connection semantics is enforced when a port is *bound* to a net. (The connection semantics in HDLs is equivalent to the binding semantics in programming languages.)

The connection semantics in VHDL is relatively simple, and is similar to the binding semantics of an ordinary programming language. The complicated part is the concept of resolution function, which basically defines how the value of a net will be determined if there are several *drivers* for the net.

The connection semantics for analog circuits differs significantly from that of digital circuits (such as that in VHDL). First of all, information carriers and information have to be separated. Each connection point (actual) is associated with a voltage (called node voltage), and each terminal (formal) is associated with a current (called terminal current). For example, Fig. 1

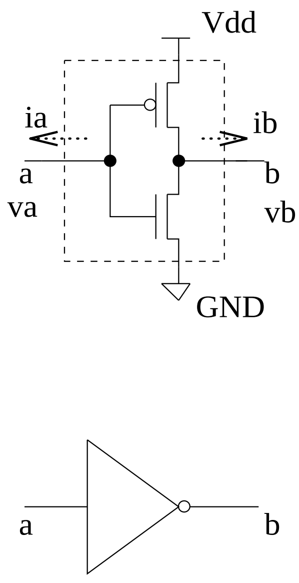

Figure 1: A CMOS inverter.

illustrates a CMOS inverter, where its logic abstraction has two signals: *a* and *b*, but its KVL/KCL abstraction involves both voltages and currents. Then the connection semantics implies that the following set of equations be satisfied:

1. Kirchoff Current Law (KCL): The summation of all the terminal currents entering a node must be equal to zero.
2. Kirchoff Voltage Law (KVL): The voltages of all the terminals connected to a node must be the same.

This is called the conservation-law (or KVL/KCL) semantics. We note that separating the concept of formal and actual of the information carriers from that of the information itself provides a much cleaner definition. Previously, both a voltage and either a current or an array of currents are associated with a node.

We also remark on another difference between a digital HDL and an analog HDL. For digital HDLs, the connection semantics can be resolved/enforced locally, where the enforcement of KVL/KCL requires a global evaluation. Digital HDLs share the same locality for both description and evaluation, whereas analog description can be local, but its evaluation must be global.

For top-down analog modeling, another type of connection semantics called *signal-flow semantics* is used. Each net is associated with only one quantity, similar to a digital signal. But the value of a quantity is determined by solving the set of equations. For such quantities, input and output modes can be defined similar to digital MODE. However, it does not necessarily specify the order of evaluation, but rather the order of dependency. Analog simulators may exploit this feature for efficient simulation.

From the language perspective, signals and quantities are objects that have hardware meanings, whereas variables, constants (and generics) are only auxiliary objects that do not have hardware counterparts. A type can be associated with an object, which specifies the property of the object. Due to the usefulness of types in avoiding first-order modeling errors, VHDL adopts the strong typing rule: only objects with the same type can connect together. This reflects in terms of connection semantics: only nets with the same (more precisely, closely related) type can connect together, otherwise type conversion is needed.

Abstraction level vs *abstraction domain.* We propose to use abstraction level to refer to different architectures for one entity, that all share the same interface signal definitions (at the same abstraction domain). We use abstraction domain to refer to those abstractions for the same real world component but with different interface signal definitions. For example, a register described at the world level and that described at the bit level belong to two different abstraction domains. A low-pass filter described at the time domain or that at the frequency domain belong to two different abstraction domains. On the other hand, a transistor-level description of an operational amplifier and its macromodel belong to the same abstraction domain but different abstraction levels. A behavior view of a digital block and its structure view (if they share the same interface) are in the same domain but at different abstraction levels.

Abstraction vs simulation. Abstraction domain is closely related to simulation mode. The types and natures of interface signals determine the types of simulation it can be used for. Simulation of two real-world components in the same abstraction domain but at the abstraction levels do not involve any conversion. However, simulation of two components in different abstraction domains involves certain conversions. Such conversions may range from the simplest such as "type conversion" to more complex such as "domain conversion", (from time-domain to frequency-domain or vice verse), to "nature conversion" (from mechanical to thermal or to electrical), etc.

3. Issues with Mixed-Signal Abstraction in VHDL

In current VHDL, a design/component may be described by a single entity with multiple architectures. This mechanism supports digital design well, where a high-level specification and various implementations can be described as different architectures. Then, changing from specification to implementation, or from one implementation to the other, can be done nicely with the use of the configuration mechanism. However, the ability of VHDL in supporting mixed abstractions is limited: all architectures of an entity must share the same interface—ports (generics) and their types must be the same as specified by the entity declaration. Thus the current entity/architecture/configuration mechanism supports mixed abstractions at the same "data-type" level but not designs described at the different "data-type" levels.

For example, consider a register declared as integer type or bit type: the user must define two different named entities. This not only causes the problem of readability, but also leads to an undesirable situation: replacing an integer-typed register with another register with exactly the same functionality but bit-typed requires updating the VHDL source code that uses the register. Nevertheless, this is not a major problem for VHDL, the entity/architecture/configuration mechanism supports most digital design applications.

The problem is getting worse for switch-level modeling [2, 10], and even serious for analog modeling [9, 11]. It is OFTEN the case for the same design/component, there are many "architectures" that not only differ in internal specifications, but also differ in types of interfaces (ports). In terms of the board-socket-chip analogy of Alec Stanculescu (see Perry's, 2rd, p. 189) [5], it is often the case that, for a socket on the board, many chips can be plugged into that socket, but the corresponding pins can not be characterized by the same type. (In other words, an implicit VHDL assumption that those chips that can be plugged into one socket must have the same interface (type) characterization, is no long true.)

4. Notion of Entity Overloading

What seems a natural extension is to overload entities. In terms of the board-socket-chip analogy, the same (component) name and PORT MAP construct will be used in describing a socket on the board (component instantiation), regardless of what kinds of chips (as long as they have the same intended function/use) to plug in (component declaration). Then for those chips that have the same pin/interface types, we declare one entity with the COMPLETELY specified PORTs. As a result, for one socket, there may be several entities—with the same name—to plug in.

More technically, we may declare several entities with the same name, each with its own complete interface definitions, and each may be associated with a set of architectures that have the same interface. Then, in order to instantiate a lower level overloaded entity in a higher level architecture, a corresponding component declaration must be given completely. If an architecture instantiates the same named component in several places, each with different interfaces, then several entity instances (several components with the same name) must be declared.

Clearly, using the mechanism of entity overloading, the user can use the exact same name for different abstraction models for the design/component (the name could be taken from the data book). Further, switching from one architecture of an instantiated component to the other—even if they differ in terms of interface types—does not involve the change of the component instantiation statement (netlist).

This extension preserves the upward compatibility to VHDL'93. All the VHDL libraries and designs created before will be valid under the extension. Indeed, to VHDL beginners and programmers who do not want to use this feature, entity overloading is an advanced topic just like subprogram overloading. On the other hand, the language factor should be considered much higher than that of subprogram overloading [3].

5. Language Implementation of Entity Overloading

Resolution of entity overloading can be done by the compiler in the similar way as for subprogram overloading.

In the simplest case, entity overloading resolution can be achieved by types/natures of signals. When there is another overloading function, subprogram, or entity involved, then a restriction rule similar to the rules in VHDL'93 for overloaded subprograms applies. A central concept here is that the user must provide enough information such that the compiler can figure out which entity should be used.

One additional issue arises in case of entity overloading: when we define an architecture, we have to say which interface (entity declaration) the architecture is going to use. This can be done in two ways. One is to request the user to put all the architectures corresponding to one interface together with the interface declaration in one VHDL file. The other is to introduce a new language structure mechanism (called model/group/cluster for example) to group together those architectures with their interface (entity) declaration.

6. Towards a Polymorphic Structural Description

Netlist reuse is motivated by the following three observations:

1. In a hierarchical and structured design environment, there are many levels of netlist description.
2. Many levels of netlist descriptions can be shared among different abstractions.
3. Parameter instantiation does not occur everywhere. For example, in the standard cell design environment, one component is used with the same geometry, and used many places in the layout design, it is reasonable to assume that all such components have the same parameters; for example, delays.

For example, consider a digital circuit that has been designed with many hierarchies. Ideally, the description of these hierarchies should be usable for logic, switch-level and electrical simulation; only the models of the leaf cells need to be changed. This is what we mean by *netlist reuse.*

In general, netlist reuse can happen at several different phases. One is at the analysis phase. When one abstraction is switched to another, no re-analysis of the netlist, but re-elaboration is needed. This mechanism may be called *ad hoc netlist overloading.* It can happen at the elaboration phase. By using late typing, or deferred typing, the types and natures of interface lists are only bound at the elaboration phase. We may call this *netlist overloading.* Finally, it can happen at the simulation phase, i.e., the types and the natures of interfaces should be determined by its input signals during simulation. We may call it *netlist polymorphism,* due to its similarity to the concept of polymorphism in programming languages. The latter two approaches are generally difficult to exploit in current VHDL (requires significant language re-design). In this section, we shall restrict ourself to netlist reuse at the analysis phase. We illustrate that it can be achieved by using entity overloading and certain modeling (coding) rules.

We need to introduce a notion of significant port. A *significant port* refers to a port that can be used across various abstractions. Note that signal carriers on a significant port can vary for different abstraction. For example, a port can carry only one analog quantity at the signal-flow abstraction or one digital signal at the logic-level abstraction, but it carries both voltage and current (two quantities) at the electrical abstraction level. Typically, either voltage or current from the electrical abstraction level is selected for abstraction at the higher level. Using this definition, ports used for power and ground at the electrical abstraction domain are insignificant, since they do not have the corresponding higher domain (logic domain) abstractions. See Fig.1 for example.

There are two application scenarios. The first scenario is that among several abstraction domains, only the types and natures of signal carriers are different. The second scenario is that not only the types and natures of signal carriers vary, but also ports are different.

We also need to distinguish two types of information associated with an interface list: *abstraction-dependent and abstraction independent.* Clearly, GENERICs are often tightened together with a specific abstraction domain (even a specific abstraction level).

In order to make a netlist re-usable, we suggest the following modeling rules:

1. Bind the abstraction-level dependent formals using the configuration mechanism.
2. Organize the abstraction-domain related information into configuration.
3. Avoid the use of GENERIC MAP in component instantiation. Use defaults in ENTITY definition or in COMPONENT statements if possible, otherwise consider using configuration. For example, in case of a netlist which contains two inverters that only differ in their delays, use the configuration mechanism.
4. Do not mix the behavioral description (that will involves the information related to abstraction) and the structural description in one architecture. In case that some calculation related to the behavior of a netlist has to be performed, for example, parameter assertion for timing checking or power consumption calculation in analog simulation, encapsulate such "passive" statements into entities (interfaces), instead of architectures.
5. Avoid the use of propositional binding of interface list, or specify the significant ports first, and then insignificant ports.
6. Map insignificant ports using the configuration mechanism.
7. Organize signal and component declarations using packages. Automatically switch packages during analysis and elaboration.

In summary, the main consideration is to use configuration (not architecture) for that information related to the abstraction domain. For example, in-significant port map, generics map, type conversion, and domain conversion (connection type conversion, nature conversion). This leads to pure structural descriptions. Combined with entity overloading, it enables us to achieve netlist reuse.

7. Conclusions

While entity overloading offers a lot of convenience and flexibility, its impact to language design is minimal. Entity overloading can be resolved in the same (even simpler) way as subprogram overloading. Its adoption into VHDL will enhance VHDL's ability in supporting mixed abstractions in general, switch-level and analog modeling in particular. This extension is up-compatible to VHDL'93.

Acknowledgment: Some ideas presented in this paper originate from the AHDL project sponsored by United States Air Force Rome Laboratory, under contract F30602-93-C-0209. Several people contributed to the development of the concept of entity overloading into its current form. In particular, the author wishes to thank Jean-Michel Bergé, Ernst Christen, Steve Drager, Doug Dunlop, Dan FitzPatrick, Jim Hanna, Erich Marschner, Alain Vachoux, and Martin Vlach for their support and helpful discussions.

References

[1] J. Barby, A. Mantooth, C.-J. Shi and P. Subramaniam, "AHDL modeling for analog and mixed-signal top-down design," *Tutorial Handouts, 32rd IEEE/ACM Design Automation Conference*, June 16, 1995.

[2] K. Khordoc, M. Biotteau and E. Cerny, "Swich-level models in multi-level VHDL simulations", pp. 43-62 in *VHDL for simulation, synthesis and formal proofs of hardware*, Jean Mermet (ed.), Kluwer Academic Publishers, 1992.

[3] Oz Levia, S. Maginot and J. Rouillard, "Lessons in language design: cost/benefit analysis of VHDL

features", pp. 447-453 in *Proc. IEEE/ACM 31st Design Automation Conference*, June 1994.

[4] *IEEE Standard VHDL Language Reference Manual*, ANSI/IEEE Std 1076-1993, IEEE, NY, 1994.

[5] D. L. Perry, *VHDL*, 2rd, McGraw-Hill, 1994.

[6] R. A. Rutenbar, "Analog design automation: where are we? where are we going", pp. 13.1.1-13.1.8 in *Proc. IEEE Custom Integrated Circuits Conference*, 1993.

[7] David L. Rhodes, "Analog modeling using MHDL", *Modeling in Analog Design*, Jean-Michel Bergé, Oz Levia and Jacques Rouillard, eds. (Kluwer Academic Publishers, 1995): 47-92.

[8] C.-J. Shi, E. Christen, P. Liebmann, S. Krolikoski, and W. Zhou, "VHDL-A: Analog extension to VHDL", *IEEE International ASIC Conference & Exhibit*, Sept. 1994: 160-165.

[9] C.-J. Shi and A. Vachoux, "VHDL-A design objectives and rationale", *Modeling in Analog Design*, Jean-Michel Bergé, Oz Levia and Jacques Rouillard, eds. (Kluwer Academic Publishers, 1995): 1-30.

[10] Alex Stanalescu, "Switch-level modeling in VHDL", pp. 74-98 in *Applications of VHDL to Circuit Design*, R. E. Harr and A. G. Stanculescu (eds.), Kluwer Academic Publishers, 1991.

[11] B. R. Stanisic and M. W. Brown, "Behavior modeling of mixed analog-digital circuits", pp. 74-98 in *Applications of VHDL to Circuit Design*, R. E. Harr and A. G. Stanculescu (eds.), Kluwer Academic Publishers, 1991.

[12] Alain Vachoux and Jean-Michel Bergé, "VHDL-A: Analog and mixed-mode extensions to VHDL", pp. 475-480 in *Proc. EUROSIM'95*, Elsevier Science B.V. Publishers, Sept. 1995.