Requirements and Concepts for Hardware/Software Codevelopment: VHDL in 1998 and 200X

Lt Col Mike Mills    Capt Greg Peterson
WL/AASH
Wright Laboratory, Avionics Directorate
2241 Avionics Circle  Rm N3-F22
Wright-Patterson AFB, OH 45433-7334
(937)-255-6653 ext. 3591
millsm@msrc.wpafb.af.mil    gdp@el.wpafb.af.mil

Abstract

After ten years of standardization, VHDL continues to revolutionize electronics design by helping in the description of portable, accurate models for simulation, analysis, documentation, and synthesis. As dictated by IEEE rules, the VHDL standard requires reballoting by the end of 1998. This provides an opportunity to reconsider the strengths and weaknesses of VHDL and to determine the future of the language and its role in the development of electronic systems. One area identified as critical to current and future development of electronic systems, hardware/software codevelopment, pertains to the concurrent development of hardware and software system components throughout the design effort. This paper investigates, from the perspective of hardware/software codevelopment, the requirements for VHDL in 1998 (and beyond) to help identify what changes are needed. We include with the requirements the rationale for each.

Introduction

To comply with IEEE rules for its standards, restandardization of the VHDL standard must occur at least every five years. This provides an excellent opportunity to improve, refine, or extend the VHDL language based on the experiences of users and tool implementers from the previous five years. The VHDL Analysis and Standardization Group (VASG) is responsible for maintaining the standard; therefore, the VASG leads efforts currently under way which target revising the VHDL standard, IEEE 1076-1993, by the end of 1998. The schedule for restandardization activities must be aggressive to meet the 1998 deadline; hence, the VASG plans on a conservative approach for defining 1076-1998. Concurrently, the VASG plans on beginning work for more aggressive language revision activities targeting some time further in the future, but no later than 2003. Thus we have a conservative language revision effort for 1998, with more aggressive language change proposals deferred to 200X.

In the ten years since the initial standardization of VHDL, electronic design efforts have dramatically increased in the size, speed, and complexity of the systems being designed. Today, VHDL models must be able to describe with high-fidelity the behavior of complex, embedded systems often comprised of several ASICS, numerous commercial off the shelf (COTS) parts such as processors and memory, operating systems with their associated schedulers, and application software written in languages such as C, C++, and Ada. Perhaps the largest departure from past VHDL usage, and the most demanding area now and in the future, is the extent of the need for support of electronic hardware and its interaction with software components.

The representational needs of system designers span more than simply the capability of modeling software and hardware functionality. To help optimize architectural design, a single high-level VHDL specification needs to be capable of supporting tradeoffs
between hardware and software implementations. For example, a computer manufacturer may wish to implement a low-end, microcode implementation of a peripheral or processor assembly which is compatible at the register or instruction set level with high-end implementations fully manifest as custom hardware. Similarly, legacy electronic systems may include hardware components implementing functionality best migrated to newer, high-performance microprocessors.

The expressibility and abstraction capabilities of VHDL provide tremendous power for describing systems throughout the design process. The general problem of multidisciplinary system design is very difficult, with the need for capturing and back-annotating the results of many analyses including area, weight, power, cost, manufacturability, maintainability, mechanical effects, and analog effects, to name just a few. Numerous standardization and research activities address these issues and their relation to VHDL. Software, as an intangible, mathematical product, is a very complex domain for design and analysis. Nonetheless, the consideration of hardware and software behavior together is a critical capability for the design and documentation of systems. Therefore, we explore the current state of the art in designing hardware and software. We then discuss what needs or requirements there are for VHDL to support co-design and then investigate some of the areas we can address in the context of the short and long term VHDL revision efforts. Finally, we consider possible solutions and draw conclusions about the best approaches for supporting co-design with VHDL.

**Current Codedesign with VHDL**

The current standard practice (not co-design-oriented) for embedded systems design includes concurrent hardware and software development processes. The hardware and software design paths often begin from an initial specification of the hardware based on preliminary software requirements. The hardware design effort then focuses on completing the hardware components of the system. Concurrently, the software design effort develops the software code for the system consistent with the initial hardware specification. When the hardware and software design tasks are complete, an integration and test process begins. See Figure 1.

![Figure 1: Current System Design Process](image)

Often there is little interaction between the hardware and software design efforts because of a lack of a unified representation, simulation, and synthesis framework [1]. Because the integration and test phase of the design process is typically the first time the hardware and software are joined, a variety of problems are often encountered. First, the lack of communications between the hardware and software design teams results in inconsistencies in the interpretation of design specifications, thus the hardware and software do not share the same view. This results in potentially incompatible development efforts. Secondly, changes in the hardware design often are not communicated with the software design team, so the software is developed for the wrong hardware configuration. Third, during integration and test, performance bottlenecks and hardware bugs are discovered. Due to the high cost of redesigning hardware, these problems are typically rectified by making modifications to
the system software, resulting in code that is late and over budget. At this point, additional software engineers may be added to the effort, which can exacerbate the schedule and budget problems [2]. The software development effort often receives the blame for the overall project difficulties, when in reality the problems come from communications and coordination shortcomings during the specification and development.

For software development and maintenance on processors or processor families with long life cycles, this can be a particularly frustrating problem. Initial implementation flaws or outdated features may result in requirements which future generations must maintain for compatibility, often at substantial cost in performance, time, and money.

To address this problem within the context of electronic systems design, the simultaneous consideration of hardware and software throughout the design process is needed. This is referred to as the co-design of hardware and software. Hardware/software co-design tasks include the specification of the system (sometimes referred to as code specification), the concurrent software code generation and hardware development, and the verification of the integrated system (typically via co-simulation of hardware and software models, executing software on emulated hardware, or physical prototyping). Note that the verification task needs to be performed throughout the system design effort, with different levels of detail being appropriate for each stage in the system development. Similarly, validation represents an important system design task. Design verification indicates the degree of conformance to the specification; whereas validation aims to ensure that the specification conforms to the needs and requirements of the customer.

To begin the system design process, the customer and/or designers create a system specification. This initial specification should be detailed enough to concretely describe the required interface behavior, functionality, and relevant non-functional constraints without unnecessarily constraining the potential design space. In terms of hardware and software, this specification, or co-specification, should allow flexibility by allowing particular tasks to be implemented in hardware or software based on which best meets the system requirements for performance, cost, etc. In current practice, however, the system specification often includes a decomposition of the system into functional blocks to be implemented in hardware or software.

Ongoing research and development efforts target VHDL language support for system specification. The VSPEC system, is a Larch Interface Language for VHDL which targets formal requirements representation in the digital systems domain. Specifically, designers use VSPEC to represent requirements for automated synthesis and static analysis of DSP and avionics systems [3]. In order to better support abstract designs with more flexibility in representing time, instances, and interfaces, ICL developed VHDL+ for use in their design tools. The VHDL+ extensions for interface specifications enable different levels of design abstraction, including the use of incomplete specifications [4].

With system specification in hand, the system design activity progresses to concurrent and (hopefully) coordinated hardware and software design. Assuming the system specification includes a description of the entire system’s behavior, this functionality must then be decomposed into some number of subsystems or blocks of functionality, which may be implemented in either hardware or software. This is referred to as partitioning the design. Next, we develop a candidate design architecture including the decision about whether to implement each block in hardware or software. This design activity is mapping. Following the partitioning and mapping steps, some method must be employed to evaluate the candidate architecture to determine if it meets the system functional and non-functional requirements. This is often done by estimations and back-of-the-envelope calculations, but using the performance modeling techniques developed for VHDL, we can evaluate the architecture with relative ease and fidelity [5,6].

Performance modeling with VHDL provides a mechanism for quickly evaluating different candidate system architectures by using a package of token-based VHDL
constructs for the hardware elements, software modules, interconnection networks, and the data flowing through the system. Abstract models of processors, application specific integrated circuits, and interconnection networks focus on the amount of time spent on each token of data they handle before producing appropriate output tokens. As the hardware design progresses, more accurate models can be created to take into account design decisions and to enable modeling with higher fidelity. Similarly, the software models may include different levels of detail ranging from an estimated instruction count, an estimate of the instruction mixes, to the final operational software code for each software module’s processing. As the design progresses and more details are determined, the detail of the performance model can correspondingly increase. Capabilities for mixing models of different abstraction levels are needed in order to allow designers to flesh out areas of the system design at different rates. This arises when designers wish to study in more detail subsystems which have higher risk associated with their design or when portions of a design are reused [7,8,9].

With the system specification and architectural design tasks complete, the hardware and software components are developed using standard design practices including the use of synthesis and, in certain domains, autocoding. Numerous papers already discuss the state of the art methods for hardware and software design activities in great detail; therefore, we encourage the reader to peruse the proceedings of the VHDL International Users’ Forums, the Design Automation Conferences, the Software Engineering conferences, IEEE Workshops on Codesign, etc. With regard to codesign, these design practices often provide some points where the hardware and software components can be exercised together for verification and validation. Nevertheless, the design activities are not as tightly integrated as they could be, and the hardware and software components are typically represented using different languages and computational models. Consequently, system design processes often must integrate and support different methodologies, languages, and tools for the hardware and software components.

**Emerging Codesign with VHDL**

Based on the number of tools and designers using co-simulation, one must conclude it is currently the most mature aspect of hardware/software codesign with VHDL. Typical co-simulation consists of integrating register transfer level or gate level models of hardware components with an external instruction set architecture model of the included processors and their associated software. Performance modeling proves quite useful at higher levels of abstraction. Detailed co-simulation helps provide confidence that the system will function correctly before the fabrication of the hardware. The models of the hardware and software components may be very detailed and large, hence these co-simulations may require extremely long execution times. Using timely co-simulations can significantly reduce the impact of design errors by catching them early enough to avoid costly redesign efforts.

![Figure 2: Current VHDL Co-simulation](image)

Because most ISA models have been implemented in a software programming language such as C, the VHDL simulator needs to interface with the ISA simulator. Most often, designers use the VHDL foreign attribute as the mechanism for interfacing to
the external simulator. As noted in the VHDL language reference manual, the foreign attribute is a potentially non-portable language construct, thus its use can result in detailed system models tied to a particular VHDL simulator and ISA simulator [10]. See Figure 2 for how designers typically co-simulate hardware and software with VHDL.

A related issue to co-simulation as discussed above is the co-simulation of VHDL and Verilog models. The Open Modeling Forum addresses this issue and has developed the Open Modeling Interface (OMI) to support co-simulation with different hardware description languages and to protect intellectual property [13].

Throughout the design process, verification of the system continues. Using specification tools or animation capabilities, analyses can be completed at the most abstract levels. Performance modeling provides verification during the architectural design. As the detailed design continues, simulation at behavioral, structural (RTL), and gate levels, with the associated software models, provides confidence in the system design. See Figure 3 for more details. In the next section, we explore the outstanding language needs of designers to support a codesign-oriented design process.

**Needs for Codesign with VHDL**

After discussing the current practice for hardware/software codesign with VHDL, we now focus on the specific language-related needs of designers using VHDL. As mentioned above, the available time for revising VHDL by 1998 dictates that only conservative changes to the standard are likely to be implemented by this time. More aggressive changes to VHDL will be deferred to the 200X revision of the language.

Designers need to be able to specify systems containing hardware and software components in a manner independent of the particular implementation and mapping chosen. This specification method needs to support incomplete specifications in order to support the incremental development of systems from very abstract levels to detailed gate level models and to provide the potential for adding capabilities further along in the life cycle as mission needs dictate and technology improvements allow. The specification needs to be able to collect functional and non-functional requirements for the system. Functional requirements include the algorithmic description of how the system behaves along with the timing. Non-functional requirements include

---

**Figure 3: VHDL Codesign Methodology**

How to best model memory elements remains an important issue impacting simulation efficiency for performance models to gate-level models [11]. Similarly, research efforts address the best means of modeling interfaces in a portable and efficient manner. The Standard Virtual Interface developed during the RASSP program attempts to support flexible design upgrades and reuse by defining VHDL modeling techniques for encapsulating standard interface functionality [12]. Similar models of standard interfaces are under development for performance models [5].
considerations such as power, size, weight, manufacturability, reliability, maintainability, and cost. Some programs advocate the use of simulatable or executable specifications which include the behavior (or function) of the system, a set of constraints (or non-functional requirements), and a test bench for verifying that the system meets functional requirements [14,15].

In addition to the functional aspects of a subsystem or component, we need to capture non-functional requirements including constraints on size, weight, power dissipation, and other physical characteristics. In addition, constraints on fabrication processes, materials, and machines impact the manufacturability of system components and the capability to redesign the components later in the system's life cycle. Although these constraints impact activities late in the design cycle, early consideration of these constraints can minimize the adverse effects of manufacturing problems. As discussed above, the specification needs to be able to describe the system without overly constraining the designer to implement particular tasks in hardware or software.

Co-specification spans a wide area of problems, and numerous researchers continue to work in this area. Two of the more mature and noteworthy research efforts, VSPEC and VHDL+, have shown promising results and help to address key aspects of this area [3,4]. Nevertheless, because of the relative immaturity of the research and the scope of language changes to support co-specification, this area is best categorized a long-term problem involving basic research. Designers do indeed need the support of co-specification capabilities within VHDL, so language revision efforts for 200X already are considering this requirement.

Providing design documentation in a standard language, one of the primary goals driving the initial development of VHDL, remains an important need for designers, particularly for complex systems with long life cycles. Therefore, revisions should not include unnecessary data (such as tool-specific settings or controls), nor should they render design information unintelligible or unnecessarily difficult to navigate.

The increasing use of VHDL can perhaps be attributed to the development of hardware synthesis capabilities. Future revisions of VHDL need to support current design processes for hardware, including synthesis. To support abstract modeling and codesign, VHDL needs to continue support for more than an RT level subset; the language needs to support advances in synthesis technology, including behavioral synthesis.

Object-oriented software engineering methods promise improved productivity and maintainability due to encapsulation, abstraction, and reuse. As object-oriented programming languages propagate, hardware/software codesign methodologies, languages, and tools will need to support or at least interoperate with object-oriented software engineering practices. The semantic and syntactic differences between object-oriented software programming languages and hardware description languages can create difficulties in interfacing hardware models with software application code. Preventing potential problems caused by the difference in representation between object-oriented software languages and hardware description languages may be the most convincing argument for creating object-oriented extensions to VHDL. Many object-oriented features already exist in VHDL; debate currently focuses on which capabilities are needed and possible implementations [16]. The required functionality, not to mention the language crafting details, remain to be completed, although the Object-Oriented VHDL Study Group and the VASG are considering the possibility of object-oriented changes to VHDL for the 200X language revision.

Designers use VHDL simulations to provide some level of confidence that an implementation, at whatever modeling level of detail, is correct. In a formal sense, each simulation simply proves that the model acts correctly for a certain set of input vectors. As the number of vectors simulated increases, the confidence in the correctness of the design correspondingly increases. Note that correctness is not proven until every possible input vector and state is exercised, which is impractical in most cases. Nevertheless, the designers wish to simulate models with the highest fidelity and the largest number of input vectors possible. Therefore, speeding up
the simulation speed of VHDL models is a priority now and for the foreseeable future.

Designers need the capability to perform formal analyses of models because simulations do not provide sufficient confidence for critical systems. In particular, a standard definition of the formal semantics of VHDL is needed. In addition, potential changes to the language need to include some consideration for the possible impact on formal analysis tools. One area drawing considerable interest, the development of an application programming interface (API) or a programming language interface (PLI) to VHDL, may result in external applications having the ability to change the state of VHDL simulations in a manner unpredictable by formal analyses. In addition, some current aspects of VHDL complicate formal analysis. For example, the access and values of shared variables are currently not well defined, although this area is being clarified by adding monitors to VHDL shared variables. The foreign attribute does not sufficiently define the interface to other implementations for formal analysis tools. Other areas identified in the language reference manual as potentially non-portable and lacking precise meaning within VHDL include the use of timing resolutions finer than femtoseconds, using more than 32 bits to represent time for longer simulations, and communications via files or TEXTIO [10].

While perhaps being obvious, an important need for designers is that future revisions to VHDL do not invalidate their current methodologies, tools, and models. Therefore, backward compatibility with past versions of the VHDL standard remains a priority, with little possibility of the removal of language features in 1998. The VASG intends to flag which language features may be removed by 200X within the 1998 language reference manual in order to provide a period to wean users away from those features. Nevertheless, language revision targeted towards removing or speeding up particular language features of VHDL does need consideration for 200X.

As discussed in the last section, the current state of practice for VHDL-based hardware/software codesign focuses mainly on co-verification via co-simulation. The chief needs of designers in this area are to provide the fastest possible number of simulated cycles and to help in standardizing the way models are constructed and tools interfaced.

To optimize simulation speed for co-simulation, designers need simulation efficiency oriented standard models or modeling guidelines for interfaces, memories, and processors. The IEEE DASC Parallel Simulation Study Group collected a taxonomy and standard practice documents to address these issues for parallel and distributed simulation kernels [17,18]. There are proposals to define a cycle-based VHDL semantic to support co-simulation efficiency, although such a definition most likely will not be completed for 1998. The RASSP program developed a modeling taxonomy to standardize types of models [19] and a performance modeling interoperability guide [20], although neither directly addresses performance optimizations.

In general, we need to be able to interface a VHDL model and its associated simulator with a variety of other applications such as other VHDL models and simulators, Verilog models and simulators, and software models with instruction set architecture simulators. The VHDL foreign attribute provides the functionality to interface VHDL models to external applications, but can cause problems with portability [10]. Currently, the VHDL language reference manual does not define the allowed types or modes for foreign attribute calls. When one uses a foreign attribute, the syntactic and semantic benefits of strong typing are lost. The foreign attribute definition needs to be refined to reduce portability problems and to help in the creation of standard practices for modeling hardware and software components and for interfacing to VHDL simulators. A proposal for how to refine the foreign attribute definition follows in the next section.

Intellectual property protection and language interoperability needs are being addressed by the Open Modeling Foundation (OMF) and Virtual Socket efforts. The OMF effort has produced a preliminary specification for its Open Modeling Interface [13], so language revisions need to cooperate with the OMF efforts.

The issue of interfacing a VHDL model and simulator with other simulators, models, or applications introduces many related concerns.
and details. For example, how one synchronizes different representations of time requires potentially addressing different time resolutions, physical units, and representations. How much visibility and control of a VHDL model to provide an external model or application is a vexing problem with wide-ranging implications. In the event of multiple timed simulators or models interacting, control of time becomes a critical issue, particularly given the potential for such distributed simulations to deadlock without proper control mechanisms [21]. The VHDL design community perceives many needs for codesign, but the language revision efforts need to proceed very carefully and intentionally with regard to these issues.

Possible Solutions

One goal of this paper is to investigate minor modifications to VHDL that would support hardware/software codesign. Interfacing VHDL models and simulators with external applications stands out as a focused problem area which needs refining; therefore, we propose to refine the standard to help make the foreign attribute more portable.

As currently defined, the VHDL foreign attribute designates architectures or subprograms to be implemented in a different simulation language or programming language. The parameters can be of types that are allowed within VHDL but can be elaborated differently from what VHDL specifies. Statements within foreign models can also be elaborated differently depending on implementation. The implementation dependence of VHDL features associated with the foreign attribute hinders portability of a VHDL model. Yet, the implementation needs to conform to the rules of the language in which it is written which may conflict or extend what is allowed in VHDL.

So how do we make the foreign attribute more portable? Models associated with the foreign attributes have no semantic guidance in the VHDL standard. If standard packages were added to instantiate the foreign attribute to specific languages such as Verilog, Ada 95, C, C++, or another VHDL model or simulator then the current implementation dependencies of the foreign attribute could have more explicit semantics and, therefore, become more portable. The suggested standard packages need not be so restrictive as to hinder its application but specific enough to guide implementations along the same path to increase portability.

Given the similarities between VHDL and Ada, consideration of how Ada was changed to support interfacing to other languages provides valuable insight into some possible solutions to VHDL interfacing issues [22]. Ada 95 expanded its initial capability to interface to programs written in other languages by substituting Ada 83’s pragma INTERFACE with pragmas IMPORT, EXPORT, CONVENTION, and LINKER_OPTION in addition to providing separate interface packages for C, COBOL, and FORTRAN [23]. (Ada pragmas are compiler directives from within the Ada language.) The old pragma INTERFACE was limited to calling programs in other languages but could not be called by programs in other languages [24]. This limitation hindered interactions with programs in other languages. Since the foreign attribute only associates VHDL constructs to an implementation, such restrictions do not appear in the current standard, thus providing great flexibility to model developers. However, with little guidance, non-portable implementations result.

Attributes are associated with a variety of VHDL constructs. VHDL signals have a type part which could be associated to an external model using a foreign attribute. Since the parameters of foreign models can be signals, an item of interest is how the signal should be treated by the called model. The signal from a VHDL model may not have any meaning in a procedural language such as C or Ada. However, signal amplitude values at an instant of time could be interpreted as constants or possibly variables that get updated. Also, VHDL signal parameters have a type part which could be represented in an external model. Signals that can be represented in an external model may have to be transformed to a different time reference, for example, interfacing a VHDL model with a Verilog model.

For VHDL 1998, we recommend that some standard low level packages be constructed to
define at least some of the semantics of foreign for certain languages that are likely to interface with VHDL. For VHDL 200X, a more comprehensive way of interfacing to external modules is quite appropriate.

The general control and synchronization of VHDL simulators and other applications must be addressed when considering changes to the foreign attribute and the external interfacing capabilities of VHDL. These issues become particularly important if we allow external applications to have access to the internals of a VHDL model, including the capability to write to signals or otherwise change the internal state of the VHDL simulation. This turns into the general problem of maintaining causality as experienced by the parallel and distributed simulation communities and requires some form of a synchronization protocol [21], which would significantly impact the complexity of the language and tools. Support for signal parameters to foreign architectures or subprograms, particularly signals of mode out or inout, implies the need for an application program interface for the external implementation. Proposed efforts for creating a standard API or PLI will need to consider this issue as well. Given the scope of this work, such changes are most likely to be completed for the 200X language revision.

Given the importance of supporting co-simulation, cycle simulation promises to easily increase the number of simulated cycles per unit of time. To do so, we need to precisely define the cycle simulation semantics for VHDL. This represents a significant departure from the current semantics of VHDL. The behavior of VHDL models with more detailed timing in the context of a cycle simulation impacts the potential of this approach. The problem of synchronizing models with different timing granularities arises in a similar way with VHDL-AMS or with the mixed abstraction modeling we see with the hybrid performance modeling [5].

The VHDL codesign community needs a recommended practice document to address the best methods of developing models, the most effective methodologies for applying codesign techniques with VHDL, and how to interface VHDL models and simulators with external applications of simulations. Standard taxonomies, model definitions, and formats will also help.

An issue indirectly impacting the VHDL support of hardware/software codesign is the organization of the standard and its relationship to derived and related standards. When the base VHDL standard changes, the potential impact on several related standards, the status of these related standards, and how conflicting standards issues will be resolved all come into question. To reduce the difficulties in maintaining and synchronizing multiple standards, some users propose the inclusion of related and/or derived standards within the base VHDL standard. Forcing compliant tools to implement the base language and all the included standards may not be in the best interests of users or vendors. One solution to this issue is to have the organization of the VHDL standard follow example of Ada and its included annexes. In Ada, a number of annexes are appended to the standard, and compliant implementations may optionally implement some or all of these annexes. This approach would achieve the aim of producing tighter synchronization between the base language and related standards without forcing all tool vendors to implement all the VHDL-related standards.

**Conclusions**

With the upcoming VHDL restandardization by 1998, the electronic design automation community has the opportunity to consider what changes can be made in order to support hardware/software codesign with VHDL. By exploring the general language and tool needs of the codesign community, the priority of these needs, and the effort required to support these needs, we determined long and short term goals for codesign support. Because the foreign attribute as currently defined within the VHDL standard presents potential portability problems while offering functionality critical to hardware/software codesign efforts, we investigate requirements, issues, and possible language changes to improve the foreign attribute. Other codesign-related VHDL standards tasks, such as developing a recommended practices or standard packages, should be considered for
the VHDL language revision of 200X or forwarded to the IEEE DASC Codesign Study Group. See http://vhdl.org/vi/codesign for more details.

Acknowledgments

We would like to thank John Willis, the VHDL Issues Screening and Analysis Committee (ISAC) chair, for his insightful comments on the proposed changes.