

VHPI, A Programming Language Interface for VHDL

Doug Dunlop

Cadence Design Systems, Inc.
11912 Hunting Ridge Ct
Potomac, MD 20854
dunlop@cadence.com

1.0 Abstract

This paper describes VHPI, a C programming language interface for VHDL that is under development at Cadence. The intended use of VHPI is explained and the important characteristics of the interface for accessing VHDL design information and interacting with a VHDL tool are described. Finally, the need for a standard way of including user-written C code with VHDL models in a VHDL environment is discussed.

2.0 Introduction

An important and widely-used aspect of Verilog HDL is the Programming Language Interface (PLI, see [1]). The Verilog PLI allows applications written in C to access a Verilog design hierarchy and to interface to Verilog HDL tools such as logic simulators. Over the years the Verilog PLI has matured based on user experience and the most-recent generation of the Verilog PLI, the Verilog Procedural Interface (VPI), provides a powerful and elegant mechanism for accessing Verilog HDL design data and interacting with a Verilog HDL tool.

Recognizing the utility of a capability like this, VHDL tool vendors often provide mechanisms with their products that provide functionality that is roughly analogous to portions of that provided in the Verilog PLI. However, these solutions are all vendor specific and there is considerable variability across products in the capabilities that are

provided. In light of this, there recently has been growing interest in the development of an open, standard programming language interface for VHDL.

In this paper we describe VHPI (the VHDL Procedural Interface), an interface under development at Cadence that allows C programs access to VHDL design information and the ability to interact with a VHDL simulator. In the following section we first motivate the idea of VHDL PLI. We then explain the different kinds of design and simulation information that are available through VHPI and discuss the modeling formalism used in specifying this information. The corresponding programming language interface is also described and simple examples are provided to illustrate the basic concepts. This is followed by a section that explains the VHPI callback mechanism that allows dynamic interaction with a VHDL simulator. A description of related work is then presented and the paper concludes with a brief summary.

3.0 Why a VHDL Programming Language Interface?

It was not long after the first VHDL implementations were available that C programming interfaces were developed to allow user-written C code to be incorporated into a VHDL simulation. Many VHDL simulation vendors recognized the value added by supporting such interfaces and included them in their VHDL tools. Unfortunately, these interfaces

are often proprietary and tend to vary from vendor to vendor, making it difficult for the user to develop C code that is portable across VHDL implementations.

From a historical perspective, VHDL is based largely on the programming language of Ada. As a result, some of the mindset associated with the original version of Ada is apparent in VHDL. One aspect of this mindset is the perception that the language is primarily stand-alone, self-contained and with relatively little need for interacting with the "outside world". Its worth noting that in the recent revision of Ada (see [5]), one of the most significant shifts in emphasis is from this rather restricted point of view to a more open one in which standard, language-defined mechanisms allow Ada programs to work effectively with software written in other programming languages. It is not difficult to argue that VHDL would benefit from a similar shift in thinking, in which non-proprietary ways of incorporating C models in VHDL simulations are developed and standardized.

The VHDL 93 revision effort recognized the need to develop language mechanisms that would allow non-VHDL code to be included in a VHDL design. However, the 'Foreign attribute introduced in VHDL 93 is only a partial step in this direction and in practice does not contribute significantly to the ability to combine C and VHDL in a portable way.

In general, there are several important reasons for developing and standardizing a C programming language interface for VHDL. It can be very useful to be able to instantiate a model of a device written in C from a VHDL design or to be able to invoke a C function from VHDL. C is an extremely flexible language and it can be convenient to use C for complex or detailed functionality that would be awkward or inefficient to code in VHDL. It can also be useful to be able to easily take advantage of existing C code libraries in VHDL designs. There are applications that can be performed most easily and most efficiently by giving the designer access to the design hierarchy and the ability to read and set values of objects and interact with the basic functions performed by the simulator. Examples here include delay annotators and calculators, design rule checkers, customized debugging and display applications, design hierarchy browsers, and de-compilation or translation utilities. Finally, it is very desirable that the VHDL designer using C and VHDL in these ways not be tied to a particular VHDL tool vendor and that his or her work be portable in the same way VHDL is portable.

4.0 VHPI Overview

In this section an overview of VHPI is presented. The reader is cautioned that the interface described here represents work in progress at Cadence. There are a number of details in the interface that are still being investigated and aspects of the interface may change from what is described here. However, it is expected that the fundamental aspects in the design of the interface (e.g., that the access to VHDL design information is based on an information model, that the mechanism for simulation interaction is based on a callback mechanism, etc.) will not change.

At a high level, VHPI provides two general capabilities. The first of these is the ability for C code to extract information about the VHDL design. The other is the ability for C code to interact with the VHDL simulator. These capabilities are described in the two subsections that follow. This is followed by two subsections that discuss several miscellaneous aspects of VHPI and that enumerate the current set of VHPI routines.

4.1 Information Access

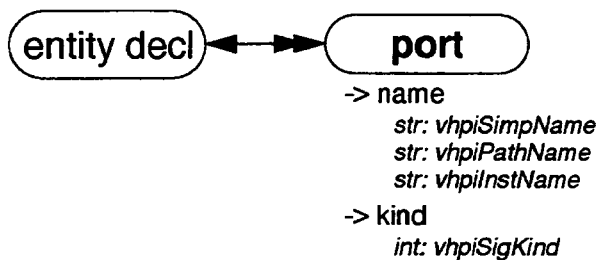
The access to VHDL design information in VHPI is based on the concept of an *object*. Objects represent items in the VHDL design such as signals and processes. Objects have a formally defined information content that consists of properties and relationships. A *property* represents a basic characteristic of an object. For example, an object representing a VHDL loop statement has a property that indicates the name of the label on the loop. A *relationship* documents an association from an object to another object or a set of objects. Continuing with the loop statement example, an object representing a loop statement has a relationship that indicates the iteration scheme associated with the loop as well as a relationship indicating the sequence of sequential statements that make up the loop body. The former relationship is called a *singular* relationship since the relationship is to a single object; the latter relationship is called a *multiple* relationship since the relationship is to a collection of objects.

There are two important aspects to this methodology for accessing VHDL design information in VHPI. One of these is the specification of the available information; the other is the corresponding programming interface. These aspects are described in the following two subsections. Both aspects are based on similar approaches in VPI (see [1]).

4.1.1 The VHPI Information Model

In the definition of VHPI the different kinds of objects and the properties and relationships that exist for those kinds of objects are defined with an *information model*. The information model serves as a formal specification of the information content for VHPI objects and provides a convenient framework for describing the corresponding programming interface. The VHPI information model is expressed in set of object diagrams which graphically show relationships and define the properties associated with particular kinds of objects.

The following fragment of an object diagram illustrates the information modeling methodology used in the definition



of VHPI. In this diagram, **entity decl** and **port** are kinds of objects. They are used to represent VHDL entity declarations and ports, respectively. There is a multiple relationship (indicated by the double arrow head) from an **entity decl** object to a collection of **port** objects; this relationship represents the ports of the entity declaration. There is also a singular relationship (indicated by the single arrow head) from a **port** object that represents the entity declaration on which the port exists. The diagram also shows that **port** objects have four properties — three name properties (*vhpISimpleName*, *vhpIPathName*, and *vhpIInstName*) and a kind property (*vhpISigKind*). The name properties are string-valued (corresponding to the VHDL ‘SimpleName’, ‘InstanceName’, and ‘PathName’ attributes on the port, respectively) and the kind property is integer-valued (and gives an encoding of the VHDL signal kind of the port).

Although the object diagrams used in defining the VHPI information model are intuitive and useful for most purposes, an equally-formal text-based representation of the information model is currently being explored to facilitate machine processing of the model.

The VHPI information model covers three major facets of VHDL design information. These are the unelaborated form of a VHDL design unit, an elaborated design hierarchy, and VHDL simulation-time information. These three aspects provide incremental insight into the model data; if the user is accessing a design hierarchy through VHPI, the uninstantiated (i.e., unelaborated) data is also available; if VHPI is being used in a simulation context, not only is simulation specific information available but also queries can be made about instantiated and uninstantiated data. As an example, in a simulation context, the user can obtain the current value, instance name, and subtype of a signal, which are respectively runtime, instantiated and uninstantiated data. The VHPI information model shows all the information available for all three aspects of VHDL design information. Depending on how the VHPI application was started, some information might not be available, and an appropriate error condition will be returned at this point.

4.1.2 Programming Access to VHPI Objects

The programming interface in VHPI that provides access to the information described in the previous section utilizes the concept of an object handle. A *handle* is the means in VHPI of referencing an object. The C type *vhpiHandle* is defined in the interface for handles. This type is essentially a private type in the sense that no assumptions can be made by the user about the representation of handles. Further, all access to objects is made through handles, e.g., objects are never accessed directly using pointers to structures. The implementation approach for objects and the means by which handles reference objects are intentionally not specified in VHPI. The only meaningful C operation on handles is to compare a handle against NULL (e.g., using `==`). The routine *vhpi_compare_objects* is provided that compares two handles to determine if they refer to the same object.

The VHPI interface includes a routine *vhpi_handle_by_name()* that returns a handle to an object from the name of the object. VHPI also provides generic functions for basic tasks such as traversing relationships and determining property values. Singular relationships are traversed with the routine *vhpi_handle()*. In the following code fragment, a handle to the entity declaration which contains a particular port is

```

vhpiHandle port, entity;
port = vhpi_handle_by_name(":e:p1:",
                          NULL);
  
```

```
entity = vhpi_handle(vhpiEntityDecl,
                    port);
```

obtained from a handle to that port. This example utilizes the information model described in the previous section concerning entity declarations and ports. The call to `vhpi_handle_by_name` retrieves a handle to signal `e(a1):s1` and assigns it to the `vhpiHandle` variable `signal`. The `NULL` second argument in this call directs the routine to search for the name from the top level of the design. The call to `vhpi_handle` traverses the singular relationship from the port to the entity declaration. The first argument in this call corresponds to the name of the relationship; in this case, since the relationship was unlabeled in the object diagram, the name of the relationship is derived from the kind of object targeted by the relationship. The net effect of the example is to assign to `entity` a handle for the object representing entity declaration `e`.

Properties of objects may be obtained using routines in the `vhpi_get` family. The routine `vhpi_get()` returns values of integer and boolean properties. The routine `vhpi_get_str()` reads string properties. As an illustration, to retrieve a pointer to the hierarchical instantiated name of the object referenced by handle `entity` from the previous example, the following call could be made:

```
char *instname =
    vhpi_get_str(vhpiInstName, entity);
```

Here, the character pointer `instname` will point to the string `":e(a):"` assuming the architecture `a` is associated with this particular instance of entity declaration `e`.

In VHPI, multiple relationships are traversed with an iteration mechanism. The routine `vhpi_iterate()` creates an object of type `vhpi_iterator`, which may then be passed to the routine `vhpi_scan()` to traverse the individual objects. In the following example, the instantiated name of each port of entity `e` is displayed:

```
vhpiHandle itr;
itr = vhpi_iterate(vhpiPort, entity);
while (port = vhpi_scan(itr))
    vhpi_printf("\t%s\n",
                vhpi_get_str(vhpiInstName,
                             port));
```

If `p1` is the simple name of a port of the entity declaration `e`, the above code will include the output `":e(a):p1:"`. The

routine `vhpi_printf` is a VHPI routine analogous to the C `printf` library function.

The act of obtaining a handle to an object by using routines such as `vhpi_handle_by_name`, `vhpi_handle`, `vhpi_scan`, etc., may involve the allocation of memory. VHPI is being defined to include a routine to indicate to the implementation that a handle to an object is no longer needed so that this storage may be reclaimed.

Most properties in the VHPI information model have boolean, integer, or string values. These properties may be read using the `vhpi_get` and `vhpi_get_str` routines discussed previously. Certain properties however are of more complex types and require specialized support. For example, the value of a data object is read with a specific routine called `vhpi_get_value`. This routine deals with the variety of types for data objects that is possible in VHDL, including general arrays and records. The routine allows the user to request a particular format to be used in representing the value (e.g., a printable string representation); otherwise, the returned value is represented in the native format for that type of data. A companion routine `vhpi_set_value` can be used to set the value of a data object.

4.2 Interaction During Execution

Dynamic interaction between a VHPI application and the VHDL simulator is achieved with a registered *callback* mechanism. VHPI callbacks allow the user to request that the simulator call a user routine when a specific activity occurs. For example, the user might request that the user routine `my_monitor()` be called when a particular net changes value, or that `my_cleanup()` be called when the VHDL tool has completed. The VHPI callback mechanism is modeled after the callback scheme in VPI (see [1]).

In VHPI callbacks provide the user with the means to dynamically interact with the simulator, detecting the occurrence of value changes, advancement of time, end of simulation, etc. This capability allows applications that provide integration with other simulation systems, specialized timing checks, complex debugging features, etc.

When a callback is registered with the simulator, a *reason* is given that reflects the circumstances under which the callback is desired. The VHPI reasons for simulation-

related callbacks can be separated into three main categories:

- *Simulation event.* These callbacks occur when a particular event occurs during simulation. Examples include callbacks that occur after a value change on a particular signal or just prior to the execution of a specific behavioral concurrent statement.
- *Simulation time.* These callbacks occur at points related to simulation time or the VHDL simulation cycle. Examples include callbacks that occur after a certain amount of simulation time has elapsed or just before the postponed processes are executed in the current simulation cycle.
- *Simulator action/feature.* These callbacks occur at various stages in VHDL simulation. Examples include callbacks that occur at the end of elaboration or when simulation is restarted from a saved state.

VHPI simulation-related callbacks are registered by the user with the function `vhpi_register_cb()`. This routine is passed the specific reason for the callback, the routine to be called, and what system and user data should be passed to the callback routine when the callback occurs. Registered callbacks can be removed with the routine `vhpi_remove_cb()`. Certain kinds of callbacks are persistent in the sense that they are not automatically removed after they occur.

A related facility in VHPI is the ability to register callbacks for foreign functions. A *foreign function* is a C routine that is used as the implementation of a VHDL process, subprogram, or architecture. In VHDL source text, the use of a foreign function as the implementation for one of these VHDL constructs is indicated by an attribute specification for the predefined 'Foreign attribute. When a callback is registered for a foreign function, information about the function such as its name and return type is passed along with addresses of a pair of callback routines. The first of these routines is called once during initialization (prior to any invocation of the function); the second is called each time the foreign function is invoked.

Somewhat coupled with this mechanism of callbacks for foreign functions is the ability in VHPI to establish an array of function pointers for routines that will automatically be called when the simulator starts up. One of the primary uses of this array, called

`vhdl_startup_routines`, is in registering callbacks for foreign functions.

VHPI callbacks for foreign functions are analogous to system task and function callbacks in VPI (see [1]).

4.3 Miscellaneous Features

On an error condition, VHPI routines of type integer return zero, and routines of type `vhpiHandle` or character pointer return NULL. Note that these values may also be returned when there is no error condition. To determine if an error occurred, the routine `vhpi_chk_error()` can be used. This routine indicates whether an error occurred in the previously called VHPI routine and if so provides detailed information about the error. There is also a callback reason established for run-time errors.

VHPI provides a set of routines to support text-based input/output. The notion of a *multi-channel descriptor* (MCD) is provided which can be used to route output to a set of files or logical devices. Routines analogous to the C `printf` library function are available for MCDs and for the general logging of run-time output.

4.4 VHPI Routine Summary

The following table gives an alphabetical list of the routines currently provided in VHPI. The descriptions given for the routines are extremely brief and are intended only to convey a high-level feel for the purpose of the routines.

VHPI Routine Name	Purpose
<code>vhpi_chk_error()</code>	Checks/returns error info
<code>vhpi_compare_objects()</code>	Compares handles
<code>vhpi_free_object()</code>	Deallocates an object
<code>vhpi_get()</code>	Reads the value of a property
<code>vhpi_get_cb_info()</code>	Obtains callback information
<code>vhpi_get_delays()</code>	Extracts delay information
<code>vhpi_get_str()</code>	Reads the value of a property
<code>vhpi_get_foreignf_info()</code>	Obtains foreign function info
<code>vhpi_get_time()</code>	Gets current simulation time
<code>vhpi_get_value()</code>	Reads the value of a data object

VHPI Routine Name	Purpose
<code>vhpi_get_vhdl_info()</code>	Returns tool invocation info
<code>vhpi_handle()</code>	Follows a singular relationship
<code>vhpi_handle_by_index()</code>	Returns element from an index
<code>vhpi_handle_by_name()</code>	Obtains object from its name
<code>vhpi_iterate()</code>	Follows a multiple relationship
<code>vhpi_mcd_close()</code>	Closes MCD channels
<code>vhpi_mcd_name()</code>	Gives name of MCD channel
<code>vhpi_mcd_open()</code>	Opens MCD channels
<code>vhpi_mcd_printf()</code>	Prints to MCD channels
<code>vhpi_printf()</code>	Performs C-like printing
<code>vhpi_put_delays()</code>	Sets delay information
<code>vhpi_put_value()</code>	Sets the value of a data object
<code>vhpi_register_cb()</code>	Registers callback
<code>vhpi_register_foreignf()</code>	Registers foreign. function
<code>vhpi_remove_cb()</code>	Cancels callback
<code>vhpi_scan()</code>	Gets next list/set element

5.0 Related Work

VHPI is based on the Verilog VPI (see [1]). A number of simulator-specific programming interfaces for VHDL have been developed that provide capabilities similar to that provided by portions of VHPI.

The Open Model Forum (OMF) has developed a set of open programming interfaces that define a simulator-independent and HDL-independent form of interaction between an EDA tool such as a simulator and a model (see [2]). This integration interface is intended to protect the intellectual property in simulation models and to support efficient simulation. The OMF interfaces use an approach similar to that described in section 4.1 as the means by which the simulator extracts information about the model interface. The OMF interfaces also utilize a callback mechanism that allows the simulator and model to interact during execution. The OMF callback mechanism is necessarily more general than that in VHPI since its focus is not restricted to VHDL simulation.

In recent years there has been considerable interest in interoperability between VHDL and Verilog, that is, in the development of guidelines and techniques that allow Verilog models to be used effectively in VHDL simulations and vice-versa. In a related way, it would be useful to have an HDL-neutral programming interface that could be supported in both VHDL and Verilog simulation environments and that would allow interoperability between a C application written to the interface and an HDL implementation supporting the interface. Work on the development of such an interface has begun at Cadence. Unlike the OMF work, this interface deliberately exposes internal model information and is intended to support a wide range of EDA applications.

6.0 Summary

There is a need for a standard programming interface that provides access to VHDL design information and allows interaction with a VHDL tool such as a simulator. The VHDL programming interface described in this paper is flexible and powerful and would be a reasonable starting point for the development of such a standard. VHPI utilizes a model-driven means of accessing design information and a comprehensive system of callbacks for simulator interaction. The interface leverages off techniques used in the latest version of the Verilog PLI.

VHDL is already complemented by a set of related standards such as VITAL (see [4]) and the 1164 standard (see [3]). These related standards enhance the interoperability of VHDL models and design techniques. This notion of a related standard would be a suitable standardization approach for a VHDL programming interface.

7.0 Acknowledgments

The author is grateful to Francoise Martinolle for her work in developing the VHPI specification. Several parts of this paper are adapted from portions of this document.

The author is also grateful to Kathy McKinley for her comments and suggestions on an earlier draft of this paper.

8.0 References

- [1] IEEE Std 1364, Verilog Hardware Description Language Reference Manual.

[2] Open Model Interface Draft Standard V1.0, see <http://www.cfi.org/OMF/>.

[3] IEEE Std 1074.4 VHDL Initiative Toward ASIC Libraries (VITAL).

[4] IEEE Std 1164-1993, IEEE Standard Multivalued Logic System for VHDL Model Interoperability (Std_logic_1164).

[5] Ada 95 Reference Manual, The Language, The Standard Libraries, ANSI/ISO/IEC-8652:1995.