VHPI Standard Specification

Deleted: 6

Draft 4.7

Deleted: 68
Deleted: *68*
Deleted: *69*
Deleted: *70*
Deleted: *72*
Deleted: 73
Deleted: *74*
Deleted: *74*
Deleted: *76*
Deleted: *77*
Deleted: *77*
Deleted: *78*
Deleted: 79
Deleted: *80*
Deleted: *80*
Deleted: *82*
Deleted: 83
Deleted: *83*
Deleted: *84*
Deleted: *84*
Deleted: *86*
Deleted: *87*
Deleted: 88

Deleted: *88*
Deleted: 89
Deleted: *89*
Deleted: 90
Deleted: *90*
Deleted: 91
Deleted: *91*
Deleted: *91*
Deleted: *92*
Deleted: *93*
Deleted: **94**
Deleted: 94
Deleted: 94
Deleted: 95
Deleted: *95*
Deleted: *96*
Deleted: *97*
Deleted: *97*
Deleted: *97*
Deleted: *98*
Deleted: *98*
Deleted: *98*
Deleted: *99*
Deleted: *99*
Deleted: *100*
Deleted: **100**
Deleted: 100
Deleted: *101*
Deleted: *107*
Deleted: 108
Deleted: *108*
Deleted: *109*
Deleted: **111**
Deleted: 111
Deleted: **111**
Deleted: **111**
Deleted: **112**
Deleted: **112**
Deleted: **112**
Deleted: **112**
Deleted: **112**
Deleted: 113
Deleted: *113*
Deleted: **113**
Deleted: **113**
Deleted: **115**

Deleted: 253
Deleted: 253
Deleted: 253
Deleted: 253
Deleted: 253
Deleted: 253
Deleted: 253
Deleted: 253
Deleted: 253
Deleted: 253
Deleted: 254
Deleted: 254
Deleted: 254
Deleted: 254
Deleted: 254
Deleted: 254
Deleted: 254
Deleted: 254
Deleted: 254
Deleted: 255
Deleted: 255
Deleted: 255
Deleted: 255
Deleted: 256
Deleted: 256
Deleted: 257
Deleted: 258
Deleted: 258

VHDL Procedural Interface

1. Overview

1.1 Scope and Purpose of the VHDL Procedural Interface

Design a standard procedural interface for VHDL. The outcome should be a specification that is implementor independant and which can be used on any VHDL compliant tool.
Supports the current standard version of VHDL and any past versions as needed.
The interface should define the semantics for a mixed language design and define the elaboration/instantiation and access methodology during runtime of foreign models.
The interface provides a mechanism to interact, control and communicate with a VHDL compliant tool.
The interface is an ANSI C procedural interface, which guarantees C source code portability across all tools compliant with the interface. Even though the interface defines symbolic constant values and C data structures which are a step towards binary portability, C object code binary portability across same harware/OS is not guaranteed. Vendors or application writers should document application binary interface issues which would affect their integration or would not be compatible with their PLI implementation.
The interface provides a VHPI header file which defines the required VHPI access. The compliance with the standard requires a vendor to preserve the VHPI standard file. This VHPI interface provides extensibility for a vendor VHPI implementation

1.1.1 VHDL Procedural interface requirements and guidelines

There are 10 requirements:

#1 functionality:

The procedural interface functionality can be divided in 2 parts: the **core** functions and the **utility** functions.
<u>core</u>: The interface should provide core functionality that enables the development of applications such as:

- design traversals, netlisters,
- connectivity extractors,
- co-simulation, backplane interfaces,
- behavioral models,
- debugging environments,
- simulation testbench and verification,
- VHDL code profilers and coverage tools,
- VHDL decompilers,
- delay calculators.

These various types of applications require different capabilities to be supported by the VHDL procedural interface; they can be classified in 4 categories:

1. access to static VHDL design data
2. access and modification of specific VHDL objects
3. simulation interaction and control
4. foreign model instantiation and intercommunication mechanism with VHDL design

Class 1 functionality should provide access to the elaborated model. Complete (with the exception of protected data see requirement # 2) behavioral and structural access is highly desirable for back end tools such as synthesis tools, delay calculators design verification tools. If the procedural interface provides complete access to the static design data it should be possible to decompile a design that was analyzed and regenerate an equivalent VHDL description. Delay calculation and back annotation through VHPI is considered as lower priority and is not addressed in this first version of the standard. Read access to generic values which specify delays are however possible with this version of the standard. There are several reasons behind pushing out providing the delay calculation and annotation capabilities:

1. No existing C interface has this capability,
2. SDF back annotation or proprietary delay calculation or back annotation tools have been used for VHDL designs.
3. A group under the DASC is working on a new standard for delay calculation and annotation (DPC) and will fullfill this need. http://www.vhdl.org/dpc

Class 3 functionality should provide the **ability to change values** of VHDL objects during elaboration or simulation.Valid objects that can be modified will be specified.
Class 3 functionality should provide **simulation interaction** such as the ability to schedule events and transactions, query the simulation state and time queue and interrupt the simulation engine at defined times or for various reasons.

Class 4 functionality should provide a mechanism to **elaborate foreign models** with VHDL models. This mechanism should specify how one can use a foreign model in a VHDL model and how the foreign model should access and propagate values to and from the VHDL model.

The interface should provide support for **event-driven** as well as for **alternative algorithms for simulation**.

Utilities: Core utility functions such as printing, displaying and comparison functions are necessary. The interface should provide an error handling mechanism, strong error detection and a set of known error codes that the user can reference. All of the PLI functions should give an indication of how and why they fail if they fail.

#2 **restricted access to protected data**:

VHDL does not defined a mechanism to specify protected source code. The access to protected models should be restricted to the information that can be found in a model vendor library and what is necessary for interfacing the library cells in the VHDL design.

#3 Memory management:

The procedural interface sould provide functionality to allow the application to manage the lifetime of the memory allocated by the PLI.

#4 Hide internal data representation:

The application cannot make assumptions about the underlaying data implementation. It must use the defined mechanisms for data manipulation.

#5 Portability:

The PLI should be ANSI C compatible.
The mechanism that the PLI interface provides for integrating PLI applications or models should be easily portable to major platforms, i.e. UNIX or NT.

#6 Allowing multiple concurrent PLI applications:

The PLI should support simultaneous use and parallel read-only access to the same data. Note:Parallel modification (write access) to the same data may be indeterministic and may be implementation dependant. The design solution may determine the outcome of this issue.

#7 VITAL:

The working group will evaluate if the PLI interface should require VITAL specific information.

#8 Saving/Restoring:

The PLI should support a mechanism to save and restore PLI application state; after a restore operation, the VHDL compliant tool should be able to continue.

#9 Resetting a simulation state:

For simulation tools, the PLI should provide a mechanism to reset to time 0 which is the time just after simulation initialization.

#10 evaluate co-simulation requirements:

TBD

Guidelines

There are 7 guidelines.

#1 Do not preclude mixed signal VHDL.

#2 Provide for smooth crossing of VHDL/Verilog domains

#3 performance

The interface should provide fast access to data. In particular, when the interface functions are used for communicating with a simulator, where speed is the most important thing.

#4 capacity

The procedural interface should be able to handle large designs and should manage memory internally.

#5 versioning

A mechanism should be provided to determine adequate version information for the PLI interface, the simulator and relevant models.

#6 Function, data type names should be intuitive and seem natural to somebody who knows VHDL.

#7 The working group will evaluate relevant existing interfaces with the possibility of leveraging prior work.

1.1.2 VHPI capability sets and conformance

The standard define clusters of VHPI capability, a vendor may claim conformance to several VHPI capabilities. The claim of conformance to a VHPI capability requires that a vendor provides a compliant implementation to all the methods, properties and functions referred by that capability.
All vhpi functions defined by the standard should, if not implemented return an unimplemented error message.
An integer property vhpiCapabilitiesP can be queried from the tool class to check the vendor supported VHPI capabilities. There is an enumeration constant defined for every capability.
A tool which supports a given phase shal support the vhpiStartOf phase name and vhpiEndOf phase name that phase.

VHPI defines ten capabilities:

. the hierarchy capability: access to elaborated regions, design units, object declarations, types and subtypes. vhpi_handle_by_name, vhpi_get_value of declared objects.
This capability enumeration constant is vhpiProvidesHierarchy.

. the VHDL static access capability: complete post elaboration static access including statements, expressions, access to values of declared items after elaboration (vhpi_get_value).
The capability enumeration constant is vhpiProvidesStaticAccess. It is a superset of the hierarchy set. It requires the hierarchy capability.

. the connectivity capability: access to VHDL drivers and contributors, port connections. It also requires the hierarchy capability.
The capability enumeration constant is vhpiProvidesConnectivity.

. the post analysis capability (uninstantiated VHDL access) allows traversal of statements, expressions, design units, etc…This capability set provides VHPI access after VHDL analysis and traversal of specific relationships from the instantiated model to the uninstantated model (section 5.8). Also provides vhpi_get_value of objects which are initialized to locally static expressions and vhpi_handle_by_name in the uninstantiated domain.
The capability enumeration constant is vhpiProvidesPostAnalysis.

. the basic foreign models capability: ability to create foreign architectures and subprograms (vhpi_register_foreignf, vhpi_get_foreignf_info), foreign architecture and subprogram information model access, specific foreign model callbacks, general callbacks, vhpi_get_value and vhpi_put_value.
The capability enumeration constant is vhpiProvidesForeignModel. It also requires the hierarchy capability.

. the advanced foreign models capablity: creation of foreign drivers and processes, scheduling of transactions.to foreign drivers.
The capability enumeration constant is vhpiProvidesAdvancedForeignModel. It requires the basic foreign model capability.

. the save/restart capability: save and restart of foreign models and applications, vhpi_put_data, vhpi_get_data, callbacks for save and restart, vhpiIdP, vhpiSaveRestartLocationP. The capability enumeration constant is vhpiProvidesSaveRestart.

. the reset capability: reset to time zero for foreign models and applications, callbacks for reset.
The capability enumeration constant is vhpiProvidesReset.

. the debug and runtime simulation capability: vhpi_control, vhpi_get_time, vhpi_get_next_time, vhpi_handle_by_name, access to the supported static information model, including at a minimum accessing objects, reading and modification of object values, scheduling transactions to drivers, registering object value change callbacks as well as time and action callbacks.
The capability enumeration constant is vhpiProvidesDebugRuntime.

. the dynamically elaborated capability: access to dynamically elaborated objects (VHDL subprograms, for loops). The compliance set enumeration constant is vhpiProvidesDynamicElab. It requires the debug and runtime simulation capability.

1.2 Interface Naming Conventions

The VHDL Procedural Interface is denoted by the short name of **VHPI** which stands for the **VH**DL **P**rocedural **I**nterface.

1. All standard functions, classes, types, relationship tag names, enumeration constants defined by the interface starts by the prefix **"vhpi".**

2. The VHPI standard function names are lower case characters and have an underscore between each word; all other names (classes, relationship tags, enumeration constant identifiers will have no underscore and each word after the VHPI prefix will start by a upper case letter followed by lower case letters for the remaining of the word.
3. All defined VHPI types will end in a capital **T**.
4. One to many relationship tag names have an s (lower case) at the end.
5. VHPI uses some short name conventions: for example decl for declaration, stmt for statement, conc for concurrent, seq for sequential, subp for subprogram…
6. The VHPI class kind names end by the letter **K**.
7. The VHPI property names end by the letter **P**.

1.3 Procedural Interface Overview

The VHDL procedural interface is based on:

- the definition of a VHDL ***information model*** that represents the static and dynamic VHDL data that is accessible by the procedural interface,
- a ***small set of functions*** that operate on this model to access data, query about some particular piece of information, modify data, interact with the tool that supports the model or provide utilities such as for printing or checking errors for example…

The VHDL PLI information model is based on an *object-oriented* representation of the VHDL *post-elaborated* and *simulation data*. It partitions the VHDL data into *VHPI classes* that are connected by *relationships*. We use the same terminology that is used in object-oriented software design. A VHPI class is a set of VHPI data types which share the same functional *methods* and *properties*. An instance of a class is called an *object*. A class can have zero or more *member* classes. Member classes are said to be derived from the class they belong to (the parent class). Member classes *inherit* the methods and properties of their parent classes. The relations between a class and the rest of the information model are defined by the information model and methods are available to traverse the external class links. There are basically three classes of relationships: the *one-to-one* relationship, the *one-to-many* relationship and the *many-to-many* relationship.
The first relationship describes the fact that given an object of a certain class, and given a destination class type, at most one object of the destination class can be obtained. The second relationship describes the fact that, given an object of a certain class, and a destination class type, there can be many reachable objects of that target class. The third type of relationship specifies the fact that, given two or more objects of the same or different class type, and given a destination class type, more than one object of that target class can be obtained. VHPI functions are provided to traverse these relationships. These three classes of relations are sufficient to describe the navigation throughout the VHPI information model. At present the VHPI information model only uses the first and second classes of relationships. A class can also have *properties* that generally describe inherent characteristics or attributes of that class. Property values can be queried with some predefined VHPI functions. Other functions are provided to *get* or *modify* VHDL values; these functions are only available from some classes of objects. Interaction between the VHPI interface and the tool is achieved via *callbacks*. Finally, *utility* functions are provided for printing, checking errors for example In summary, the procedural interface contains about thirty functions, from which less than ten are used for accessing the complete VHDL information model.

2. VHPI Handles

2.1 Objects and handles

Object definition: The VHPI information model represents VHDL static and runtime data. In the VHPI information model, there are static and dynamic objects represented as classes with associated properties and methods.

Handle definition: A handle is a reference to an object of the information model.

The VHPI functions manipulate VHDL data at some abstraction level. The user only gets back *handles* which basically are an abstract representation of some VHDL object such as for example, an instance, a signal or a transaction. A handle is an *opaque* pointer to some VHDL object represented in the information model. The handle identifies some static elaborated and/or dynamic runtime VHDL information, the VHPI interface knows how to relate handles with the object they represent. Users cannot make an assumption about the underlying internal representation of a handle.

The C type of the handle (vhpiHandleT) is predefined by the interface (typedef PLI_UINT32 *vhpiHandleT). The VHPI interface functions manipulate and create handles. A VHPI handle is used to reference any VHDL object that is defined in the information model that is identifying static or dynamic data. The VHPI interface defines a meta model that describes the mechanisms on how to access information. In the meta model, any handle has one property called *vhpiKindP* that identifies the class of which the VHPI handle is an instance. The class defines the type of VHDL information the handle points to. For example, if a VHPI handle is a handle to a variable declaration, the kind property will return the integer constant corresponding to the variable declaration class (#define vhpiVarDeclK <number>). The interface predefines an integer kind for each leaf class of the information model. Depending on the handle class, some relations (methods), properties (attributes) are available; these relations and properties are described by the VHPI information model. VHPI interface functions only apply to the leaf classes.

2.2 Handle management functions

In this section, we describe how handles to objects denoted by the information model are allocated and freed.

2.2.1 Handle creation

Handles are created by the VHPI interface navigation and creation functions. An interface implementation may choose to share handles between various applications and to return the same handle each time the same object is accessed. Handle creation can be optional for certain classes of objects such as callbacks where the user is given the choice to request explicitly the creation of a handle. All access navigation functions such *as vhpi_iterator(), vhpi_scan(), vhpi_handle(), vhpi_handle_by_name()* and *vhpi_handle_by_index(),* (refer to chapter 3) create and return handles. Handles are owned by the VHPI client application.

2.2.2 Handle release

All handles need to be explicitly released by the VHPI user. A function (*vhpi_release_handle()*) is provided to request the release of a handle. If a handle is shared between VHPI applications, the release of the handle may not be effective until all client applications have requested the handle release. After a handle is released, **no** reference should be made to that handle. The user cannot assume that the handle still exists neither that it refers to the same object. It is recommended that VHPI users release handles when they are not needed. An iterator is automatically released by the VHPI interface at the end of the iteration.

2.2.3 Handle comparison

Two different handles may identify the same VHPI object. The interface provides a function to compare two handles (*vhpi_compare_handles()*). This function will return true if the handles refer to the same object, false otherwise.

2.3 Lifetime of objects and handles

2.3.1 Object lifetime

A static object comes into existence at a particular point in time in the tool's execution and lives until the tool exits. When an object comes into existence, it is possible to obtain a handle to the object. For example, a component instance in the design hierarchy is a static object that comes into existence sometime during elaboration.

A dynamic object comes into existence and may cease to exist sometime later. They exist for as long they are required or until they are removed. For example, subprogram's formal parameters are dynamically elaborated when the subprogram is called and cease to exist when the subprogram completes and returns. Transactions on drivers are created with waveforms and may be cancelled by future waveform edits. Callbacks are created and removed by the VHPI user.

2.3.2 Handle lifetime

A handle comes into existence when it is returned to the user. It lives until the user releases it. There are various methods of navigating the VHPI information that create and return handles to the user, e.g., *vhpi_handle, vhpi_iterator, vhpi_create*. The user releases a handle with *vhpi_release_handle*.

2.3.3 Invalid handles

When a handle to an object is obtained, a VHPI client may access through this handle some object properties, access or modify the object runtime value (*vhpi_get_value, vhpi_put_value*) or navigate to related objects. If the object is a dynamic object, it may cease to exist. A handle to a dynamic object that no longer exists is called an **invalid** handle. The handle exists, the object doesn't. For any handle, a boolean property is defined to check the validity of that handle (**vhpiIsInvalidP**). An invalid handle may be released by a VHPI application (keeping invalid handles is not very useful).

2.3.4 Referential integrity

With the above terminology, we can define the concept of referential integrity of handles. In this context, it means that for as long as a handle exists, it is safe to reference it. You may use it in any VHPI function that accepts a handle, and that function will attempt to perform its operation. Regardless whether that attempt is legal in the information model or results in a VHPI runtime error, it will not cause the tool to crash.

In particular, invalid handles have referential integrity. It may certainly be treated as an error if you reference an invalid handle, depending on the particular type of handle and operation requested. Handles to mature callbacks also have referential integrity. A handle to a mature callback has very little value for a VHPI client: it cannot be re-enabled, and it cannot be discovered via traversal of the information model. It should be deleted by the VHPI server,unless the client(user) has previously obtained a handle to the transaction. If the client has a handle, he has ownership, albeit to something of marginal value. He can query some of its properties and methods it or just waste the memory resource. It follows that, after all such handles are released with *vhpi_release_handle()*, the mature callback should be deleted by the VHPI server. The VHPI server is free to waste resources itself, but the point is, it has ownership of the callback memory.

If a handle is released regardless how it is released either explicitly by the user or by the tool, it has no longer referential integrity. If you reference a handle after you have released or after an iterator is exhausted, it is an egregious error which can cause the tool to crash. It is similar to referencing freed memory in a C program.

2.4 Meta handles

The information model also defines meta classes. Meta classes do not represent any VHDL object. For example iterator and collection classes (*vhpiIteratorK* and *vhpiAnyCollectionK* kinds) represent respectively iteration lists and ordered collections of objects. Meta handles are subject to the same referential integrity rules as other handles.

2.4.1 Iterator class

Iterator handles are handles defined by the interface to access many objects of the same class type. Iterator handles are used to traverse one-to-many relationships that connect classes. Iterator handles are slightly different from non-meta handles in the sense that they cannot be shared between applications because they hold the state of the current iteration. The *vhpiKindP* of an iterator handle is *vhpiIteratorK*. The iterator class is a sub-class of the base class. An unique iterator handle is created by each call to *vhpi_iterator*. *vhpi_scan* takes an iterator handle and can be used to return a handle to each iteration element. When there is no more element to return, *vhpi_scan* returns a NULL handle. The iterator handle is released automatically by the simulator at the end of the iteration. Reference to the iterator handle after the end of an iteration is erroneous. It the iteration is not exhausted, the user should explicitly release the iterator handle to avoid a memory leak.

Adding elements to an iteration while processing an iterator has unspecified behaviour. For example, while iterating on callbacks, register a new callback, or while iterating over the members of a collection, add a new member has unspecified behaviour.

2.4.2 Collection class

The *vhpiKindP* of a collection handle is either a *vhpiAnyCollectionK* or a specialized collection of objects of the same kind, for example a collection of drivers is *vhpiDriverCollectionK*. The collection class represents a user-defined collection of VHPI objects. The collection contains an arbitrarily sized, ordered set of VHPI objects. The collection may be created at any time, provided, of course the desired object members exist at the time. The purpose of the class is for the organizational convenience of an application or model. Atomic operations are defined on the collection class (see operations). The UML model defines a oneToMany method to iterate over the members of the collection (*vhpiMembers*). Iterating on the members of a collection handle only returns the valid handles.

2.4.2.1 Construction of a collection object

A collection object is created with *vhpi_create*. The first call provides a handle to the first object to be added to the collection and returns a handle to the collection object:

```
vphiHandleT myCollection;
myCollection = vhpi_create(vhpiAnyCollectionK, NULL, vhpiHandleT
anyObject);
```

Objects may be added to the collection, one at a time, as follows:

```
myCollection = vhpi_create(vhpAnyCollectionK, myCollection, vhpiHandleT
anotherObject);
```

The return value is a handle to the modified collection object or NULL if an error occured. The original collection object handle shall be passed as the second parameter, the handle to the object to be added to the collection shall be passed as the third parameter. The ordering of the collection set corresponds to the order in which objects are added to it. There is no restriction on when a collection may be created or when objects may be added to an existing collection.

NOTEs:

As is the case for all VHPI handles, a handle to a collection does not remain valid across process boundaries that may exist in the architecture of a particular VHPI-compliant tool.

Interleaving addition of elements to the collection while iterating over the members of the collection has unspecified behaviour.

2.4.2.2 Collection Object Lifetime

A collection exists from the time it is created until its handle is released. No navigation VHPI function ever returns collection handles. It is the application/model responsibility to keep a handle to the collection created. It is also its responsibility to release the collection handle when it is no longer needed. Releasing the collection handle does not release the handles the collection contains.

2.4.2.3 Referential Integrity

With respect to a collection of dynamic objects, if a handle to one of the collection members is obtained by iterating over the ***vhpiMembers*** relationship or accessed via *vhpi_handle_by_index*, it should always return a handle that is safe to reference. It does not matter whether the handle to the object of interest was released, this is a new handle obtained by accessing the collection member object. Moreover, if the collection is a collection of dynamic objects (callbacks for example), it does not matter if the dynamic object was removed or ceased to exist, the reference to it in the collection still exists in the same manner that a callback handle kept by a user can be referenced even after the callback has been removed.
Another way of stating the expected behavior is that the referential integrity of a collection transitively includes the referential integrity of a handle to any of its underlying objects.

2.4.2.4 Operations on a collection

There is a powerful generality with a collection that is possible, but it is being used in only one narrow context in the current VHPI specification. VHPI operations are defined on some classes of the information model. A VHPI operation can be applied to a class which possesses that operation, the operation consists in modifying/accessing some of the class internal data. Such operations are for example vhpi_put_value, vhpi_get_value, vhpi_schedule_transaction, vhpi_register_cb… Operations operate on an object which reference handle is passed as an argument to the VHPI operation. It is desirable in many contexts to apply the same operation to a set of objects by a series of identical sequential operation calls where the reference handle changes to point to a different object. It would be a very convenient shorthand to allow certain VHPI operations to accept a handle to a collection. Such an operation would appear in the information model as an operation defined on a collection, and you can think of its definition being a "delegated" operation defined on the individual members of the collection. There is no supported use of collections in this manner.
This is a also a mechanism that can be used when a series of operations on separate handles must be treated as an atomic operation. A collection of driver handles that represent the sub-element drivers of a composite unresolved signal may be used as the reference handle with *vhpi_schedule_transaction* to schedule a composite transaction. This is the only context that is being proposed for a collection.
Property access or method navigation do not work by delegation: a common property of the elements of a collection cannot be queried on a reference handle of the collection.

2.4.2.5 Error Handling

An operation applied to a collection may have an error associated with one or members of the collection. Such an operation will be recognized as a single error which may be reflected in the return value of function as well as be accessible with *vhpi_check_error*. The correctness of use of an operation on a collection is the transitive correctness of that operation on each member of the collection. Specific error conditions are defined with the VHPI function that accepts a collection.

Future Considerations

The formal information model may be extended with notation that reflects operations on an object that may be delegated to collections of such objects. That notation will indicate that the collection must be a homogenous set of objects of the kind that support the operation and its delegation. This will allow the actual C binding of the operation to indicate that a collection handle or a object handle is allowed for this function. The function reference may provide details of what kinds of collections are allowed. This detail should be inferable from the formal information model as well.

This concept of a collection may also extended to support other methods for collection construction. For example, it may be reasonable to navigate an association that returns the collection of drivers of a composite with a simple call to vhpi_handle.

3. Interface function overview

3.1 Information Access Routines

The VHPI interface distinguishes two types of access:

- accessing a handle of some class type from a reference handle of a given class type; this is a *single* or *one-to-one* relationship.
- accessing a list of handles of the same class type from a reference handle of a given class type; this is a *multiple* or *one-to-many* relationship.

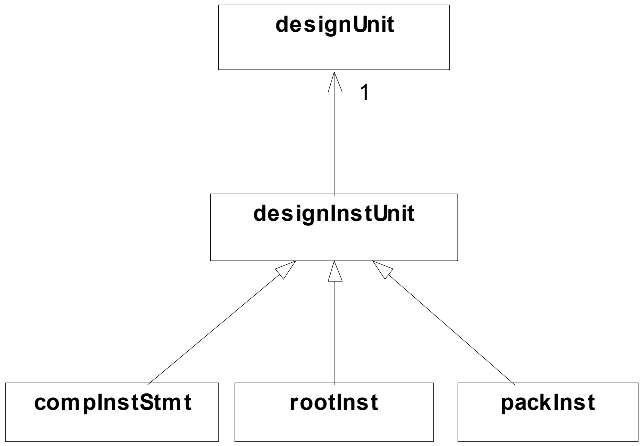
In order to perform these two classes of operations, the VHPI interface defines respectively two mechanisms:
vhpi_handle() to traverse single relationships and *vhpi_iterator()* in conjunction with *vhpi_scan()* to traverse multiple relationships.

Note: A relationship is also called an **association**.
Please refer to section 4.1 for a description of the UML notation used by the diagrams in this chapter.

3.1.1 Single relationship traversal function

The interface provides one function *vhpi_handle()* to traverse one-to-one relationships between objects. The one-to-one relationships are defined by the information model. A one-to-one relationship exists between a reference (source) class and a target (destination) class if there is **at most one** handle of the target class type that can be obtained by traversing this relationship. Given a reference handle of a VHPI kind (refHdl), and given a VHPI class kind, the interface returns a handle of that destination VHPI class reflecting the traversal of the relation (example 1). The single relationship can also be marked with a relation name, which name should be used instead of the destination class type (example 2). Named associations are used to disambiguate the relationship to traverse when it is possible to reach the same class type from the same reference handle or to add specific semantic information. These relationship are said to have a tag. The information model defines the set of one-to-one relationships.

<u>Example 1:</u> unnamed relationships
From a *vhpiCompInstStmtK* reference handle, it is possible to access the design unit that is bound to the component instance by following the relationship between a *vhpiCompInstStmtK* and the *vhpiDesignUnit* class.



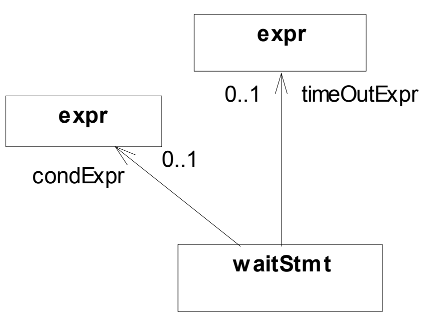

```
vhpiHandleT instHdl, duHdl;
void get_binding_info(instHdl) {
vhpiHandleT instHdl;
switch (vhpi_get(vhpiKindP, instHdl)) {
    case vhpiCompInstStmtK:
    case vhpiRootInstK:
    case vhpiPackInstK:
        duHdl = vhpi_handle(vhpiDesignUnit, instHdl);
        vhpi_printf ("design unit name %s", vhpi_get_str(vhpiUnitNameP,
duHdl));
        vhpi_printf("in library %s\n", vhpi_get_str(vhpiLibLogicalNameP,
duHdl));



,
        break;
    default:
        break;
}/* end switch */
}/* get_binding_info() */
```

<u>Example 2:</u> tagged relationships

From a *vhpiWaitStmtK* reference handle, it is possible to access the expression of the condition by following the directed tagged relationship *vhpiCondExpr*. It is also possible to access the time out expression by following the directed tagged relationship *vhpiTimeOutExpr*.



```
vhpiHandleT stmtHdl, condHdl, timeHdl;
if (vhpi_get(vhpiKindP, stmtHdl) == vhpiWaitStmtK) {
    condHdl = vhpi_handle(vhpiCondExpr, stmtHdl);
    timeHdl = vhpi_handle(vhpiTimeOutExpr, stmtHdl);
}
```

3.1.2 Iteration functions and vhpi_handle_by_index

The interface provides a mechanism to traverse one-to-many relationships. These relationships are defined by the information model. This is a two phase mechanism: first, an iterator handle for the class of objects one wants to iterate on is created and initialized with the function *vhpi_iterator()*, second, a function *vhpi_scan()* is provided to scan the list of handles designated by this iterator. The *vhpi_scan()* function returns a handle for each of the objects of the requested iteration type in the iteration list. vhpi_iterator() returns NULL if there is no element in the iteration list. The iteration list can be a dynamic list for callbacks, or transaction iterations. The iteration reflects the current simulation state. Therefore handles to dynamic data returned by *vhpi_scan()* may become invalid as simulation progresses. Callbacks can be added or removed during simulation by either the VHDL simulator or by the VHPI functions such as *vhpi_register_cb(), vhpi_remove_cb();* transactions can be scheduled with *vhpi_put_value()* or *vhpi_schedule_transaction().*

Note: Unless specified by the information model, an iteration does not define an order therefore user code should not be dependent upon the order the handles are returned in order to be portable.

Example:
Please refer to the scope class diagram.

```
vhpiHandleT instHdl, instIter;

/* get all sub-instances of a scope instance */
instIter = vhpi_iterator(vhpiInternalRegions, instHdl);
if (instIter)
while (instHdl = vhpi_scan (instIter)){
    vhpi_printf("found instance %s\n", vhpi_get_str(vhpiNameP, instHdl));
}
```

For any iteration which is qualified as "ordered" in the information model, it is possible to request a handle to a specific object of that iteration by providing an index n. The function vhpi_handle_by_index() shall take a reference handle which is the same as the one which would be passed to vhpi_iterator() and an index n, and would return the handle which would have been returned by the n^{th} vhpi_scan() call.

3.2 Simple property access functions

The interface provides functions to access class properties. There are several classes of properties: boolean or integer properties, string properties, real or physical properties. The interface provides a function to query about boolean or integer properties *vhpi_get(),* a function to retrieve the string value of a string property *vhpi_get_str(),* a function to get the value of a real property *vhpi_get_real()* and a function to get the value of a physical property *vhpi_get_phys().*
These functions take a handle to an object, the property of interest and respectively return an integer, a string, a real or a physical value. The details are given in the following paragraphs.

3.2.1 Integer or boolean properties

The function vhpi_get returns the value of an integer or boolean property.
(vhpiIntT) vhpi_get(vhpiIntPropertyT propertyTag, vhpiHandleT handle);

propertyTag: is the tag name of the integer property.

handle: is a handle to an object of the information that must possess this property

returns: an integer constant corresponding to the value of the requested property for the given handle,
vhpiUndefined if an error ocurred.

vhpiIntPropertyT is an enumerated standard type of all the integer and boolean properties defined by the interface.
vhpiIntT is a VHPI typedef; an implementation should define it to be able to represent the entire range of VHDL integers.
#define vhpiUndefined -1

For boolean properties we define:
#define vhpiTrue 1
#define vhpiFalse 0

Specific values are defined for each integer property. For instance, vhpiModeP property can return the following values vhpiInMode, vhpiOutMode, vhpiInoutMode, vhpiBufferMode, vhpiLinkageMode or vhpiUndefined.
#define vhpiInMode 1001
#define vhpiOutMode 1002
#define vhpiInoutMode 1003
#define vhpiBufferemode 1004
#define vhpiLinkageMode 1005

Since some integer properties may return negative values; for example the vhpiLeftBoundP or vhpiRightBoundP of a negative range of an array, it may not be possible in all cases to determine that an error occurred by looking at the returned values. In these rare cases, a possible error must be checked according to the error checking mechanism.

Procedural Interface References:
See "vhpi_get_str()" to get a string property value.
See "vhpi_get_real()" to get a real property value.
See "vhpi_get_phys()" to get a physical property value.

Enumeration type for the integer or boolean properties is vhpiIntPropertyT.

Errors:
For most of the properties, a returned value of vhpiUndefined will indicate that an error occurred.

3.2.2 String properties

The function vhpi_get_str returns the value of a string property such as vhpiNameP, vhpiFullNameP...
The available string properties which can be queried with this function are listed by the vhpiStrPropertyT enumeration type in the vhpi_user.h header file.

const PLI_BYTE8 * vhpi_get_str(vhpiStrPropertyT propertyTag, vhpiHandleT handle);

propertyTag: is the tag name of a string property.
handle : denotes a handle to an object which possess the string property.

<u>returns:</u> a pointer to a static string buffer that has been filled up with the string value of the property successfully retrieved or NULL on failure.

Notes:

1. The next call to vhpi_get_str will overwrite the previous string value.
2. Since VHDL identifiers can contain special and graphic characters, the user must be cautious when using the C string library functions or C printf functions when manipulating VHDL strings for name properties. The name property of an extended identifier contains the starting and ending \ character.

.

Example:

```
line
1          library lib;
2          use lib.p.all;
3          entity top is
4            postoned assert FALSE report "top level";
5          end top;
6
7          architecture struct of top is
8          signal mySig : integer;
9          for cpu_1 use entity lib.e(a);
10          begin
11            Cpu_1 : mycpu;
12            process(MYSIG)
13            begin
14              wait for 1 ns;
15            end process;
16          end;

For a handle to the root instance:
vhpiNameP = ":"
vhpiFullNameP = ":"
vhpiDefNameP = "work:top(struct)"
For a handle to the primary design unit (entity declaration) that is
bound to the root instance:
vhpiUnitNameP = "lib.top"
For a handle to the assert statement in the entity declaration:
vhpiNameP = "_0"
vhpiFullNameP = ":_0"
For a handle to the signal declaration in the architecture:
vhpiNameP = "mysig"
vhpiFullNameP = ":mysig"
vhpiCaseNameP = "mySig";
vhpiFullCaseNameP = ":mySig"
For a handle to the component instance labelled cpu_1 :
vhpiNameP = "cpu_1"
vhpiCaseNameP: "Cpu_1"
vhpiFullNameP = ":cpu_1"
vhpiFullCaseNameP = ":Cpu_1"
vhpiDefNameP = "lib:e(a)"
For a handle to the process statement in the architecture body:
vhpiNameP = "_1"
vhpiFullNameP = ":_1"
For a handle to signal s declared in package P:
vhpiFullNameP = ":lib:p:s"
```

Procedural Interface References:

See "vhpi_get()" to get an integer based property value.
See "vhpi_get_real()" to get a real property value.

See “vhpi_get_phys()” to get a physical property value.

Enumeration type for the string properties is vhpiStrPropertyT.

See ANNEX B for description of each string property.

Errors:
A NULL returned indicates that an error occurred.
The requested property does not apply to this object class kind.
Another property access function must be used for this property class.

3.2.3 Real properties

vhpi_get_real() will return a double, is only used to get the vhpiFloatLeftBound and vhpiFloatRightBound of a floating range. The caller must use the error checking mechanism to determine if an error occurred.

Procedural Interface References:
See “vhpi_get_str()” to get a string property value.
See “vhpi_get()” to get an integer based property value.
See “vhpi_get_phys()” to get a physical based property value.

Enumeration type for the real properties is vhpiRealPropertyT

3.2.4 Physical properties

vhpi_get_phys() will return a vhpiPhysT structure. It is used to get the value of a physical property (for example vhpiResolutionLimitP of the simulation or vhpiPhysLeftBound of a physical type). The value returned is a two member structure.

```
typedef struct vhpiPhysS {
   PLI_INT32 high;
   PLI_UINT32 low;
} vhpiPhysT;
```

Procedural Interface References:
See “vhpi_get_str()” to get a string property value.
See “vhpi_get()” to get an integer based property value.
See “vhpi_get_real()” to get a real based property value.

Enumeration type for the physical properties is vhpiPhysPropertyT.

Errors:
The requested property does not apply to this object class kind.
Another property access function must be used for this property class.

3.3 Look up by name

The interface provides a function to get a handle to an object in the VHDL design hierarchy by relative or absolute name, vhpi_handle_by_name(). This function is operational in the post analysis, post elaboration and runtime domains. This function returns a handle to any object which possesses the vhpiFullNameP property.

3.4 Value manipulation functions

3.4.1 Value access function

The interface provides a function to get the value of an object (vhpi_get_value). Only certain classes of objects have a value that can be accessed with this function. Valid classes include VHPI handle kinds denoting VHDL objects, VHDL names or literals. The standard requires vhpi_get_value to support the access to values of locally static names only. Note that it is not possible to get the value of any complex expression directly. Subprogram parameter values can only be fetched when the subprogram is executed. The function gets the current value of the designated object, therefore default or initial values of VHDL objects can be fetched at the beginning of simulation, thereafter the value of the object is the simulation value at the present time. The vhpi_get_value function takes a handle to an object that possesses the vhpi_get_value method, a pointer to a value structure that has been allocated by the user and fills up the appropriate value field in the requested format.

3.4.2 Value formatting function

The interface provides a function to format a value (vhpi_format_value). This function takes as the first parameter, the input VHPI value structure, and as the second parameter a VHPI value structure which format field should be set to specify the new format in which the value must be formatted.

3.4.3 Value modification functions

3.4.3.1 Immediate update

The interface provides a function to immediately update the value of an object. Only certain classes of objects can be modified with this function such as handles denoting signals or variables. VHPI classes of objects that are valid for this function are marked in the information by having the vhpi_put_value method. Different update modes are available: deposit, deposit and propagate, force until release, force propagate with event, and release the value.

3.4.3.2 Value scheduling

The interface provides a function to schedule a transaction on a signal driver or collection of drivers (vhpi_schedule_transaction()). This allows VHPI to participate in signal value update and resolution. Different scheduling modes are available and mimic the inertial and transport mode. A delay can be specified enabling to modify the future waveform of a signal. The function takes a handle to a driver or collection, a value structure, a delay mode and an optional delay time value.

3.5 Foreign Model Support

VHPI supports the creation of foreign architectures and foreign subprograms with semantics equivalent to anything that can be written in VHDL directly.
VHPI supports registering foreign models procedurally with vhpi_register_foreignf() and querying registration with vhpi_get_foreignf_info(). New drivers can be created with vhpi_create() and vhpi_assert() emulates a VHDL report statement

3.6 Callbacks

3.6.1 Functions for registration, removing, disabling, enabling callbacks

The interaction between the PLI models or applications and the tool is done via the callback mechanism. The interface provides functions to register, remove, disable or enable callbacks. The interface provides a rich set of callback reasons.

VHPI supports accessing registered callbacks through an iteration mechanism. These can be the complete set of callbacks registered or callbacks registered on signals, ports and so on. Multiple applications can register these callbacks; when iterating on callbacks an application can get a hold of a callback registered by another application.

3.7 Utilities and Miscellaneous Functions

The VHPI functions discussed above cover the full range of information access for the uninstantiated, instantiated, and runtime information models. There remains a small number of utility and miscellaneous functions to complete the VHPI functional interface overview.

3.7.1 Error checking

VHPI calls can produce errors. There are two mechanisms in VHPI to handle them:

1) checking a global error status,
2) registering a callback on errror.

The first mechanism requires calling vhpi_check_error() immediately after each VHPI function call to see if the last VHPI function call queued an error. The second is by using an error callback mechanism, where the callback function will be called whenever a VHPI call produces an error. However the drawback of this last mechanism that VHPI does not keep track of which callbacks are registered for which application; therefore a VHPI client may get a callback for an error it did not produce. It is necessary for that VHPI client to recognize its own callbacks: the callback function called on error should be tied to the application or model that produced that error.

Note: John would like to remove the callback on error as it is dysfunctional.

3.7.2 Printing to stdout and log files

The function vhpi_printf() writes output to the output channel of the tool which invoked the VHPI application. vhpi_printf() uses the same formatting capabilities as the C printf function vhpi_printf() does not provide the capability to write to VHDL files.

3.7.3 Optional Save/Restart Support

If the VHDL tool supports a save/restart operation, provisions must be made to save private data associated with VHPI applications and foreign models and restore it later. Two routines, vhpi_put_data() and vhpi_get_data(), support this functionality.

Francoise: Should we mention anything about reset even though it does not add any other interface functions?

3.7.4 Miscellaneous Functions

vhpi_compare_handles() – checks if two handles refer to the same object.

vhpi_control() – provides some control capabilities over the tool, such as stopping or finishing execution.

vhpi_get_time() – returns the current simulation time

vhpi_get_next_time() – returns the next simulation time at which some activity is scheduled

vhpi_protected_call() – executes operations on variables of a protected type

vhpi_release_handle() – releases resources associated with a handle

4. The VHDL PLI information model

4.1 Formal notation

We use the standard Unified Modeling Language (UML) to formally express the VHPI information model. UML is a graphical language to model object-oriented software design. It defines a rigorous notation and a meta model of the notation (diagrams) that can be used to describe object-oriented software design. We use the *class diagram* technique of UML to express the VHDL PLI information model. A class diagram specifies the VHPI class types and the way they are connected together. In UML, class inheritance is denoted by a hollow arrow directed towards the parent class. Relationships between classes are called *associations* and are denoted by straight lines between classes. Associations have descriptive parameters such as *multiplicity, navigability and role names.*

UML notation quick reference

A class

A member class

The link shows **inheritance** between a class and its derived classes. A derived class inherits properties and operations from its parent classes. The hollow arrow points to the parent class.

An expanded class shows two compartments, the first one displays the **properties** (attributes in Object Oriented terminology) with their names and return type, the second one displays the **operations** (methods in Object-Oriented terminology) that are defined within this class. Properties and operations inherited from parent classes may not appear in the compartment boxes of the derived classes.

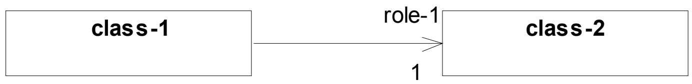
Associations

Associations are links between classes that represent their inter-relationships.
Navigability, multiplicity and role names can be used to further describe the relationship.
Navigability expresses the direction of access and is represented by an arrow. An association can be bi-directional in which case arrows may be shown at both ends.
Multiplicity expresses the type of relationship between the classes: singular (one, zero or one), multiple (zero or more, one or more) and is represented by numbers at the end of the association to which it applies. It can be one the following:

- 1 for access to one object handle
- 0..1 for access to zero or one object handle
- * for access to zero or more object handles of the same class
- 1..* for access to one or more object handles of the same class

A **role name** is a tag name on one end of the association. It may be used to indicate more precisely the relationship or to distinguish this relationship from another relationship that leads to an object of the same class. In the example below, role-1 is the name of the relation that accesses an object of class-2 from an object of class-1. The relationship it denotes is a singular relationship.
In the diagrams, we use the following convention:
if a role name is not specified, the method name for accessing the object pointed by the arrow is the target class name.

class-1 accesses class-2, method name is role-1



scope class accesses decl class, method name is decls.

The VHPI iteration or one-to-many method is modeled by an association with a multiplicity of either zero or more (*), or one or more (1..*) to indicate that the iteration may contain zero elements or will contain at least one element. The direction or navigability indicates the class of the handles created by the iteration. In the example above, we show that there is a one-to-many relationship between a scope class and a decl class. A singular or one-to-one method will be represented by a navigable association with a multiplicity of one (1) if the method should always return a handle of the destination class or a multiplicity of zero or one (0..1) if the method may not return a handle. In the example above, the diagram shows a one-to-one relationship that allows to traverse the association named “role-1” between a handle of class-1 and a handle of class-2. Note that the diagrams only express the possible access flow; for example there is no method that allows to get a handle of class-1 from a reference handle of class-2.
An iteration qualified as “ordered” indicates that the elements of the iteration are always returned in a given order and that this order must be the same in all VHPI compliant implementations. If an iteration is qualified as ordered, it also means that the vhpi_handle_by_index function is available on the reference handle and can be used to return the n^{th} element of the ordered iteration.

4.2 Classes overview

The information model is partitioned into a few UML packages, each package contains several class diagrams which are related to a given capability. A class diagram shows the traversal relationships between related classes.

- the standard hierarchy package includes class diagrams used to traverse VHDL design hierarchy, it typically describes the hierarchy capability set,
- the standard design unit package includes class diagrams to VHDL design units, and the lexical scope class diagram..
- the standard declaration package includes class diagrams describing object declarations.
- the standard type package includes class diagrams describing types and subtypes.
- the standard spec package includes class diagrams describing attributes, disconnection, and configuration specifications.
- the standard subprogram package includes class diagrams describing subprogram declarations and subprogram call access.
- the standard statement package includes class diagrams describing concurrent and sequential statements.

the

- the standard expression package includes class diagrams describing expressions.
- the standard connectivity package includes class diagrams describing the drivers and port connections access.
- the standard engine package includes class diagrams describing tool access
- the standard callback package includes class diagrams describing callback access
- the standard foreignf package includes class diagrams describing access to to foreign models.
- the standard meta package includes class diagrams of the base class, iterator and collection classes.

class
The class hierarchy defined by the VHPI model is the following. Each indentation indicates that a lower level in the hierarchy and that the indented class type denotes a child class of the immediately preceding class.
(*) indicates a class that inherits from multiple ancestor classes.
All classes inherit from the vhpiBase class.

The class “vhpiToolK” designates the simulator, elaborator or tool that is executing the VHDL model.

The base and null classes are meta-classes; which means that they do not belong to the information model of VHDL but are defined for modelling the access.

(1) The base class is the top of the class hiearchy.
It has 2 properties the vhpiKindP and vhpiKindStrP properties.
All defined classes inherits from the base class.

(2) The Null class denotes the VHDL elaborated or uninstantiated design.
It is named the null class because a null pointer handles is used to refer to it.
For example: vhpi_handle(vhpiRootInst, NULL) will return the top level instance of the current elaborated VHDL design.
vhpi_get_phys(vhpiResolutionLimitP, NULL) returns the resolution.

(*) denotes multiple inheritance for the class
(?) Do we need this class?

- Base (see note 1)
    - Null (see note 2)
    - Region
        - EqProcessStmt(*)
        - BlockStmt(*)
        - GenerateStmt(*)
        - DesignInstUnit
            - CompInstStmt(*)
            - RootInst
            - PackInst
        - SubpCall(*)
        - ForLoopStmt(*)
    - Decl
        - TypeDecl(*)
            - ScalarTypeDecl
                - EnumTypeDecl
                - IntegerTypeDecl
                - FloatingTypeDecl
                - PhysicalTypeDecl
            - CompositeTypeDecl
                - ArrayTypeDecl
                - RecordTypeDecl
            - FileTypeDecl
            - AccessTypeDecl
        - SubtypeDecl(*)
        - SubpDecl
            - FuncDecl
            - ProcDecl
        - AliasDecl
        - AttrDecl
        - ElemDecl
        - UnitDecl
        - ObjDecl
            - FileDecl
            - ConstDecl
            - VarDecl
            - SigDecl
            - InterfaceDecl
                - portDecl(*)

- genericDecl
- ParamDecl
  - ConstParamDecl
  - SigParamDecl
  - VarParamDecl
- FileParamDecl

- Subtype
  - TypeMark (? Do we need this)
    - SubTypeDecl(*)
    - TypeDecl(*)
  - SubtypeIndic
- Range
  - IntRange
  - FloatRange

- AttrSpec
- DesignUnit
  - PrimaryUnit
    - EntityDecl
    - PackDecl
    - ConfigDecl
  - SecondaryUnit
    - ArchBody
    - PackBody

- StackFrame
  - EqProcessStmt(*)
  - SubpCall(*)
    - FuncCall(*)
    - ProcCallStmt(*)
- Signal
  - PortDecl(*)
  - SigDecl(*)
  - SigParamDecl
  - SelectedName
  - IndexedName
  - PredefAttrName(*) (predefined signal attribute)
  - GuardSignal (? do we need it?)
- InterfaceElem
  - Port
  - Signal
  - Conversion
  - Expr
- Source
  - Driver
  - FuncCall
  - Port
  - Signal
  - Conversion(*)
- Stmt
  - ConcStmt
    - EqProcessStmt(*)
      - ProcessStmt
      - ProcCallStmt(*)
      - CondSigAssignStmt(*)

```
            SelectSigAssignStmt(*)
            AssertStmt(*)
        CompInstStmt(*)
        GenerateStmt(*)
            ForGenerate
            IfGenerate
        BlockStmt(*)
    SeqStmt
        WaitStmt
        ReportStmt(*)
        AssertStmt(*)
        IfStmt
        CaseStmt
        LoopStmt
            ForLoopStmt
            WhileLoppStmt
        NextStmt
        VarAssignStmt
        SeqSigAssignStmt(*)
        NullStmt
        ExitStmt
        ReturnStmt
        ProcCallStmt(*)
SigAssignStmt
    CondSigAssignStmt(*)
    SeqSigAssignStmt(*)
    SelectSigAssignStmt(*)

CondWaveform
SelectWaveform
WaveformElem
Transaction
Callback
TimeQueue

AssocElem
expr
    UnaryExpr
    BinaryExpr
    PrimaryExpr
        Operator
        Allocator
        Conversion(*)
        QualifiedExpr
        FuncCall(*)
        Aggregate
        Literal
        Name
            SimpName
            PrefixedName
                SelectedName
                DerefName
                IndexedName
                SliceName
                AttrName
```

UserAttrName
PredefAttrName(*)
    IndexedAttrName
    SimpAttrName

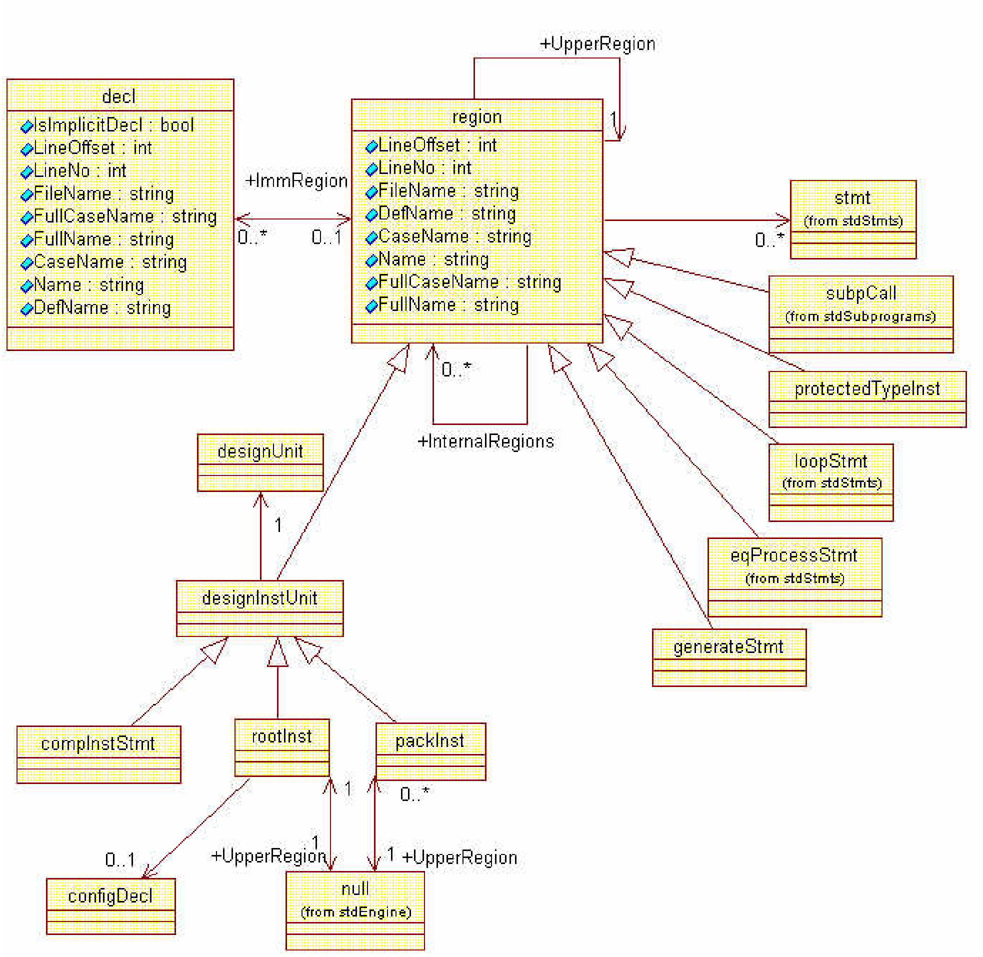
All traversal methods and properties defined by the information model if not implemented shall return a non implemented error message.

The information model is organized as a set of UML packages, each one containing one or more UML class diagrams. A UML package is a logical directory containing class diagrams depicting related functionality.

4.3 Standard hierarchy package (hierarchy capability set)

4.3.1 The region inheritance class diagram

This access represented by this diagram is part of the hierarchy capability with the exceptions of:
1) Access to a vhpiLoopStmt and vhpiSubpCall is only part of the dynamicElab capability.
2) Iteration on vhpiStmts is part of the static access capability.
3) The one to one relationship from a vhpiRootInstK class to a vhpiConfigDecl class is part of the post analysis access capability set.



Note:

1. Iteration on internal regions may return any region kind except loopstmt and protectedType kinds. A variable of a protectedType creates a protected type region. Upper region of a protected type shall return the region of the variable declaration. The name of a protected type region is the name of the variable. The fullname of a protected type region is the fullName of the variable. vhpi_handle_by_name of such fullName shall return a handle to the variable.

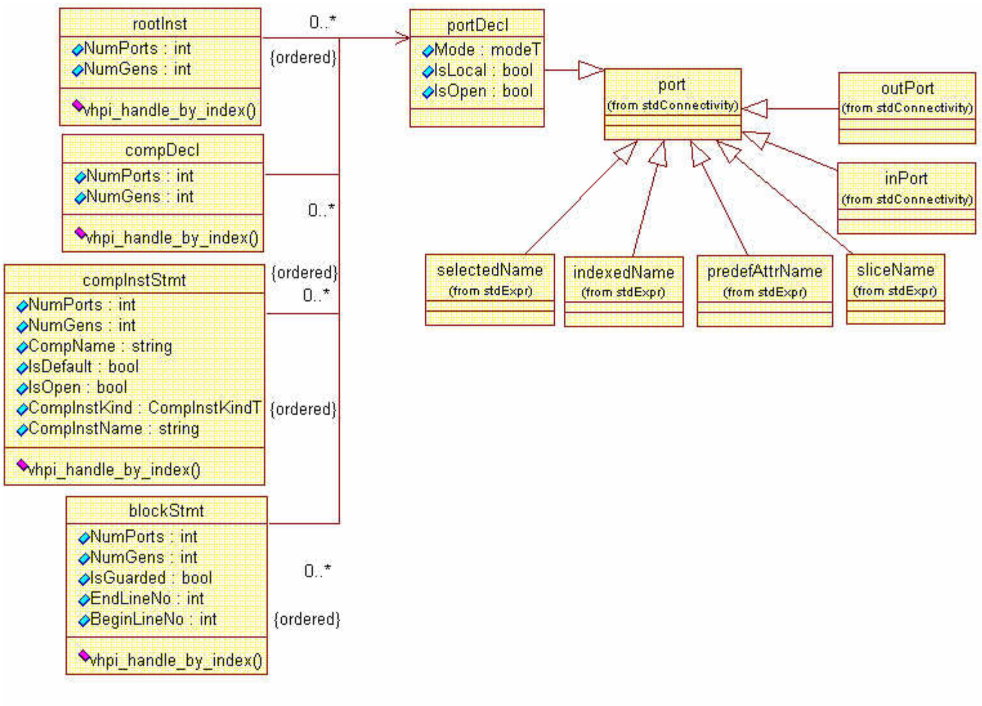
Deleted: ¶

Notes:

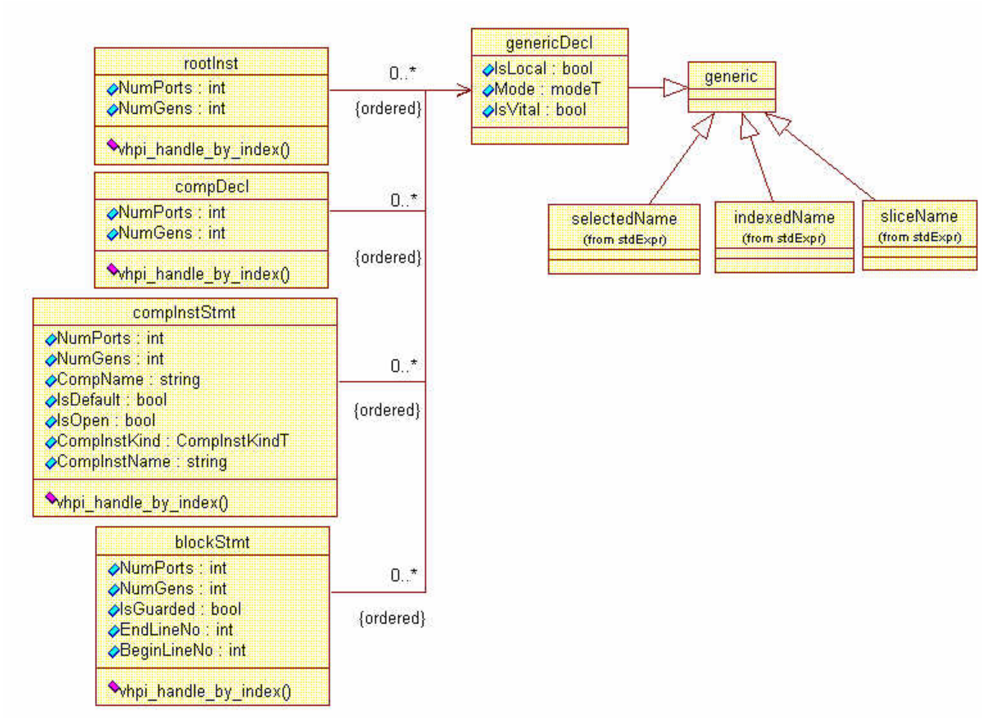
2. Iteration on internalRegions only return the elaborated regions.
3. Iteration on stmts return all statements, instantiated or uninstantiated.

4.3.2 The port class diagram

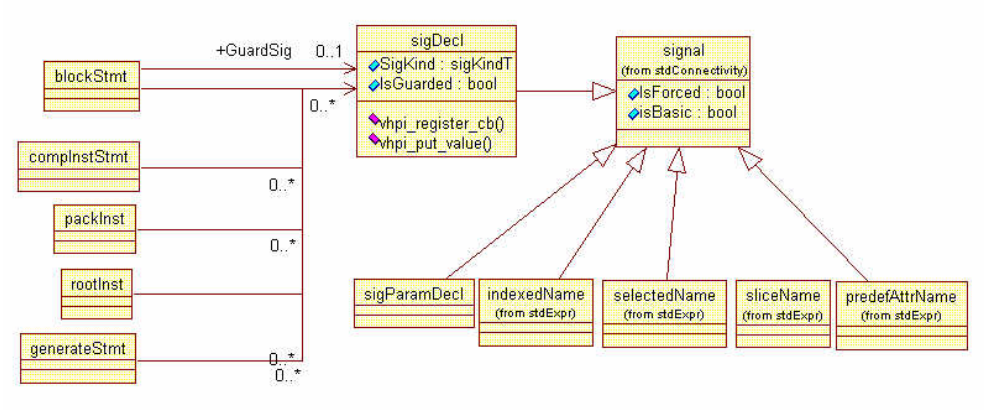
The access described by this diagram is part of the hierarchy capability.
The access described by this diagram is also part of the post analysis capability with the exception that a vhpiRootInstK class of object may never be obtained in the post analysis domain.



4.3.3 The generics class diagram



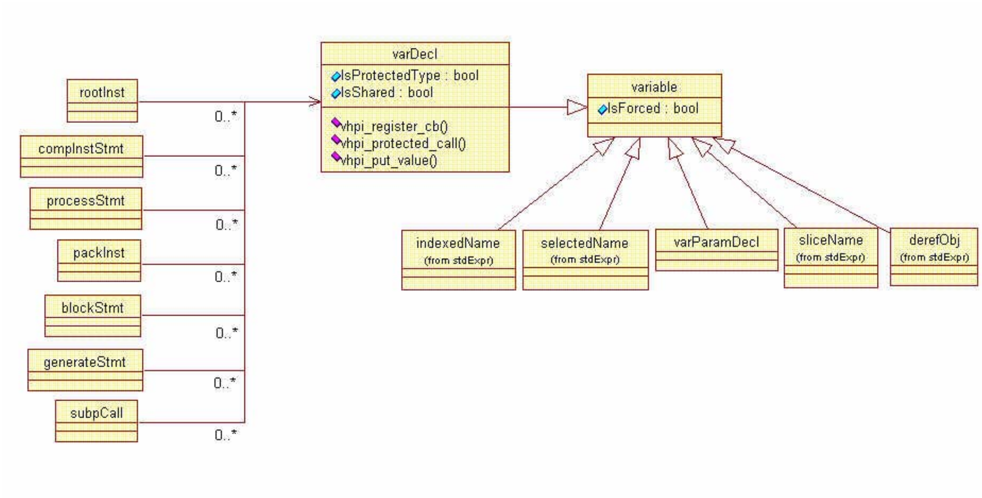
1 ### 4.3.4 The signals class diagram



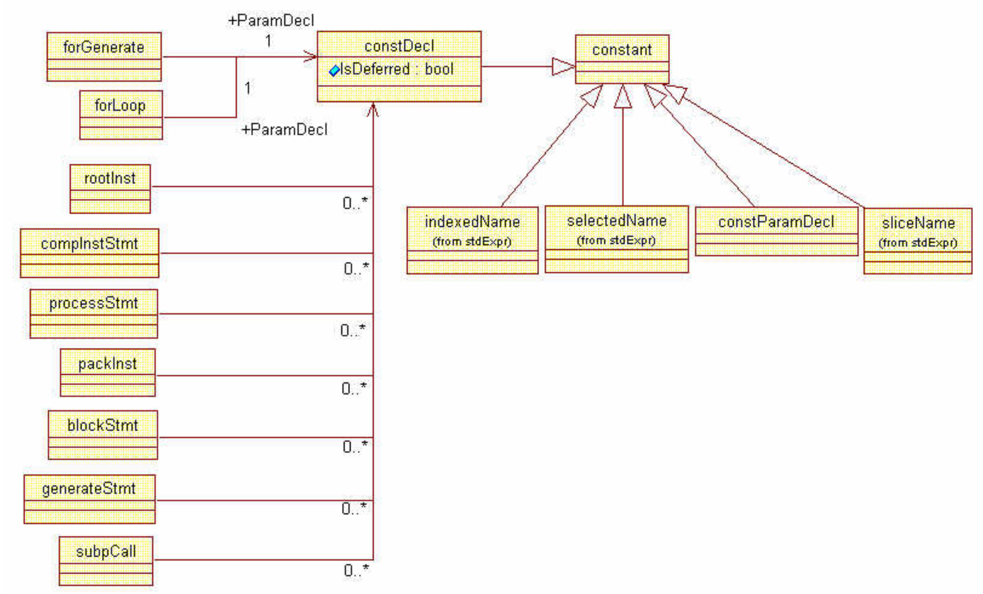
2

4.3.5 The variable class diagram

The access described by these diagrams is part of the static access and post analysis capabilities with the exception of vhpi_put_value which is part of the debug and runtime capability.
In addition, vhpi_put_value on is also part of the basic foreign model capabilities.
vhpi_register_cb for varDecl is also part of the debug and runtime capabilities and basic foreign model capabilities.
vhpi_protected_call() is also part of the debug and runtime capability.

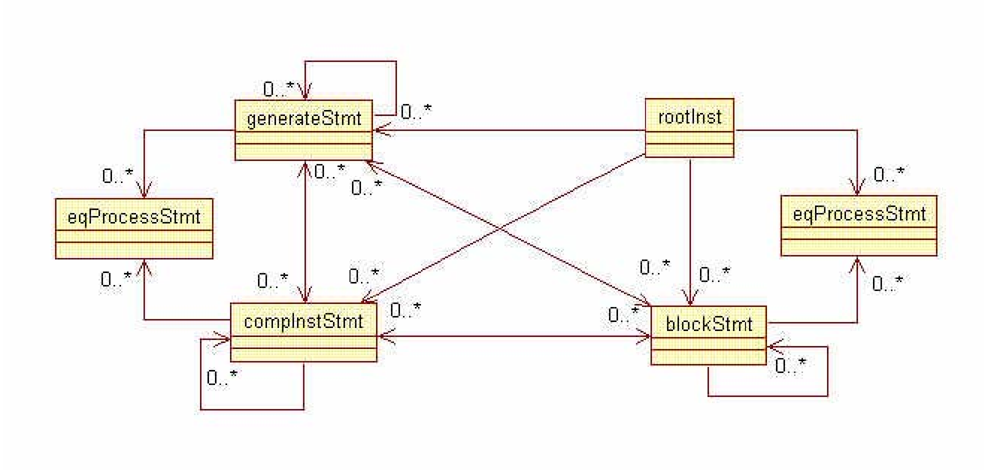


4.3.6 The constant class diagram



4.3.7 The structural class diagram

The access described by this diagram is part of the hierarchy capabilities.



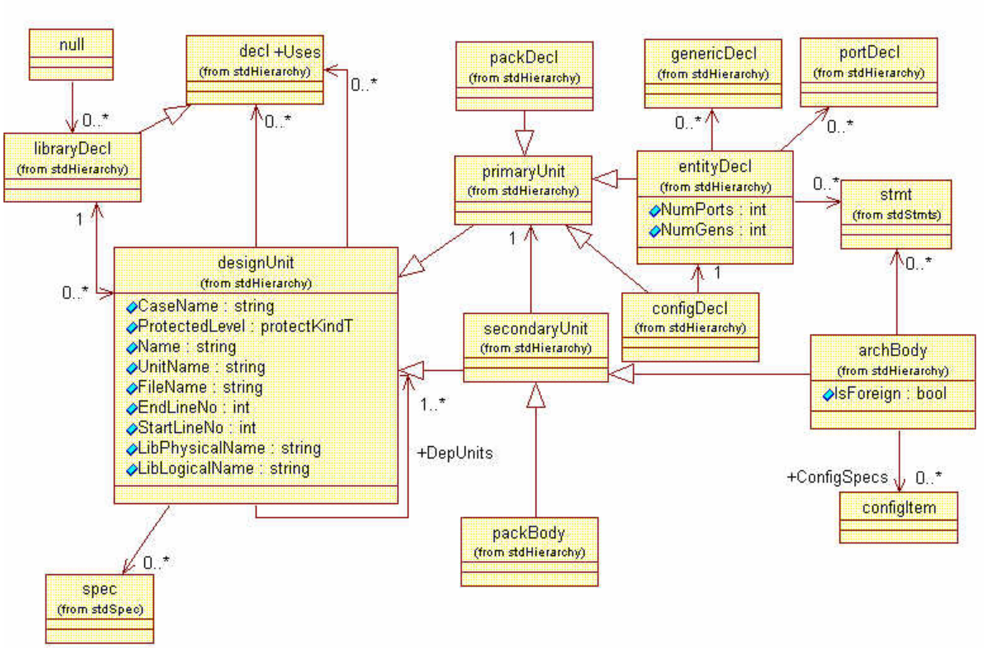
Notes:

4. The iterations are specialized iterations of the vhpiInternalRegions iteration. In a pure VHDL design, they are equivalent to iterating on vhpiInternalRegions and filtering a special kind of region. These iterations are useful when traversing a mixed language design. (May be put an informative Annex on use of VHPI and VPI when traversing mixed language designs.)
5. These iterations only return handles to elaborated regions.

4.4 Standard uninstantiated package (post analysis capability set)

4.4.1 The design unit class diagram

This diagram describes access which is part of the post analysis and static access capabilities.



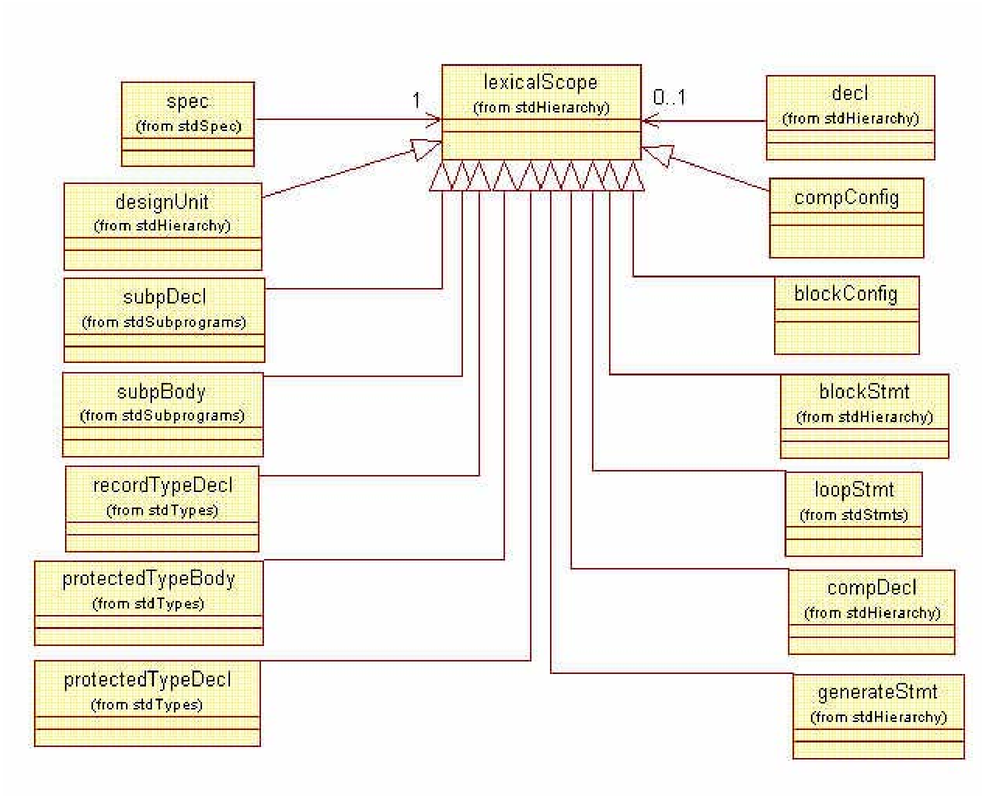
Note:

6. The iteration vhpiUses returns external declarations referenced by the design unit.
7. Access to the library declarations is not directly available but can be extrapolated from the uses iterations.

4.4.2 The lexical scope diagram

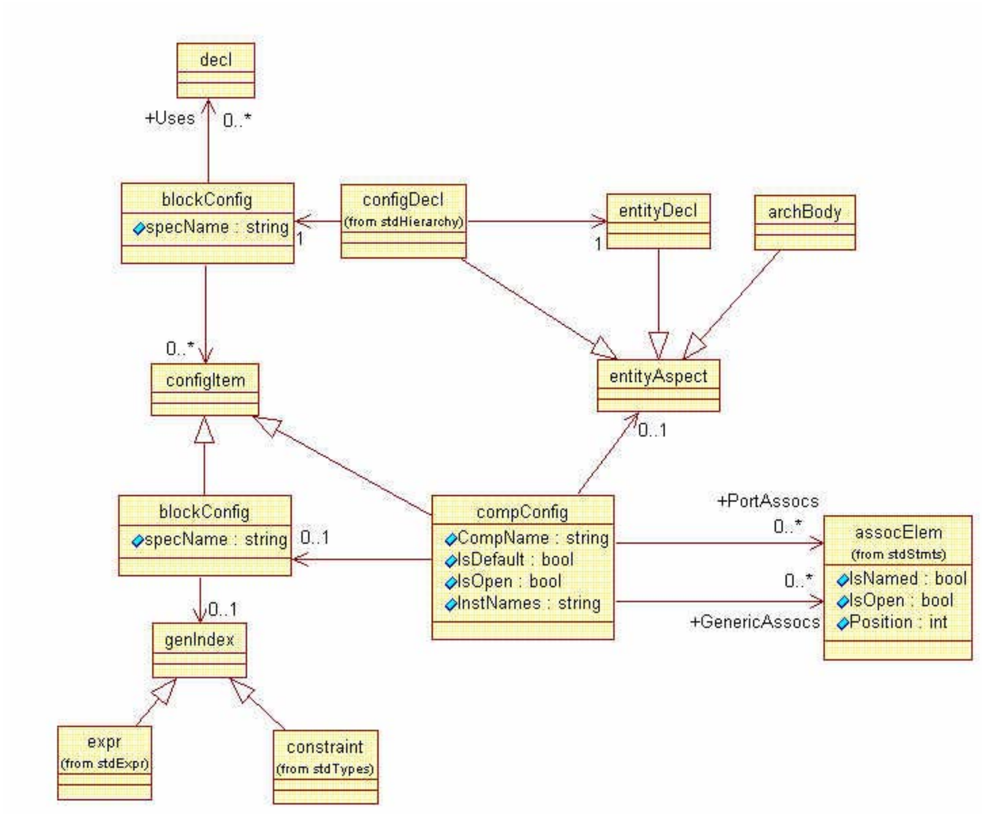
The access described by this diagram is also part of the post analysis capabilities.

.



4.4.3 The configuration declaration class diagram

The access described by this diagram is part of post analysis capability.



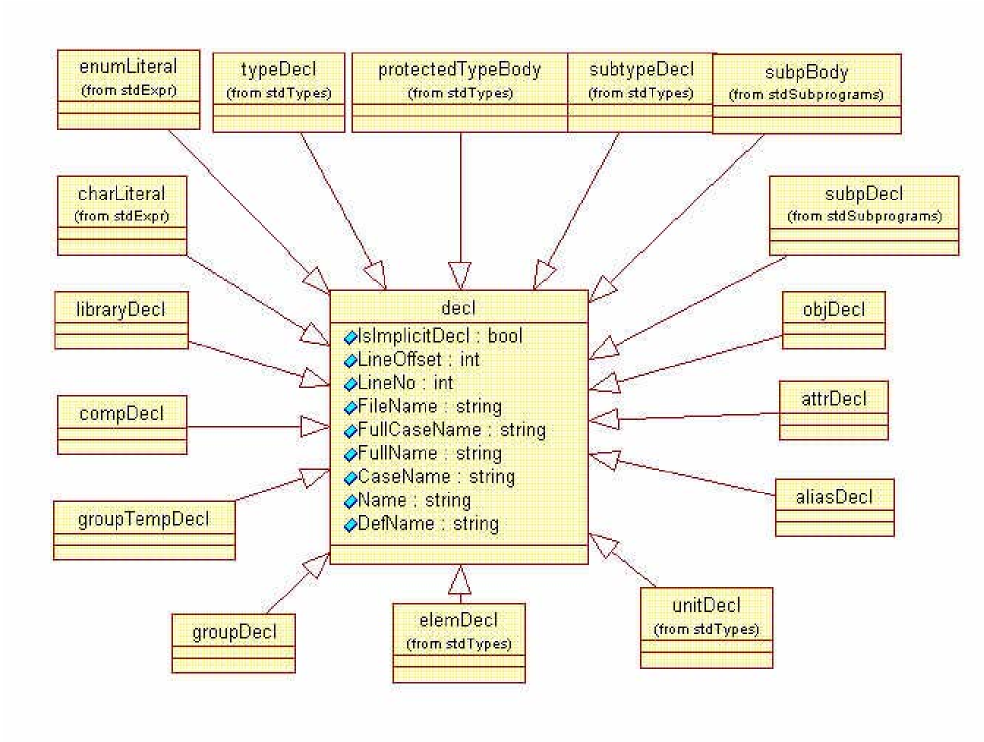
Notes:

8. The iteration vhpiUses returns the external declarations referenced by the design unit.
9. If vhpiIsOpen or vhpiIsDefault is true, then vhpiEntityAspect, vhpiPortAssocs and vhpiGenericAssocs shall return null.
10. The binding indication is obtained from the compConfig by traversing to the entityAspect. An entity aspect can either be a entitydecl, a configdecl or an archBody.
11. vhpiInstNames returns “all”, or “others” or the list of the instance names as it appears in VHDL,

4.5 The standard declaration package (hierarchy and static capability sets)

4.5.1 The declaration class inheritance diagram

The access described by this diagram is part of the static access and post analysis access capabilities.



Notes:

12. Iteration on vhpiDecls does not return implicit declarations.

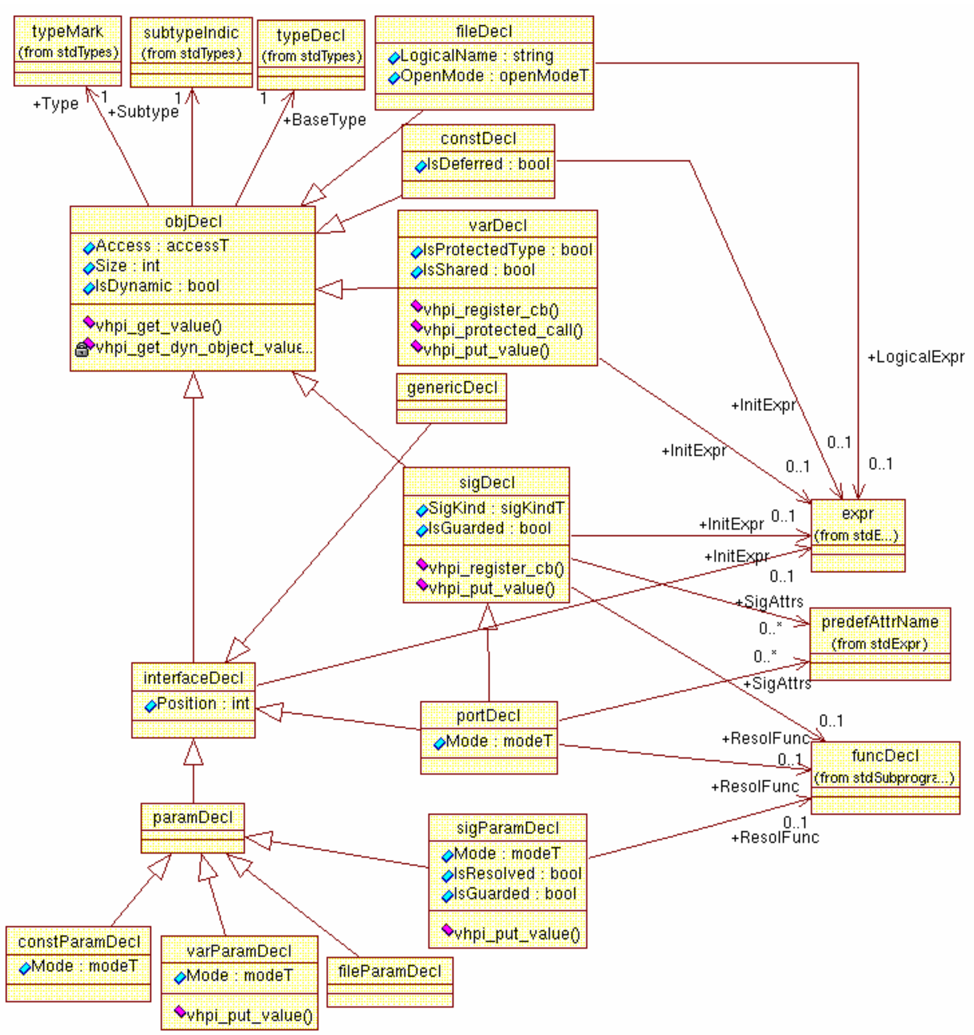
4.5.2 The object class diagram

The hierarchy capability includes the access described by this diagrams with the exception of getting values and modifying object values, registering callbacks and protected calls.
The post analysis capability includes the access described by this diagrams with the exception of getting values and modifying object values, registering callbacks and protected calls.

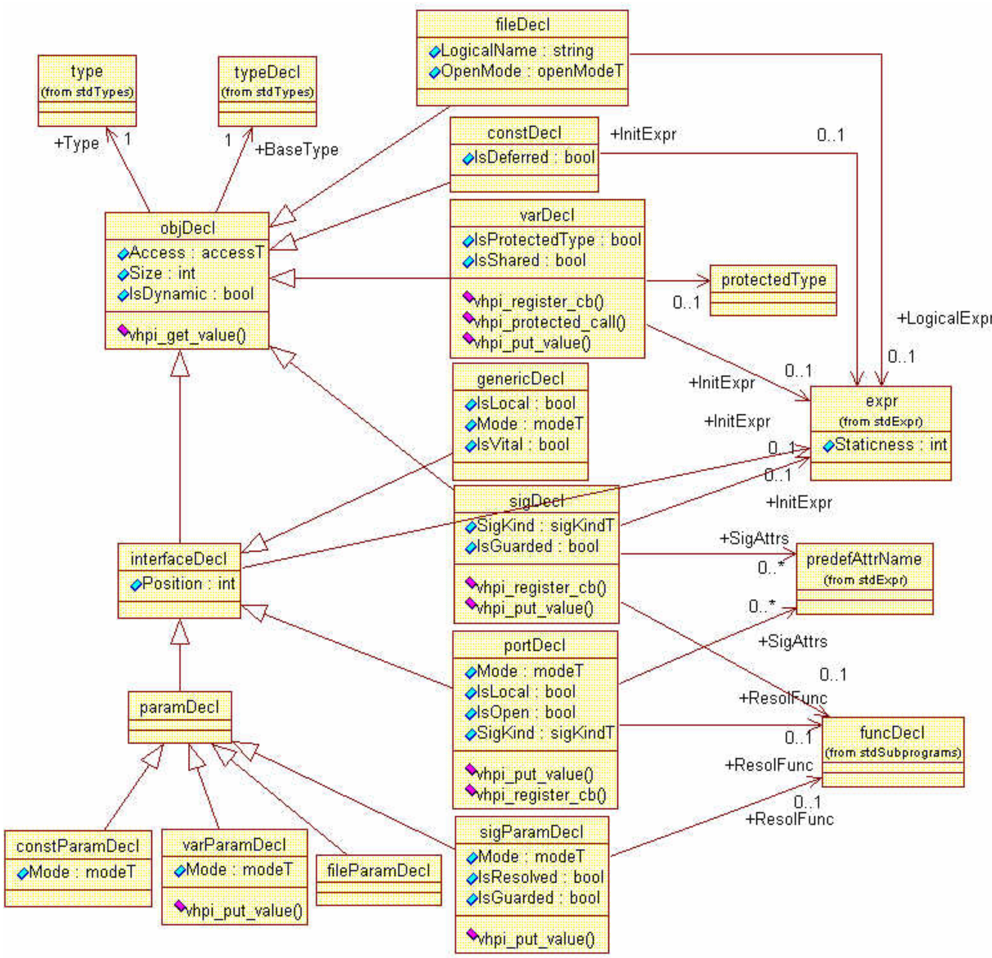
Deprecated:

- One-to-one relationship vhpiSubtype from objDecl to vhpiSubtypeIndicK
  This method is deprecated in favor of the vhpiType one-to-one relationship which provides a more direct and simplified way of obtaining the type of an object. The vhpiSubtypeIndicK kind is deprecated.
- vhpiTypeMark class is renamed vhpiType.
  The type of the object may be an implicit declared type which does not have a type name, hence the renaming of vhpiTypeMark to vhpiType which does not imply that the type has a type name. The rename of the method also provides a recursive way of traversing the type hierarchy.

Deprecated diagram:


typeMark
(from stdTypes)
subtypeIndic
(from stdTypes)
typeDecl
(from stdTypes)
fileDecl
LogicalName : string
OpenMode : openModeT
+Type
+Subtype
+BaseType
1
1
1
constDecl
IsDeferred : bool
objDecl
Access : accessT
Size : int
IsDynamic : bool
vhpi_get_value()
vhpi_get_dyn_object_value...
varDecl
IsProtectedType : bool
IsShared : bool
vhpi_register_cb()
vhpi_protected_call()
vhpi_put_value()
+LogicalExpr
genericDecl
+InitExpr
+InitExpr
0..1
0..1
0..1
sigDecl
SigKind : sigKindT
IsGuarded : bool
vhpi_register_cb()
vhpi_put_value()
+InitExpr
0..1
expr
(from stdE...)
+InitExpr
0..1
+SigAttrs
predefAttrName
(from stdExpr)
0..*
0..*
+SigAttrs
interfaceDecl
Position : int
portDecl
Mode : modeT
0..1
+ResolFunc
0..1
funcDecl
(from stdSubprogra...)
+ResolFunc
paramDecl
sigParamDecl
Mode : modeT
IsResolved : bool
IsGuarded : bool
vhpi_put_value()
0..1
+ResolFunc
constParamDecl
Mode : modeT
varParamDecl
Mode : modeT
vhpi_put_value()
fileParamDecl

New Diagram:



Notes:

13. If a variable is of a protected type (vhpiIsProtectedType property return TRUE), the 1-to-1 relationship vhpiProtectedType shall return the protected type region instance of this variable. The protectedType name is the same as the name of the variable.

14. If the variable is a shared variable of a protected type, access or modification of the values of the local declarations of the variable protected type region shall be done through a vhpi_protected_call.

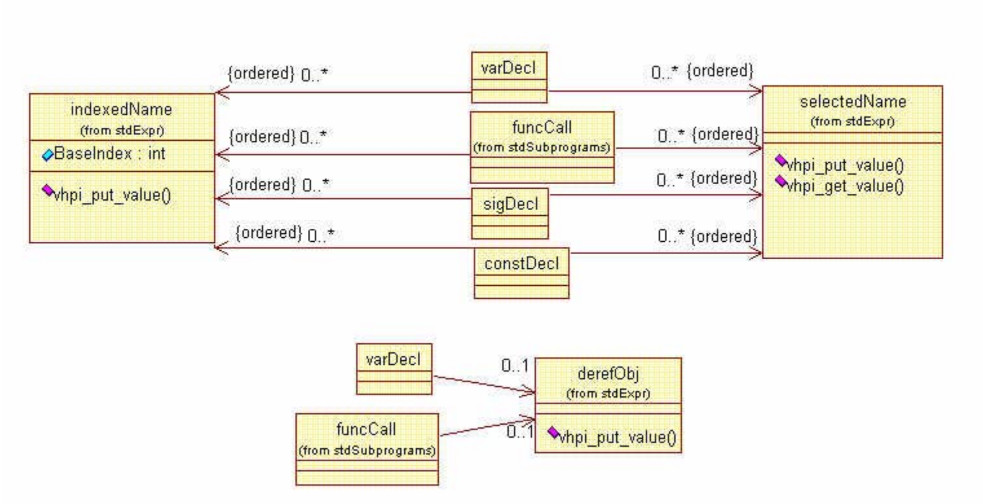
15. vhpi_get_value() can only return the name of the file which has been opened when applied to a filedecl.

16. The property vhpiAccessP can return either vhpiNoAccess or a combination of the bit flags vhpiRead, vhpiWrite, vhiConnectivity. This property can be used to determine any tool restricted access on a given object declaration.

Issue: is this property applicable to a filedecl?

4.5.3 The composite object class diagram

The hierarchy capability includes the access described by this diagrams with the exception of getting values and modifying object values.



Notes:

17. The vhpiDerefObj relationship returns NULL if the variable declaration or function call is not instantiated. deref

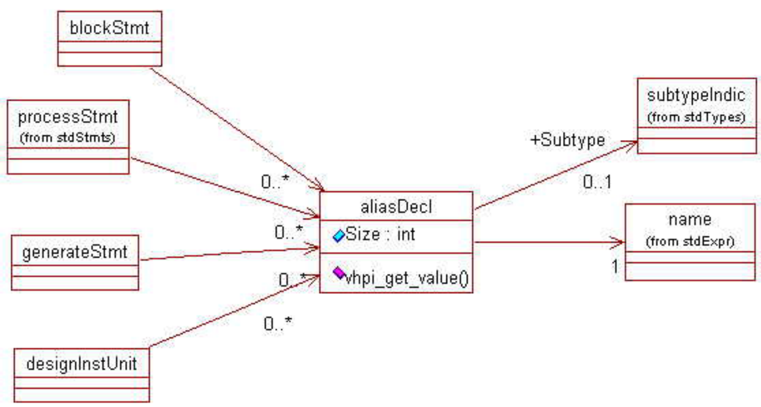
Can I ask for the prefix?If it represents an actual memory location, prefix is undefined, how you get there may have multiple paths?

4.5.4 The alias declaration diagram

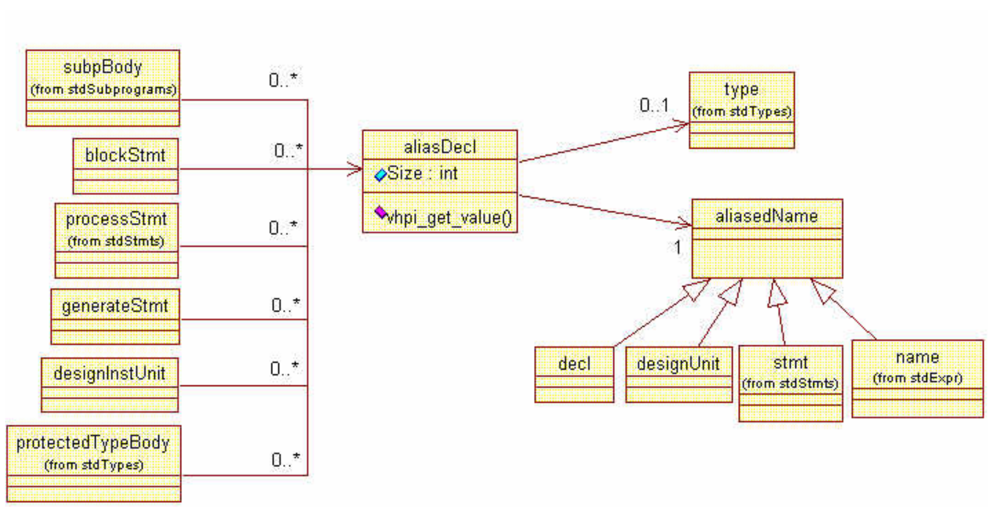
The access described by this diagram is part of the static access capability.
The access described by this diagram is part of the post analysis access capability.
Note: Size property and vhpi_get_value will return a value for a locally static expression, of object aliases.

Note: Iteration on vhpiAliasDecls, attrDecls, BlockStmts, compInstStmts
and sensitivities can only applied to the sub-class of the class
region for which they be possible.

Deprecated diagram with respect to the vhpiSubtype method:



New diagram:

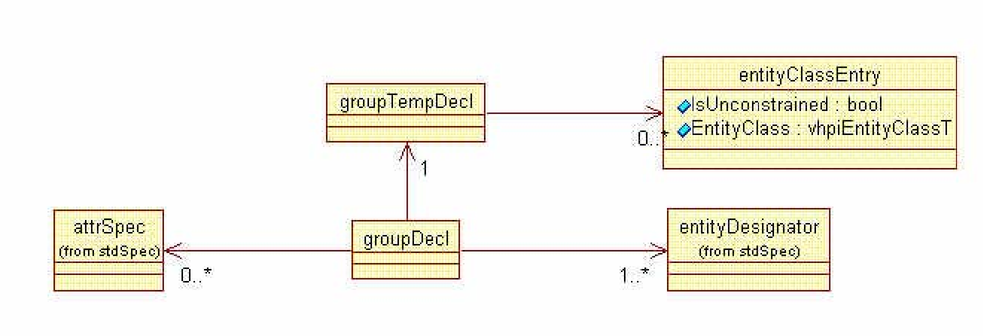


Notes:

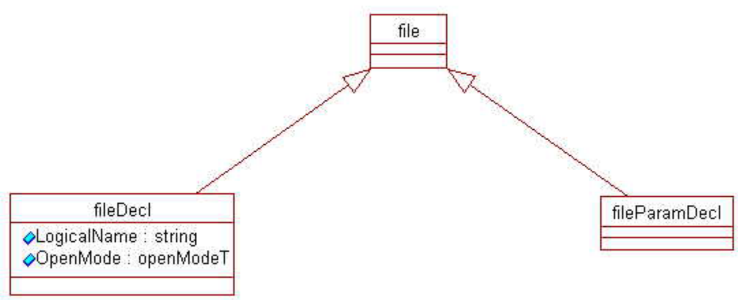
18. vhpi_get_value only applies to alias of objects or of names.
19. vhpiType returns a null handle for non-object aliases..It returns the resulting type of the alias as defined by the rules in the VHDL LRM alias Declaration section.

4.5.5 The group declaration diagram

The access described by this diagram is part of the static access and post analysis capabilities.



4.5.6 The file inheritance diagram



4.6 The standard type package (static access capability set)

4.6.1 The type and subtype class diagram

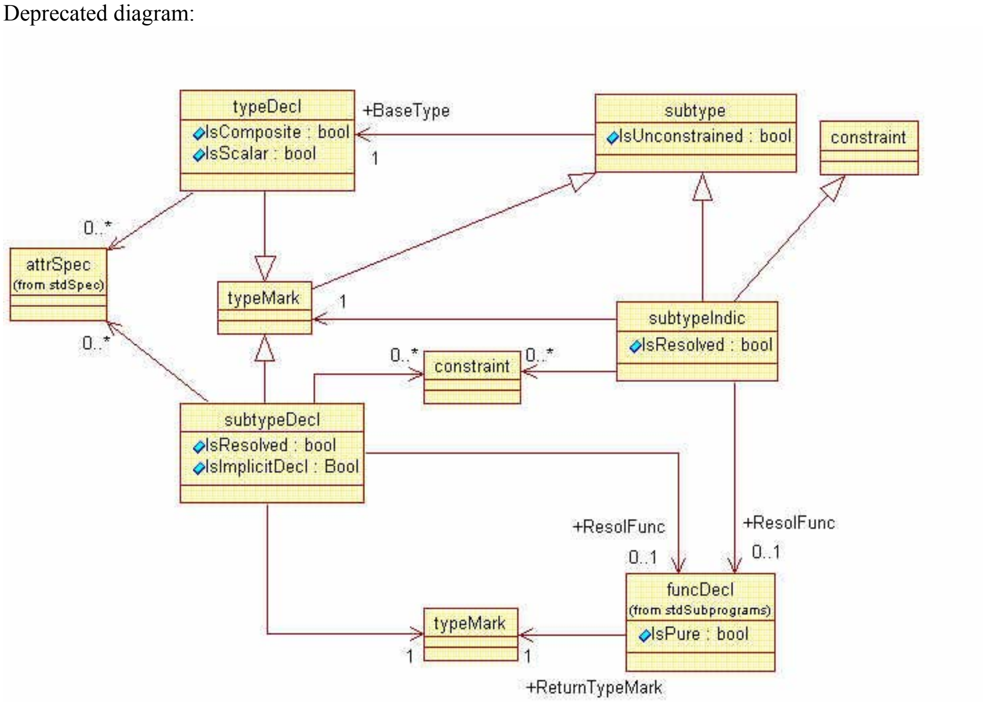
The access described by this diagram is part of the hierarchy access with the exception of access to the attribute specifications. Access to attribute specification is part of the static access.
The access described by this diagram is also part of the post analysis capabilities.
Replace typemark with type, replace retrunTypeMark with ReturnType, remove subtypeIndic. Make type class inherit from constraint

Deprecated:

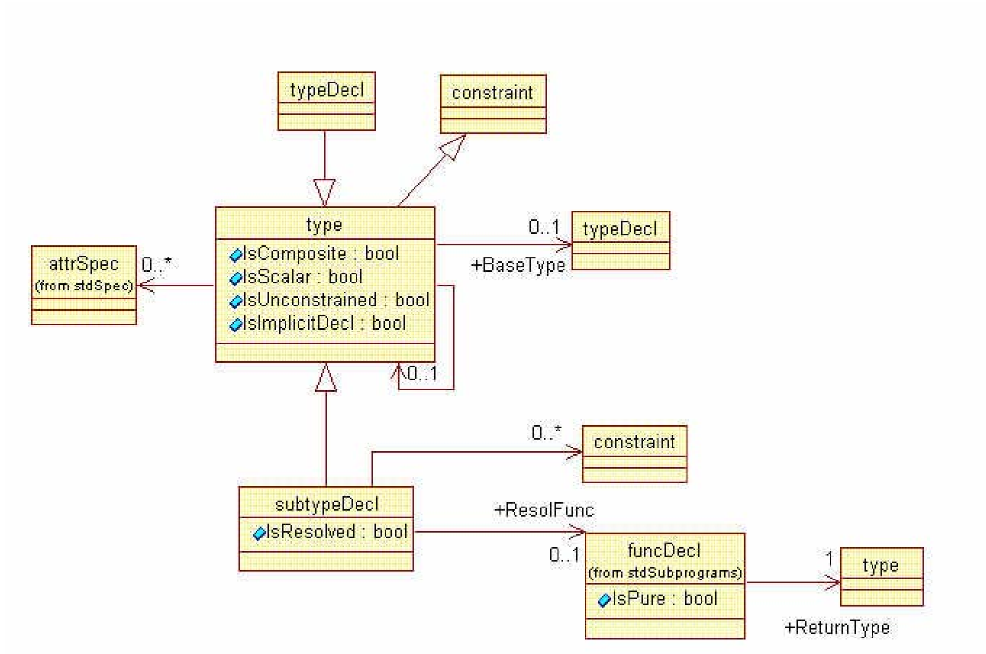
- vhpiTypeMark one-to-one relationship is renamed vhpiType, vhpiTypeMark class is renamed vhpiType class.
- vhpiSubtypeIndicK class is deprecated
- vhpiSubtype class is deprecated
- vhpiIsAnonymous is deprecated

Remove subtype class, move isUnconstrained to the type class. BaseType relationship 0..1 from type to typedecl. BaseType always returns null from a typedecl. vhpiType always returns null from a typedecl.
Move IsComposite to class type. Move isImplicitDecl to type class. Remove isAnonymous
Move isScalar to the type class

Deprecated diagram:



New diagram:



Ask a question to Peter on where to provide types examples in the LRM.

Notes:

20. The base type of a type is not the base type defined by the VHDL LRM but the user defined type.
21. The recursive vhpiType method ends at the base type.
22. The vhpiBaseType of a TypeDecl handle returns null.

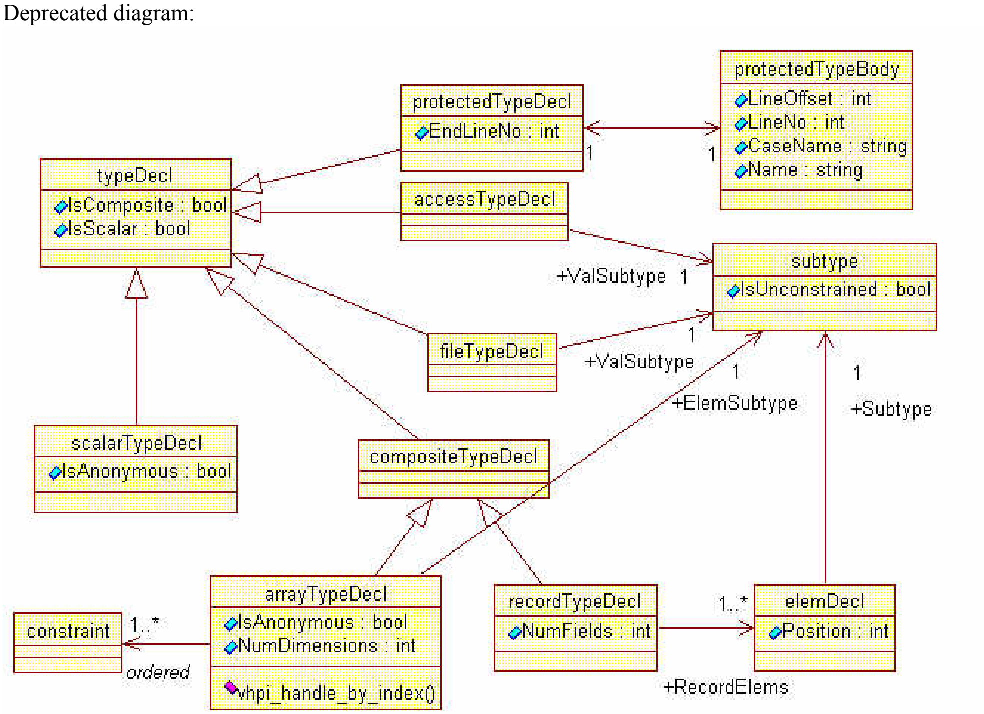
4.6.2 The type inheritance class diagram

The access described by this diagram is part of the hierarchy access and post analysis capabilities.

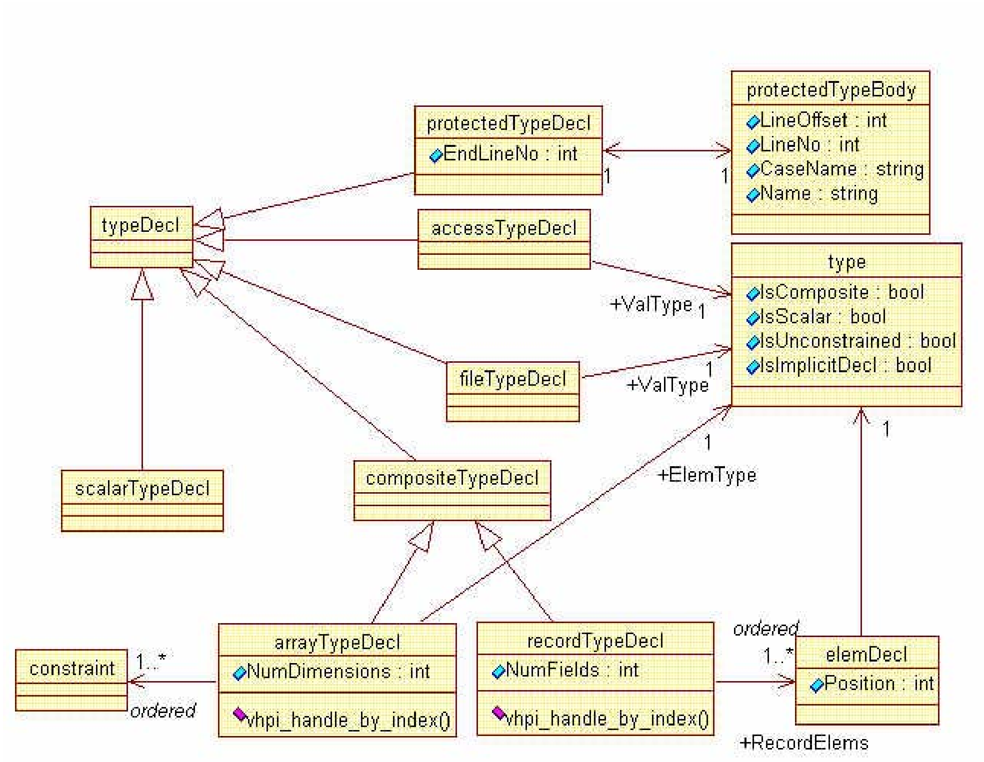
Deprecated:
Replace subtype class with type class replace the tags valsutype with valtype, elemsubtype with elemtype and subtype with type,
Remove isAnonynous

- vhpiSubtype class is deprecated.
- The tagged relationships vhpiValSubtype, vhpiElemSubtype and vhpiSubtype are deprecated.
- vhpiIsAnonymous is deprecated.

Deprecated diagram:



New diagram:

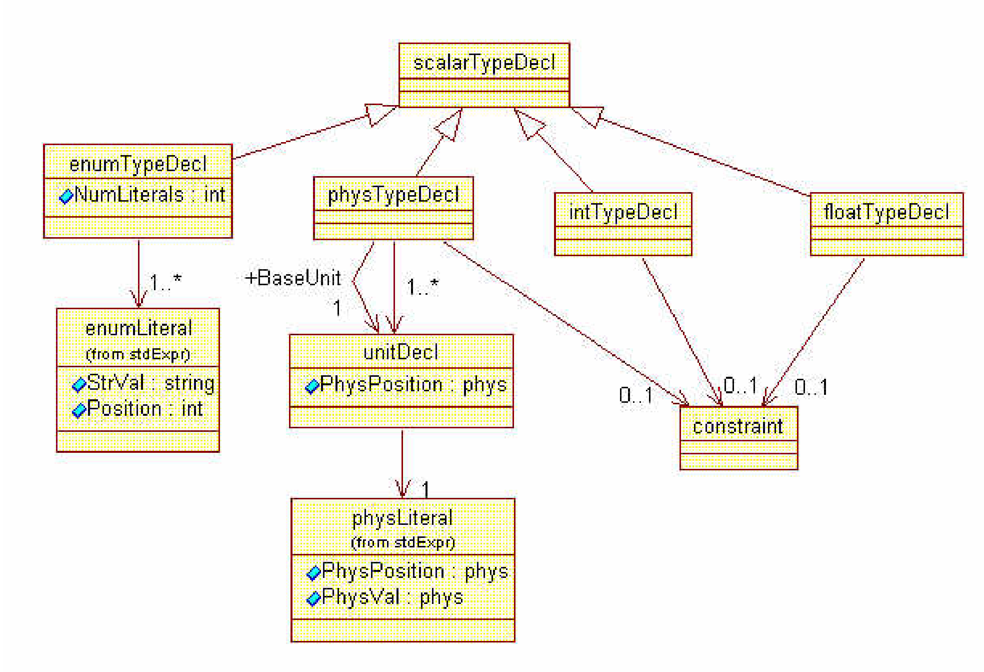


Notes:

23. vhpiBaseType should return the user defined type or the predefined type definition.

4.6.3 The scalar type class diagram

The access described by this diagram is part of the hierarchy access and post analysis capabilities.



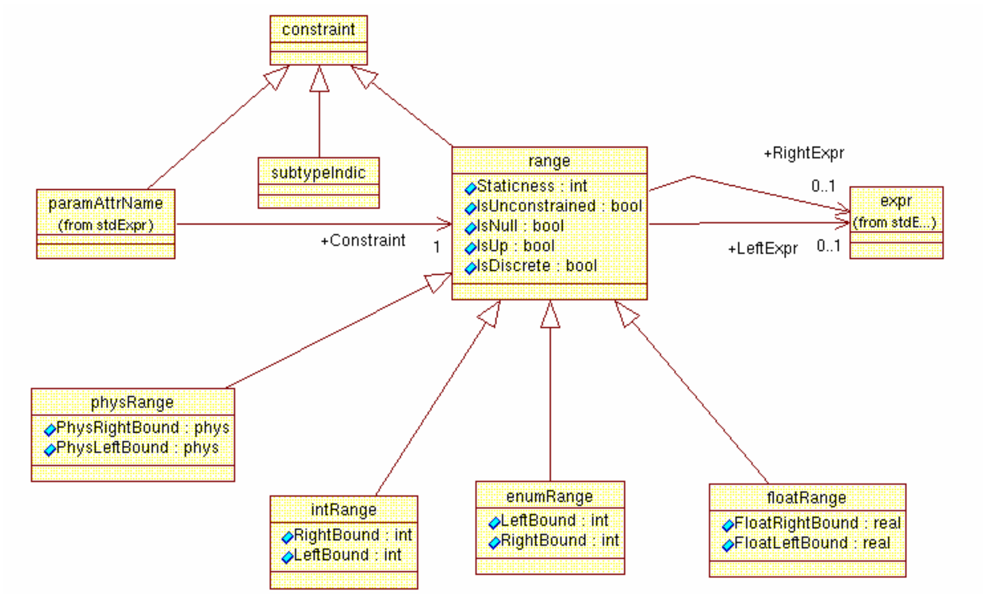
4.6.4 The constraint class diagram

The access described by this diagram is part of the static access (unless it is needed to traverse the subtype to get the value of an object and in that case it should be part of the hierarchy capability)
The access described by this diagram is also part of the post analysis capabilities.

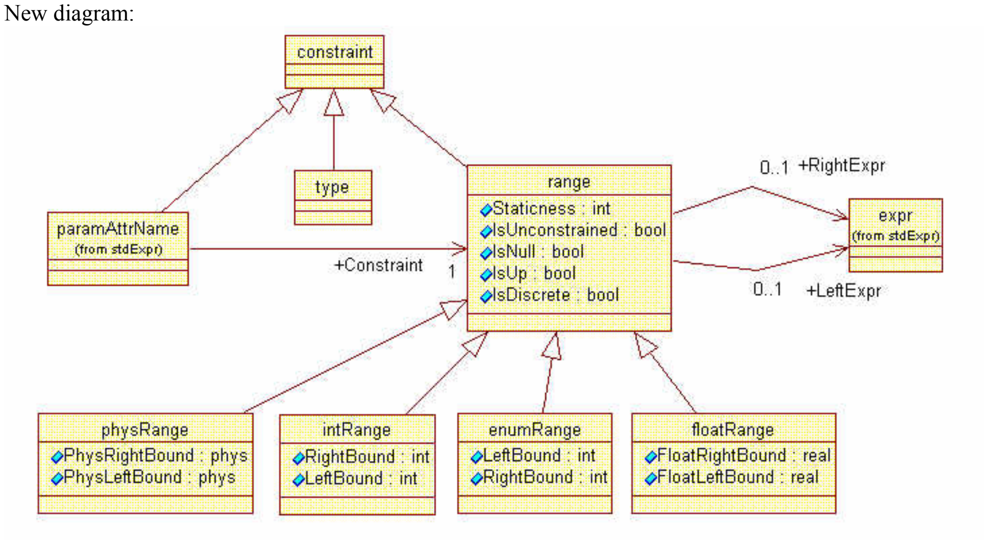
Deprecated:

- vhpiSubtypeIndicK class is deprecated.

Deprecated diagram:



New diagram:



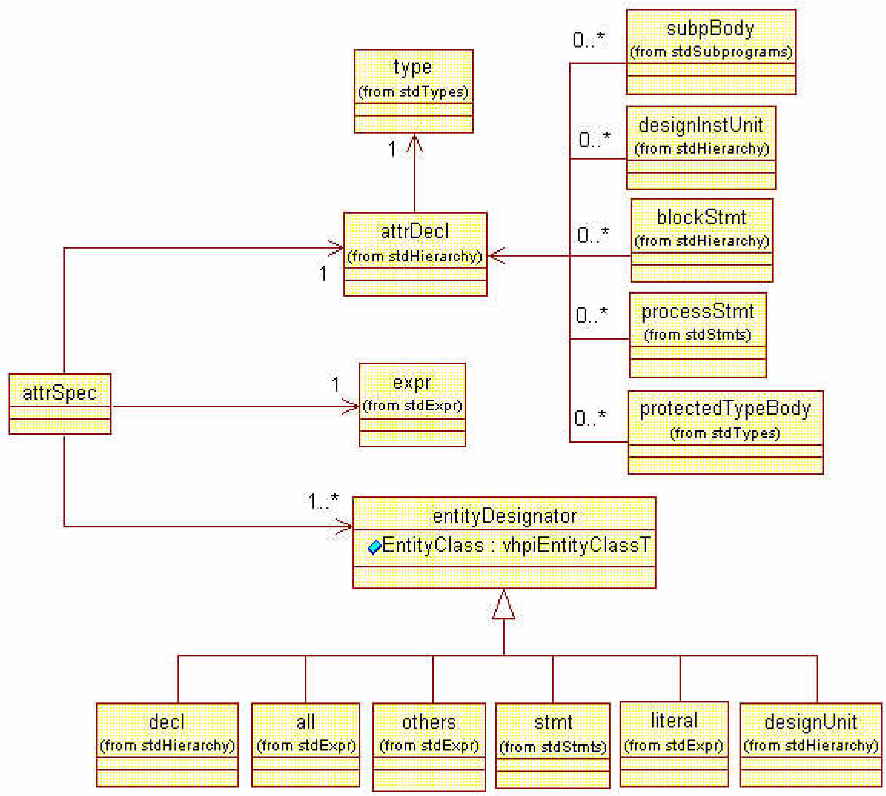
Notes:

24. vhpiConstraint from a paramAttrName returns the range represented by the attribute ‘range or ‘reverse_range as described by the VHDL LRM.

4.7 The standard specification package (static access capability set)

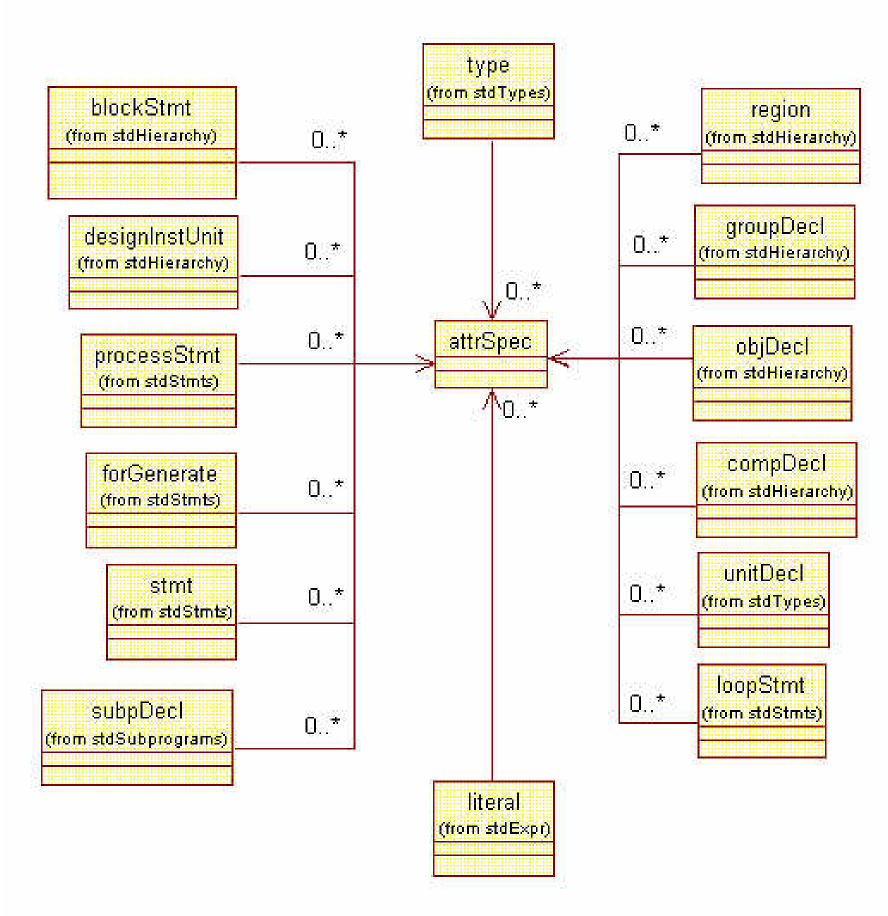
4.7.1 The attribute declaration and specification class diagram

The access described by this diagram is part of the static access and post analysis capabilities.



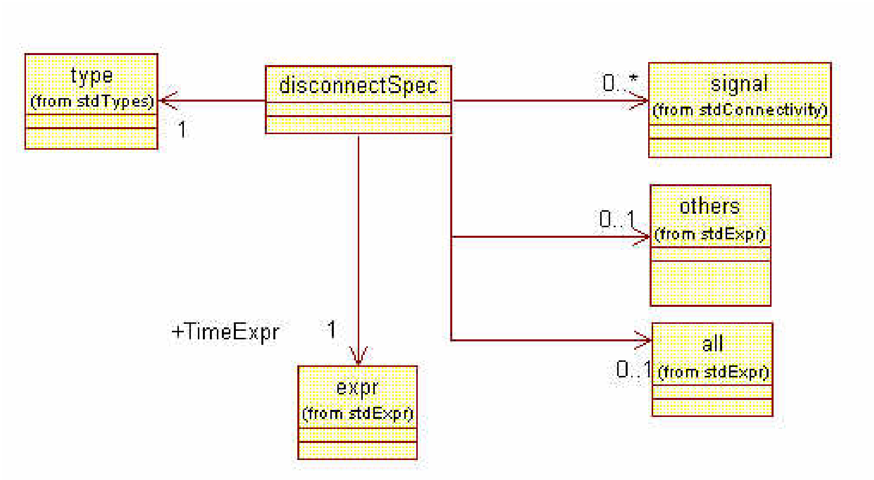
4.7.2 The attribute specification associations

The access described by this diagram is part of the static access and post analysis capabilities.



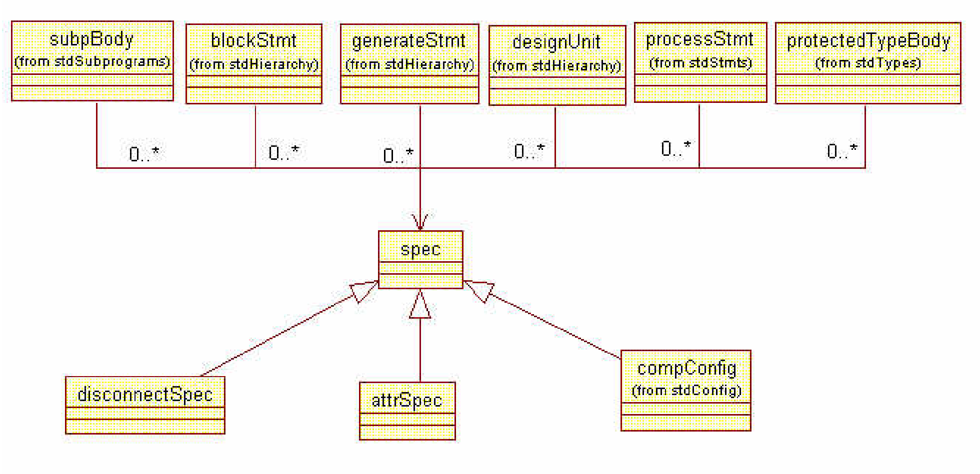
4.7.3 The disconnection specification class diagram

The access described by this diagram is part of the static access and post analysis capabilities.



4.7.4 The specifications diagram

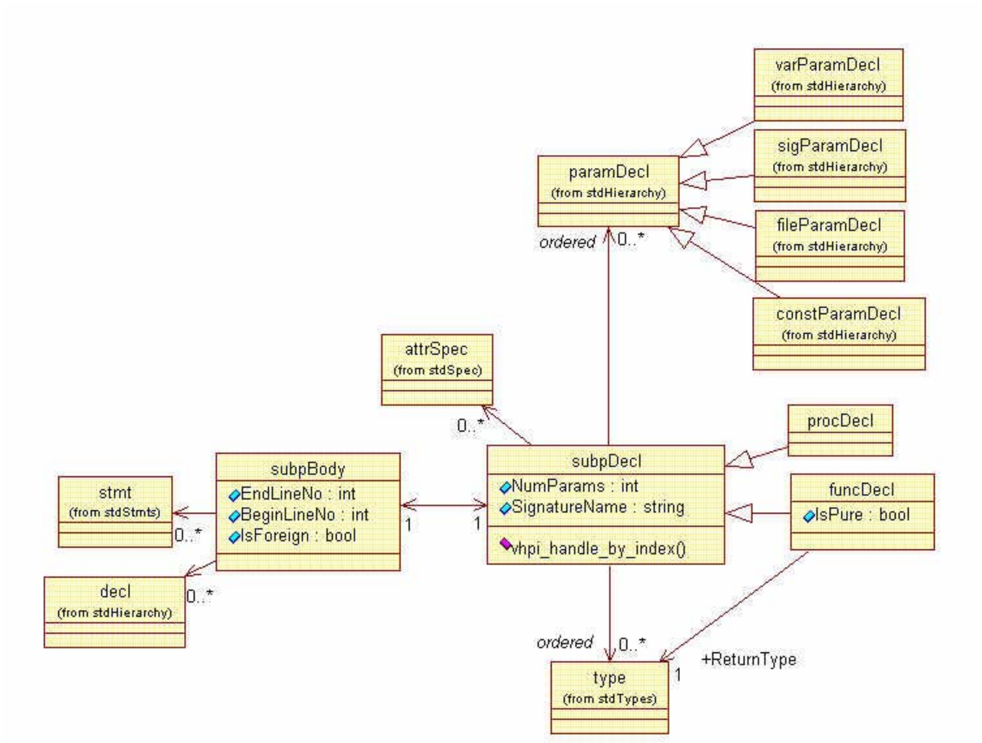
The access described by this diagram is part of the static access and post analysis capabilities.



4.8 The standard subprogram package (static access and dynamic elab capability sets)

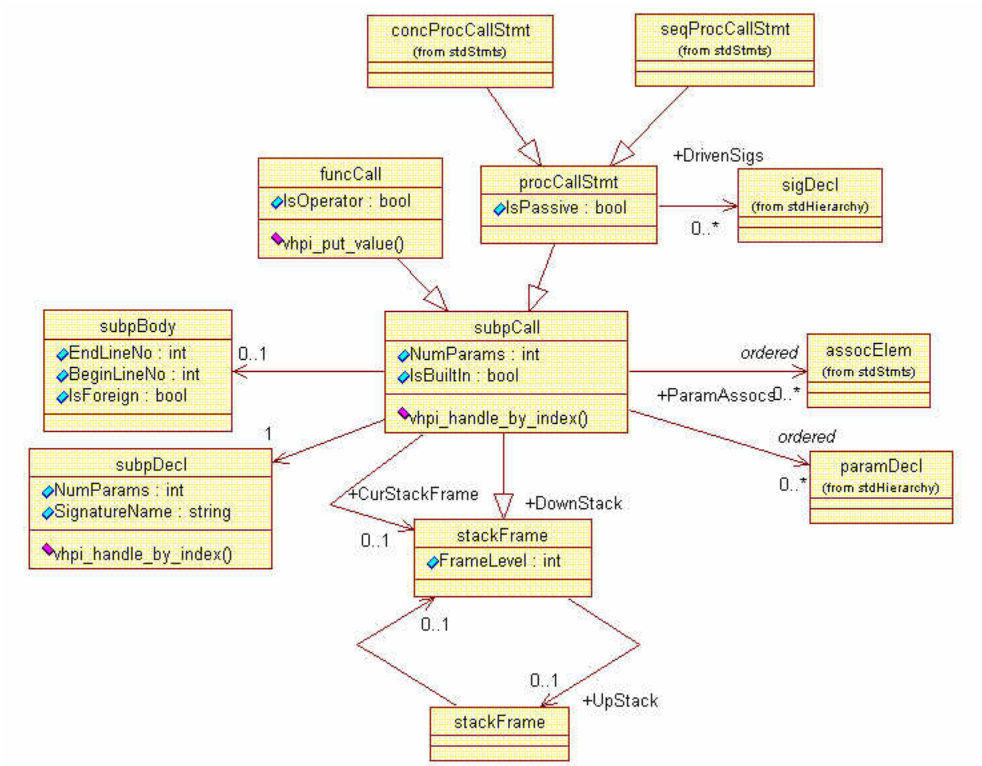
4.8.1 The subprogram declaration class diagram

The access described by this diagram is part of the static access and post analysis capabilities. The access described by this diagram is part of the basic foreign model capability. However if the subpbody is a foreign model, iteration on statement, declarations and attribute declarations shall return NULL.



4.8.2 The subprogram call class diagram

The access described by this diagram is part of the basic foreign model capability.
The access described by this diagram is part of the dynamic elaboration access capability except for vhpi_put_value on function calls.
The access described by this diagram is part of the static access capability with the exception of access to and from stackFrames, vhpi_put_value of function calls and iteration on driven signals.
Iteration on driven signals is part of the connectivity capability.



Notes:

25. Iteration on paramDecls from an instantiated subpCall (active) returns the formal parameters to which the associations of the actuals are in effect
26. A subpCall is only a stackframe if the subpcall is instantiated

27. Handles obtained from a stack frame which represents a protected type method of a shared variable do not require a vhpi_protected_call as the lock has already been obtained by the subprogram executing.

28. The property vhpiIsOperatorP is TRUE if a function call denotes an operator call. There are 3 categories of operators: predefined operators from the TEXTIO or STD package, built-in or accelerated operators for example operators defined in the stdlogic or vital packages, user-defined operators.
29. When applied to a subpCall handle, the property vhpiIsBuiltInP returns TRUE if the subpboday is unavailable and has a private implementation. If the property return FALSE and the subpCall is a

Deleted: ¶

Formatted: Bullets and Numbering

funcCall for which the property vhpiIsOperatorP is TRUE, the function call denotes a user defined operator call.

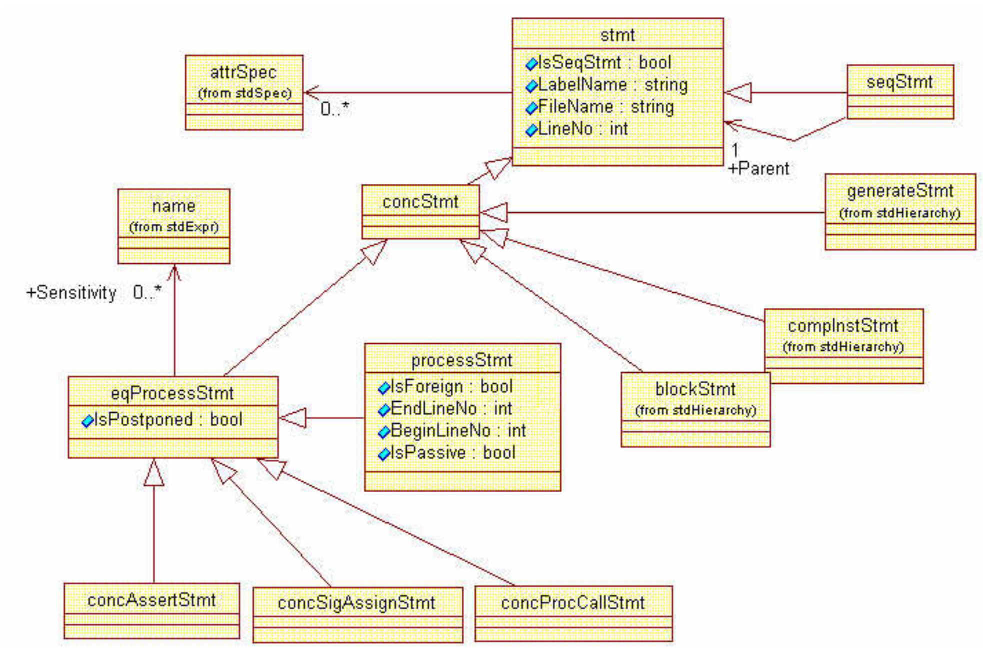
4.9 The standard statement package (static access capability set)

4.9.1 The concurrent statement class diagram

The access described by this diagram is part of the static access capability.
Note: The iteration on sensitivity only applies to concurrent statements : concurrent assert statement, concurrent signal assignment statement and concurrent procedure call statements.
Refactoring the eqProcessStmt with assertion/sigassign/procallstmt
Or add 6 classes concassert, concsigassign…
Note:

30. Explain what is the sensitivity (explicit or as inferred by the LRM). Union of the wait sensitivity?
    Static sensitivity: list of the signals on the explicit sensitivity list. Return null for the processes having wait stmts.

Formatted: Bullets and Numbering



4.9.2 The structural statement class diagram

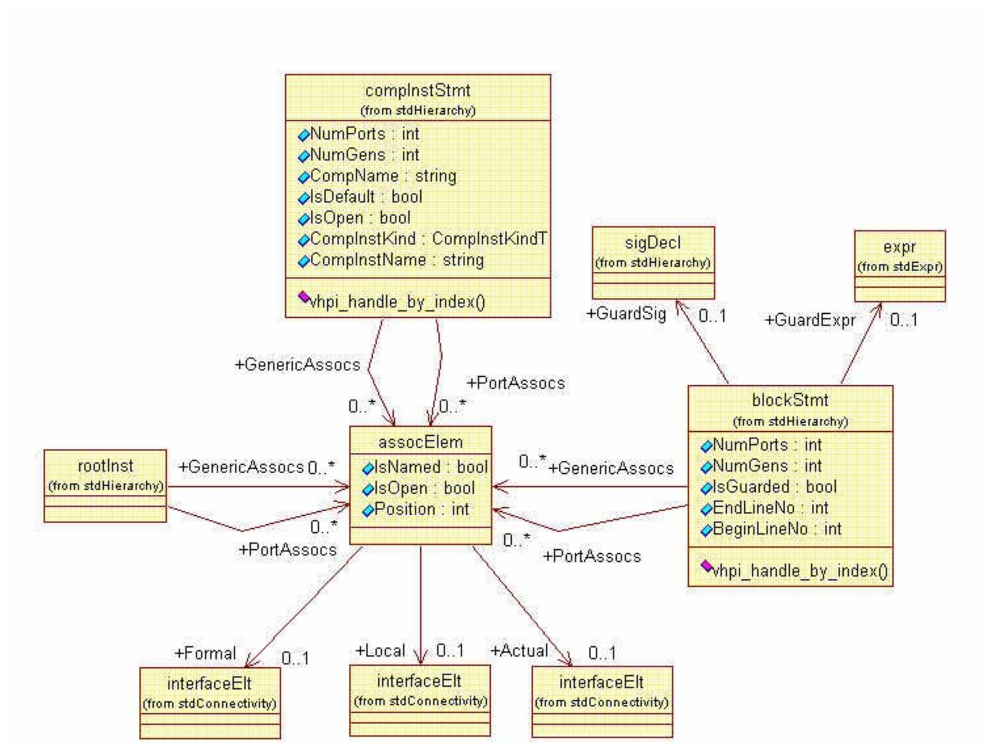
The access described by this diagram is part of the static access capability except with the exception of accessing the formal of an association element which is part of the connectivity capability.

The access described by this diagram is part of the post analysis capability with the following exception: Access to the formal of an association element is part of the connectivity capability.

Notes:

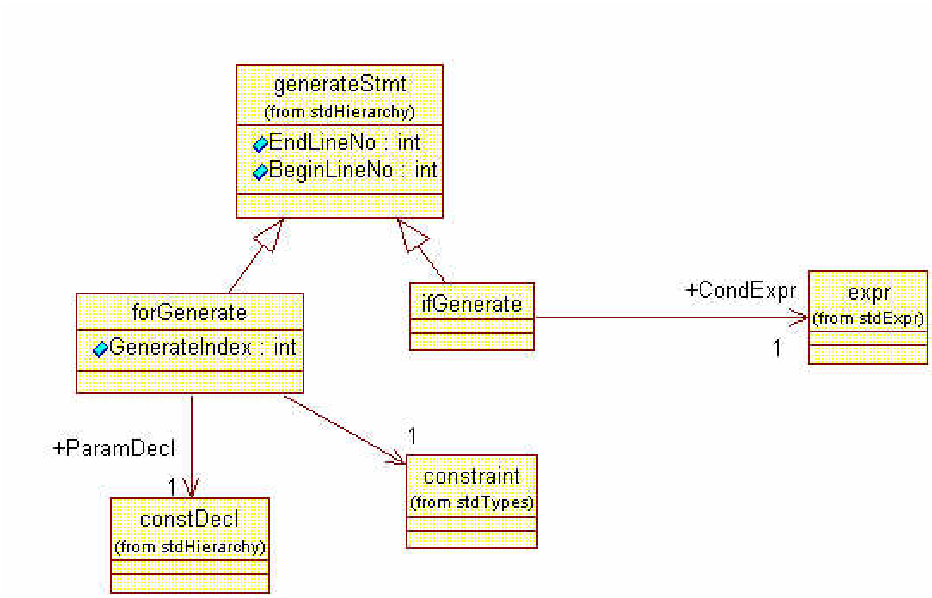
31. If vhpiCompName returns null then the component instance statement is a direct instantiation, otherwise if IsDefault is true, there is no component configuration, and the name of the bound entity is the same as the name of the component, else if isDefault is false, there is either a component configuration in the architecture containing this component instance or in a configuration declaration.
32. The position of an assocElem is the position of the formal in the interface list.
33. Formal is only accessible from an association element obtained from a direct component instance statement. Local is only accessible from an association element obtained from a non direct component instance statement.

Formatted: Bullets and Numbering



4.9.3 The generate statement class diagram

The access described by this diagram is part of the static access and post analysis capability.



Notes:

34. In the instantiated domain, if generate with condition expression evaluating to FALSE are not returned as handle. In the uninstantiated domain, all generate statements are returned. For example, iteration on stmts would return if generate with false condition.
35. In the instantiated domain a forGenerate handle is returned for each index. In the uninstantiated domain, a single forGenerate is returned.
36. The vhpiCondExpr relationship returns NULL if the ifGenerate is instantiated (vhpiIsUninstantiatedP property is FALSE) and an error is created. If the ifGenerate is uninstantiated, vhpiCondExpr returns a expression handle.
37. The vhpiGenerateIndexP property returns the generate index value if the forGenerate is instantiated; the property returns -1 and a error is created if the forGenerate is uninstantiated.

Formatted: Bullets and Numbering

38. The relationships vhpiParamDecl and vhpiConstraint are only available from an uninstantiated forGenerate handle.

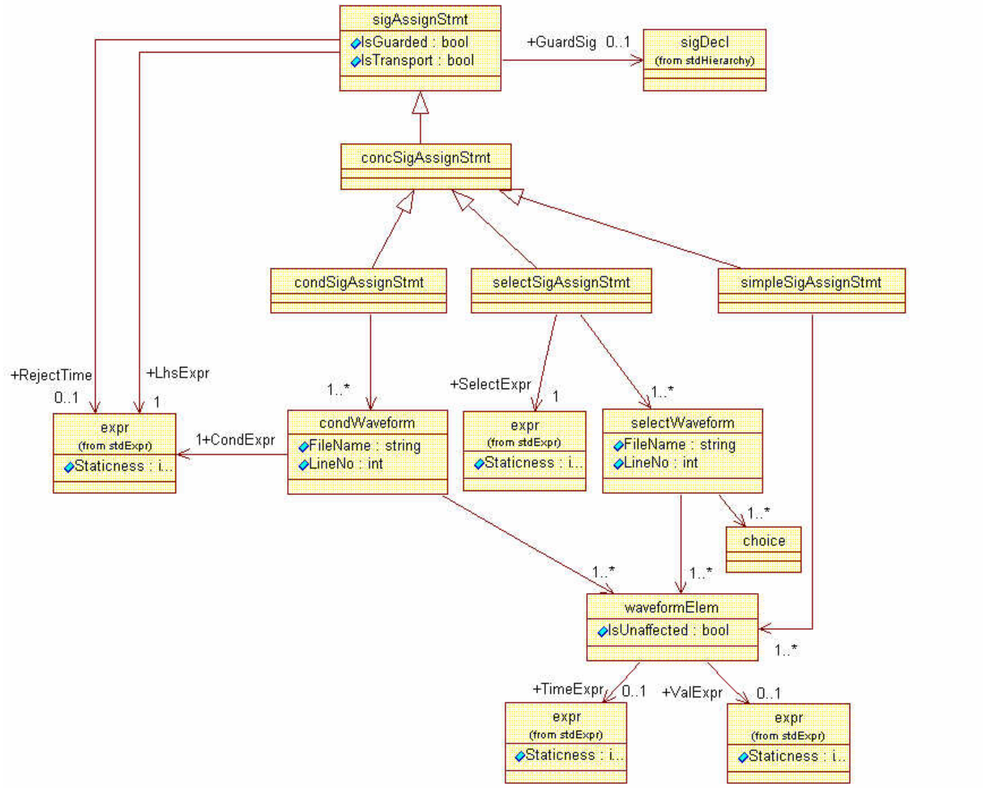
Issue uninstantiated/instantiated representation: decide if an error is created and if the relation ship should say 0..1

4.9.4 The concurrent signal assignments class diagram

The access described is part of the static access and post analysis capabilities.

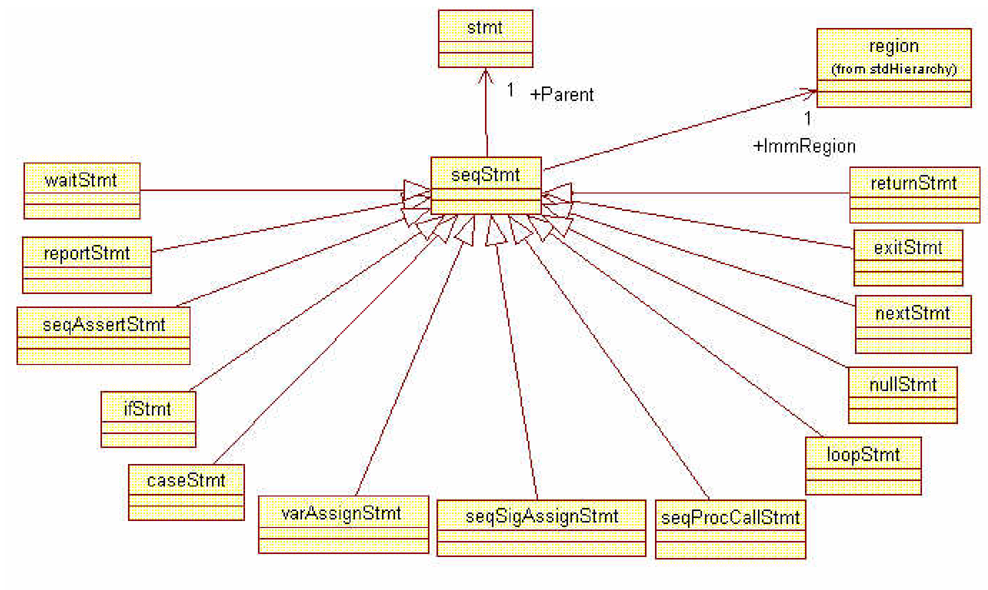
Deleted: capability.

Deleted: The access described by this diagram is part of the post analysis access capability with the exception of accessing the region from a sequential statement.¶



4.9.5 The sequential statement inheritance class diagram

The access described by this diagram is part of the static access capability.
The access described by this diagram is part of the post analysis access capability with the exception of accessing the region from a sequential statement.


stmt
region
(from stdHierarchy)
1
+Parent
1
+ImmRegion
seqStmt
waitStmt
reportStmt
seqAssertStmt
ifStmt
caseStmt
varAssignStmt
seqSigAssignStmt
seqProcCallStmt
loopStmt
nullStmt
nextStmt
exitStmt
returnStmt

1

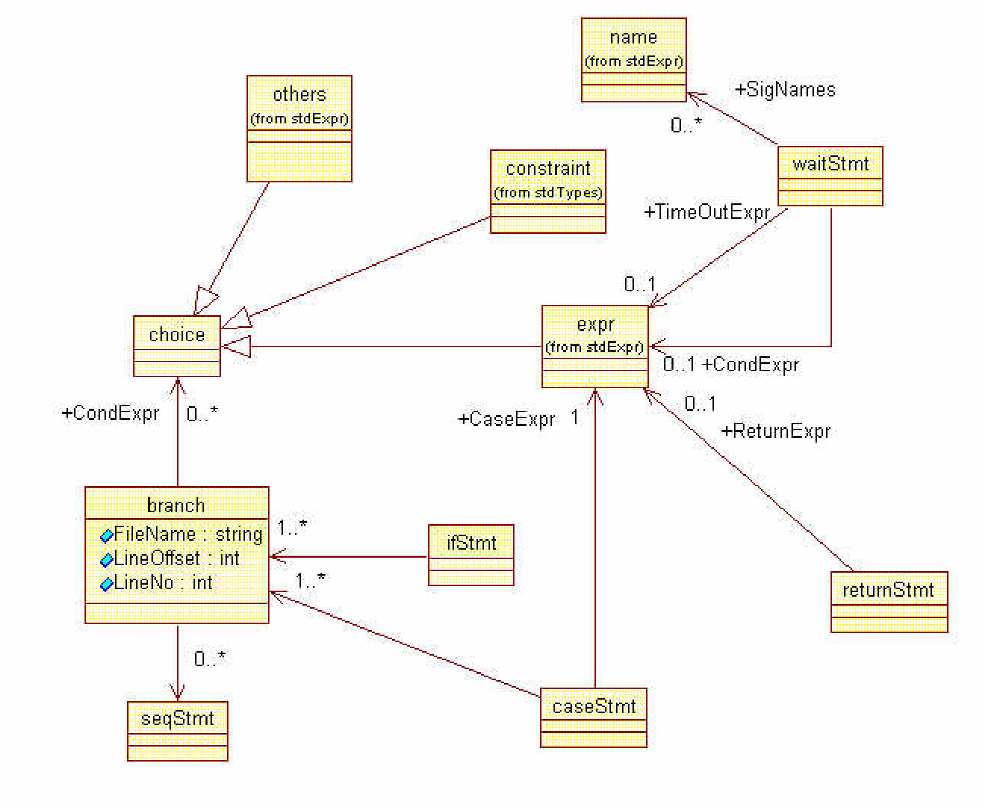
4.9.6 The sequential case, if, wait and return statement class diagram

The access described by this diagram is part of the static access and post analysis capabilities.

Notes:

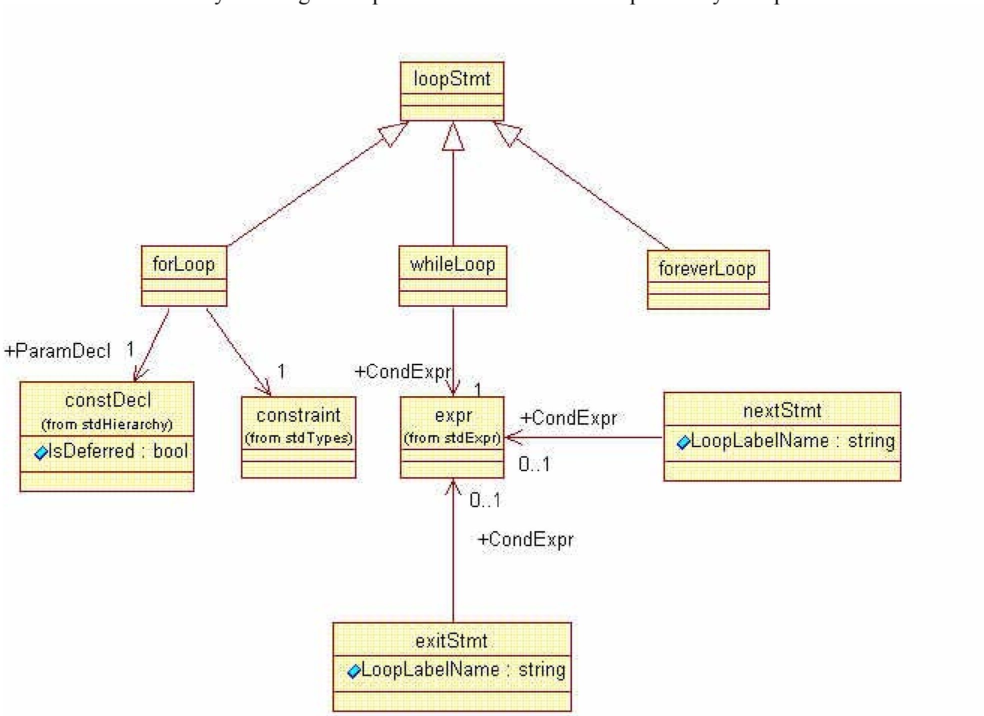
39. how branches are returned for an ifStmt, the iteration on branch from the ifStmt class will return the branch of the if, then the branch of the elsif (if any) then the branch of the else (if any).
40. condExpr iteration from a branch of an ifStmt returns only one choice, condExpr from a caseStmt may return multiple choices.

Formatted: Bullets and Numbering



4.9.7 The sequential loop, exit and next statement diagram

The access described by this diagram is part of the static access and post analysis capabilities.



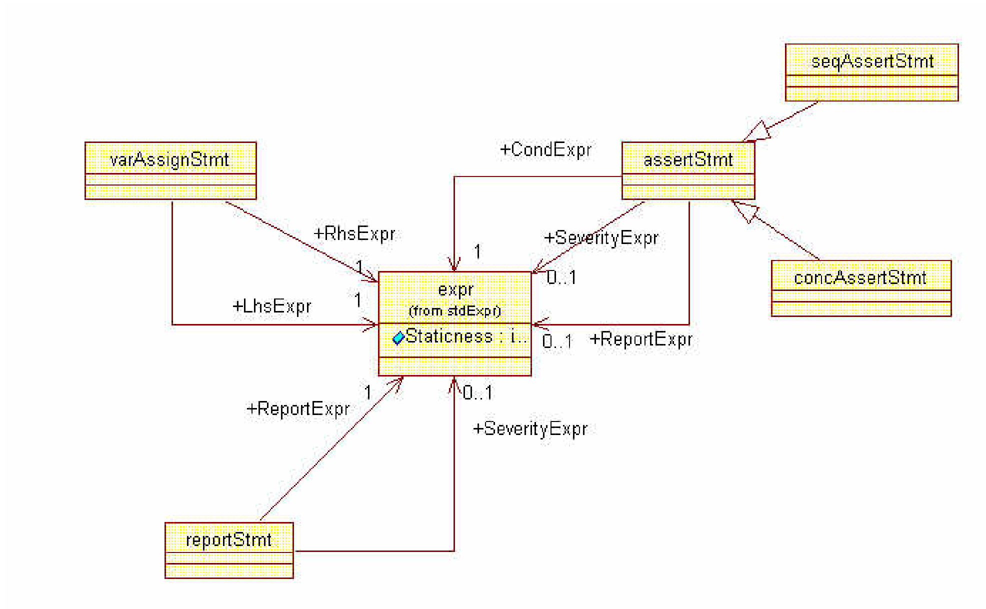
Note:

41. LoopLabelNameP property shall return if there is no loop label name provided in the next statement.

Formatted: Bullets and Numbering

4.9.8 The sequential variable assignment, assert and report statement diagram

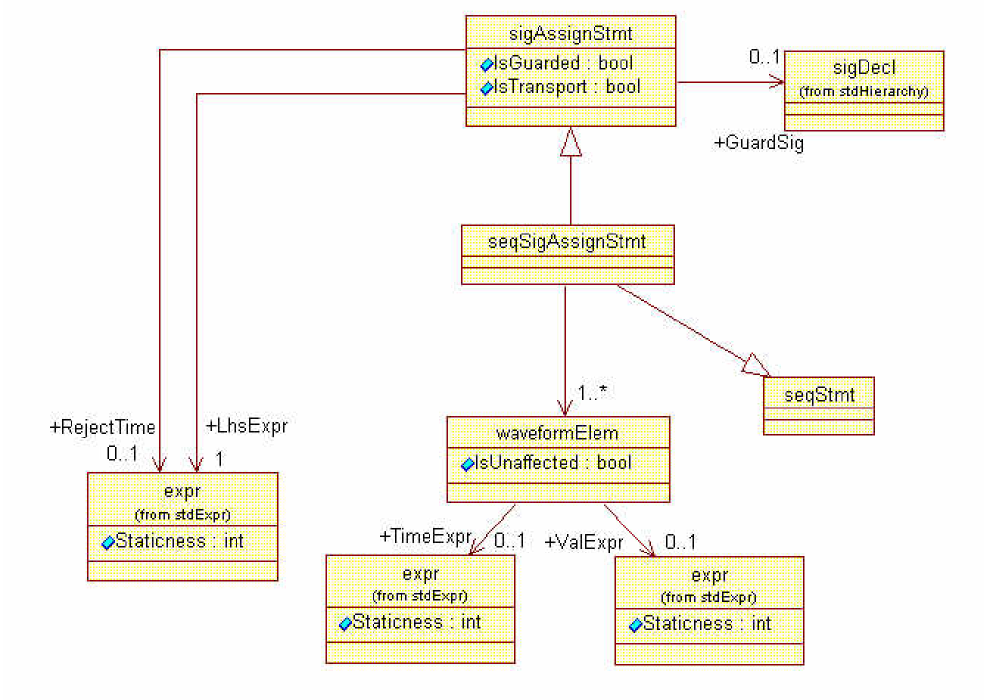
The access described by this diagram is part of the static access and post analysis capabilities.


seqAssertStmt
varAssignStmt
+CondExpr
assertStmt
+RhsExpr
+SeverityExpr
1
1
0..1
concAssertStmt
expr
(from stdExpr)
+LhsExpr
1
Staticness : i...
0..1
+ReportExpr
+ReportExpr
1
0..1
+SeverityExpr
reportStmt

1

4.9.9 The signal assignment statement class diagram

The access described by this diagram is part of the static access and post analysis capabilities



Notes:

42. If the property vhpiIsUnaffected is TRUE, then vhpiValExpr and vhpiTimeExpr relationships shall return a null handle.

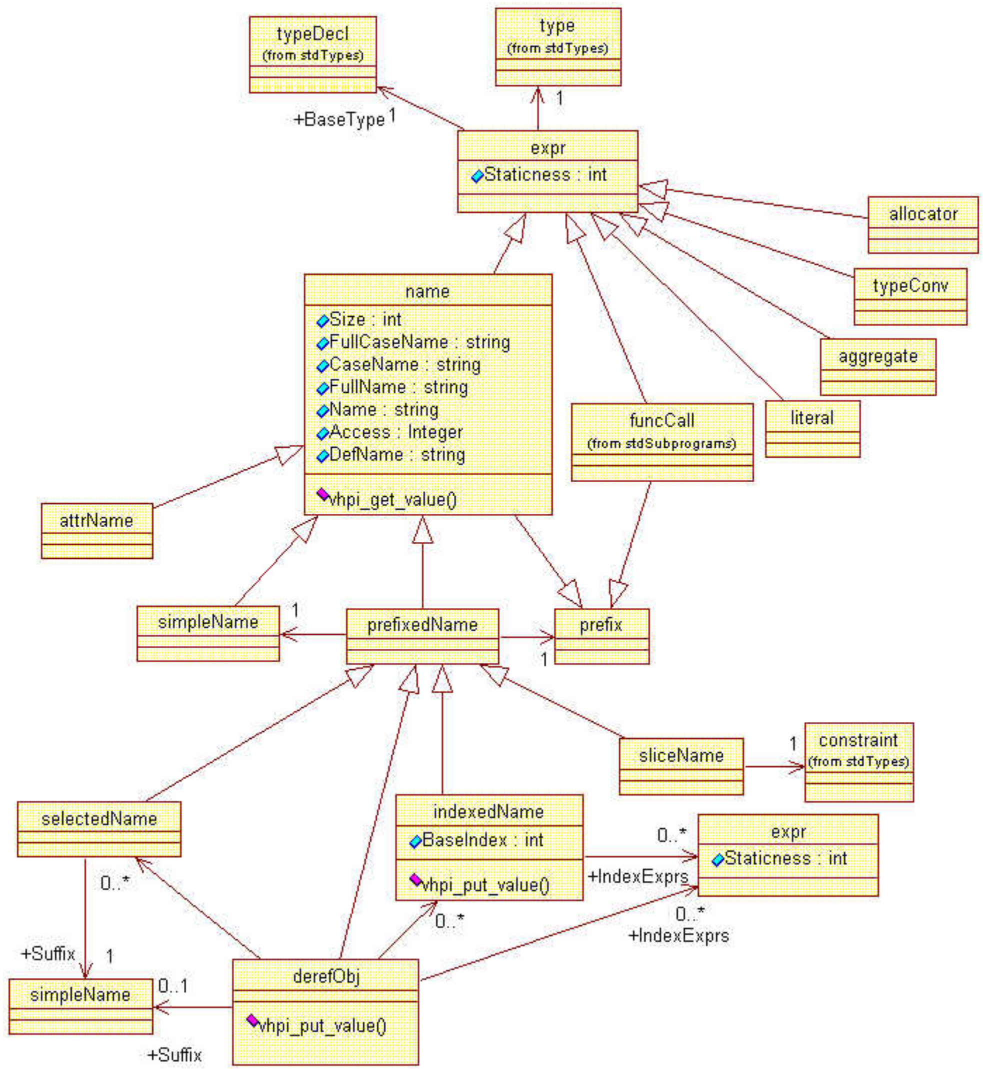
Formatted: Bullets and Numbering

43. The VHDL expression was the null transaction if the vhpiValExpr return a null handle and the property vhpiIsUnaffected is FALSE.

4.10 The standard expression package

4.10.1 The expression inheritance diagram

The access described by this diagram is part of the static access and post analysis capabilities except for vhpi_put_value which is part of the debug and runtime and basic foreign capabilities.
vhpi_get_value and vhpi_put_value of dereference objects is only part of the debug and runtime capabilities.
Issue: need to add access to indexedNames and selectedNames from a derefObj, do they return a derefObj?



Notes:

44. The iterations vhpiSelectedNames andvhpiIndexedNames are only available when the vhpiDerefObjK is instantiated. The relationships vhpiSuffix, vhpiIndexExprs and vhpiPrefix are only available when the vhpiDerefObjK is uninstantiated. vhpi_put_value() and vhpi_get_value() are only available when the vhpiDerefObj is instantiated.

Formatted: Bullets and Numbering

Issue: should we say return NULL if instantiated? Do we generate an error or warning?

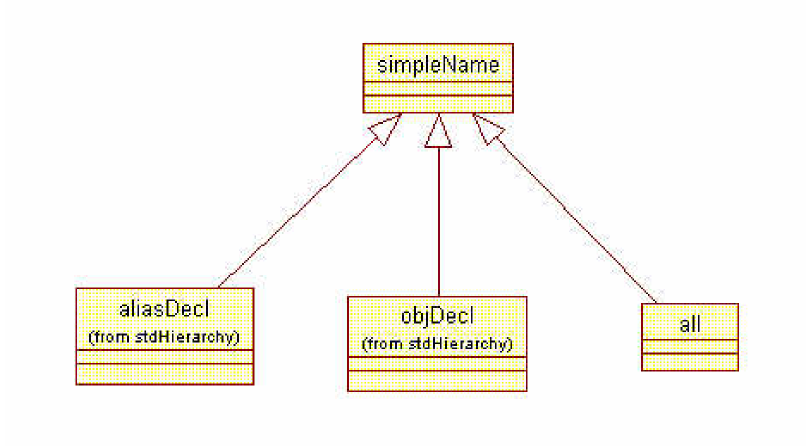
45. The vhpiSuffix relationship return the simpleName vhpiAllK when accessing the ".all" suffix of a vhpiDerefObjK.

Formatted: Bullets and Numbering

46. The vhpiSelectedNameK class represents only selected names of a record as all other VHDL selected names (package items selected names) have been resolved to their reference.

4.10.2 The simple name class diagram

The access described by this diagram is part of the static access and post analysis capabilities.



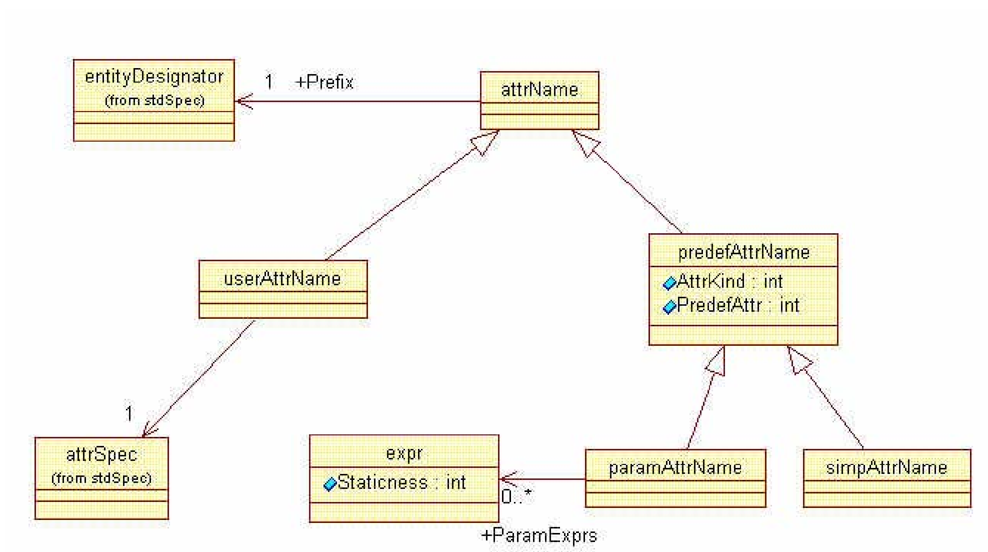
Note:

47. A simpleName is the vhpiAllK class when denoting the all suffix of a uninstantiated vhpiDerefObjK.

Formatted: Bullets and Numbering

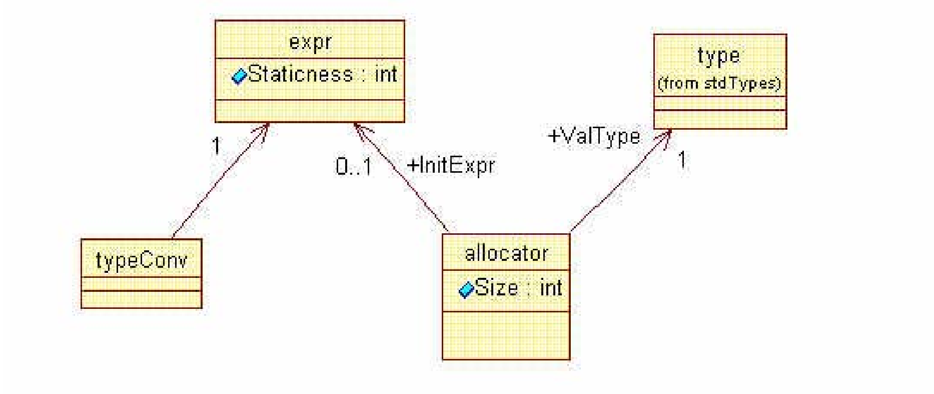
4.10.3 The attribute class diagram

The access described by this diagram is part of the static access and post analysis capabilities.



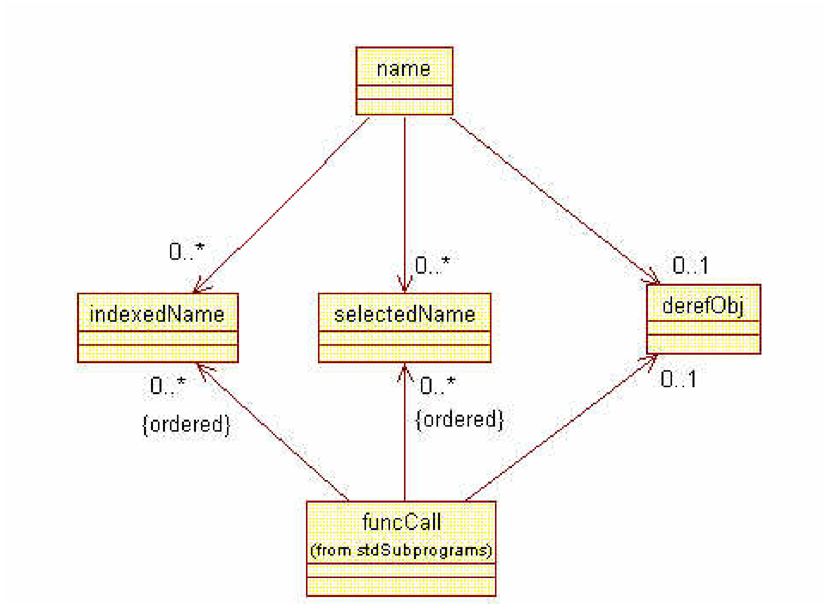
4.10.4 The type conversion, aggregate class diagram

The access described by these diagrams is part of the static access capability.
The access described by these diagrams is part of the post analysis access except for iteration on indexednames, selectednames which is part of the static access capability.
Access to the derefObj is only part of the debug and runtime capability.



4.10.5 The name access class diagram

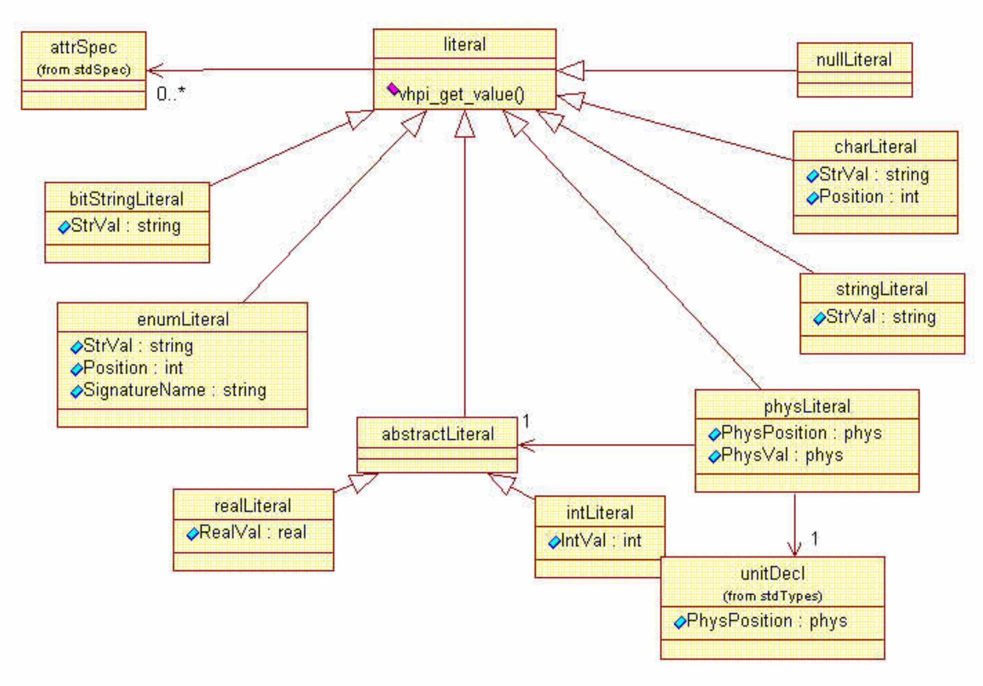
The access described by these diagrams is part of the static access capability.
The access described by these diagrams is part of the post analysis access except for iteration on indexednames, selectednames which is part of the static access capability.
Access to the derefObj is only part of the debug and runtime capability.
Access to indexedNames, selectedName and derefObj is part of the advanced foreign model capabilities.


name
0..*
0..*
0..1
indexedName
selectedName
derefObj
0..*
{ordered}
0..*
{ordered}
0..1
funcCall
(from stdSubprograms)

1

4.10.6 The literal class diagram

The access described by this diagram is part of the static access and post analysis capabilities.



Note:

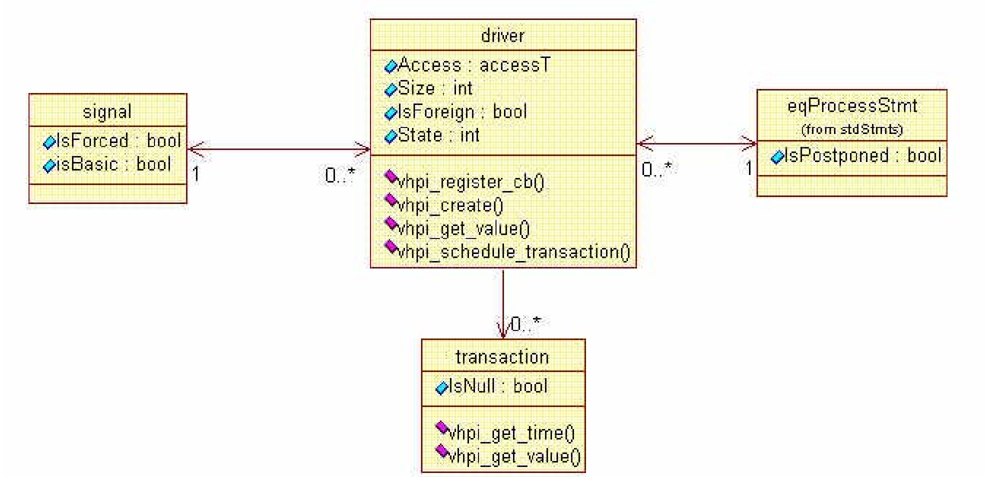
48. The vhpiNullLiteralK class represents the null access value for any access type.

Formatted: Bullets and Numbering

4.11 The standard connectivity package

4.11.1 The driver class diagram

The access described by this diagram is part of connectivity access capability with the following exceptions:
vhpi_schedule_transaction to a driver, iteration on transactions, vhpi_register_cb on a driver are part of the debug and runtime capability.
vhpi_create of a driver is part of the advanced foreign model capability (creation of a foreign driver).



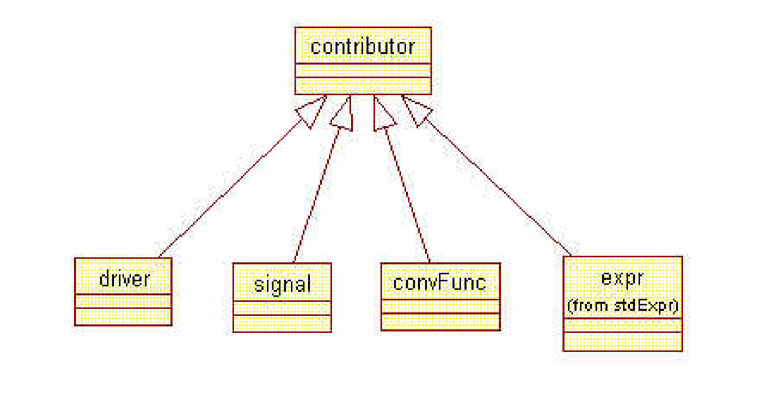
Notes:

49. The iteration on transactions returns the future scheduled transactions for the driver.
50. Iteration on vhpiBasicSignals should return the basic signals as defined by the VHDL LRM. (John to provide additional description)
51. If vhpiIsForeign is TRUE, the foreign driver may or may not have a process associated with it.

Formatted: Bullets and Numbering

4.11.2 The contributor inheritance diagram

The access described by this diagram is part of the connectivity capability.



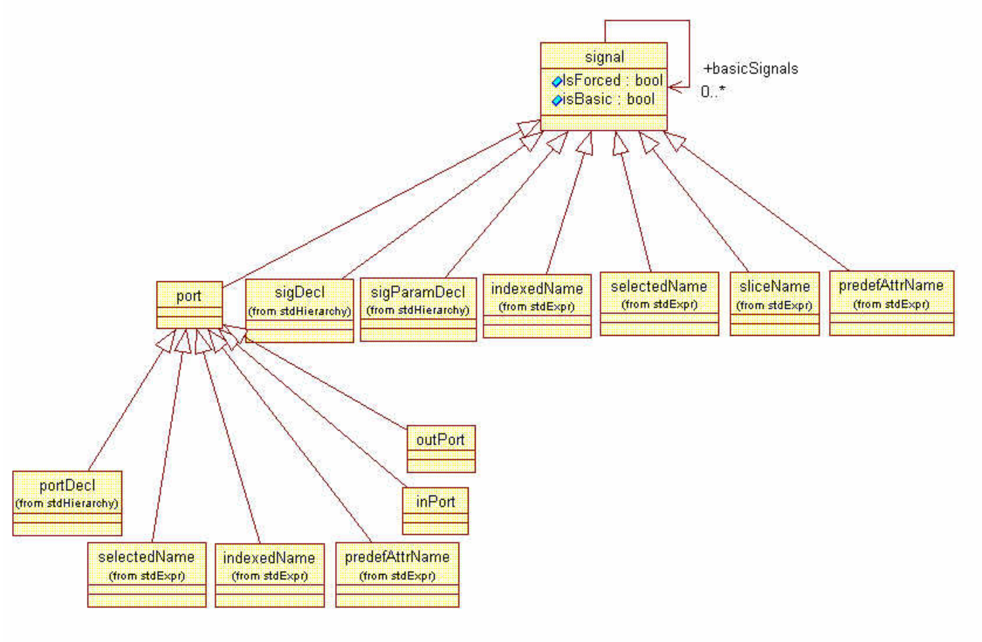
Notes:

52. Undriven signals or ports that are left opened may be assigned a globally static initial expression in their declaration; the iteration on contributors from such handles will return the initial expression of the declaration. If such objects do not have an explicit initial expression but takes its driving value from the default value of its subtype, the iteration on contributor shall return NULL. An expression can also be returned as a contributor if the INPUT, INOUT, or buffer port is associated with a globally static expression. get a null handle back only if the expression is the default value.

Formatted: Bullets and Numbering

4.11.3 The basic signal class diagram

The access described by this diagram is part of the connectivity capability.



Notes:

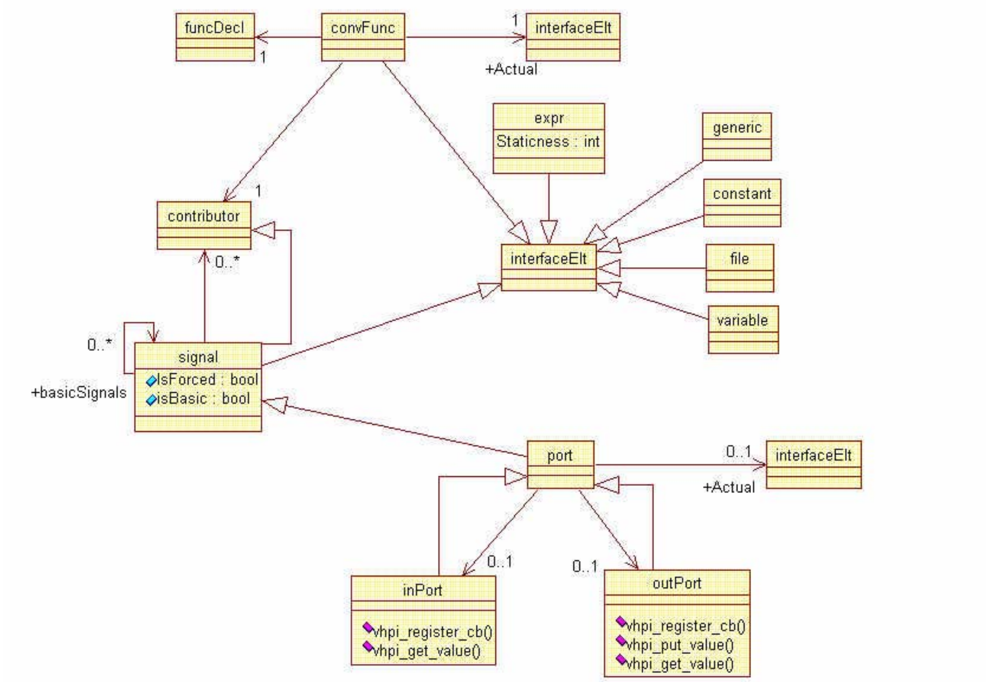
53. If the property vhpiIsBasicP is TRUE for a signal, it will be FALSE for any lower level subelement of the signal.

Formatted: Bullets and Numbering

4.11.4 The connectivity diagram

The access described by this diagram is part of the connectivity capability with the following exceptions:
vhpi_put_value and vhpi_register_cb on to OutPort are part of the basic foreign models and debug and runtime capabilities.

Deleted: Issue: no equivalent of vpiPortInst for a signal (okay)¶



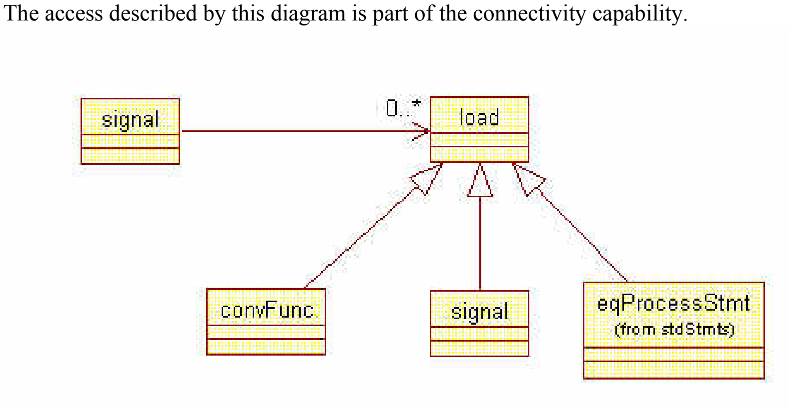
Notes:

54. The vhpiOutPort and vhpiInPort relationships are only available for an INOUT port
55. Iteration on contributors are not allowed from a signal for which the vhpiIsBasicP property is FALSE.

Formatted: Bullets and Numbering

4.11.5 The loads class diagram

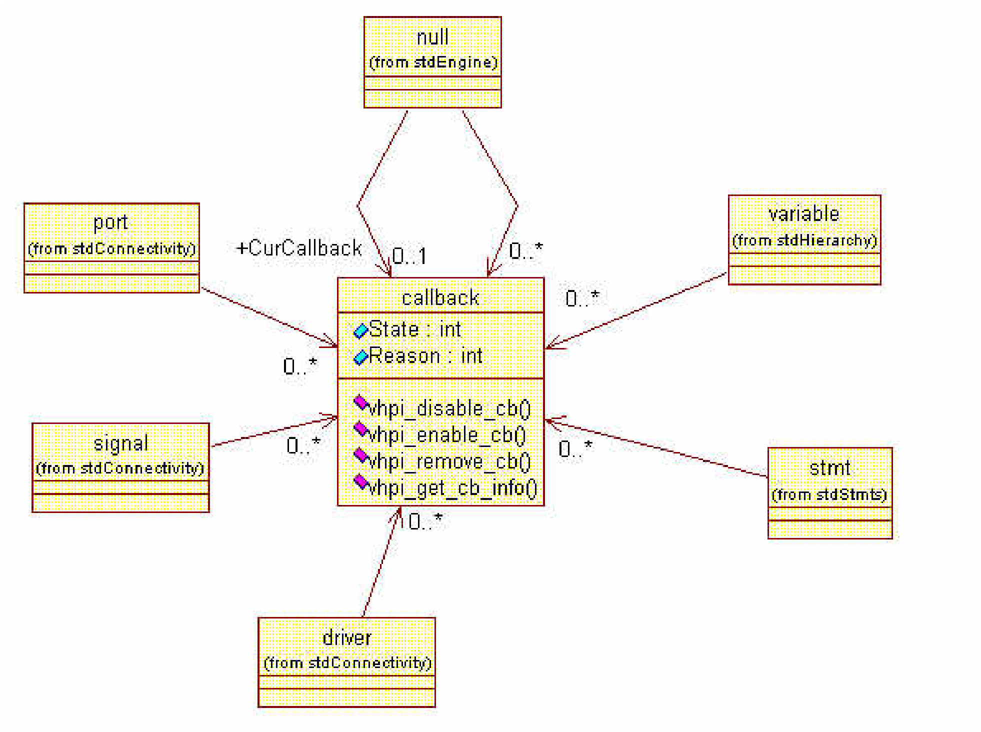
The access described by this diagram is part of the connectivity capability.



4.12 The standard callback package

4.12.1 The callback statement class diagram

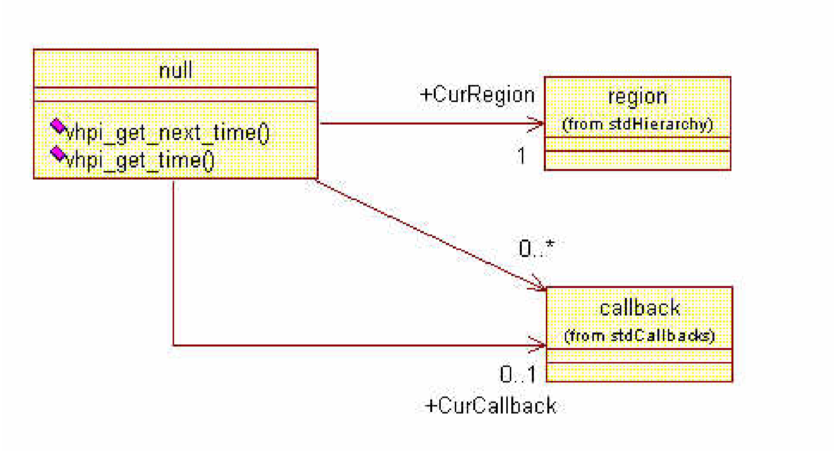
The access described by this diagram is part of the debug and runtime and basic foreign models capabilities.



4.13 The standard engine package

4.13.1 The simulator kernel class diagram

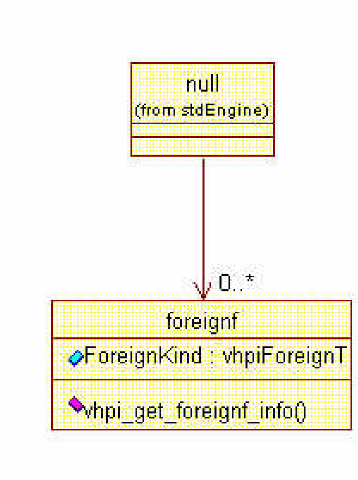
The access described by this diagram is part of the debug and runtime capability.



4.14 The standard foreign models package

4.14.1 The foreign model class diagram

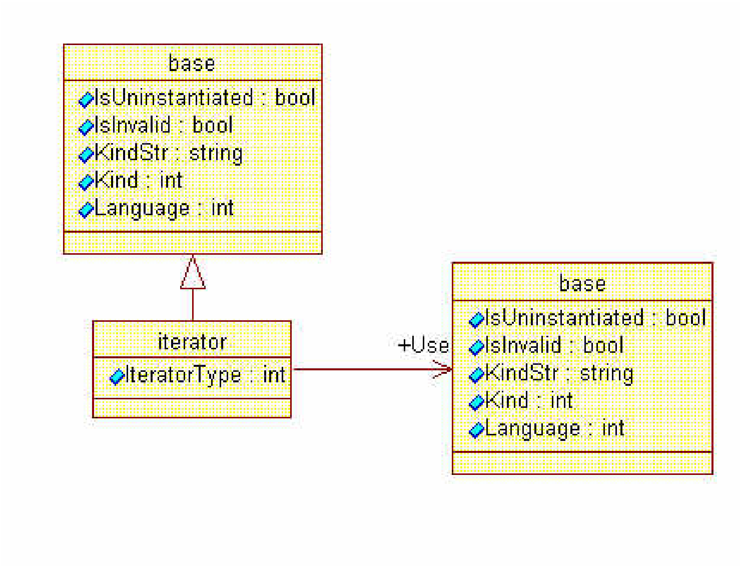
The access described by this diagram is part of the basic foreign model capability.



4.15 The standard meta package

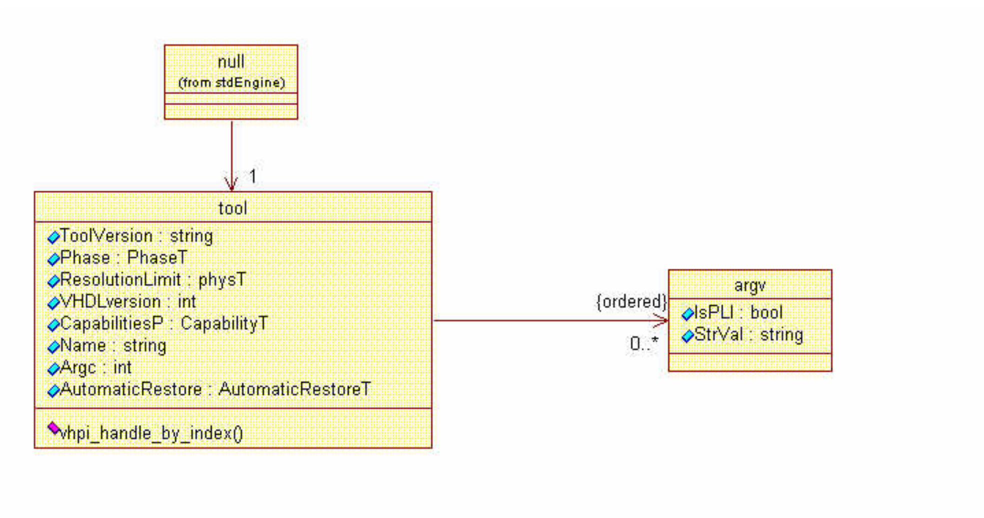
4.15.1 The iterator diagram

The access described by this diagram is part of the static access capability??



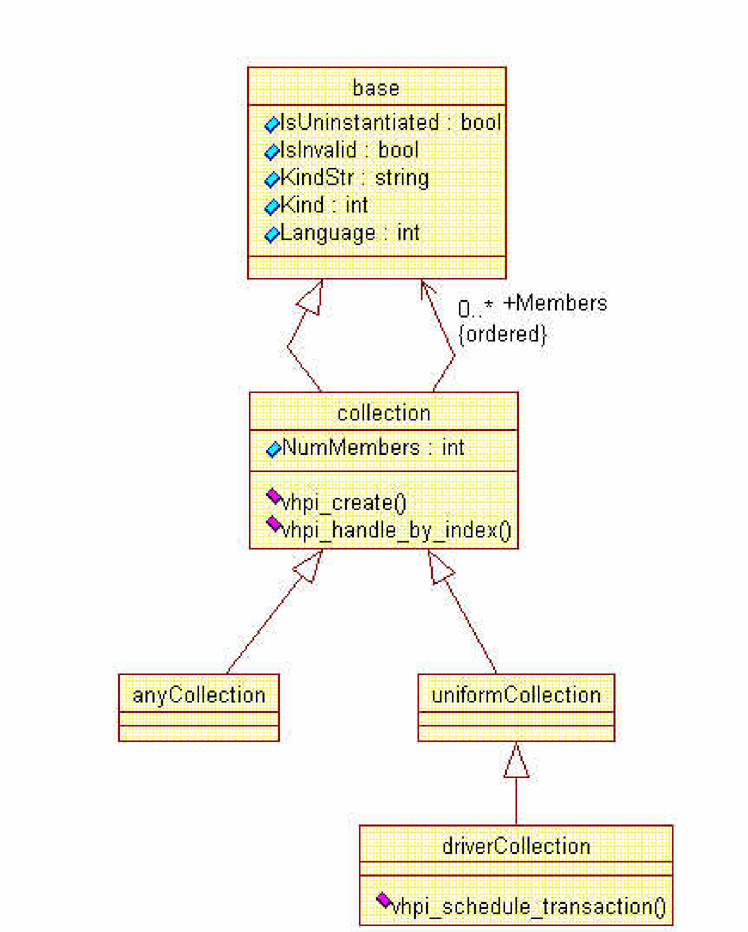
4.15.2 The tool class diagram

The access described by this diagram is part of ?.



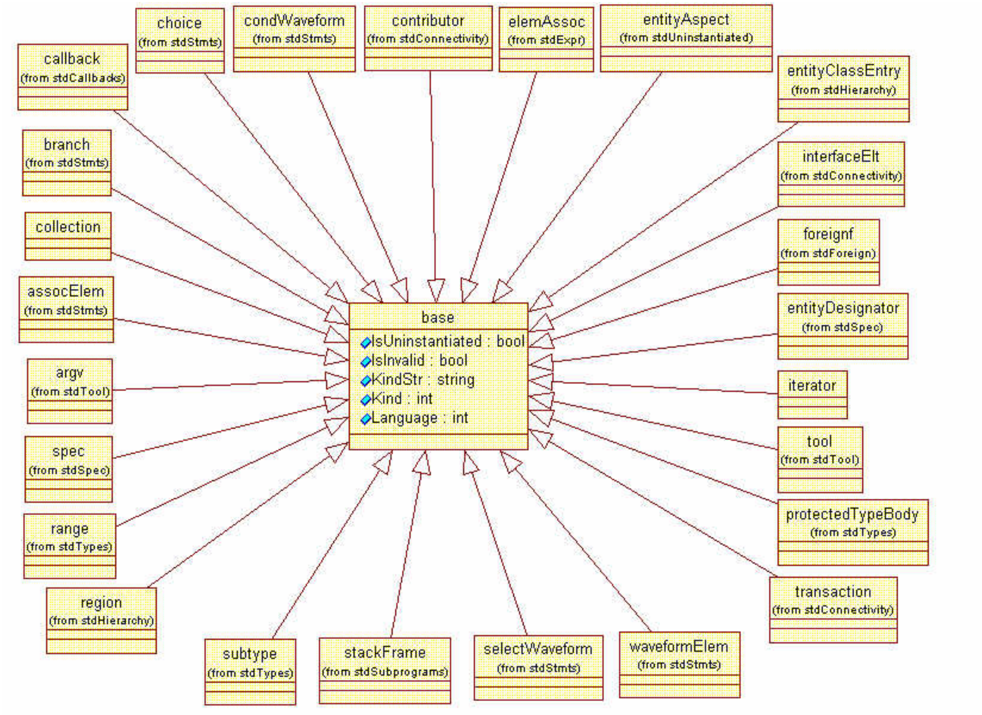
4.15.3 The collection class diagram

The access described by this diagram is part of the debug and runtime capability.



4.15.4 The base inheritance class diagram

The access described by this diagram is part of all capabilities. Not all classes may be supported.



5. Access to the Uninstantiated model

5.1 Scope

This document describes the functional specification for providing VHDL uninstantiated access, that is the VHPI model for accessing analyzed (non-elaborated) VHDL information. The access described in this chapter is part of the post analysis capability set.

In an analyzed VHDL model, the basic units are design units that reside in design libraries. A design unit is one of:

1) entity declaration
2) architecture body
3) package declaration
4) package body,
5) configuration declaration.

Access to an uninstantiated model is provided by referencing the library name and a design unit name.

The objective is to be able to recreate the VHDL source file (not necessary to preserve the same statement ordering as in the original source file). In particular, user should have access to component declarations, component instantiations, formals/actuals, unconstrained array types, configuration specifications, configuration body, etc... However, certain items processed during the analysis phase cannot be reverted back. Examples of some of these are locally constant expressions, comments…

Uninstantiated access is useful for tools that need access to the source VHDL via an API and do not need access to the hierarchy or connectivity information. Examples of such tools are: design rule checkers, coding style checkers and translators.

5.2 VHPI Application Contexts

VHPI supports two information models:

- the uninstantiated information model

- the instantiated information model.

Given the various phases of a tool:

vhpiCbStartOfTool -> vhpiCbStartOfAnalysis-> vhpiCbEndOfAnalysis->vhpiCbStartOfElaboration->vhpiCbEndOfElaboration->vhpiCbStartOfSimulation->vhpiCbEndOfSimulation->vhpiCbEndOfTool

The uninstantiated information model is available for access from vhpiCbStartOfTool till vhpiCbEndOfTool, while the instantiated information model is available for access from vhpiCbStartOfElaborationStart till vhpiCbEndOfTool. Specify that the tool capability must include vhpiProvidePostAnalysis.

Note: Library browsing can be done at CbStartOfTool.

Given a handle to an object in the uninstantiated information model, it is not possible to traverse a relationship to get to a handle of an instantiated object.

The reverse is not true, that is, there are cases where from the instantiated information model, you can traverse a relationship to get to an object in the uninstantiated information model. Section 5.3.1 enumerates all such cases.

For the instantiated information model, the context of operation for a VHPI application is the **elaborated design context**. This context consists of:

- the entire elaborated design including the top-level package instances.

So a VHPI program can get access to the instantiated information model in one of two ways:

- by navigation from the NULL handle (get a handle to the root of the elaborated design, get handles to package instances etc…)

- get a handle to an elaborated object via vhpi_handle_by_name().

For the uninstantiated information model, a VHPI application only accesses uninstantiated information.

So a VHPI program can get access to the uninstantiated information model in one of four ways:

- get a handle to a library (working library or other design libraries).

- get a handle to an uninstantiated object via vhpi_handle_by_name().

- navigate from the instantiated to the uninstantiated model (see section 5.8).

In this chapter, we describe the uninstantiated information model and access only.

5.3 VHPI Uninstantiated Access

The VHDL procedural interface is defined by:

the definition of a VHDL uninstantiated information model that represents the VHDL data that is accessible by the procedural interface,

a subset of the functions that operate on this model to access, modify, and interact with the data.

The next sections define which subset of the VHPI information model is available, what additional new information is available, and what are the functions that can legally be used in on the uninstantiated information model.

5.3.1 Uninstantiated Information Model

During VHDL analysis, design units described in a design file are analyzed and each successful analysis results in the analyzed design unit data being placed in a design library. The VHPI uninstantiated data is the data resulting from the analysis of a design unit. A design unit is comprised of use clauses (dependent analyzed units or declarations) and a single library unit which is either an entity, an architecture, a package, package body or configuration declaration. Design units have only uninstantiated data. Any data obtained by traversing a relationship from a design unit handle will lead to uninstantiated data. Any data obtained by traversing a relationship from a reference handle of uninstantiated data gives back handles to uninstantiated data.

From the uninstantiated information model, no access to the instantiated information model is allowed. Classes of objects which are not part of the uninstantiated information model are classes which denote elaborated or runtime data. For example, a handle of the region class can never be obtained in the uninstantiated model because it denotes an elaborated region instance.

It is possible to obtain a handle to uninstantiated data in the following cases:

1) When calling vhpi_handle_by_name() with the unit name of the design unit or with the string returned by the vhpiDefNameP property. The handle obtained is of class vhpiEntityDeclK, vhpiArchBodyK, vhpiPackDeclK, vhpiPackBodyK, vhpiConfigDeclK or a handle to uninstantiated data. Revisit after chapter 6 review.

2) When calling vhpi_handle_by_name() with the name of the library. Revisit after chapter 6 review

3) When traversing a relationship from a handle of uninstantiated data.

4) When applying the 1-to-many relationship vhpiLibraries to a NULL reference handle; this will yield all the libraries made available to the tool.

```
vhpi_iterate (vhpiLibraryDecls, NULL)
```

(The association of the physical library to the logical library is tool-specific).

5) When traversing a specific relationship from the instantiated information model. This set of relationships is defined later in the document.

In the uninstantiated information model, it is not possible to access information which pertain to elaboration, connectivity or runtime. In addition, only handles to objects which have an existing reference in the VHDL description can be obtained. For example it is not possible to iterate on vhpiIndexedNames from a handle of a variable declaration which is of an array type.

5.3.2 New additions

5.3.2.1 Lexical Scope Class

Given a handle to an uninstantiated declaration, the 1-to-1 method vhpiLexicalScope will return a handle to the enclosing lexical scope of that declaration. The lexical scope relationship will return NULL for a reference handle denoting a library declaration.

The lexical scope relationship returns a handle of the lexical scope class which is one of the following classes:

- sub-classes of the design unit class: vhpiEntityDeclK, vhpiArchBodyK, vhpiPackDeclK, vhpiPackBodyK, vhpiConfigDeclK

- some sub-classes of the class decl: vhpiFuncDeclK, vhpiProcDeclK, vhpiProtected-TypeDeclK, vhpiProtectedbodyK, vhpiCompDeclK, vhpiRecordTypeDeclK,

- some sub-classes of the class stmt: vhpiBlockStmtK, vhpiLoopStmtK, vhpiForGen-erateStmtK,

- a class of kind vhpiBlockConfigK, vhpiCompConfigK

The vhpiLexicalScope class is only valid in the uninstantiated information model with a reference handle to an uninstantiated declaration. Following is the class diagram for the lexical scope class:

Lexical Scope class diagram (see diagram 4.4.2)

5.3.3 Expanded Names

Move this to the name class diagram.
In aVHDL source file, there may exist a number of out-of-scope references, called **expanded names**. Examples of these are:

```
IEEE.NUMERIC_BIT.UNSIGNED
WORK.ALU(RTL)
MY_PACK.SIG_A
```

VHPI does not provide full information about expanded names references. This has the drawback to not being able for decompilation applications to exactly produce the original source. This is more efficient for synthesis oriented applications and more inline with the information retained by analyzers.

5.3.4 Unsupported classes

The following **class kinds** of vhpi handles are not supported in the uninstantiated information model:

```
vhpiDriverK,
vhpiDriverCollectionK,
vhpiForeignfK,
vhpiInPortK,
vhpiOutPortK,
vhpiPackInstK,
vhpiProtectedTypeK,
vhpiRootInstK,
vhpiTransactionK.
```

5.3.5 Unsupported 1-to-1 relationships

The following **1-to-1 relationships** are not valid in the uninstantiated information model:

Issue: missing association from the forGenerate and ifGenerate classes

```
vhpiBasicSignal,
vhpiContributor,
vhpiCurCallBack
vhpiCurEqProcess,
vhpiCurStackFrame,
vhpiDerefObj,
vhpiDownStack,
vhpiImmRegion,
vhpiInPort,
vhpiOutPort,
vhpiProtectedTypeBody,
vhpiRootInst,
vhpiUpStack,
```

```
vhpiUpperRegion.
```

5.3.6 Unsupported 1-to-many relationships

The following **1-to-many relationships** are not valid in the uninstantiated information model:

```
vhpiBasicSignals,
vhpiContributors,
vhpiCurRegion,
vhpiDrivenSigs,
vhpiDrivers,
vhpiForeignfs,
vhpiIndexedNames,
vhpiInternalRegions,
vhpiPackInsts,
vhpiSelectedNames,
vhpiSigAttrs,
vhpiTransactions.
```

Deleted: s

Deleted: Issue I019 (curEqProcess)

A vhpi error will be generated if the functions vhpi_handle() and vhpi_iterator() are used for a uninstantiated handle and the iteration method or the 1-to-1 method is one of the invalid methods listed above. In these cases, the functions will return a null handle. The vhpi error can be checked immediately after the call to vhpi_handle() and vhpi_iterator() by calling vhpi_check_error().

5.3.7 Unsupported integer properties

The following **integer properties** are not valid in the uninstantiated information model:

```
vhpiAccessP, (read, write, or no access at all)
vhpiForeignKindP,
vhpiFrameLevelP,
vhpiGenerateIndexP,
vhpiIsBasicP,
vhpiIsDefaultP, (binding of the component
instance is default binding)
vhpiIsForcedP,
vhpiIsForeignP,
vhpiIsOpenP,
,
vhpiLoopIndexP,
vhpiResolutionLimitP.
```

A vhpi error will be generated if the functions vhpi_get() and vhpi_get_str() are used with an integer /string property which is invalid in uninstantiated mode. The function vhpi_get() will return an unspecified value and vhpi_get_str() will return null. The vhpi error can be checked immediately after the call to vhpi_get() or vhpi_get_str() to check for error by calling vhpi_check_error().

5.3.8 Unsupported functions

The following vhpi functions are not valid in the uninstantiated information model:

```
- vhpi_protected_call

Value access and modifications:
- vhpi_get_value : will only returns values
for locally static expressions
- vhpi_put_value
- vhpi_schedule_transaction

Simulation access and control function:
- vhpi_get_time
- vhpi_control
- vhpi_put_data

Foreign function registration:

- vhpi_register_foreignf
- vhpi_get_foreign_info
```

Notes:
1. vhpi_handle_by_index (invalid when creating a handle to an indexedName or selectedName because the iterations on indexedNames and selectedNames are illegal in uninstantiated access).
2. Callback functions:
- vhpi_register_cb
- vhpi_disable_cb,
- vhpi_enable_cb,
- vhpi_get_cb_info,
(Action callbacks are allowed, object and stmt callbacks cannot have uninstantiated handles. Time callbacks are not allowed…).
3. vhpi_create is not allowed to create a driver or a processStmt handle.

The use of functions which are not supported in the uninstantiated model will generate a runtime vhpi error. The functions which return a handle or a status code will respectively return a null handle or the code for failure.

5.3.9 vhpi_handle_by_name

See section 6.2

5.3.10 Instantiated to uninstantiated model

Issue: mark these associations in the information model.
Issue: add associations from the information model which return uninstantiated handles.

This section lists the list of legal relationships that can be used to cross from the instantiated information model to the uninstantiated information model:

1) vhpiCompInstStmtK to designUnit class
2) vhpiRootInstK to vhpiConfigDeclK
3) vhpiCompInstStmtK to vhpiConfigSpecK

4) vhpiCompInstStmtK to vhpiCompDeclK
5) vhpi_handle_by_name passing the string of the vhpiDefNameP property
6) subprogram
7) forloop

No other accesses shall be allowed.

5.3.11 Additional Comments

When iterating on statements in the uninstantiated model, you get handle to an if generate statement even if the conditional expression is locally static (and false). On the contrary, in the instantiated model, an ifGenerate statement whose condition is false will not be returned since it has not been elaborated.

6. VHPI names properties, Access by name lookup

Move this paragraph as an introduction to chapters 4, 5, 6
VHPI provides two related but distinct formal information models. The instantiated design model is an instance hierarchy, i.e., the root instance, its component and block instance hierarchy, etc., that results from elaboration of a VHDL design. The uninstantiated information model is a library of design units previously analyzed. The relationship between them is that during the elaboration phase, the analyzed design units are instantiated into the instance hierarchy. After elaboration, instances in this hierarchy have a relationship back to corresponding design units in a library; the structure and behaviour of an instance are defined by the design units to which they are bound. If a VHPI tool claims to support post analysis capability, it supports access to the uninstantiated information model.

Many object classes in each information model have name properties, strings that convey naming information to VHPI client applications. These are useful in referring to these objects in written output in terms the end user can relate back to the original VHDL source.

VHPI clients have a variety of mechanisms to obtain handles to objects in either information model. From an initial handle, one may navigate to other handles. Alternatively, one may search for a handle by name. The search string may be an absolute or relative path name. A search may be limited to a specific region or design unit

The next section identifies the name properties available in both the library and design information models. After that, vhpi_handle_by_name functionality is defined. Open issues are identified and rationale included, as appropriate.

6.1 VHPI Name String Properties

The name string properties can be understood by examining the UML formal information model and using the definitions below. Both the instantiated and library information model objects share some name properties. To avoid confusion, their descriptions are repeated.

Deleted: uninstantiated

Issue: replace uninstantiated with library unit and instantiated with design hierarchy

6.1.1 Name Properties - Instantiated Information Model (design hierarchy access)

The instantiated information model includes everything that is constructed during VHDL elaboration phase and can be broadly divided into the design instance hierarchy and the elaborated packages referenced by the design.

With respect to the UML of the design model, the named objects are primarily regions (see the region inheritance class diagram) and decls (see the declaration class diagram).

6.1.1.1 vhpiNameP

This property returns the name of the designated object in unspecified case for basic identifiers (VHDL is case insensitive) or case preserved for extended identifiers. Broken down by class:

6.1.1.1.1 decl - for a declared item, its identifier

The vhpiNameP of a declaration is the declaration identifier name. The vhpiNameP of a subprogram declaration (vhpiSubpDeclK) or subprogram body (vhpiSubpBodyK) is the subprogram identifier name. The vhpiNameP of a vhpiLibraryDeclP is the library logical name.

Issue: This is *not* equivalent to the 'simple_name predefined attribute. For example, 'simple_name is guaranteed to return all lower case. Resolution : A name property equivalent to vhpiSimpleNameP was not determined necessary.

Note: since an enumeration literal or subprogram can be overloaded, a property vhpiSignatureNameP can return the parameter type profile string of the enumeration literal or subprogram. The vhpiNameP property of an enumeration literal or subprogram shall return the simple name of the enumeration literal or subprogram. For example for a enumeration literal named red and belonging to the enumeration type "color_type", the vhpiNameP property shall return "red" and the vhpiSignatureNameP shall return "return color_type" in order to distinguish it from another enumeration literal of the same name belonging to another enumeration type.

Issue: vhpiNameP property can be queried on a handle of an implicit declaration and returns the declaration name.

For example, guard signals have a simple name of "GUARD". These can be found with vhpi_handle_by_name.

The name property applied to implicitly defined operators (for example "+" from the the standard package) follow the syntax for overloaded functions. The vhpiNameP of an operator declaration should include the double quotes.

6.1.1.1.2 compInstStmt – its instance label

For a vhpiCompInstStmtK reference handle, the vhpiNameP property returns the component instance statement label.

6.1.1.1.3 rootInst – the entity name

For a vhpiRootInstK reference handle, the vhpiNameP property returns the entity name
(it was agreed for mixed language interoperability verilog returns the name of the top level module)

6.1.1.1.4 packInst - its package name

6.1.1.1.5 blockStmt - the block label

6.1.1.1.6 loopStmt – the loop label or an implicitly created label

<vhpiFullNameP of upperRegion>:<vhpiNameP of loopStmt>

Deleted: t

Deleted: should be

Deleted: [

Deleted:]

Deleted: This is consistent with the syntax for an overloaded parameterless function whose return type is color_type (see section 5.2.1.11). This is how the VHDL LRM describes enumeration literals.¶

Deleted: Add vhpiSignatureNameP property for vhpiSubpDecl and vhpiEnumLiteralK ; reconcile with vhpiFullName¶

Deleted: (reconcile with signatureName)

See Rule 1 below for generating implicit loopStmt labels. Because the loop stmt variable is dynamically elaborated, it is advanced functionality. If the VHPI server conforms to "dynamically elaborated" capability, handles to loop variables may be obtained (See section 1.1.2 VHPI levels of capability). The vhpiNameP property of a loop variable would be its identifier.

6.1.1.1.7 generateStmt - the <label_name>[(generate index)]

There are two forms of generate, a conditional generate and an iterative generate. For a conditional generate, the vhpiNameP is the label name. With an iterative generate stmt, each iteration replicates the contents of the generate stmt, i.e., generates a block instance of its declarations and concurrent stmts. The generate_index is the value of the for generate constant that defines the iteration. This implicit constant declaration is defined with a discrete range, either an integer or enumerated type. For the purpose of the vhpiNameP string, the value of the generate_parameter of an enumerated type will be given by the string representation of the enumeration literal.

6.1.1.1.8 VHDL name

IndexedName, SelectedName, SliceName, derefObj, ParamAttrName, SimpAttrName, UserAttrName, (see name class diagram) the VHDL string name equivalent to the object the name refers to. It may not be returned as it appears in the VHDL source because a single handle may represent expressions of the same object. Example: s(1), s(1 + 0), s(c) where C is the integer constant 1, may have the same vhpiNameP of "s(1)"

The name may be returned with or without case preserved depending if there is reference to an extended identifier within the name.

Example: selected name: "f.a", indexed name: "r(j)", "a.all", "s'delayed", "t'high", "my_attribute", etc…

Note: a vhpiDerefObjK handle will only have a name if it refers to a VHDL source dereference name. vhpiDerefObj handles which are obtained by way of applying the vhpiDerefObj method do not have a name because they denote a memory location in the VHDL heap.

6.1.1.1.9 EqProcessStmt – the process label or implicitly generated label

See Rule 2 below for generating EqProcessStmt labels. An equivalent process statement is either a process statement or one of the concurrent statements - procedure call, assertion, or signal assignment.

6.1.1.1.10 protectedType

A vhpiProtectedTypeK handle represents the instantiated variable of a protected type. The vhpiNameP of a vhpiProtectedTypeK handle is the name of the variable. The vhpiNameP of protected type declaration vhpiProtectedTypeDeclK or vhpiProtectedTypeBodyK is the protected type identifier name

6.1.1.1.11 subpCall and stackFrames

a) name of a procedure call stmt:

When the procedure call statement is executed, a stack frame is created which represents the state of the procedure call. For a concurrent procedure call, the vhpiNameP property shall return the explicit or automatically generated name of the equivalent process representing the concurrent subpcall.
For a sequential procedure call the vhpiNameP property shall return name of the of the subprogram. The vhpiNameP property returns the same strig value whether or not the procedure is active.
There is a special case when the procedure call denotes a method of a shared variable of a protected type; in that case the vhpiNameP of the procedural call or function call should be:
<variable_name>.<vhpiNameP of the subpCall>
When an object of a protected type is used in VHDL, it is accessed through protected procedure and function calls. Its name is defined consistently with subpCall objects. The procedure call name is <variable_name>.<subp_name>().

Deleted: T

Deleted: of a subpCall should be the

Deleted: concurrent or sequential

Deleted: When the procedure call is not executing, the vhpiNamePame should return the procedure declaration name.

Deleted: ¶

We should provide handles to concurrent procedure calls when traversing hierarchy. Access to concurrent procedure calls should not be restricted by the VHPI server. Concurrent procedure calls are transformed in their equivalent processes with sequential procedure calls. Dynamic elaboration applies to sequential procedure calls.

Deleted: I sent out an email to Paul and Alex asking if the dynamic elaboration was applying to concurrent procedure calls.¶
Paul replied that concurrent

Deleted: ¶

Deleted: ¶

b) name of a function call

A function call is an expression.

The vhpiNameP of a function call is the name of the function declaration. Ex: "my_func". In the case of a function call applied to a shared variable of a protected type, the name should include the shared variable name:

Deleted: <variable_name>.<vhpiNameP of the subpCall>

6.1.1.2 vhpiSignatureNameP

This property is available for subpCall and enumLiteral.
The property returns a string which represent the signature of the subpCall or enumeration literal. An enumeration literal is represented as a parameterless function call.

[[<vhpiNameP of parameter type declaration>]{**,** <vhpiNameP of parameter type declaration>}][**return** <vhpiNameP of return type>]**]**

6.1.1.2.1 Rule 1 - Naming of unlabelled loop statements:

VHPI would generate a loop label which starts by the concatenation of the "_L" or "_l" string and an integer which denotes the sequence appearance number of the loop statement in the VHDL source text of the declared region. The numbering starts at 0 and increments by 1. For example the auto-generated vhpiNameP of the first loop statement in process or postponed process would be "_L0". Numbering of loops is reset for each internal region.
Rationale: A loop variable is an important object to access by name. Because the label is optional, 'path_name attribute can be ambiguous. The vhpiNameP property must produce reliable names that are unique references to an object to support handle_by_name. Because the loop variable is dynamically elaborated, look up by name of the loop variable is advanced functionality. The fullname of the loop variable would include the process label, the explicit or automatically generated loop label and the loop variable.
Example:
"Process_label:loop_label:loop_var"

6.1.1.2.2 Rule 2 - Naming of unlabelled equivalent processes:

VHPI would generate an equivalent process label name which starts with the concatenation of the "_P" or "_p" string and an integer which denotes the sequence appearance number of the equivalent process in the VHDL source text of the declared region. The numbering starts at 0 and increments by 1. For example the auto-generated vhpiNameP of the first equivalent process statement in an entity declaration would be "_p0". Numbering of equivalent processes in the architecture follows the numbering sequence used for the entity. The number used for the first process in the architecture will be either 0 if the entity did not contain any unlabelled processes or n+1 where n is the number used for naming the last unlabelled equivalent process of the entity. Numbering of processes is reset for each internal region (block, generate or component statement).

example: "_P2"is the vhpiNameP of the second occurring process for this entity/architecture pair.
note: _P2 is not a legal VHDL identifier (should be escaped) this ensures that this identifier is not used in the rest of the design.
Rationale: Because the label is optional, 'path_name attribute can be ambiguous. The vhpiNameP property must produce reliable names that are unique references to an object to support vhpi_handle_by_name.

6.1.1.3 vhpiCaseNameP

Returns the case preserved name of the declared identifier.

I propose that vhpiNameP of implicit declarations returns the implicit declaration identifier (for example for the guard signal, the operator name for an implicit operator etc... vhpiCaseNameP would return the same as vhpiNameP for implicit declarations.
In the case of:
- a declaration appearing in both the subprogram declaration and its body,
- a subprogram declaration and its subprogram body,
- an incomplete type and its full type,
- a protectedTypeDeclaration and its body,
- a package declaration and its body,

the vhpiCaseNameP shall return the case preserved name of the declaration in the subprogram declaration, the incomplete type, the protectedType declaration or a package declaration.
Issue: check rational + specify new compliance capability for names

Deleted: ,

6.1.1.4 vhpiFullNameP

This is a string describing the path through the elaborated design hierarchy, from the top level entity or package to this object. The string is defined in terms of the vhpiNameP property to produce a unique reference to the given object. Objects which have the vhpiFullNameP property are explicit declarations, sub-elements of these declared objects, or regions.
The vhpiFullNameP property returns a string that is often identical to X`PATH_NAME attribute, but will differ because of ambiguities and features of the inherent in the VHDL LRM definition.
Since VHDL is case insensitive, the case of the vhpiFullNameP string is not specified unless there is an extended identifier.
Note: vhpiFullCaseNameP should be used to retrieve the hierarchical name with case preserved characters for the declared items only.

6.1.1.4.1 Elaborated design object :{<vhpiNameP>:}[vhpiNameP]

The vhpiFullNameP property should return the concatenation of the vhpiNameP strings of each instance or item found on the hierarchy path. The character ‘**:**’ is used between two successive names returned by vhpiNameP.
The vhpiFullNameP of an object declared in a subprogram would only be obtainable if the subprogram is elaborated and consequently the object is elaborated too. Therefore the fullName of that object would contain the vhpiNameP of the subprogram call as defined above.
Ex:The vhpiFullNameP of a variable A defined as a subprogram parameter, elaborated as part a concurrent subprogram call, would be:
<vhpiFullNameP of the base eqProcessStmt representing the concProcCallStmt>::A

The fullname of an to uninstantiated declarations is equivalent to the vhpiDefNameP (fullname in the library information model).
The vhpiFullNameP of the root instance is ":<entity_name>. This is sufficient to refer to a unique root, VHDL 1993 and 2000 only allow one top-level design unitThis brings it inline with `path_name. It is also the only way to deal with the stated goal of interoperability with Verilog and VHDL.

Deleted: For

Deleted: its full name

Deleted: <vhpiNameP of subpCall

Deleted: Issue: reconcile with

Deleted: applied

Deleted: => A declared item in a function declaration does not have a fullname as it is not elaborated¶

6.1.1.4.2 Library declaration

The vhpiFullNameP of a vhpiLibraryDecl is: @<lib_logical_name>

6.1.1.4.3 Elaborated package object @<lib_logical_name>:<pack_name>[:vhpiNameP]

The vhpiFullName for an object in an elaborated package instance returns a string that is nearly identical to the `path_name attribute, but differs in order to resolve ambiguities in the VHDL LRM definition. The leading : is replaced with an '@' character to disambiguate a name reference to an elaborated design object from a reference to a package object with the same name. For example, "work" can be both a logical library name and the entity name of the design root.
Construction of full names for items elaborated in package instances is defined as follow:
@<lib_logical_name>:<pack_name>:<vhpiNameP of declared_item>
The declared_item_name is equivalent to the vhpiNameP property of the declared object.

6.1.1.4.4 subpCall

:<vhpiFullNameP of upperRegion of base eqprocess stmt>:<vhpiNameP of subpCall>{:<vhpiNameP of subpCall>}
This would work for both concurrent and sequential procedure calls.
For concurrent procedure calls, the names would reduce to the name of the eqprocessstmt it corresponds to.
For sequential procedure calls, the name would be formed of the name of the enclosing eqprocessstmt, to which would be appended the vhpiNameP of the procedure.

6.1.1.4.5 vhpiPathNameP

Rationale: provides an ability to write foreign models and application output consistent with the simulator. This is a string describing the path through the elaborated design hierarchy, from the top level entity or package to this object. The vhpiPathNameP is identical to the VHDL predefined attribute, 'path_name, as defined in the LRM 1076. A VHPI tool should guarantee that the same value will be used for `path_name attribute during simulation.

6.1.1.4.6 vhpiInstanceNameP

The vhpiInstanceNameP is identical to the VHDL predefined attribute, `instance_name, as defined in the LRM 1076. This is a string similar to `path_name, but includes the names of the entity and architecture bound to each component instance in the path. (Same rational as 5.1.1.3.3).

6.1.1.4.7 vhpiCaseNameP

This property returns the case preserved string of the item declaration. The string returned will reflect lower or upper case characters used in the identifier declaration. Note that for extended identifiers, or unlabelled loop statements or equivalent processes, the vhpiCaseNameP string will be exactly the same as the vhpiNameP string.

6.1.1.4.8 vhpiFullCaseNameP

The string returned is formed by the concatenation of each single vhpiCaseNameP string on the hierarchical path to the designated object. The ':'character is the delimiter between each simple case name.

> **Note**: All these properties vhpiNameP, vhpiCaseNameP, vhpiFullNameP and vhpiFullCaseNameP apply to the name class (see expression diagram) and region class.

6.1.1.4.9 vhpiDefNameP

This property returns the full name of the associated object in the uninstantiated information model; this property is available for all objects which have a name.
vhpiDefNameP syntax use . separator and is defined in terms of vhpiUnitNameP,
Note: This property is available in both the elaborated model and uninstantiated model.

6.1.1.4.10 vhpiUnitNameP

This property is available in both the elaborated model and uninstantiated model. See uninstantiated model properties for description.

6.1.1.4.11 vhpiFileNameP

This property returns the physical file system path name of the VHDL source file where the item designated by the handle appears. This property is applicable for every VHPI class kind that has a vhpiLineNoP (line number property). Among these are: declared items, design units, etc...
Note: the VHPI specification does not imply the physical organization of a file system and makes no normative reference to file system specifications.

6.1.1.5 Name Properties - Uninstantiated Information Model (Library unit access)

The uninstantiated information model is a library of previously analyzed design units related to the current execution of the underlying tool providing VHPI. During the elaboration phase, some of these analyzed design units are instantiated into the hierarchy. After elaboration, instances in the design hierarchy have a relationship to design units in a library, they are defined by them (see design unit class diagram 4.4.1).

6.1.1.6 The uninstantiated model

The UML diagrams shares object classes between the uninstantiated and instantiated model, but these object classes are understood to have differences. See chapter on uninstantiated access to understand these differences. Issue: How do we represent this in UML?
The design unit class in the unelaborated data model is unique and has a number of unique properties.
There are no regions in the instantiated model, but rather lexical scopes which affect some of the name properties.

6.1.1.7 vhpiLibLogicalNameP - the logical name of the library in which the design unit was compiled

This property returns the logical name of the library in which the design unit was analyzed.

6.1.1.8 vhpiLibPhysicalNameP - the physical name of the library

No interpretation implied by VHPI since that mapping is a tool issue, not an LRM issue.

6.1.1.9 vhpiUnitNameP: for design unit class

The name of the declared design unit in the VHDL source. This property is only applicable to the designUnit class. The separator is the . character.
The name is returned in unspecified case for basic identifiers or case preserved for extended identifiers.
The vhpiUnitNameP of a design unit of the following class is:

EntityDecl: lib_name.entity_name
Arch body: lib_name.entity_name:arch_name
PackDecl: lib_name.pack_name
Pack Body: lib_name.pack_name:BODY
note: all variations of upper and lower case letters for BODY are allowed.
Config: lib_name.config_name

This property is allowed in the instantiated and uninstantiated design.

6.1.1.10 vhpiNameP – exists for declaration and name class and should produce the same string as in the elaborated model.

The string returned conforms to and is the same as the string returned in the instantiated model with the following exceptions:
For forGenerate, the vhpiNameP string returns the label without the index.

For non locally static names, the vhpiNameP string may return a name that is different from the name in the elaborated model. For example, the vhpiNameP of an indexedname when the index is globally static may be S(gen) in the analyzed model but S(0) in the elaborated model.

6.1.1.11 vhpiFullNameP, vhpiDefNameP syntax: @<vhpiUnitNameP>{.<vhpiNameP of the vhpiLexicalScope>}[.<vhpiNameP>]

The vhpiFullNameP of an uninstantiated handle is identical to the string which would be returned by vhpiDefNameP. This is a string describing the path through the analyzed design unit, from the library through lexical scopes to this object. The string is defined in terms of the other properties to produce clear, unique references to the object.

6.1.1.12 vhpiCaseNameP no change

6.1.1.13 vhpiDefCaseNameP – analogous to vhpiDefNameP but case preserved

6.1.1.14 vhpiFileNameP – no change

6.1.2 Other Name Properties

Other string properties are defined:

6.1.2.1 vhpiCompNameP for compInstStmt, compConfig

vhpiCompNameP returns the component name specified for a component instance statement or component configuration or null if direct instantiation is used.

6.1.2.2 vhpiCompInstNameP for vhpiCompInstStmtK

Returns the string written in VHDL of either the configuration name, component name or entity architecture name. The name string includes the library name either explicit or default.

Deleted: Francoise to complete:

6.1.2.3 vhpiLabelNameP for the stmt class

returns the label name if it exists for the statement or null.

6.1.2.4 vhpiLoopLabelNameP for vhpiNextStmtK and vhpiExitStmtK

This property returns the label name of which to jump to or to exit from if the next or exit statement have an explicit label; null otherwise

6.1.2.5 vhpiLogicalNameP for vhpiFileDeclK class

This property returns the logical name which identifies an external file in the host file system which is associated with the file declaration; otherwise null

6.2 Access by name lookup

The goal is to define vhpi_handle by name in terms of name string properties for consistency in user interface input and output. The name properties have a clear definition in both information models. VHPI provides vhpi_handle_by_name function to obtain a handle to an object in either the instantiated or uninstantiated information models. The latter capability is referred to as the post analysis capability in section 1.1.2 for compliance. vhpi_handle_by_name is only allowed on classes of objects which possess the vhpiFullNameP property. If a compliance capability requires access to a given class which has the vhpiFullNameP property, it is also required that vhpi_handle_by_name be supported as well.

The vhpi_handle_by_name function uses a case insensitive comparison of the search string to the vhpiFullNameP value, except where a component of the name uses extended identifiers. That component of the name will be compared with case sensitivity.
It may be possible to specify a name string that is ambiguous, i.e., that refers to more than one object. If vhpi_handle_by_name detects ambiguity, it will return a NULL handle. If this is not detected, the object handle returned will be the arbitrary choice of one such object. VHPI does not require detection of this condition.
Implicit signal attributes cannot be returned by handle_by_name. In order to find the signal attributes implicitly defined by a design on a given signal, use the vhpiSigAttrs iteration relationship.

6.2.1 Instantiated Model Access(Design hierarchy)

The interface provides a vhpi_handle_by_name function, which given a reference scope handle and an instantiated hierarchical name which identifies a given VHDL item, returns a handle to the designated item if an item of that name exists in that scope. This function (*vhpi_handle_by_name()*) can only be used for VHPI classes that possess the *vhpiFullNameP* property.
A scope reference handle of NULL denotes the top level of the design. The reference handle can be any of the packages instances or root instance of the design hierarchy or any region handle in the entire design. Any other type of handle to an instantiated model object is an illegal scope reference.
The vhpiFullNameP property is similar to the VHDL `path_name attribute, but has been extended to prevent many of the ambiguous name references of `path_name. These ambiguities arise from region labels that are optional and anomalies in the LRM concerning `path_name. For example, unlabelled equivalent process and overloaded subprogram calls are disambiguated in vhpiFullNameP.

vhpiFullNameP is defined for explicit and implicit declarations.
vhpi_handle_by_name() can be used with names denoting declarations, indexedNames, selectedNames, attrNames denoting user defined attribute names, or signal valued attributes which denote signals (predefined attributes which vhpiAttrKindP property return vhpiSignalK).
For slice names, vhpi_handle_by_name is only able to find handles for slices that are visible in the original VHDL source. Indices if specified in the name, must only involve literals.
vhpi_handle_by_name of a vhpiFullname to a subprogram identifier name returns the subprogram declaration (vhpiFuncDeclK or vhpiProcDeclK) except if there is a dynamically elaborated subprogram call of the same name in the same region, in that case the subpCall shadows the subpDecl
Additionnally if the handle to look up denotes an enumeration literal that is overloaded, the vhpiSignatureNameP property must be appended to the vhpiFullNameP of the enumeration literal in order to return unambiguously a handle.

6.2.1.1 Find by Absolute Path Name

The vhpi_handle_by_name function uses a case insensitive comparison of the search string to the vhpiFullNameP value, except where a component of the name uses extended identifiers. That component of the name will be compared with case sensitivity.
An absolute path name is indicated by a search string which begins with ':' or '@' (@ for elaborated package references). The name provided must exactly match the vhpiFullNameP of an object, whose handle will be returned.
The scope reference handle must be consistent with the search string, i.e., either be a NULL handle, the handle of a region along the path to the desired object, or an elaborated package instance in which the object can be found.

The scope reference handle functions to narrow the domain of the search. For a region handle, the search is restricted to the region, any of its internal region, or their sub regions. For an elaborated package, the search is restricted to the package's elaborated declarations.

6.2.1.2 Find by Relative Path Name

This is indicated by a search string which does not begin with a ':' or '@' character. The search string will produce the same result as if the vhpiFullNameP of the scope reference handle was prepended to the search string, formulating an absolute search. The scope reference handle shall not be null.
The scope reference handle effectively establishes the top most region or an elaborated package context from which to search. There is no manner in which a relative name search will find an object above this region or package. In particular, this is not a search that mimics VHDL name resolution, in which some declarations outside an immediate region might be visible.

6.2.2 Uninstantiated Model Access (Library unit access)

Having access to the uninstantiated model in VHPI is a separate capability identified as the "vhpiProvidesPostAnalysis" capability (see 1.1.2). The UML description and name properties provide the framework for vhpi_handle_by_name working uniformly across both information models.
In the uninstantiated context, the vhpiFullNameP returns a string identical to the same string returned by the vhpiDefNameP property.
Additionnally if the handle to look up denotes an enumeration literal or a subprogram call that is overloaded, the vhpiSignatureNameP property must be appended to the vhpiFullNameP of the enumeration literal or subpCall in order to return unambiguously a handle otherwise the specification of the returned handle by vhpi_handle_by_name is not specified by the standard.

Deleted: .

The string value of the vhpiDefNameP property passed to vhpi_handle_by_name returns a handle to a post-analysis object.

6.2.2.1 Find by Absolute Path Name

This is indicated by a search string which begins with'@'.   The name provided must match the vhpiDefNameP of an object whose handle will be returned.
The vhpi_handle_by_name function uses a case insensitive comparison of the search string to the vhpiDefNameP value, except where a component of the name uses extended identifiers;that component of the name will be compared with case sensitivity.
The scope reference handle must be consistent with the search string, i.e., either be NULL, the handle of the logical library along the path to the desired object, or the design unit containing the desired object.
The scope reference handle functions to narrow the domain of the search. For library access, it allows the search to be restricted to a specific logical library or design unit.
A null scope reference handle indicates that the search is done over all the logical libraries known to the tool.

6.2.2.2 Find by Relative Path Name

This is indicated by a search string that is recognizable as a relative path, meaning it does not begin with a '@'. The search scope reference handle shall not be null. The search string will produce the same result as if the vhpiDefNameP of the scope reference handle was prepended to the search string, formulating an absolute search.

7. Foreign models interface

This chapter describes how to interface VHPI/ANSI-C foreign models or applications with a VHDL tool. It describes the specification, invocation and execution of foreign VHPI models and applications. This chapter is organized as follows:

- section 7.1: overall flow of execution of a mixed VHDL/VHPI foreign design,
- section 7.2: VHDL specification of foreign models,
- section 7.3: registration of VHPI foreign models and applications,
- section 7.4 : elaboration of VHPI foreign models,
- section 7.5: execution of VHPI foreign models and applications,
- section 7.6: VHDL context passing
- section 7.7: save, restart and reset of foreign models.

7.1 The phases of execution of a VHDL/VHPI mixed design

As defined by VHDL, there are two kinds of foreign models, architectures and subprograms which can themselves be either functions or procedures. VHPI foreign models have a VHDL declaration and a VHPI/ANSI-C behavior implementation. The VHDL declaration denotes the interface of the foreign model and other local declarations: for a foreign entity/architecture, it includes the entity and architecture declarative parts, and for a foreign subprogram, it includes the subprogram declaration (if present), or subprogram specification (if no declaration). Note that a foreign subprogram does not need to have a subprogram body. VHPI also provides the capability of invoking a foreign VHPI application at a given point during a VHDL session such as analysis, elaboration or simulation. That application is an autonomous VHPI program that can run concurrently with the simulation or can be called at a defined point in time during a VHDL session. Typical third-party tool applications such as signal monitoring, VHDL code profilers, hierarchy browsers can be developed and integrated as VHPI foreign applications.

We distinguish several phases in the life of foreign models and applications:

1. Registration
2. Elaboration
3. Initialization
4. Simulation run-time execution
5. Termination
6. Save/restart or reset.

The following paragraphs describe the different phases in details. All phases except for the registration and save/restart/reset phases, refer to already known phases in a VHDL tool.

Registration:

Registration is the first stage of execution of a VHDL/VHPI session. VHDL tools must support a registration mechanism for the VHPI foreign models and applications. The registration phase must occur before the vhpiCbStartOfTool callback which is the first defined time point for callback registration. Typically a developer of foreign models and applications would have to provide a C dynamic or static library which contains the compiled code of the VHPI based models and applications and at least a function to register the foreign models and applications into a VHDL session. After registration, the foreign model and application behaviors are defined, that is, a given VHPI based implementation has been associated with a foreign model or application. The required associations are defined in section 7.3. The registration of a model or application records the association(s) between C behavior and a VHDL model and does not necessarily do the symbol binding. The symbol binding may occur between the registration phase and the phase when the symbol functions need to be used. For VHPI applications the symbols need to be resolved before the vhpiCbStartOfTool callbacks.

NOTES:

1 - The registration function or also called bootstrap function is the only required globally visible entry point for registering a library of VHPI C models or a foreign application.

2 - There is no predefined name for the bootstrap function. The name of the bootstrap function must be supplied to the VHDL tool using an implementation defined mechanism.
3 - There is no predefined name for the foreign libraries. As a consequence, several libraries can be registered in the same VHDL session.
4 - The tool has the flexibility to determine when the binding of the model name to the C functions occurs (during registration or late during elaboration/simulation). The VHDL tool is free to register all the foreign models included in a library at once, or to register them one at a time when the foreign model is encountered during elaboration. However, the various C functions providing elaboration, initialization or simulation of a foreign model need to be known at the execution of the given phase.
5 - A library of foreign models or a foreign application may have several bootstrap functions.
6 - The binding can occur anytime from the point of registration until the point of use. The point of use of an application is immediately prior to the vhpiCbStartOfTool callback.

Elaboration:
Elaboration consists of the creation of the model instances that are involved in a VHDL design. Elaboration occurs after the model that is being elaborated has been registered and before simulation initialization. The declarative parts of VHPI foreign architectures are statically elaborated while the formal parameters of VHPI foreign subprograms are dynamically elaborated (there is no elaboration of the subprogram declarative part). The elaboration of the foreign models including the elaboration of the declarative and statement body parts is defined in section 7.4. It is an error if when elaborating of a foreign model, the C function for the elaboration of the foreign model is not known or not found. Initial values computed during the elaboration of an object declaration may involve the execution of a foreign function. Creation of a VHPI process and driver may be done during elaboration of a foreign architecture. The initial values of the ports of the foreign models or driving values of the VHPI drivers can be set with vhpi_put_value by the elaboration function of a foreign architecture.

Initialization:
Simulation initialization refers to the phase where signal driving and effective values are initially computed and processes are executed for the first time. The computation of a signal effective or driving value may involve the execution of a foreign function (resolution or conversion). The initialization function of each foreign architecture gets called during the initialization phase. The simulation function of a foreign subprogram may get called as a result of process execution. It is an error if the initialization function of a foreign architecture or simulation C function of a foreign subprogram is not found at the initialization phase if it needs to be executed.

Simulation runtime execution:
Simulation consists of the execution of the previously elaborated and initialized design. The execution of the foreign models or applications at appropriate times is accomplished through registration of callback functions for various reasons. These callbacks are either dynamically registered as the simulation proceeds or may have been registered during initialization. The control flow of a concurrent region defined by a foreign architecture can be emulated by callbacks.

Termination:
The termination phase is the last phase after which execution of other phases is not permitted. Termination consists of going back to a clean state before exiting the present VHDL session. The termination of a VHDL session can happen for several reasons (end of simulation, fatal error…). Termination is the last stage of execution of a VHDL session. Termination involves calling the registered *vhpiCbEndOfTool* callbacks. This callback reason gives the ability for foreign models and applications to take proper action to free any allocated resources, close files and terminate cleanly.

Save, Restart, Reset:
The mechanisms for saving, restoring or resetting foreign models are described in section 7.7. Foreign models and applications have the capability to save, restart or reset their state. These operations occur at a clean state of the simulation cycle: all scheduled events for that simulation cycle must be executed before the operation takes place.

Informative note: Foreign models should not assume that the memory they allocate or the files they open persist between any of these phases. In fact, any of these phases could belong to a different process.

7.2 Foreign models specification

The string of a foreign attribute that is decorating a VHPI foreign architecture or subprogram follows a VHPI standard syntax. There are two standard syntax: one which specifies an indirect binding of the foreign model behaviours and another one which specifies a direct binding of the foreign model behaviours.

7.2.1 Foreign attribute syntax

7.2.1.1 Standard indirect binding mechanism

Foreign VHPI architectures:

attribute FOREIGN of <architecture_name>: architecture is
“VHPI <library_name> <model_name>”

Foreign VHPI procedures and functions:

attribute FOREIGN of <subprogram_name[signature]>: procedure | function is “VHPI <library_name> <model_name>”

1) The "VHPI" identifier indicates that this foreign model has a VHPI based implementation and that the binding is a standard binding.
2) <library_name> is the logical name of the C library. The mapping to the physical library is implementation dependent.
3) <model_name> identifies a VHPI based model implementation for a foreign architecture or subprogram.
4) The foreign attribute string must be locally static and the string syntax must be as follows:

"VHPI {<space_character>} {graphic_character}{<space_character>} {graphic_character}"

The space character can either be the space or the non breaking space character.

<space_character> := SPACE | NBSP

The string should start by VHPI keyword and should consists of two sets of graphic characters separated by at least one space character.
VHDL LRM 1076-1993 modifications needed page 72 for foreign attribute specifications.

NOTES:
- The analysis of the foreign string does not yield any interpretation of the value of the foreign string.
- The elaboration of the foreign attribute specification may involve some checking depending on a vendor implementation (checking for the existence of the C library, and existence of the foreign model).
- The name of the C library containing the foreign models need not to be the same as the name of the VHDL logical library which contains the VHDL component declarations and entity shell declarations of the foreign model. In the example below, the VHDL library which contains the foreign model component declarations is the VHDL library “foreignmodels” while the C foreign models implementation live in a C shared library named foreignC.<platform dependent_suffix>.

Example:

```
-- VHDL source file containing the specifications of foreign components
-- and subprograms
-- the package containing the declarations
   package packshell is
     component C_and
```

```
      port(p1, p2: IN bit; p3: OUT: bit);
     end component;
  end package;

  -- the foreign procedure and function declarations
  procedure myproc(signal f1: OUT bit ; constant f2: IN integer);
   attribute foreign of myproc: procedure is
              “VHPI foreignC myCproc”;
  function myfunc(signal f1: IN bit) return integer;
  attribute foreign of myfunc: function is
              “VHPI foreignC myCfunc”;

end package packshell;

-- VHDL source file containing the design units
-- the entity/architecture declarations of the foreign architecture.
entity C_and is
  port (p1, p2: IN bit; p3: OUT: bit);
  end C_and;


  architecture My_C_gate of C_and is
    -- foreign attribute
  attribute foreign of my_C_gate :architecture is
            "VHPI foreignC myCarch";
  begin
  end architecture My_c_gate;


library foreignmodels; -- the VHDL library which contains the VHDL shell
                       -- declarations for entity/architectures/
                       -- subprograms
use foreignmodels.packshell.all;  -- use clause selecting a package in
                             -- the library

  entity top is
  end top;
  architecture my_vhdl of top is
    constant val: integer:= 0;
    signal s1, s2, s3: BIT;
  begin
    u1: C_and(s1, s2, s3); -- instantiation of a foreign VHPI model
C_and
    myproc(s1, val);       -- concurrent foreign procedure call
                           -- statement myproc;

    process (s1)
    variable va: integer:= val;
    begin
       va = myfunc(s1);   -- foreign function call myfunc
    end process;
  end my_vhdl;
```

7.2.1.2 Standard Direct Binding mechanism

This binding mechanism accomplishes both the registration and binding of the foreign model. A VHPI implementation may also provide direct binding to the C behavior by providing the C library and function names for the foreign model in the foreign attribute string.

In the case of foreign architectures, the string shall contain four tokens, each separated by one or more SPACE characters. Escaped identifiers can be used as tokens to specify platform specific names.

The first token shall be the keyword VHPIDIRECT.
The next token shall either be the NULL token or the name of the C library where the C functions modeling the behaviour of the foreign model are defined.
The next token shall be the name of the C function modeling the elaboration of the foreign architecture or NULL if none is required.
The next token shall be the name of the C function representing the initialization of the foreign architecture or NULL if none is required.
The library name shall not have any suffix.
Note 1: The library name is is useful to distinguish C functions of the same name living in different C libraries. If the library name is NULL, the search of the C functions is vendor specific.

In the case of foreign subprograms, the string shall contain three tokens, each separated by one or more SPACE characters.
The first token shall be the keyword VHPIDIRECT.
The next token shall either be the NULL token or the name of the C library where the C functions modeling the behaviour of the foreign model are defined.
The next token shall either be the NULL token or the name of the C function modeling the execution of the foreign subprogram. See Note 2.

Note 2: If the NULL token is given in place of the name of the execf function of the foreign, the name of the C function is assumed to be the name of the VHDL subprogram declaration in the case specified in the VHDL file.

Foreign VHPI architectures:

attribute FOREIGN of <architecture_name>: architecture is
"VHPIDIRECT <library_name> <elabf_name> <execf_name> "
where <library_name> <elabf_name> or <execf_name> can be the NULL tokens.

Foreign VHPI procedures and functions:

attribute FOREIGN of <subprogram_name[signature]>: procedure | function is
"VHPIDIRECT <library_name> <execf_name>"

where <library_name> and <execf_name> can be the NULL tokens.
The foreign attribute string must be locally static and must be a string that starts with the VHPIDIRECT keyword.

7.3 Registration

7.3.1 Delivery and packaging of libraries of foreign VHPI models or applications

The standard does not define how a foreign library name and its corresponding bootstrap function are exchanged between a VHDL tool and a library developer.
The standard requires that models and applications be delivered packaged into either C dynamic (shared) or static (archive) libraries.

Foreign models and applications can be registered through a tabular file. A tabular registry file shall be provided when using the VHPI indirect foreign attribute syntax,

7.3.1.1 Tabular registry format

The tabular registration consists of providing a textual registry file which contains the registration information of foreign models and applications in a standard defined format. Each line of the registry table defines the registration of exactly one model or application or library of models.
The format of each line of the registry file is the following:
For a foreign architecture:
<library_name> <model_name> vhpiArchF <elab_fctn_name> | null <initialization_fctn_name>
For a foreign subprogram:
<library_name> <model_name> vhpiFuncF | vhpiProcF null <execution_fctn_name> | null
For a foreign application:
<library_name> <application_name> vhpiAppF <bootstrap_fctn_name> null
For a library of foreign models:
<library_name> null vhpiLibF <boostrap_fctn_name> null

Comments may be included in the file, each comment line must start by a "--" character. The library, model, application and function names must be formed with graphical characters and can be extended identifiers. The elaboration, execution and bootstrap function names should be the C source function names. One or more spaces can occur between names. The null token should be entered in the place of a C function name if no function name is provided. In the case of a null execution_fctn_name for a foreign subprogram, the name of the function defaults to the name of the model name.
If several bootstrap functions are associated with a library, an entry for each bootstrap function must be in the registry file.

Example:

```
Registry_file contents example
--<library_name> <model_name> <kind> <elab_fctn_name> <sim_fctn_name>
-- registration of a foreign architecture
myClib orgate vhpiArchF     elab_gate  init_gate
-- registration of a foreign function
myClib myfunc vhpiFuncF     null       sim_myfunc
-- registration of a foreign application
myCapp app1 vhpiAppF  boot_myapp null
-- registration of a library of models
myClib null   vhpiLibF  bootlib    null

Example of a library of models function registration
void boot_lib
{
 for each model in the library
  vhpi_register_foreignf()
}
```

Note: The names, number of and locations of the registry files are not predefined by the standard.

7.3.2 Registration functions for foreign models and applications

7.3.2.1 Registration and binding of a foreign model

A bootstrap function which registers models of a library shall use the standard VHPI function *vhpi_register_foreignf()*. This function is called during the registration phase for each foreign model and provides information such as C function entry points for the various phases defined for this kind of foreign model. Different pieces of information are needed for a foreign architecture, procedure or function. A data structure of type *vhpiForeignDataT* is filled with the necessary information by the caller (bootstrap function). A pointer to that data structure is passed as an argument to *vhpi_register_foreignf().*

Note : The tabular registration of foreign architectures and subprograms should not use the function vhpi_register_foreignf(), as all the pieces of information required are expected to be in the tabular registration file.
Correct since it is a textual entry rather than a programmatical entry but the tool who is reading the tabular form may use vhpi_register_foreignf()

The registration of a foreign model corresponding to a foreign architecture should provide back to the VHDL tool the following pieces of information:

- the library name,
- the model name,
- the model kind,
- a function pointer to the elaboration function for the architecture statement part or null if there is no user-defined elaboration,
- a function pointer to the initialization execution function for the architecture statement part.

The registration of a foreign model corresponding to a foreign procedure or function should provide back to the VHDL tool the following pieces of information:

- the library name,
- the model name,
- the model kind,
- a function pointer to the simulation function for the procedure or function body statement part.

Procedural Interface References:

See “vhpi_register_foreignf()”.
See “vhpiForeignDataT”.
See “vhpi_get_foreignf_info()”.

7.3.2.2 Registration of foreign applications

Direct binding of foreign applications is unspecified by the standard.
The registration of a foreign application consists of executing the bootstrap function of the foreign application at the registration phase. A VHDL tool must execute the bootstrap function of each VHPI foreign application that is needed for a particular VHDL session. The standard only requires that the developer of the application provides the library and the name of its bootstrap/registration function. The function name should be a visible symbol of that library. It is left to the vendor and provider to determine how the library is bound (for example dynamic, static linking or dynamic loading from command line arguments).
Applications may be put in the tabular form which is the standard registration mechanism. If they are used they must be bootstrapped before the vhpiCbStartOfTool callback. An implementation can define the way a user may indicate if an application is used for that session. The boostrap function may register callbacks as defined in chapter 8.
vhpi_register_foreignf() can also be used for registering an application.
The registration of a foreign application should provide back to the VHDL tool the following pieces of information:

- the library name,
- the application name,
- the model kind, (in that case vhpiAppKind)
- a function pointer to the main function of the application program.

Procedural Interface References:

See “vhpi_register_cb()”

See “vhpiCbDataT”.

7.3.3 Registration and binding errors

1. Library name and/or model name used by a foreign attribute is not found in the registry.
2. The library cannot be located.
3. The registered C functions cannot be bound. This applies to functions registered for applications, libraries or models.

7.3.4 Restrictions

During the registration phase, the only part of the VHPI information model that can be accessed is the tool class (including the tool class properties, methods and operations). Other operations allowed are calls to *vhpi_register_foreignf(), vhpi_get_foreignf_info()* and *vhpi_register_cb(), vhpi_get_cb_info()*; registration of callbacks is also restricted to reasons that do not require to provide handles to elaborated objects. Also permitted are VHPI function calls to print (*vhpi_printf()*), check VHPI error *vhpi_check_error()*, and emit an assertion *vhpi_assert()).*

7.4 Elaboration of foreign models

7.4.1 Elaboration of foreign architectures

The elaboration of foreign architectures involves the elaboration of the declarative part of the entity and architecture.
Note: 156 and 157 of the VHDL LRM 1076-1993 need to be changed. The VHDL tool shall call the optional elaboration function which was registered for the foreign architecture.

7.4.2 Elaboration function

The elaboration function of the architecture is called in place of the elaboration of the statement body part of the foreign architecture. The prototype of the elabf function shall be identical to the prototype of callback functions:

```
PLI_VOID elabf(const vhpiCbDataT * cbDataP );
```

The *elabf* function is called with a pointer to a callback data structure of type *vhpiCbDataT*, the obj field should be a handle to the architecture instance (*vhpiCompInstStmtK* or *vhpiRootInstK*). The reason code shown by the cbDataP argument (type vhpiCbDataT *) of the *elabf* function shall be “*vhpiCbStartOfElaboration*”. From the instance handle, information pertaining to this architecture instance and its elaborated entity can be obtained: for example elaborated ports and generics of that instance The region class diagram depicts the relationships that can be traversed; the access permitted is described below.
The elabf() function can:
- Access the current elaborated component instance and all its elaborated declarative part.
- Access the value of any of the instance declared items (including the generic propagated value)
- Create foreign drivers *(vhpiDriverK)* and foreign processes *(vhpiProcessK)* for that instance.
- Set the initial driving value of output ports and internal signals with vhpi_put_value().

7.4.3 Elaboration of foreign subprograms

The elaboration of foreign subprograms involves dynamic elaboration of the subprogram. The subprogram formal parameters are elaborated when the subprogram call statement is encountered. No special elaboration C function entry point is needed. The “*elabf*” function pointer of the registration for a foreign subprogram shall be NULL.
Note: LRM modifications needed pages 156, 157, 163.

A foreign function can be called during elaboration phase to initialize declared items. The *execf* function is used to provide the initial value of the declared object.
Note: LRM modification needed for elaboration of declared objects involving foreign functions.

7.5 Simulation run time execution

7.5.1 Simulation of foreign architectures

During non-postponed process execution phase of the simulation initialization, the *initialization* functions of the foreign architectures are executed. This is the ONLY time the initialization function is invoked automatically by the simulator.

7.5.2 Initialization function

Architectures execution will be started once automatically at the simulation initialization phase by the invocation of the "*execf*" function and must sustain themselves throughout the entire simulation session by registering other C callback functions for simulation event reasons. The initialization function is specified by the *execf()* function. The prototype of the *execf* function is identical to the prototype of callback functions.

```
PLI_VOID execf(const struct vhpiCbDataS * cbDataP  );
```

The *obj* field of the cbDataP argument should be set to the handle of the architecture instance that is initialized (*vhpiCompInstStmtK*, or *vhpiRootInstK*). The reason code of the "*execf*" function should be *vhpiCbStartOfInitialization*. Memory allocated by the foreign architecture can be stored in the *user_data* field of the cbDataP of future registered callback functions.

The initialization function has access to the entire design, and has access to any VHPI function. There is no restriction on what the initialization function can do except calling vhpi_register_foreignf (registering a foreign model).

Informative note: there is no further control-flow for this foreign architecture instance except for callbacks that are registered during the initialization by the *execf* function.

7.5.3 Simulation of foreign subprograms

When a foreign subprogram call is encountered during VHDL execution, the simulation execution function is called: the control flow of a foreign subprogram call is determined by the VHDL simulation semantics.

7.5.4 Execution function

The simulation function for a foreign subprogram is specified by the *execf* function. The prototype of the execution function is identical to the prototype of a callback function.

```
PLI_VOID execf(const vhpiCbDataT * data  );
```

The *obj* field of the callback data structure of type *vhpiCbDataT* should be a handle to the subprogram call being executed (*vhpiFuncCallK* or *vhpiProcCallStmtK*). The reason code for a foreign subprogram call should be: "*vhpiCbStartOfSubpCall*". The *user_data* field has no defined value.
There is no restriction on what the execution function can do. Handles to dynamically elaborated objects are only valid for the duration of the subprogram call. The user should not expect these handles to be valid

after the subprogram call has returned, nor that these handles be the same ones the next time the same subprogram call is executed. Note that the call-data context may be different for each subprogram call. Some vendors may do static elaboration of concurrent subprogram class and therefore handles to objects living in subprograms may be valid across subprogram calls. A property vhpiIsInvalidP is provided to check the validity of a handle.

Foreign functions must return a value. *().* In order to set the return value of a function, the function call handle is the handle that should be used to set a value through *vhpi_put_value*. If the function return type is composite other than an array of scalars, then the users should iterate over the call handle to get to the level of a scalar or an array of scalars in order to set the return value. In the case of the return type being an array of scalars a single call to *vhpi_put_value* can be used to set the return value. The case of a function returning an unconstrained array requires a different treatment as outlined in the following paragraphs.

If the function returns an unconstrained array of scalars, then a single call to *vhpi_put_value* will suffice, in order to set the values of all scalar subelements. Alternatively, the user can choose to set the number of elements that the interface should expect in the return value, and then iterate over these elements using the relation *vhpiIndexedNames* and set each of the scalars separately. The function call handle will be the reference handle for this iteration. In order to achieve this, the first call to *vhpi_put_value* should use the flag *vhpiSizeConstraint*, with the *numElems* field in the value stucture containing the number of elements that the interface should expect.

If the function returns an unconstrained array of composites, then the users are required to call *vhpi_put_value* with the flag *vhpiSizeConstraint* and set *numElems* in the value structure to the number of elements that the function will return. Subsequently, users can iterate over the subelements of the array using the relationship *vhpiIndexedNames* on the function call handle and set the value of each subelement separately.

For all calls to *vhpi_put_value* that set function return values, the flag parameter can be either *vhpiDeposit, vhpiDepositPropagate, vhpiForce or vhpiForcePropagate*. The semantics of setting the return value of the function are all exactly the same irrespective of the flag used. The return value is available immediately after the function return within the context of the expression that has the function call. If the function call is on the right hand side of a signal assignment statement, then the return value will be used in scheduling a transaction on that signal upon function return. The flag *vhpiRelease* has no effect when used with a function call handle or on any of the subelements of a function call handle for composite return types. In that case, an error should be generated.

The VHPI interface will check that the number of elements passed using *vhpiSizeConstraint* with *vhpi_put_value* matches the number of subelements that are actually set by the user with subsequent calls to *vhpi_put_value*, the interface should issue a warning in case of a size mismatch. If the number of elements returned by a function does not match the number of elements expected from the function within the context of the call, a runtime size error will be issued by the tool.

Calls to *vhpi_put_value* with *vhpiSizeConstraint* replace the previous size constraint for the specified reference handle. For vector of scalars, setting an explicit constraint is not necessary as a call *to vhpi_put_value* which supplies a vector of scalars will define an implicit constraint from the number of elements in the vector. However if an explicit size constraint was previously set, it is an error if the implicit constraint is different from that explicit size constaint.

An example to illustrate the use model for functions returning unconstrained types.

```
function foo(p1 : in std_logic;
             p2 : in std_logic)
             return std_logic_vector is
begin
end;
```

```
attribute foreign of foo:function is “VHPIDIRECT:mylib:fooC”;

signal bar : std_logic_vector := foo(‘0’, ‘1’);
signal foobar : std_logic_vector(3 downto 0);

P : process (clk, reset)
begin
  if (reset = ‘0’) then
    foobar <= “0000”;
  elsif (clk’event and clk = ‘1’) then
    foobar <= foo(‘1’, ‘0’);
  end if;
end process;
```

In the first call to function foo, where it is the initialization expression for a signal declaration, the return value can be any number of scalars. The range and direction of the constrained anonymous subtype of signal bar will be determined by the implementation, while the size will come from the foreign implementation of foo. In the second call to function foo, where it is used on the right hand side of a signal assignment statement, the size has to be four. The constraints (3 downto 0) is assumed by the implementation. In order to set the return value in either of the two calls to foo, the VHPI foreign function can either use a single call to *vhpi_put_value* or use a *vhpi_put_value* with *vhpiSizeConstraint* and then iterate over the scalar subelements and set them one at a time.

```
function finit(p1 : in std_logic)
               return std_logic_vector is
begin
end;

attribute foreign of finit:function is “VHPIDIRECT:mylib:finitC”;

function foo(p1 : in std_logic)
             return std_logic_vector is
begin
end;

attribute foreign of foo:function is “VHPIDIRECT:mylib:fooC”;

signal bar : std_logic_vector := finit(‘1’);
signal foobar : std_logic_vector(3 downto 0);

P : process (clk, reset)
begin
  if (reset = ‘0’) then
    foobar <= “0000”;
  elsif (clk’event and clk = ‘1’) then
    foobar <= foo(foobar(0));
  end if;
end process;
```

In the call to function finit, where it is the initialization expression for a signal declaration, the return value can be any number of scalars. The range and direction of the constrained anonymous subtype of signal bar will be determined by the implementation, while the size will come from the foreign implementation of finit. In the call to function foo, where it is used on the right hand side of a signal assignment statement, the size has to be four. The constraints (3 downto 0) is assumed by the implementation. In order to set the return value, the VHPI foreign functions can either use a single call to *vhpi_put_value* or use a *vhpi_put_value* with *vhpiSizeConstraint* and then iterate over the scalar subelements of the call handle and set them one at a time.

```
void finitC(vhpiCbDataT* pCbData)
{
  int i;
  vhpiValueT value;
  vhpiHandleT param;
  vhpiHandleT callHandle;
  vhpiHandleT subelement;

  // get the call handle and the first and only parameter
  callHandle = pCbData->obj;
  param = vhpi_handle_by_index(vhpiParamDecls,
                               callHandle, 0);

  // get the value passed into this call
  value.format = vhpiEnumVal;
  vhpi_get_value(param, &value);
  vhpi_release_handle(param);

  // set the size constraint to be eight, to indicate we intend
  // to return a vector of eight elements
  value.numElems = 8;
  vhpi_put_value(callHandle, &value,
                 vhpiSizeConstraint);

  // we don't have to set anything in the value structure at this
  // point as we intend to use the value passed in as the value of
  // each subelement of the vector going out

  // iterate over the subelements of the call handle and set each
  // of them
  for (i = 0; i < 8; i++) {
    subelement = vhpi_handle_by_index(vhpiParamDecls,
                                      callHandle, i);
    vhpi_put_value(subelement, &value, vhpiDeposit);
  }
}

void fooC(vhpiCbDataT* pCbData)
{
  int i;
  vhpiHandleT param;
  vhpiHandleT callHandle;
  vhpiValueT value;
  vhpiEnumT enumval;
  vhpiEnumT vector[4];

  // get call handle and the first parameter
  callHandle = pCbData->obj;
  param = vhpi_handle_by_index(vhpiParamDecls,
                               callHandle, 0);

  // get the value of the parameter
  value.format = vhpiEnumVal;
  vhpi_get_value(param, &value);
  vhpi_release_handle(param);

  // we intend to return a four element vector where each scalar
  // subelement is the flipped version of what was passed in
```

```
  for (i = 0; i < 4; i++)
    vector[i] = (isStdLogicZero(value.value.enumval))? vhpi1 : vhpi0;

  // set the return value in one shot
  value.format = vhpiEnumVecVal;
  value.value.enums = vector;
  value.numElems = 4;
  vhpi_put_value(callHandle, &value, vhpiDeposit);
}
```

7.5.5 Restrictions and errors

Any VHPI function except *vhpi_register_foreignf* can be called by foreign subprograms and functions. Scheduling a zero delay transaction with *vhpi_schedule_transaction* or *vhpi_put_value* should generate a runtime error if called during postponed process phase.
Informative notes: Foreign function can reach and update out-of-scope objects. The behaviour of foreign functions declared as pure is not checked or enforced by the VHDL tool.

Procedural Interface References:

See "vhpi_register_foreignf()" for registering foreign models.
See "vhpiForeignDataT" for passing foreign model information.
See "vhpi_put_value()"for setting the returned value of a foreign function call.

7.6 Context passing mechanism

This section describes the mechanism by which interface and parameters are passed between the VHDL and foreign-C functions, and for the case of foreign functions, how return values are passed back to the VHDL tool.

The foreign C-function prototypes bear no relationship to the number or types of parameters used in the VHDL declaration, parameters are accessed indirectly through the VHPI traversal methods if used when the call to the C run-time function is executed.
A handle to the architecture instance or subprogram call is passed as the *obj* field of the callback data structure which is the single argument of each *elabf* or *execf* function.
VHPI access functions can be used on that handle.

Some VHPI methods which can return foreign instances, subprogram calls or callbacks can be used to transfer context between VHDL and C:

vhpiCurRegion shall return the currently active executing region instance from which the foreign model call derived. vhpiCurRegion can return a foreign architecture instance or the foreign function procedure call being executed.

7.6.1 Architecture instance

The architecture instance handle is passed through the obj field of the cbDataT structure. During elaboration, only access to the entity architecture elaborated items are allowed.
During initialization of the architecture behaviour, all VHPI access is allowed. During initialization, the architecture installs its behaviour by registering future callbacks; memory allocated by the foreign architecture instance can be stored in the user_data field of the callback data structure of the registered callbacks.

7.6.2 Subprogram Calls

When the flow of execution of VHDL code encounters a foreign subprogram call, call instance (context) information must be passed to the C-function implementation of the subprogram in order to perform the desired behavior. Context information is passed in a pointer to a callback data structure of type *vhpiCbDataT*. Information is provided as a handle to the VHDL subprogram call from which all necessary context information such as formal parameters can be obtained. Available data access is described by the class diagram for subprogram calls (refer 4.8.2). Because the subprogram call is an instance of a foreign subprogram, some relationships such as iteration on the sequential statements, will return NULL.
Informative note: As a good programming rule, the foreign function code should access the formal parameters of the VHDL declaration and not attempt to reach outside the VHDL subprogram scope.

For foreign subprograms that are functions, a return value must be passed back to the caller. The VHPI function *vhpi_put_value()* should be used to indicate the return value of the function call. It will be a run-time error if :
- the C function fails to return the value,
- the size indicated by *vhpi_put_value* with *vhpiSizeConstraint* flag does not match the size of the value set by the subsequent *vhpi_put_value* calls (case of a foreign function returning an unconstrained array type),
- the value is the wrong size for the context of the call.

VHDL formal parameters are accessed through handles by traversing the relationships depicted by the class diagram of subprogram call. The VHDL passing mechanism for IN, INOUT and OUT parameters applies also to foreign subprograms.
The VHDL language observes some special handling of parameters in respect to the parameters being of mode IN, OUT or INOUT, as well as of class “constant”, “variable”, "signal" or "file".

The VHDL LRM 1076-2001 defines the following mechanisms which are being amended by VHPI
1. Values of **input** parameters of class **variable** or **constant** are **copied** from the actual parameters to the formal parameters at the **beginning** of the execution of the procedure or function. vhpi_get_value applied to formal parameters of input or inout mode shall return the value of the actual parameter as set at the beginning of the execution of the subprogram.
2. Values of the **output** parameters of class **variable** are **copied** from the formal parameters to the actual parameters at the **end** of execution of the procedure. Values deposited on output or inout formal parameters by vhpi_put_value are copied to the actual parameters at the end of the execution of the subprogram call.

When a parameter is of class **signal**, a **reference** to the actual signal parameter is **passed** into the procedure. This implies that changes made to the actual OUTSIDE of the procedure call will be **reflected** in the formal, similarly, if the formal is modified, the actual will reflect that change. In particular, transactions scheduled on a driver of the formal signal parameter of mode OUT are equivalent to transactions scheduled on the actual signal and vice-versa. Transaction scheduling from the C foreign code is performed with the function *vhpi_schedule_transaction()* or *with vhpi_put_*value (*vhpiDepositPropagate* or *vhpiForcePropagate*) *vhpi_put_value* with the other flags (*vhpiDeposit*, *vhpiForce*) deposits or forces a value on the actual signal.

Parameters of class **file** are also passed by reference. The opening mode of the file may be specified explicitly by the file declaration or be the default mode. VHPI access to a file parameter declaration is defined by the information model. Any other operation on a file parameter declaration is undefined.

Handles representing dynamic elaborated objects belonging to the subprogram call are only valid during the subprogram call execution. The user should not assume that the objects refered by these handles exist after the subprogram call completes neither that the same handles will be returned by the interface for the same subprogram call later during simulation. These handles are only valid (methods properties and operation can be obtained while the subprogram is active).

vhpi_get_value() method can be applied to formal parameters of mode IN or INOUT; this accesses the value of the VHDL formal parameter.
vhpi_put_value() method can be applied to formal parameters of mode OUT or INOUT; it will update the value or schedule a zero delay transaction on the VHDL formal parameter depending on the flags and class of the parameter. *vhpi_schedule_transaction* can be applied to a formal signal parameter of mode OUT or INOUT.

Note: Page 20 and 21 of the LRM has to be updated with foreign subprograms.

Procedural Interface References:

See *vhpi_get_value()* to access the value of a formal parameter of mode IN.
See *vhpi_put_value()* to deposit a value into a formal parameter of mode OUT.
See *vhpi_schedule_transaction()* to schedule a new transaction to a signal formal parameter of mode OUT.
See *"vhpi_register_cb()"* to register callbacks.
See *"vhpi_handle_by_index()"* to access a given formal parameter handle in the ordered iteration formal list.

7.7 Save, Restart and Reset

If the simulator supports save, restart and reset of a VHDL design then this capability can be extended to enable the foreign models and applications to save their state, restore from a simulation checkpoint or reset to time zero. In order to support this functionality, VHPI has defined save, restart and reset callback reasons as well as VHPI functions to write to *(vhpi_put_data())* and read from *(vhpi_get_data())* a saved checkpoint location. A foreign model/application that is interested in saving its state must register a callback for save. VHPI provides two callback reasons for each save, restart or reset action. The *vhpiCbStartOfSave, vhpiCbStartOfRestart, vhpiCbStartOfReset* callback functions are called respectively at the beginning of a save, restart or reset operation, while the *vhpiCbEndOfSave, vhpiCbEndOfRestart, vhpiCbEndOfReset* are called respectively at the end of the save, restart or reset operations. This is provided as a convenience to the user so that actions that need to be serialized can be guaranteed to happen in the correct order.
Note: A model does not need to register for both startOf and endOf reasons.
Foreign applications can also use the save/restart callbacks to save and restore their data from the simulation save location (see 7.7.4).

7.7.1 Saving foreign models

The standard requires that a compliant VHPI implementation save at least the restart callbacks during a save operation. This is so that when a restart operation is initiated all the models and applications that saved data are given the opportunity to restore saved data through their restart callbacks.
The standard also allows implementations to save handles, callbacks and user data and restore all of these with referential integrity. A tool vendor may indicate his capabilities to save and restore this information through the *vhpiAutomaticRestoreP* property. This property is an integer valued set of flags which can be queried from the tool class. The property expected values are vhpiRestoreAll, vhpiRestoreHandles, vhpiRestoreCallbacks, and vhpiRestoreUserData. A return value of *vhpiRestoreUserData* for *vhpiAutomaticRestoreP* implies that the tool will automatically save all user data in memory at the point of save and restore all data back into memory with referential integrity at the point of restart. For each flag of this property that is not set the user is responsible for saving necessary data to be able to recreate the state of the simulation at restart.

VHPI provides a function to write data in a save location:

```
PLI_INT32 vhpi_put_data(PLI_INT32 id, PLI_VOID *dataLoc, PLI_INT32 numBytes)
```

numBytes: the number of bytes to write out, must be greater than zero.
dataLoc: the address of the data to be saved.

id: a unique identifier of the location of the saved data that is used to retrieve the data during a restart operation. A new id is obtained by calling vhpi_get(vhpiIdP, NULL).
returns: the number of bytes saved.

The function will write "*numBytes*" of the data starting at "*dataLoc*" into a simulation save location. The argument *id* identifies the saved foreign data set. This id determines how the data is written in the file. Data from multiple calls with the same id MUST be stored by the simulator in a manner that allows the opposite routine *vhpi_get_data()* to pull out the data of the same id in the order it was put in; data with different ids can be retrieved in any order.
Note 1: This allows the restart operation to be independent of the order the foreign models were saved.
The function returns the number of bytes that were successfully saved.
Note 2: The caller is responsible for determining if the number of bytes written corresponds to the number of bytes requested to be written.

The behaviour of *vhpi_put_data()* is only defined when invoked from a callback function that was registered for reason *vhpiCbStartOfSave* or *vhpiCbEndOfSave*.
There are no restrictions on:
* how many times the *vhpi_put_data()* function can be called for a given id,
* how many ids, a foreign model can create,
* the order the foreign models put data into the saved location using different ids.

It is an error if the id passed in is zero or is an unknown id for this simulation session. It is an error if numBytes is equal to zero.

See example in the procedural interface reference for *vhpi_put_data()*.

The property tag *vhpiIdP* is defined vhpi_get(vhpiIdP, NULL) will generate and return a unique id to the caller. The id is unique for the simulation session. The id returned is different from zero. The id corresponds to an area in a saved location. The id is then used by vhpi_put_data to indicate where to save or by vhpi_get_data to indicate from where to restore data. vhpi_get(vhpiIdP, NULL) can only be called during a save operation. *vhpi_put_data() and vhpi_get_data()* functions will check if the id passed in is a legal id for this simulation session.
The id for a given foreign model or application should be different for each save operation of the same simulation session allowing to save different simulation checkpoints.
Can I use same id across different check points?

7.7.2 Restarting foreign models

VHPI provides a function to read data from a saved location:

```
PLI_INT32 vhpi_get_data(PLI_INT32 id, PLI_VOID *dataLoc, PLI_INT32 numBytes)
```

numBytes: the number of bytes of data to retrieve, must be positive.
dataLoc: the address of the data in which to place the data read.
id: a unique identifier that is used to specify the location of the data to read from the saved file. Must be greater than zero.

returns: the number of bytes retrieved.

The function will read "numBytes" from a simulation save location and place it at the address pointed by dataLoc. The memory for dataLoc must have been properly allocated by the caller and must be sufficient to

hold the data read. The first call for a given “id” will retrieve the data starting at what was placed into the save location with the first call with the same id to vhpi_put_data(). The return value will be the number of bytes retrieved. Each subsequent call will start retrieving the data where the last call left off for the given id. The vhpi_get_data() function can ONLY be called from a callback function that was registered for reason vhpiCbStartOfRestart or vhpiCbEndOfRestart.
Callbacks for both restart reasons must be registered by either the callback user functions of reasons vhpiCbStartOfSave of vhpiCbEndOfSave. A compliant VHPI implementation will hence support a user’s intent to pass any unique ID for a particular save operation to the associated restart callbacks, so that they can faithfully restore saved data using one or both the restart callback reasons. Any type of data (for example int, long, char) can be retrieved, but pointer links and structures may need to be manually rebuilt on a restart. It will be a warning for the foreign model to retrieve more data than what was placed into the simulation save location for a given id. If this happens, the dataLoc will be filled with the data that is left for a given id and the remaining bytes will be filled with ‘\0’. It is acceptable for a foreign model to retrieve with a given id less data that what was stored for that id. It is an error if the id passed in is zero or an unknown id.
The vhpi_get_data() function MUST only be called from a callback routine that was registered for reason vhpiCbStartOfRestart or vhpiCbEndOfRestart.

See example in the procedural interface reference for vhpi_get_data().
Clarify when the id is obtained and passed to the restart operation. Save/restart model with ids. Where is the restart registered? Save is repetitive.

The restart sequence consists of

1. The tool loads the saved model,
2. The tool executes the vhpiCbStartOfRestart callback functions,
   The VHPI client application checks the restore capabilities of the tool by querying the vhpiAutomaticRestoreP property.. Handles, callbacks and/or user data may not be restored by the tool:
   automatic = vhpi_get(vhpiAutomaticRestoreP, toolH))
   The vhpiAutomaticRestoreP property returns an integer value which indicates what has been restored; the defined standard integer values are defined by the enumeration type vhpiAutomaticRestoreT.
   Depending on what is not automatically restored by the tool, he client or application may need to re-register the foreign models callbacks during the restart operation (remap the function pointers to the C functions), re obtain handles or rebuild its user data using vhpi_get_data().

3. The tool executes the vhpiCbEndOfRestart callback functions,

4. T_c = T_s (time of the save). The tool starts simulation from the point of save.

Deleted: vhpiAutomaticRestoreT ¶

```
typedef enum {
       vhpiRestoreAll    = 7,
       vhpiRestoreUserData        = 1,
       vhpiRestoreHandles    = 2,
       vhpiRestoreCallbacks = 4,
} vhpiAutomaticRestoreT ;
```

Deleted: ¶

Formatted: Bullets and Numbering

7.7.3 Reset of foreign models state

On a reset operation, after the vhpiCbStartOfReset callbacks have been executed, a compliant VHPI implementation shall remove all callbacks with the exception of *vhpiCbEndOfReset* callbacks. The removed callbacks will include those that were registered prior to simulation initialization. The

vhpiCbEndOfReset callback of a client or application will have to register all callbacks required to exist during and after simulation initialization.
The reset operation winds the simulation time back to zero and to the beginning of initialization, corresponding to step 1.0.1 in the annotated simulation cycle. Whenever a reset operation is initiated, a compliant VHPI implementation will first execute all callbacks registered for reason vhpiCbStartOfReset. The vhpiCbStartOfReset can be thought as "prepare for reset, clean up data structures, release any handle which will become invalid after reset". After te vhpiCbStartOfReset callbacks have executed, the current simulation time will be rewound to zero, followed by the execution of all user registered vhpiCbEndOfReset callbacks. The initialization phase corresponding to step 1.0.1 in the annotated simulation cycle will then be initiated. No callbacks other than vhpiCbEndOfReset shall be remaining after the simulation time has been reset to 0 due to a reset operation. This means that any callback registered by the user before simulation initialization is not required by the standard to remain between or after vhpiCbStartOfReset and vhpiCbEndOfReset callbacks are run. The callback reason vhpiCbStartOfReset is provided to client applications as a point at which they can cleanup any memory that has been allocated or any state dependent data in memory. The callback reason vhpiCbEndOfReset is provided to enable client applications to re-register any callbacks that existed at the start of initialization, including any vhpiCbStartOfInitialization callbacks before the simulator begins execution of the initialization phase. As all foreign model initialization code is re-run as part of the simulation initialization phase, client code does not need to do anything for reset, other than re registering any callbacks that were registered before initialization or re-registering debug environmental callbacks.

Note 1 : All callbacks that were registered during the course of simulation, starting at initialization phase 1.0.1 will be removed during reset. After all vhpiCbStartOfReset callbacks are run, a compliant simulator will remove all callbacks, active, disabled or mature and all scheduled transactions will be anulled.

All handles that pertain to static data are still valid after a reset. Handles to dynamically elaborated regions become invalid at the reset.

The reset sequence consists of:
1. Execute the vhpiCbStartOfReset callbacks, time = Tc (current time)
   The client code is responsible for freeing the handles it requested, in particular handles which will become invalid after the reset operation is completed (callback handles, transaction handles…)
2. The simulator removes all scheduled transactions and all user registered callbacks with the required exception of vhpiCbEndOfReset callbacks.
3. Reset the VHDL simulation state to the beginning of initialization, Tc = 0 ns, ready to commence execution of initilization phase 1.0.1 in the annotated simulation cycle.
4. Execute all user registered vhpiCbEndOfReset callbacks, opportunity for a client application to register callbacks.
5. Initialization phase starts at 1.0.1 in the annotated simulation cycle.

7.7.4 Save, restart and reset of VHPI applications

Applications can use the save/restart callbacks to save and restore their state. An application which uses the VHPI save/restart mechanism must request ids. Application data will be saved in the simulation saved location identified by the id. Applications can request several ids. The restart callback for the application must communicate the id.
An application can register callbacks for vhpiStartOfReset and vhpiEndOfReset to reset its internal state when the simulation gets reset to time 0 ns.

7.7.5 Getting the simulation save and restart location

A string property is defined to be able to get the name of the simulation saved location.
vhpi_get_str(vhpiSaveRestartLocationP, NULL) returns the physical name string of the saved or restart location. The save or restart location is determined by the tool. This property returns null if the tool is not in a save or restart phase.

7.7.6 Restrictions

vhpi_put_data() can only be called from a callback routine that was registered for reason vhpiCbStartOfSave or vhpiCbEndOfSave.
vhpi_get_data() and vhpi_get(vhpiIdP, NULL) can only be called from a callback routine that was registered for reason vhpiCbStartOfRestart or vhpiCbEndOfRestart.
The user_data field of a callback data structure of reason vhpiCbStartOfRestart or vhpiCbEndOfRestart cannot be a pointer into memory because the simulation executable can restart at a different process address. The size of the user_data field for a restart callback is assumed to be an unsigned long.
The property vhpiSaveRestartLocationP returns a non null string when called during a save or restart operation.

Procedural Interface References:

See *vhpi_put_data()* to save data into a save location.
See *vhpi_get_data()* to retrieve data from a save location.
See *"vhpi_register_cb()"* to register save/restart and reset callbacks.
See *"vhpi_get_str()"*to access a string property.

8. Callbacks

Callback is the mechanism used for communication between the VHPI/C code and the VHDL tool. Typically the VHPI user code would register callbacks to happen for specific conditions. The VHPI interface defines a set of reasons for the callback conditions. This chapter begins with an overview of the callback mechanism. The VHPI functions that apply specifically to callbacks are discussed. Where callbacks appear in the information model and what information can be obtained is explained. The semantics of eack callback is then defined. Finally, the execution of a callback is discussed, including what information is passed to it, and referential integrity considerations.

8.1 Callback Overview

A VHPI client application initially gains control at registration when its bootstrap function is called. Similarly, a foreign model gains initial control when its elaboration and/or initialization function defined at registration is called. Thereafter, VHPI allows an application or model to gain control at virtually all semantically significant points during the execution of the tool. The client first registers a callback of the desired kind, providing the function to be called and its relevant data. At the appropriate point, the callback is said to be triggered and VHPI calls its callback function. Some types of callbacks may only be called once, others repetitively. Callbacks are objects in the information model. A handle may be returned at registration or obtained later by navigation. VHPI provides the ability to manage callbacks, including disabling, enabling, or removing them.

8.2 Callback VHPI functions

Callbacks are objects in the VHPI information model. VHPI provides a few specfic functions to create, obtain information, and manage them.

8.2.1 Registering callbacks

vhpiHandleT vhpi_register_cb(vhpiCbDataT *cbdatap, PLI_UINT32 flags);

The registration of callbacks can happen during analysis, elaboration, initialization or simulation run-time execution. The caller can request to have a callback handle returned by the registration function by setting the flag argument to *vhpiReturnCb*. The information model for a callback is discussed in the next section. A callback has state: it is either enabled, disabled, or matured.The callback handle that is returned can be checked to determine the callback state. An integer property *vhpiStateP* can be used to get the callback state: a state could be either *vhpiMature* if the callback has occurred*, vhpiDisable* if the callback was disabled*,* or *vhpiEnable* if the callback is still active. The callback registration is immediate upon the call to *vhpi_register_cb()*. The callback is enabled by default but may be disabled at the installationregistration if the callback registration flag (second argument) is set to *vhpiDisableCb*. The flag argument can be set to both vhpiDisableCb and vhpiReturnCb.

```
Ex:
       cbHdl = vhpi_register_cb(&cbdata, vhpiDisableCb | vhpiReturnCb);
```

The information to set up the callback is passed by the user through a data structure of type *vhpiCbDataT* which must be allocated by the user. All memory for that structure must be allocated by the user, including the *vhpiTimeT* and *vhpiValueT* structures if needed. The cbDatap contents are only used to convey information to the VHPI server on the type of callback to register. The *obj* field of cbDatap may be set to a handle; the client code is free to release that handle after the callback has been registered with no impact on the callback registration. For certain time related callback reasons, the time data structure is needed to pass the time of the callback to be created. For callback on value change, the value structure is needed to pass the format in which the value of the object needs to be returned. The user allocated callback data structure can be reused to register multiple callbacks.

The callback data information is passed by the caller and must provide at least the following information:
- the callback reason,
- the callback function C pointer.

All other fields may or may not be filled up depending upon the callback reason (see sections **Error! Reference source not found.**, 8.4.6, 8.4.8).

The callback registration is immediate upon the call to *vhpi_register_cb()*.The cbDatap contents are only used to convey information to the VHPI server on the type of callback to register. The user allocated callback data structure can be immediately reused to register other callbacks.

The following is the type definition of the 1st parameter passed to vhpi_register_cb:

```
typedef struct vhpiCbDataS
{
  int reason;                          /* callback reason */
  void (*cb_rtn)(const struct vhpiCbDataS *cbdatap); /* callback
                                                       routine */
  vhpiHandleT obj;                     /* trigger object */
  vhpiTimeT *time;                     /* callback time */
  vhpiValueT *value;                   /* trigger object value */
  void *user_data;                     /* pointer to user data to be
                                          passed to the callback
                                          function */
} vhpiCbDataT;
```

The specification of any callback is defined in this 1st parameter and must provide at least the following information:
- the callback reason,
- the callback function C pointer.

All other fields may or may not be filled up depending upon the callback reason and user preference. If the time and value fields are not null, they must indicate a valid format for the callback. The time value is always representedin the base unit of the type TIME. The user_data field provides a mechanism to associate any essential client information needed by the callback function when it is executed. It is never examined or dereferenced by VHPI itself. It may be ignored or cast as a pointer to memory or any other data value of equivalent size to a void* data type to suit the client's purpose.
Further details of the CbDataS usage are discussed in 8.4 Callback Semantics.

8.2.2 Disabling and enabling callbacks

(PLI_INT32) vhpi_disable_cb(vhpiHandleT cbHdl);

(PLI_INT32) vhpi_enable_cb(vhpiHandleT cbHdl);

Any callback can be disabled and enabled respectively with vhpi_disable_cb() an d vhpi_enable_cb(). When a callback is disabled and its trigger condition becomes true, the callback function shall not be called. If it is a one time callback, its trigger condition can never become true again and its state is changed to vphiMature. If it is a repetitive callback, it remains in the vhpiDisable state. Any callback in the vhpiDisable state may be enabled, but a callback in the vhpiMature state can never be enabled. Repetitive callbacks never mature. Note that re-enabling a repetitive callback does not change any of its specification, only whether it is called or not when its trigger condition becomes true. For example, a vhpiCbRepAfterDelay callback is called after a specific delay that starts when it is registered and repeats. Disabling and enabling such a callback has no effect on the time it will next be triggered.
Both functions return 0 on success and 1 on failure. A status of 1 should be returned if the callback is:

- being disabled, but is already disabled (vhpiDisable)
- being enabled, but is already enabled (vhpiEnable)
- or has already matured (vhpiMature).

The severity of this error condition is vhpiWarning, which may be obtained by calling vhpi_check_error(). can be used to determine the severity of the error. In that case it should be a vhpiWarning.
The following callback reasons are repetitive callbacks: vhpiCbValueChange, vhpiCbForce, vhpiCbRelease, vhpiCbStmt, vhpiCbResume, vhpiCbSuspend, vhpiCbStartOfSubpCall, vhpiCbEndOfSubpCall, vhpiCbTransaction, vhpiCbRepAfterDelay, vhpiCbRepNextTimeStep, vhpiCbRepStartOfCycle, vhpiCbRepStartOfProcesses, vhpiCbRepEndOfProcesses, vhpiCbRepStartOfPostponed, vhpiCbRepEndOfPostponed, vhpiCbRepEndOfTimeStep, vhpiCbQuiescense, vhpiCbPLIError, vhpiCbEnterInteractive, vhpiCbExitInteractive.

Note all the vhpiRep* calbacks are registered in a repeated manner for the same simulation cycle point as their respective non repetitive callback reasons. vhpiCbRepAfterDelay callback causes the callback function to be triggered after every elapsed simulation time equal to the delay specified in the time field. If a vhpiCbRepAfterDelay callback is enabled after it was disabled, the callbacks will be re-enabled for the times it was initially registered. The effect of disabling such a callback results in temporarily inhibiting it from triggering, the effect of enabling this callback has the result to allow it to trigger again.

8.2.3 Getting callback information

(PLI_INT32) vhpi_get_cb_info (vhpiHandleT cbhdl, vhpiCbDataT *cbData_p);
Given a callback handle *cbhdl*, the VHPI server will fill up a vhpiCbDataT structure that has been allocated by the user with the equivalent original information which was passed by the user in the *cbData* structure at the time of registration of that callback handle. All memory for the cbDatap structure must be allocated by the user. This function can be called at any time by a VHPI application which holds a valid callback handle. The validity of a handle has to do with referential integrity and not the state of the callback. Information is available for any callback, whether is it enabled, disabled, or mature. The function returns 0 on success and 1 on failure.

8.2.4 Removing callbacks

(PLI_INT32) vhpi_remove_cb(vhpiHandleT cbHdl)

Given a callback handle, this function will remove the callback; the callback will not occur anymore. It will also free the callback handle, thus invalidating it. In contrast, just freeing the callback handle with vhpi_release_handle() frees the memory associated with the callback handle but does not remove the callback.In this later case, the callback will still be triggered according to its specification and an equivalent callback handle can still be re-obtained by the callback access methods (see section 8.3.1) depicted by the callback class diagram. The function returns 0 on success and 1 on failure.

8.3 Callback Information Model

The callback UML class diagram in Chapter 4 illustrates the methods and properties that are available for callbacks. A callback object is created when it is registered. A handle to it may be obtained at registration or at a later time using the methods described below. The handle so obtained remains valid until released by the user. The callback object itself exists until it is removed and there are no valid handles referencing it. One time callback objects are removed automatically under certain conditions.

8.3.1 Callback methods

Iteration on vhpiCallbacks from the tool (designated by a NULL reference handle) will return handles for all callbacks that are existing at the time of the query. It will return handles to callbacks that are either enabled or disabled, but mature callbacks are not returned. The only way to have a valid handle to a mature callback is to obtain the handle at registration or by iteration before the callback has matured. All callbacks are returned by the iteration to the caller even if the caller did not register these callbacks. VHPI has no concept of client identity that would allow otherwise. Given a callback handle, vhpi_get_cb_info can obtain the cbDataS of the original registration, then the caller can filter out callbacks of interest by looking at the function pointers or other information in the cbDataS. The preferred method of keeping track of callbacks is to retain handles obtained at registration

Informative note: Looking at the function pointer address is the only way to recognize one's own callbacks, provided there is a way of comparing the address to the complete set of functions used to register callbacks and no other applications have registered callbacks with those functions. No assumptions can be made about the contents of the user_data field which may be null or not be a valid memory address.

A one to one method (vhpiCurCallback) from a null reference handle will return the currently executing callback handle or null. This will provide a new handle of kind vhpiCallbackK owned by the client. The state of that handle will be enabled or matured. It can be used to immediately remove and/or releasethe callback, disable it, or any other operation allowed on a valid callback handle. Note that if the goal is register a one time callback that removes itself after it matures, the preferred method is never to obtain a handle and let VHPI cleanup after callback execution.

Note: An elaboration, initialization or execution function is not a callback. *vhpiCurCallback* when called from within an elaboration, initialization or execution function should return NULL.

Iteration from an object declaration will return handles to callbacks of reason vhpiCbValueChange, vhpiCbTransaction, vhpiCbForce, vhpiCbRelease that were registered with the obj field of the vhpiCbDataT argument set to the object declaration handle.

Iteration from a statement handle (concurrent or sequential) will return all vhpiCbStmt, vhpiCbResume and vhpiCbSuspend, vhpiCbStartOfSubpCall, vhpiCbEndOfSubpCall reason callbacks registered for that statement.

Iteration from an indexedName or selectedName will return callbacks of reason vhpiCbValueChange, vhpiCbTransaction, vhpiCbForce, vhpiCbRelease that have been registered for the object name indicated by the handle in the obj field.

Iterating from a driver handle will return all registered callbacks for reason vhpiCbTransaction and vhpiCbValueChange for this driver.

8.3.2 Callback properties

There are two integer type callback properties that can be queried given a callback handle.

  vhpiReasonP: gets the callback reason

  vhpiStateP: returns the callback state either vhpiDisable, vhpiMature, or vhpiEnable.

A callback state is said to be "mature" if the callback has occurred at the time of the query. This means specifically that if the vhpiStateP of the currently executing callback is obtained and it is a one time callback, it will already be in the vhpiMatured state. Repetitive callbacks never mature.

The callback reason specifies when a callback is supposed to occur.

8.4 Callback Semantics

This section defines all the specific kinds of callbacks for VHPI. They are described in categories of tool phase, object, foreign model, stmt, time, simulation cycle, action, and save/restart/reset. These are regarded as providing a basics set of callbacks for a client to gain control at all significant points during the VHDL tool's execution. Each callback is identified by a callback reason, which is provided at registration along with any additional information required to fully specify the callback. The VHDL simulation cycle is referenced in some callback definitions and is annotated with specific VHPI callback reasons to support rigorous formal semantics. It is defined below before any discussion of individual callbacks.

8.4.1 The Annotated VHDL Simulation Cycle

There are references to the VHDL simulation cycle used to describe when some callbacks are triggered. The VHDL simulation algorithm is presented below with modifications that describe when the various VHPI callback reasons occur.

0. Initialization phase:

1)
1.0.1) VHPI: cbStartOfInitialization callbacks are run including the implicitly registered foreign architectures callbacks (execf functions)
1.0.2) VHPI: cbStartOfNextCycle callbacks are run.
1.1.0) The driving value and the effective value of each explicitly declared signal are computed and the current value of the signal is set to the effective value. This value is assumed to have been the value of the signal for an infinite length of time prior to the start of simulation.

2)
2.1.0) The value of each implicit signal of the form S'Stable(T) or S'Quiet(T) is set to True.
2.2.0) The value of each implicit signal of the form S'Delayed(T) is set to the initial value of its prefix, S.

3)
3.1.0) The value of each implicit GUARD signal is set to the result of evaluating the corresponding guard expression.

4)
4.0.1) VHPI: cbStartOfProcesses callbacks are run.
4.1.0) Each nonpostponed process in the model is executed until it suspends.
4.1.1) VHPI: cbEndOfProcesses callbacks are run.

5)
5.0.1) VHPI: cbStartOfPostponed callbacks are run.
5.1.0) Each postponed process in the model is executed until it suspends.

6)
6.1.0) The time of the next simulation cycle (which in this case is the first simulation cycle), Tn, is calculated according to the rules of step f of the simulation cycle, below.
6.1.1) VHPI: cbEndOfInitialization callbacks are run. A reset of the VHDL model would bring back the model to immediately after this point in the simulation cycle.
6.1.2) VHPI: cbStartOfSimulation callbacks are run.

1. Simulation cycle:

This marks the beginning of a time. The current time (T_c) has just advanced to the next time (T_n) where events or actions are scheduled to occur. Signal effective values have not changed yet.

a)
a.1.0) The current time, T_c is set equal to T_n.
a.1.1) VHPI: cbNextTimeStep callbacks are run except when simulation is complete according to the rules of step a.2.0.
a.2.0) Simulation is complete when T_n = TIME'HIGH and there are no active drivers or process resumptions at T_n.
a.2.1) VHPI: cbStartOfNextCycle callbacks are run.
a.2.2) VHPI: cbAfterDelay callbacks are run.

b)
Signal update, resolution and propagation
The driving values of the signal drivers are computed by executing the transaction of their output waveform relevant for that time. vhpiCbValueChange and vhpiCbTransaction callbacks on drivers happen immediately when the driver has a value change or a transaction respectively. The basic signal effective values are computed. Resolution functions are executed to compute resolved signal values. The driving values of the basic signals are propagated through the port connections, conversion and resolution functions. Non basic signal values are computed from the values of their sources. The effective signal values are computed. The effective value of a signal becomes the new current value of that signal. A signal is said to be active during the delta cycle if its new current value is different from the previous signal value. If updating a signal causes the current value of that signal to change, then an event is generated for that signal in that delta cycle. vhpiCbValueChange callbacks on signals occur during this phase b if that signal had a value change.

b.1.0) Each active explicit signal in the model is updated. (Events may occur on signals as a result.)

c)
c.1.0) Each implicit signal in the model is updated. (Events may occur on signals as a result.)

d)
Process execution
The events determined at the signal update cause the resumption of processes sensitive to that signal during this delta simulation cycle. These processes execute and may cause new transactions on the signal drivers.

d.0.1) VHPI: cbStartOfProcesses, vhpiCb(Rep)TimeOut, vhpiCbSensitivity callbacks are run.
d.1.0) VHPI: cbResume callbacks are executed for the non postponed processes which are going to resume. cbResume callbacks occur before the process is executed.
For each non postponed process P, if P is currently sensitive to a signal S and if an event has occurred on S in this simulation cycle, then P resumes.

d.1.1)VHPI: cbValueChange callbacks for variables occur immediately if the current process execution causes the variables to change value

e)
e.1.0) Each non postponed process that has resumed in the current simulation cycle is executed until it suspends. VHPI: cbSuspend callbacks are executed for the non postponed processes which were suspended.
e.1.1) VHPI: cbEndofProcesses callbacks are run.

f)
f.1.0) The time of the next simulation cycle T_n is determined by setting it to the earliest of:
1) TIME'HIGH
2) The next time at which a driver becomes active, or
3) The next time at which a process resumes.
f.1.1) VHPI: If T_n != T_c cbLastKnownDeltaCycle callbacks are run and T_n is recalculated according to the rules of step f.1.0.

f.2.0) If T_n = T_c, then the next simulation cycle (if any) will be a delta cycle.

g) **Postponed process execution**

The postponed processes are executed if this is the last delta simulation cycle for the time T_c. Postponed processes are executed until they suspend. The execution of postponed processes should not cause any new delta cycle.

g.1.0) If the next simulation cycle will be a delta cycle, the remainder of this step is skipped. {i.e. go to step a.2.0}
g.1.1) Otherwise, VHPI: cbStartOfPostponed callbacks are run.
g.2.0) Each postponed process that has resumed but has not been executed since its last resumption is executed until it suspends. VHPI: cbResume callbacks are executed for the postponed processes which are going to resume. cbResume callbacks occur before the process is executed. VHPI: cbSuspend callbacks are executed after the postponed process suspends.
g.2.1)VHPI: cbValueChange callbacks for a variable occur immediately if the current postponed process execution causes the variable to change value.
g.3.0) T_n is recalculated acording to the rules of step f.
g.3.1) It is an error if the execution of any postponed process causes a delta cycle to occur immediately after the current simulation cycle.

h)
End of time step:
This phase follows the postponed process phase (if postponed processes exist) or the process execution phase. It marks the end of the current time T_c, the next time T_n is different from T_c.
h.1.0) VHPI: cbEndOfTimeStep callbacks are run.
h.1.1) If there are active drivers or process resumptions at T_n, then the remainder of this cycle is skipped,
{i.e. go to step a.2.0},
Otherwise the simulation has reached a quiescent state that may be the end of simulation if then, go to i).

i)
Quiescence:
In this phase, simulation has reached a stable state.
i.1.0) VHPI: cbQuiescence callbacks are run. This allows potentially foreign models or applications to further stimulate the design.
i.1.2) T_n is recalculated according to the rules of step f.
i.1.3) It is an error if the execution of any cbQuisecence callback causes a delta cycle to occur immediately after the current simulation cycle.
i.1.1) If there are active drivers or process resumptions at T_n, then the remainder of this cycle is skipped, {i.e. go to step a.1.0}
Otherwise this is the end of simulation, go to j)

j)
End of simulation:
j.1.0) VHPI: cbEndOfSimulation callbacks are run.
j.1.1) Simulation terminates.

The callback reason defines when the callback will happen. In the following section, when describing a callback reason, we explain when the callback function triggers. For all callback registrations, the user must allocate a callback data structure of type *vhpiCbDataT* and set the fields relevant for that callback reason. For all callbacks, the *reason* and *cb_rtn* field (callback function pointer) must be set. If the time and value fields are not nul, they must indicate a valid format for the callback. The time value is always represented in the simulator precision. Additional settings are described for each callback reason.

There are three main categories of callbacks: event, time and action callbacks. The various callbacks are described below.

8.4.2 Object Callbacks

There are many kinds of objects in the VHPI information model, distinguished as having a runtime value. The object callbacks return control to the client when a dynamic aspect of the object changes.

All object callbacks are repetitive callbacks except for the optional foreign model timeout callback which has both a non repetitive and repetitive callback reason. They remain in effect until they are removed by calling *vhpi_remove_cb()*.
If at the registration of the callback, the *value* and *time* fields of the *vhpiCbDataS* structure are not null, they indicate that the value of the object and the time of the change are requested to be provided when the callback triggers. The *value* field is allocated by the user at the registration and is only used as an indication to obtain a value in a specified format when the callback triggers. Only the format field of the value structure must be set, there is no need to allocate a buffer for formats which require a buffer. The time field of the cbDataS structure must be set to a non NULL value at the registration; this non NULL value will not be dereferenced by the VHPI implementation. When the callback triggers, the entire cbDataS structure of the callback function is allocated by the interface, including the *time* and *value* structures. The *value* and *time* structures are filled up when the callback triggers with the *value* and *time* of the object which caused the callback routine to be invoked. The cbDataS structure of the callback function is read-only for the user.

8.4.2.1 vhpiCbValueChange

This callback reason tracks value changes of variables,signals (including the signal attributes delayed, stable, quiet and transaction if they are referenced in the design) and drivers. These could be either full objects, selected names, indexed names or drivers. The object kind can be *vhpiSigDeclK*, *vhpiVarDeclK*, *vhpiPortDeclK*, *vhpiSigParamDeclK*, *vhpiVarParamDeclK*, *vhpiIndexedNameK*, *vhpiSliceNameK, vhpiSelectedNameK, vhpiDriverK, vhpiParamAttrNameK*. For signals which are not OUT mode ports (*vhpiOutPortDeclK*), the callback tracks signal effective value change. For OUT mode ports,drivers and *vhpiOutPortDeclK* (out port of an inout port), the callback tracks the driving value. The callback on signal value change will fire if an event is generated for that signal. Callback functions for value change of signals and ,drivers and predefined signal attributes which define implicit signals such as 'delayed, 'stable, 'quiet and 'transaction may be executed during signal update and propagation up until but specifically before cbStartOfProcesses callbacks are executed. For signal class objects, only one callback will occur for the whole object even if more than one scalar element changes value in the same delta cycle. For variable class objects, the callback triggers as soon as the variable changes value, therefore many callbacks for the same variable object can be executed in the same delta cycle. Value change callbacks on variables, typically occur during the process or postponed process execution phase whenever a variable is updated during VHDL execution or with *vhpi_put_value*.
The registration of a value change callback consists in setting the *obj* field to the handle of the object subject to the callback on value change. The caller may request that the value of the object resulting from the value change and/or the time of the value change be returned in the callback data structure. If so, the caller needs to allocate a value and time structures for the fields of the vhpiCbDataS structure. The time and value structures specify the formats in which time and value of the object value change should be given when the callback function executes (for valid formats see *vhpi_get_value()*). Otherwise if the value and time of the value change callback are not requested, the value and time field pointers should be set to NULL. These value and time structure are only an indication for the registration of the callback. When the callback triggers, the callback, value and time structures are allocated by VHPI.

Notes: Callback on value changes cannot be placed for aliases of objects.

8.4.2.2 vhpiCbForce

This callback reason triggers if a variable, signal, or part of a variable or signal was forced to a value. The callback registration consists in setting the *obj* field to the handle of the object of interest. The caller can also request to get the value of the object after the force, if so, the value field must point to the address of a value structure that has been allocated by the caller. A valid format must be provided for the value. A forced value may happen because of a call to the VHPI function *vhpi_put_value(objHdl, &value, flags),* where the flag value is set to *vhpiForce*, *vhpiForcePropagate* or from a force simulator command. If the *obj* field is null, then the callback should happen every time a force occurs on any object; when the callback triggers, the *obj* field will contain the handle of the object that was forced. The differences between the 2 types of forces are explained in section 9.3. For all objects, the callback triggers immediately when the object is forced.
Note: Signal valued attributes cannot be forced. Guard signals can be forced.
ISSUE: Should we list kinds? Note that drivers cannot be forced?If only a slice of the object was forced, return the entire object.

8.4.2.3 vhpiCbRelease

This callback reason triggers if a value release occurs to a variable or signal or sub-element thereof that was previously forced. The callback registration consists in setting the obj field to the handle of the object of interest. After the release has happened the value that was forced remains until it is overwritten by a new update. Consequently the value that the caller may request to get will still be the forced value. A value release may happen because of a call to the VHPI function *vhpi_put_value(objHdl, &value, flags),* where the flag value is set to *vhpiRelease*, or from a simulator release command with the same semantics. If the *obj* field is null, then the callback should happen every time a release occurs on any forced object; when the callback triggers, the *obj* field will contain the handle of the object which force was released. A release can be applied during any of the phases and the callback function triggers immediately.

NOTE: A force or release accomplished either through vhpi_put_value with vhpiForce or vhpiRelease flags or through a simulator force or release command does not trigger value change callbacks.

8.4.3 Optional object callbacks

8.4.3.1 vhpiCbTransaction

A callback registered for that reason triggers when a driver transaction matures or when a transaction occurs on the signal. The driver value has been updated according to the transaction value. The *obj* field should be set to a driver or signal handle. The value and time of the transaction can be requested to be returned at the callback registration by providing pointers to user allocated value and time structures in the value and time fields. The callback triggers during signal update and propagation before value change callbacks for the signal.

8.4.4 Foreign models specific callbacks

8.4.4.1 vhpiCbTimeOut

A callback registered for this reason triggers during the process execution phase when the simulation time specified in the *time* field of the *cbData* structure has elapsed since the callback registration. The *reason*, *cb_rtn* and *time* fields are the only required fields which need to be set at the callback registration. This callback is equivalent to a process statement which would have two statements: the VHDL statements wait for time followed by the call to the callback function cb_rtn.
vhpiCbTimeOut is the non repeated version of the callback. **vhpiCbRepTimeOut** is the repeated version
vhpiCbTimeOut is a convenience that is equivalent to registering a cbAfterDelay timeout cbk which itself registers a cbStartOfProcesses callback.It is possible to emulate a postponed process which has a time out by registering a time out callback which itself registers a start of postponed callback.

8.4.4.2 vhpiCbSensitivity

A callback registered for this reason triggers during the process execution phase when an event occurred on any of the signals indicated by the obj field. The *obj* field must be set to a single signal handle or to a collection of signals. If the value field is not null, the cbData value.int field will indicate which signals in the collection had events. The set to 1 of a specific bit of the value.intgs field will indicate the corresponding signals members in the collection in the obj field) which. The value.intgs field is allocated by VHPI and is read only when the callback function executes.
If the *time* field is not null at the callback registration, when the callback triggers, the cbData time field should be set to the current absolute simulation time. It is possible to emulate a postponed process which has a sensitivity by registering a vhpiCbSensitivity callback which itself registers a vhpiCbStartOfPostponed callback.
It is a **repeated** callback.
Note: Due to the nature of postponed processes, the value you may obtain when the startOfPostponed callback execute may not be the value which triggered the callback.

8.4.5 Statement callbacks

The *vhpiCbStmt* and *vhpiCbResume* and *vhpiCbSuspend vhpiStartOfSubpCall* and *vhpiEndOfSubpCall* callbacks are repetitive callbacks. Other statement callbacks are one time only callbacks. The only fields in the *vhpiCbDataS* structure that shall need to be set up by the user are the *reason, obj, cb_rtn* and *user_data* (if desired) fields. When a statement callback occurs, the *cb_rtn* user routine is called and is passed a pointer to a *vhpiCbDataS* structure, the *reason, cb_rtn* and *user_data* fields shall be set to the reason, *cb_rtn* and *user_data* fields which were passed in at the callback registration and the *obj* field shall be set to the original statement or another statement as detailed in the description of each specific callback reason.

8.4.5.1 vhpiCbStmt

A callback registered for that reason triggers before a sequential statement executes. The callback registration consists in setting the *obj* field to the handle of a sequential statement or to the handle of a equivalent process statement. The parent scope of the sequential statement must be a static region (concurrent statement) or a dynamically executing or suspended region (procedure or function call). The statement handle must be an instantiated statement handle. In the case where the object handle is an equivalent process statement, the callback will trigger when it is about to execute the concurrent statement. When the callback triggers, the obj field is set to point to the sequential statement that is going to be executed in the case of a process or procedure call statement. For other equivalent process statements, the obj field remains the concurrent statement handle. No other fields except the obj and reason fields need to be set. In particular, the time and value fields are ignored. The callback function executes during the process or postponed process execution phases. The callback acts like a statement breakpoint. Note that an optimized VHDL implementation may have effect on the order in which callbacks occur and may prevent the registration of certain callbacks due to loss of the HDL source mapping. If a callback cannot be registered, an error should be generated. The table below describes the behavior of cbStmt for all possible statement kinds.

vhpiWaitStmt **vhpiReportStmt** **vhpiAssertStmt** **vhpiVarAssignStmt** **vhpiSimpleSigAssignStmt** **vhpiNextStmt** **vhpiExitStmt** **vhpiReturnStmt**	One callback occurs before the statement executes. For the wait and assert statements, the callback occurs before the condition expression is evaluated.
vhpiForLoop **vhpiWhileLoop** **vhpiForeverLoop**	For a vhpiForLoop statement, the callback should occur prior to a new value be assigned to the loop parameter. For a vhpiWhileLoop, the callback occurs prior to the evaluation of

	the condition expression on every iteration of the loop. For a vhpiForeverLoop, the callback occurs when the forever loop statement is first encountered, and on every subsequent iteration of the forever loop.
vhpiIfStmt **vhpiCaseStmt**	There are two cases: 1) if the obj field is set to a handle to the if or case statement (vhpiIfStmtK, vhpiCaseStmtK): the callback will trigger before the condition expression of the if stmt gets evaluated, or before the case expression of the case statement gets evaluated 2) if the obj field is set to a handle to a branch (vhpiBranchK) of the if stmt or case stmt: the callback will trigger before the condition of the branch gets evaluated for an if statement and before the first statement of the branch of the case statement gets executed.
vhpiNullStmt	This callback triggers just before the null stmt is executed.
vhpiProcCallStmt **vhpiProcessStmt**	The callback occurs just before the sequential statement inside the procedure call or process statement executes. When the callback triggers, the obj field points to the handle of the sequential statement that is going to be executed.

8.4.5.2 vhpiCbResume

A callback registered for that reason triggers before a process resumes execution. The callback registration consists in setting the obj field to a process or concurrent procedure call. When the callback triggers, the callback data argument of the callback function sets the obj field to the statement after the wait statement from where the process resumes or is set to the first statement in the process statement, if the process has a sensitivity list. The callback function executes during the process or postponed process execution phases.

8.4.5.3 vhpiCbSuspend:

A callback registered for that reason triggers just before a process or procedure suspends. When the callback triggers, the current region is the process about to be suspended. The callback registration consists in setting the *obj* field to a process or concurrent procedure call. When the callback triggers, the obj field of the callback data argument is set to a handle to the explicit wait statement if the process or procedure suspends on a wait, otherwise it is set to the last statement of process or procedure call it was originally set to. The current scope is still the process that is about to be suspended. The callback function executes during the process or postponed process execution phases.

8.4.5.4 vhpiCbStartOfSubpCall

A callback registered for that reason triggers when a subprogram call starts execution. The subprogram can either be a concurrent or sequential call of a VHDL or foreign subprogram The callback registration consists in setting the obj field to a handle to the subprogram call. A handle to the subprogram call can be obtained by iterating on the sequential statements of a static region or dynamically executing region (subprogram call within a subprogram). A handle to a subprogram call cannot be obtained if the parent scope is not a static scope or an executing scope. When the callback function triggers, the subprogram formal parameters have been elaborated and their values assigned from the actual associations. No fields other than the obj and reason fields need to be set at the callback registration. The callback triggers whenever this specific instance of that subprogram call is invoked. This is a repetitive callback for a specific instance of that subprogram call.

8.4.5.5 vhpiCbEndOfSubpCall

A callback registered for that reason triggers when the subprogram call indicated by the obj field has completed execution. The subprogram can either be a concurrent or sequential call. The callback registration consists in setting the obj field to a handle to the subprogram call. The callback triggers when the subprogram call is about to return from execution. No fields other than the obj and reason fields need to be set at the callback registration. The intention is with this callback to be able to intercept and overwrite the returned value of the function call or the values of the out mode parameters of a procedure call before these values take effect in the calling code. This is a repetitive callback for a specific instance of that subprogram call.

8.4.6 Time callbacks

There are one time and repetitive time callbacks.
For all time related callbacks, the only fields in the *vhpiCbDataS* structure that shall need to be set up by the user are the *reason, time, cb_rtn* and *user_data* (if desired) fields. When a time callback occurs, the *cb_rtn* user routine is called and is passed a pointer to a *vhpiCbDataS* structure, the *reason, cb_rtn* and *user_data* fields shall be set to the reason, *cb_rtn* and *user_data* fields which were passed in at the callback registration and the time field shall be set to the absolute current time.

8.4.6.1 vhpiCbAfterDelay

A callback registered for that reason triggers at the absolute time computed by adding the current simulation time (at the registration Tr) plus the relative time delay (d) that is indicated by the time structure when the callback is registered. The callback triggers even if there is no transaction scheduled at this time; the callback function is executed at the beginning of the time step before values get updated and transactions processed. The cbData structure passed to the user callback function will include a time structure indicating the current simulation time. No fields other than the reason, cb_rtn and time fields need to be provided at registration. The time value of the cbData filled by *vhpi_get_cb_info* should be the relative time delay provided at the registration. All time values are given and interpreted in the simulator time precision.

8.4.6.2 vhpiCbRepAfterDelay

A callback registered for that reason causes the callback function to trigger at the current simulation time plus the relative time delay that is indicated by the time structure at the callback registration and at all subsequent intervals of that time delay value. If the current registration time is Tr, and a callback is registered for a delay of d, callbacks will be triggered at Tr + d, Tr + 2d, … Tr + md. The callback triggers even if there is no transaction scheduled at this time; the callback function is executed at the beginning of the time step. If one disables the repetitive callback and re-enables it at time Tn, the callback will be reinstalled to trigger at times Tr +d, Tr + 2d, Tr + md where for all m, Tr +md > Tn and where Tr is the time the callback was registered,
The callback is on the same schedule as defined when it was registered so that disable/enable has no effect on the schedule of the repetitive callback, except to temporarily inhibit it from triggering.

8.4.7 Simulation phase callbacks

For each type of callback, there is a single occurrence reason and a repetitive callback reason. If a phase callback is registered while or after that phase executes, the callback will not be triggered until the next time the tool executes that phase.

The only fields in the *vhpiCbDataS* structure that shall need to be set up by the user are the *reason, cb_rtn* and *user_data* (if desired) fields. Any other field setting will be ignored. When a simulation phase callback occurs, the *cb_rtn* user routine is called and is passed a pointer to a *vhpiCbDataS* structure, the *reason, cb_rtn* and *user_data* fields shall be set to the reason,, *cb_rtn* and *user_data* fields which were passed in at the callback registration.

8.4.7.1 vhpiCbNextTimeStep

A callback registered for that reason triggers at the beginning of the next time slice that has driver transactions and/or a cbAfterDelay or cbRepAfterDelay callback registered and where Tn (next time at which the callback triggers) is different from Tc (current time at which the callback is registered). The callback triggers at the beginning of a time step: the simulation time has just advanced, and no signal values have been updated yet. Immediate signal value modifications done by vhpi_put_value with vhpiDepositPropagate or vhpiForcePropagate flags, or by vhpi_schedule_transaction with 0 delay will take place in the first delta cycle of the time step if they are invoked by this callback function.
vhpiCbRepNextTimeStep is the repeated version.

8.4.7.2 vhpiCbStartOfNextCycle

A callback registered for that reason triggers at the beginning of the next simulation delta cycle either for the present time step or for the next time step if no more delta cycles are created for the current time step. The callback function executes after vhpiCbNextTimeStep, before vhpiCbAfterDelay and before signal update and propagation starts.
vhpiCbRepStartOfNextCycle is the repeated version.

8.4.7.3 vhpiCbStartOfProcesses

A callback registered for that reason triggers just before VHDL non postponed processes start execution. Zero delay transactions scheduled by the callback function at this point take place in the next simulation delta cycle.
vhpiCbRepStartOfProcesses is the repeated version.

8.4.7.4 vhpiCbEndOfProcesses

A callback registered for that reason triggers after all non postponed processes have executed for the current delta cycle just before the next time Tn is computed. Zero delay transactions scheduled by the callback function at this point take place in the next simulation delta cycle.
vhpiCbRepEndOfProcesses is the repeated version.

8.4.7.5 vhpiCbLastKnownDeltaCycle

A callback registered for that reason triggers just before the postponed processes execute (before vhpiCbStartOfPostponed but after vhpiCbEndOfProcesses of the last delta. This callback triggers when there is no more delta cycles to execute in this time step,. Zero delay transactions may be done by the callback and can create a new delta cycle for this time step.
vhpiCbRepLastDeltaCycle is the repeated version.

8.4.7.6 vhpiCbStartOfPostponed

A callback registered for that reason triggers before the postponed processes execute and after vhpiCbLastKnownDeltaCycle. No zero delay transactions may be done by the callback function because this marks the end of all 0 delay delta cycles for this time step. An attempt to place zero delay transactions

with *vhpi_schedule_transaction()* or with *vhpi_put_value* with modes of vhpiDepositPropagate or vhpiForcePropagate should result in a runtime error.
vhpiCbRepStartOfPostponed is the repeated version. These callbacks occur even if there are no postponed processes.

8.4.7.7 vhpiCbEndOfTimeStep

A callback registered for that reason triggers at the end of the time step, the current time Tc has not advanced yet to the next computed time Tn. No transaction may be done by the callback function because this marks the end of a time step. . An attempt to place any transactions with *vhpi_schedule_transaction()* or with *vhpi_put_value* with modes of vhpiDepositPropagate or vhpiForcePropagate should result in a runtime error.
vhpiCbRepEndOfTimeStep is the repeated version.

8.4.8 Action callbacks

The only fields in the *vhpiCbDataS* structure that shall need to be set up by the user are the *reason, cb_rtn* and *user_data* (if desired) fields. When an action callback occurs, the *cb_rtn* user routine is called and is passed a pointer to a *vhpiCbDataS* structure, the *reason, cb_rtn* and *user_data* fields shall be set to the reason, *cb_rtn* and *user_data* fields which were passed in at the callback registration. If an action callback is registered at a given point during the tool session and that point precludes that action to ever take place, the tool is not required to detect such a situation.

All callback reasons except *vhpiCbQuiescence* and *vhpiCbPLIError, vhpiEnterInteractive, vhpiExitInteractive, vhpiSigInterrrupt* are one time callbacks.

8.4.8.1 vhpiCbStartOfTool

A callback registered for that reason triggers when the tool starts its session, right after tool and VHPI interface initialization. The registration phase for VHPI has completed and all bootstrap functions have executed at this point. The tool class is accessible . Existing libraries and already analyzed models in those libraries are available in the uninstantiated information model.A session informally refers to the entire time a tool is executing. The only time any part of the VHPI information can be accessed is during a tool session. A session is delimited by the **vhpiCbStartOfTool** and **vhpiCbEndOfTool** callbacks.

8.4.8.2 vhpiCbEndOfTool

A callback registered for that reason triggers at the end of a tool session just before it exits. No access to any part of the VHPI information model is possible in the callback function, but final cleanup of a client application is possible including use of vhpi_printf(). All handles are invalid at this point and cannot be referenced, even to be released.

8.4.8.3 vhpiCbStartOfAnalysis

A callback registered for that reason triggers before analysis starts. The tool class is accessible. Existing libraries and their previously analyzed models are available**.**

8.4.8.4 vhpiCbEndOfAnalysis

A callback registered for that reason triggers at the end of the analysis phase. Access to post-analysis information models of previously analyzed design units analyzed before or during this session is possible.

8.4.8.5 vhpiCbStartOfElaboration

A callback registered for that reason triggers before the start of elaboration of a VHDL design. The access is the same as the one which is allowed for the cbEndOfAnalysis callback. In addition, this callback point allows the registration of callbacks which may be triggered during elaboration such as cbStartOfSubpCall of cbEndOfSubpCall.

ISSUE: I don't agree with this at all. This is virtually no different than vhpiEndOfAnalysis, except it does allow callbacks to be registered that may be triggered during elaboration for functions called in initializer expressions.

Note: Elaboration function for foreign architectures also have the reason field of the cbData structure set to vhpiCbStartOfElaboration; the access allowed in this circumstance is as defined in the foreign model chapter.

8.4.8.6 vhpiCbEndOfElaboration

A callback registered for that reason triggers at the end of the elaboration of a design. This provides a hook for elaborator back end applications Access to the post-analysis and post-elaboration information models is possible. Access to initial objects values determined by the elaboration of the object is possible and is the initial value assigned to the object as defined by the VHDL LRM section 12.3.1.4. Generic and port values at the end of elaboration are the ones determined by VHDL LRM 12

8.4.8.7 vhpiCbStartOfInitialization

A callback registered for that reason executes at the beginning of initialization. Access to post-analysis, post-elaboration and runtime information models is possible. Getting values has the same behaviour as getting values at cbEndOfElaboration. Updating values and scheduling transactions has unspecified behaviour.

8.4.8.8 vhpiCbEndOfInitialization

A callback registered for that reason executes at the end of initialization. Full Access to post-analysis, post-elaboration and runtime information models is possible. In particular getting values, updating values and scheduling transactions on drivers and signals is possible.

8.4.8.9 vhpiCbStartOfSimulation

A callback registered for that reason triggers when simulation starts, after simulation initialization. No other field settings are necessary for this type of callback. This callback occurs before any time or delta cycle callbacks. Access to post-analysis, post-elaboration and runtime information models is possible. Getting values, updating values and scheduling transactions is possible.

8.4.8.10 vhpiCbQuiescense

A callback registered for that reason triggers when a simulation reaches a quiet state and no transactions remain to be processed. The callback function can then schedule new non zero transactions with *vhpi_schedule_transaction()* and keep the simulation going. No other field settings are necessary for this type of callback. vhpiCbQuiescence shall occur before a vhpiCbEndOfSimulation callback. Access to post-analysis, post-elaboration and runtime information models is possible.

8.4.8.11 vhpiCbEndOfSimulation

A callback registered for that reason triggers when a simulation is complete normally according to the LRM. vhpi_control (vhpiFinish) will not cause a vhpiCbEndOfSimulation callback. If registered, a vhpiCbEndOfTool may follow a vhpiCbEndOfSimulation callback. Access to post-analysis, post-elaboration and runtime information models is possible. Getting values is possible.

8.4.9 Optional action callbacks

8.4.9.1 vhpiCbPLIError

Remove that callback, useless. A callback registered for that reason triggers when a VHPI error occurred. The callback function can then check the error. No other field settings are necessary for this type of callback. The error may or may not be caused by the application which registered the callback.

8.4.9.2 vhpiEnterInteractive

A callback registered for that reason triggers when the VHDL tool stops and enter the interactive mode. No other field settings are necessary.

8.4.9.3 vhpiExitInteractive

A callback registered for that reason triggers when the control is returned to the VHDL tool. No other field settings are necessary.

8.4.9.4 vhpiCbSigInterrupt

A callback registered for that reason may be triggered for an implementation defined abnormal event notification. No other field settings are necessary.

8.5 Save/Restart/Reset Callbacks

The reset and save callback are repetitive , restart callbacks are not.
No callbacks occur between a start of save and end of save or between a start of reset and an end of reset.
If the user interrupts the process during a save, then the data saved is not guaranteed to correspond to a valid restart point.
ISSUE: Should we allow other callbacks to happen in between these pairs of callbacks? Can you receive an interrupt callback? An error callback? Are both calls guaranteed to occur, unless the session is terminated?

vhpiCbStartOfSave, vhpiCbEndOfSave

A callback registered for reasons vhpiCbStartOfSave or vhpiEndOfSave triggers when a save command is processed by the tool. The vhpiCbStartOfSave callback occurs at the beginning of the saving of the VHPI models while the vhpiCbEndOfSave callback occurs at the end of the save operation. A model may not need to register for both callback reasons, but these two reasons are provided as a convenience to the user. For example, the user can use the vhpiCbEndOfSave reason to prepare for continuing simulation: as a consequence of saving data structures, the user may have to turn pointers of data structure into relocatable addresses, then after saving them and before continuing simulation, these must be fixed in memory to actual addresses. The fix up phase can be performed by the callback function registered for reason vhpiCbEndOfSave. When vhpiCbEndOfSave is called, the client is assured that all model instances that registered vhpiCbStartOfSave have been initially processed. Please refer to save and restart of foreign models Chapter 7. These callbacks can be registered at any time during simulation but the save operation and the save callbacks occur at a clean simulation state between simulation cycles (all scheduled events and processes for a delta cycle have executed and all steps for the next delta simulation cycle have not executed yet). The callback registration consists in setting the *cb_rtn* field to the callback routine to be called at the start of save or at the end of save. The user_data field can be set to the private data to be saved. All other fields are ignored. They are repetitive callbacks and remain until the end of the same simulation run or until there is a reset or a restart.
If time is requested via vhpi_get_time() during the cbStartOfSave cbEndOfSave or cbStartOfRestart callbacks, the time will be T_c from the previously completed simulation cycle.

vhpiCbStartOfRestart, vhpiCbEndOfRestart

A callback registered for reason vhpiCbStartOfRestart or vhpiCbEndOfRestart trigger when a restart command is processed by the tool. Please refer to the save and restart section of the foreign models chapter for more information on how these callbacks are used to restart foreign models or applications. The vhpiCbStartOfRestart callback occurs at the beginning of the restart of the VHPI models while the vhpiCbEndOfRestart callback occurs at the end of the restart operation. A model may not need to register for both callback reasons, but these two reasons are provided as a convenience to the user. For example, the user can use the first one to restore the foreign models private data and the vhpiCbEndOfRestart to re-register callbacks if necessary, since everything has been restored at this point. When vhpiCbEndOfRestart is called, the client is assured that all model instances that registered vhpiCbStartOfRestart have been initially processed. These callback registrations MUST happen during the save command in either of the vhpiCbStartOfSave or vhpiCbEndOfSave callbacks. The reason for that is that some information needs to be passed between the save action to the future restart action so that the foreign models can retrieve the correct saved data from the saved file (see save and restart of foreign models in Chapter 7). The callback registration consists in setting the cb_rtn field of the callback data structure to the callback function, the reason field to the vhpiCbStartOfRestart or vhpiCbEndOfRestart reason and the user_data field to an id identifying the location of the saved data. All other fields are ignored. That id is returned by the *vhpi_put_data()* function which saves data in the save file (see foreign models chapter). They are not repetitive callbacks.
If time is requested via vhpi_get_time() during the cbEndOfRestart callback, the time will be the time of the saved data that is being restarted.

8.6 vhpiCbStartOfReset, vhpiCbEndOfReset

A callback registered for reason vhpiCbStartOfReset or vhpiCbEndOfReset triggers when a reset command is processed by the tool. The vhpiCbStartOfReset callback occurs at the beginning of the reset of the VHPI models while the vhpiCbEndOfReset callback occurs at the end of the reset operation. A model may not need to register for both callback reasons, but these two reasons are provided as a convenience to the user so that the user can use the first one to reset the state of the foreign models private data and the vhpiCbEndOfReset to set up new callbacks. The callback registration can happen at anytime during simulation. The callback registration consists in setting the *cb_rtn* field to the function to be called, and the callback reason to either vhpiCbStartOfReset and vhpiCbEndOfReset. When the vhpiCbEndOfReset triggers the current simulation time is 0 ns. Please refer to the reset section under the foreign models chapter for more information on the reset sequence. The execution of all vhpiCbEndOfReset callbacks is followed by the initialization phase 1.0.1 in the annotated simulation cycle. Therefore, all foreign model initialization code is re-executed as part of initialization.
If time is requested via vhpi_get_time() during the cbStartOfReset, the time returned will be T_c from the previously completed simulation cycle. If time is requested via vhpi_get_time() during the cbEndOfReset, the time will be time 0.

Note: If simulation was restarted from time Tn, then reset sometime later, the current time after reset will become time 0 and, not Tn.

8.7 Callback function execution

The callback function specified by *cbDatap->cb_rtn()* is called when the condition (reason, time etc…) indicated by the callback registration becomes true and the callback is enabled A callback is said to be triggered, and responds by calling the callback function..
The callback function is called with a single argument. The argument (*cbDatap*) is a const pointer to a vhpiCbDataS structure. This callback data structure is allocated by the VHPI server and is not the original callback data structure that was passed by the user at the registration of the callback. The argument *cbDatap* contains information about the callback and its current state.that caused the routine to be invoked. The callback data structure contents should only be read by the client code and its contents including the handle denoted by the *obj* field are only valid for the duration of the callback function call. The handle provided is owned by the VHPI server and will not be the same handle provided by the user at the time of the registration. The client code must not release that handle.

During the callback routine execution, the access to the information model that is allowed is defined by the semantics of the specific callback and the phase of execution of the tool . The callback routine is of return type void.
The *user_data* field data may be provided at the registration of the callback, this field can be used to store private data or handles for example. The *user_data* value at registration is returned in the cbDatap parameter when the callback routine is executed.is preserved when the callback triggers. The *user_data* is never referenced by the VHPI interface and its dereference is not required to be a valid memory address. The *user_data* address will be constant for the life of the callback. The contents of that address can however be changed by the client code
No memory allocated by the user is read or written to by VHPI when a callback executes. User memory allocated at registration in a vhpiCbDataS is only read once when the callback is registered and is for conveying intent and registering the callback.

Procedural Interface References:

See “vhpi_register_cb()” to register a callback.
See “vhpiCbDataT” for passing callback data information.
See “vhpi_get_cb_info()” to retrieve a callback.
See “vhpi_remove_cb()”to remove a callback.
See “vhpi_enable_cb()” to enable a callback.
See “vhpi_disable_cb()’ to disable a callback.
See “vhpi_put_value()” to force, release or update a value.
See “vhpi_get_value()” to query an object value in various formats.

Errors:

Restrictions:

9. Value access and modification

This chapter describes how to access and modify values. This is one of the capabilities required to claim a compliance level of vhpiCapabilitiesP tovhpiProvidesForeignModel or vhpiProvidesDebugRuntime. This chapter defines the interface functions and associated data structures. Classes of objects that support value access will have a vhpi_get_value() operation defined (see information model), while those that support the modification of values will have a vhpi_put_value() operation defined. Additionally a function vhpi_schedule_transaction is defineded to schedule a transaction on a driver of a signal. In order to do that, one must access the drivers of basic signals. If a signal is not a basic signal, one must first access its basic signals according to the semantics described in Chapter ? on Connectivity.

The value functions can be used directly on objects that are scalars or arrays of scalars. In order to access values of composites that are more complex than an array of scalars, users are expected to traverse the composite to the level of a scalar or an array of scalars before using the value functions defined by the interface.

The interface defines a value structure as the mechanism to pass values between a model or application and the tool supporting the VHPI interface. The value structure and related types are defined as follows :

```
typedef struct vhpiValueS
{
  vhpiFormatT format;     /* IN/OUT: (depending on format) value
                                  format */
  int bufSize;            /* IN: size in bytes of the buffer */
  vhpiIntT numElems;      /* IN/OUT: number of array elements in the
                                 value,
                                 undefined value for scalars */
  vhpiPhysT unit;         /* IN/OUT: physical position of the unit
                               in which the physical value is
                               expressed */

  union
  {
    vhpiEnumT enumv, *enumvs;  /* OUT: enumeration */
    vhpiIntT intg, *intgs;     /* OUT: integer */
    vhpiRealT real, *reals;    /* OUT: floating point */
    vhpiPhysT phys, *physs;    /* OUT: physical */
    vhpiTimeT time, *times;    /* OUT: time */
    char ch, *str;             /* OUT:character or string */
    void* ptr, *ptrs; /* OUT: simulator representation value or access
value */
  } value;

} vhpiValueT;
```

The format values up to the vhpiRawDataVal format (included) are mandatory formats to be supported by vhpi_get_value, vhpi_put_value and vhpi_schedule_transaction. The types of the objects for which they can be used are described below. Other uses are undefined by the standard.
Implementations are free to enhance the set of supported formats. There is a standard defined format mapping to the type of the object which is supported by the functions *vhpi_get_value, vhpi_put_value, vhpi_schedule_transaction*. Other formats may be supported by the access value functionsand *vhpi_format_value* but these formatting of values are not defined by the standard.

```
typedef enum
{
```

```
    vhpiBinStrVal,
    vhpiOctStrVal,
    vhpiDecStrVal,
    vhpiHexStrVal,
    vhpiEnumVal,
    vhpiEnumVecVal,
    vhpiIntVal,
    vhpiIntVecVal,
    vhpiLogicVal,
    vhpiLogicVecVal,
    vhpiRealVal,
    vhpiRealVecVal,
    vhpiPhysVal,
    vhpiPhysVecVal,
    vhpiTimeVal,
    vhpiTimeVecVal,
    vhpiObjTypeVal,
    vhpiCharVal,
    vhpiStrVal,
    vhpiPtrVal,
    vhpiPtrVecVal,
    vhpiRawDataVal

} vhpiFormatT;
```

The definitions of vhpiEnumT, vhpiIntT, vhpiRealT, vhpiPhysT and vhpiTimeT are expected to be defined by the simulation vendors with the following requirements mandated by the standard,

- vhpiEnumT should be at least 32 bits wide
- vhpiIntT should be at least 32 bits wide
- vhpiRealT, vhpiPhysT and vhpiTimeT should be at least 64 bits wide

he value data structure and any associated buffer to hold values of composites should be allocated and managed by the users of the functions vhpi_get_value(), vhpi_put_value(), vhpi_schedule_transaction and vhpi_format_value(). The following are the interpretations of the fields in the value structure,

format
The format in which the data is desired (vhpi_get_value) or supplied (vhpi_put_value, vhpi_schedule_transaction, vhpi_format_value). This should be specified by the users. There are format tuples, one for scalars and the other for array of scalars, associated with the basic data types of enumeration, integer, character, floating point, physical and time. For each mandatory format, there is a field in the value union. The format vhpiStrVal can be used to retrieve values of objects of type string, enumeration type or array of characters . The format vhpiObjTypeVal has been provided to obtain a value in its native form without requiring the user to find out the type of the object. The interface will determine the most appropriate format in this case and change the format field to reflect that format of representation of the value, while returning the value. The format vhpiRawDataVal has been provided from a performance standpoint, to enable implementations to return the simulator representation of the value.

Note : Though time types are physical types, the standard makes a distinction in keeping with possible future extensions requiring a representation for time different from physical types.

bufSize
This field should be set by the user to the byte size of the user allocated value buffer and is required for values of array types. The corresponding value field of the union should be set to point to the value buffer. The interface will check if the buffer size is sufficient to hold the value and vhpi_get_value will return the required size if it is not. Consequently the size in bytes required to represent a value will be returned by a call to *vhpi_get_value* when setting bufSize to 0. The field bufsize is IN and is always set by the caller.

Note : For string values, the buffer has to be at least as large as the length of the string incremented by one for string termination. The string value returned for objects of array of characters should not add surrounding double quotes; a termination character \0 will be added at the end of the string value. The numElems field will be set to be the real length of the character string (excluding the extra \0); The bufSize should be set by the caller who allocates the buffer string to be filled up to the byte length of the string +1 (accounting for the \0). For all other array types , the buffer should be at least as large as the number of elements in the array multiplied by the size of the element data type.
The above rules apply to any string value obtained with any string format (vhpiStrVal, vhpiBinStrVal, vhpiHexStrVal and vhpiDecStrVal).

numElems

This field is used only for representing values of array types and should be set to the number of array elements represented by the union value field of str, intgs, reals, times, physs or ptrs . For string types, it is set to the string length including the termination character. For all scalar types, this value is undefined.

unit

This field is used for value representation of time or physical types; *vhpi_get_value* will set this field to the physical position of the unit in which the returned value is represented for physical types. A physical value represented in a vhpiValueT value structure is defined by the value.phys field and the unit field such that the multiplication of value.phys by the unit field should provide the physical value scaled to the base unit of its physical type. The unit field can be set by the user for obtaining a physical value in any physical position of unit while using the VHPI function *vhpi_format_value*. *vhpi_put_value* should accept physical values of any physical position representation. The VHPI property *vhpiPhysPositionP* can be applied to a reference handle of a unit declaration (vhpiUnitDeclK) of a physical type to query the physical position of that unit declaration. The function *vhpi_get_phys* should be used to query the *vhpiPhysPositionP* property.
The physical position of a physical literal of value integer 0 or floating point 0 is always 0.
If the unit is 1, the physical value(s) in value.phys(s) is(are) expressed in base units of the physical type.

value

This is defined to be a union, which contains the actual value of an object or expression. The following are the fields within this union,

1. **enumv**
   This field should be used for the positional values of enumeration typed object. VHPI enumeration define constants are defined for the standard logic, bit and boolean types.
2. **enumvs**
   This is a pointer field and should be used for arrays of enumeration types. This field should be set to point to a user allocated buffer.
3. **intg**
   This field is used for values of integer typed objects.
4. **intgs**
   This pointer field should be used for array of integers values. This field should be set to point to a user allocated buffer.
5. **real**
   This field should be used for values of floating point typed objects
6. **reals**
   This field should be used for values of arrays of floating point.
7. **phys**
   This field should be used for physical type values.
8. **physs**
   This field should be used for values of arrays of physical types.
9. **time**
   This field should be used for values of TIME typed objects
10. **times**
    This field should be used for values of arrays of TIME type.
11. **ch**
    This field should be used for values of character types.

12. **str**

This field should be used for values of string type and string formatting. The interface defines a separate function, *vhpi_format_value()* for doing string value formatting for other typed objects. (9.2*Formatting VHDL values*).

13. **ptr**

This field should be used to get the simulator/tool representation of the value for any type (vhpiRawDataVal format) or to get the access value of a variable of an access type (vhpiPtrVal format).

The standard does not mandate any specific data representation, nor does it make it mandatory for vendors to publish their data representations. The lifetime of this value is unspecified by the standard.

14. **ptrs**

This field should be used to get the access values for arrays of access type elements (vhpiPtrVecVal format).

There are various objects in the information model that carry values that can be accessed using the VHPI value mechanism. The class kinds that support values operations are:

<u>1. All sub-classes of the objDecl class:</u> *vhpiConstDeclK, vhpiSigDeclK, vhpiFileDeclK, vhpiVarDeclK, vhpiGenericDeclK, vhpiPortDeclK, vhpiSigParamDeclK, vhpiVarParamDeclK, vhpiConstParamDeclK, vhpiFileParamDeclK*

Their value can be fetched if the object declaration has been elaborated and the property vhpiAccessP allows reading of the object value (vhpiReadAccess bit is set) . If the value is fetched at the end of elaboration, the value is the default (`left) or initial value from the initial expression of the object declaration (as defined by the elaboration of the object). The value of a generic after elaboration will be the value after generic propagation. During simulation, the value fetched is the value of the object at this particular time.

For a *vhpiFileDeclK*, the value is the logical name of the file. The value is of type string. The value is the string value supplied in the declaration if present or the logical name the file was associated with during a call to FILE_OPEN if the file was opened during simulation. If the file is not opened at the time of the query, a warning is generated and the value structure is not filled with a value. In this case, *vhpi_get_value* returns a negative value to indicate that the call failed.

For shared variables of protected types, *vhpi_get_value* must be called through a *vhpi_protected_call* which obtains a lock on the variable. For a variable of an access type, the access value can be fetched: the access value is the address of the allocated object, the format to be used is vhpiPtrVal and vhpiPtrVecVal for arrays of access type. The dereference value designated by the current access value of the variable can be fetched from a handle to the dereference object (*vhpiDerefObjK*) which is obtained after applying the *vhpiDerefObj* method to the variable handle. If the variable has an access value of 0 (null) it does not designate an allocated object, in that case it is an error to apply the *vhpiDerefObj* method (dereferencing a null pointer); the *vhpiDerefObj* method should return a null handle and an error should be generated. The access value designates the created access object (just like a pointer). This is different from the value of the object which can be fetched by dereferencing the object. The value of a dereference name (xyz.ALL) will be the dereference value pointed by the object xyz. The handle kind of a dereference name is *vhpiDerefObjK*. Values of dereference names can be fetched with the basic formats defined the value structure: for example, a dereference value of a variable of type access to integer can be fetched with the vhpiIntVal format. The default value of a variable of an access type is NULL (NULL pointer). The initial access value of a variable with an initial expression will be the access value of the initial expression. A dereference object (*vhpiDerefObjK*) has a value.

For foreign subprogram calls, it is possible to get the actual values associated with the formal parameters if the mode of this parameter is either IN or INOUT (value can be read). For signal subprogram parameters, the value fetched is the effective value of the signal. In order to fetch the driving value, a handle to the driver must be obtained. Subprograms are dynamically elaborated therefore the values of their parameters or declared items can only be fetched when the program is currently executing or is suspended. We provide a method to get the current executing region (vhpiCurRegion) and a method to access the stack of an equivalent process (vhpiCurStack see subprogram call class diagram). The user can only fetch the values of parameters or declared items in a subprogram call if the subprogram call is either executing or is on the call stack of the current executing process or subprogram, or is a suspended process or on the stack of a suspended process.

For port or signal declarations (vhpiPortDeclK, vhpiSigDeclK), the value fetched is the effective value. For the in part of an inout port (vhpiInPortK), or for an inout mode port, the value fetched is the effective value; for the out part of an inout port (vhpiOutPortK), the value fetched is the driving value.

2. Any sub-class of the class name and class literal has a value *(vhpiIndexedNameK, vhpiSliceNameK, vhpiSelectedNameK, vhpiAttrNameK, vhpiDerefObjK).*

The mechanism is exactly the same for the name sub classes as for the sub classes of the objDecl class. It is not possible to fetch directly the value of any other expression or function call. *vhpi_get_value* should support getting values of locally static names; an implementation may optionally provide support for globally static names.
There is the possibility of getting a value for handles of any of the sub-classes of class literal. The standard defines a more convenient mechanism using properties for each sub-class of the class literal: *vhpiIntValP* for class vhpiIntLiteralK, *vhpiRealValP* for class vhpiRealLiteralK, *vhpiPhysValP* for vhpiPhysLiteralK and *vhpiStrValP* for classes *vhpiStringLiteralK, vhpiBitStringLiteralK, vhpiCharLiteralK* and *vhpiEnumLiteralK*.

3. Simulation objects:
A driver (*vhpiDriverK)*has a value which is its current driving value. This driving value can only be fetched after simulation initialization phase has been completed. See PA046. The driving value of a driver reflects the value of the last matured transaction.

Values can be fetched after simulation initialization has completed in a simulation session, or after elaboration in an elaboration session.

9.1 Accessing VHDL object values

```
(PLI_INT32T) vhpi_get_value(vhpiHandleT refHdl, vhpiValueT* valuep)
```

The interface defines a function *vhpi_get_value* for getting values. The first parameter, *refHdl* represents the handle to the object of which the value is required. The second parameter is a user allocated value structure. This structure should have the format field set by the user based on the VHDL type of the object referenced by refHdl. For a description of the formats allowed for each VHDL basic type, see the value structure description.

For scalar type object values, the interface copies the value into one of the scalar fields in the value union within the value structure. For arrays of scalars, the users should allocate a buffer large enough to hold the value and place the pointer to this allocated buffer into an appropriate pointer field in the value union within the value structure. The interface then copies the value into the allocated buffer. *vhpi_get_value* will not return a partial value.

Values can be accessed for all the classes of objects that possess values (that support the *vhpi_get_value* operation in the information model), which includes the set of classes outlined in 9.1 above.

9.2 Formatting VHDL values

A function *vhpi_format_value* is provided to change the representation of a value in a different format. This function takes two value pointers, the first one is the input value, the second one is the output value. The first value contains the value in a given format, the second value structure should have:

- the **format** field set to the requested format,
- the **bufSize** field set to size of the user allocated buffer (if formatting values requires the allocation of a buffer),
- the **unit** field set to the physical unit position for time or physical type unit conversions.
- the **union** field which corresponds to the requested format set to point the user allocated buffer.

This function can be used to format value into non native format representations as defined by the vendor or to do unit scaling for time or physical types. If specified, handle to the type corresponding to the value.if null it may limit the conversions which are possible.

9.3 Updating VHDL object values

The class of objects that support runtime modification of values is a subset of the class of objects that support fetching values. *vhpi_put_value* can be used during simulation to change the value of objects which possess the *vhpi_put_value* operation, or during elaboration to set the initial driving value of a foreign driver, the initial value of signals, ports or shared variables of a foreign architecture or the return value of a foreign function. See PA047.There are two types of update: deposit or force. Both can be applied with or without propagation of the value. Five update flags are defined; *vhpiDeposit*, *vhpiDepositPropagate*, *vhpiForce*, *vhpiForcePropagate* and *vhpiRelease*. Another flag *vhpiSizeConstraint* is also available to set the constraint of an object of an unconstrained type. Propagation of the value is only meaningful for signal class of objects. vhpiForcePropagate and vhpiDepositPropagate are identical to respectively force and deposit for objects which do not belong to the signal class. *vhpi_put_value* can be called at any point during initialization and simulation and additionally during elaboration for setting the return value of foreign functions.

Deleted: only

Formatted: Font: Italic

Deleted: on a runtime model

The valid object classes for immediate update are:
1. Subclasses of the objDecl class : *vhpiSigDeclK, vhpiVarDeclK, vhpiPortDeclK, vhpiOutPortDeclK, vhpiSigParamDeclK, vhpiVarParamDeclK*

Objects of class *vhpiConstDeclK and vhpiFileDeclK* cannot be modified using *vhpi_put_value* . The behaviour of *vhpi_put_value* on a *vhpiGenericDeclK is unspecified* by the standard. The exception to this rule is with the use of VHDL functions that have a foreign C implementation. These functions can be used in initialization expressions, and hence can be used indirectly to set the values of these classes of objects.

Parameters to subprograms can be modified only if their mode is either vhpiOut or vhpiInOut.

Immediate update can apply to GUARD signals. Direct update of implicit signals such as signal attributes is not permitted. If an event is created for a signal, its implicit signals such as the signal attributes are updated in the signal evaluation phase as a result of the corresponding signal being updated. Similarly guarded signals assignments can be triggered as a result of the GUARD signal becoming true.

Ports of all modes support modification of their value. For ports of mode *vhpiOut* and for *vhpiOutPortlK* (out part of an inout port) and or signal parameter of mode vhpiOut or vhpiOutSigParamK (out part of an INOUT sigPamDecl) this operation shall be equivalent to changing the port driving value. Similarly, modification of the values of subprogram signal parameters which are of mode vhpiOut shall be equivalent to changing the driving value of the actual. For port of other modes or vhpiInPortK (in part of an INOUT port), or vhpiInSigParamK (in part of an INOUT sigParamDecl) or vhpiSigDeclK, the value that is modified is the effective value.

Immediate update on shared variable of protected types must be accomplished through a *vhpi_protected_call* which acquires the lock on the variable, otherwise an error should be generated. The behaviour of *vhpi_put_value* is unspecified for shared variables of non protected types.

<u>2. Subclasses of the class name :</u> *vhpiIndexedNameK, vhpiSliceNameK, vhpiSelectedNameK, vhpiDerefObjK.*

vhpi_put_value shall not be used on prefixed names that are derived from *vhpiConstDeclK* as also subclasses of the class *vhpiLiteralK*.
The rules and restrictions stated in paragraph 1 for object decls also stand for sub-elements of these objects.

<u>3. Function call handles:</u> *vhpiFuncCallK*

The return values of function calls can be set by depositing a value on the function call object handle. Setting the return value for foreign functions is required by the standard. Setting the return value with *vhpi_put_value* of VHDL functions (overriding the returned VHDL value) is a legitimate vendor extension which is not defined by the standard.
If the return type of a function call is a composite type more complex than an array of scalars, the caller shall iterate over the sub-elements of the return type and call *vhpi_put_value* on each of these. The iterations on indexedNames and selectedNames are ordered iterations defined by the aggregate rule in the VHDL LRM 1076-2001.
Setting the return value of an unconstrained function shall be done by first calling *vhpi_put_value* to set the number of elements of the unconstrained function return parameter. The **numElems** field of the value data structure shall be set to the total number of elements of the returned array type. The values of the other fields of the value structure are unspecified. The flag parameter of *vhpi_put_value* shall be set to *vhpiSizeConstraint*. Then a second call to *vhpi_put_value* will modify the returned value by passing the actual value to be returned. A runtime error shall be generated if the size of the value in the second *vhpi_put_value* call does not match the size specified in the previous *vhpi_put_value* call. Same mechanism applies if a function returns a record type which one of the record elements is an unconstrained array.
From a subprogram call handle, users can traverse the call stack and access subprogram variables. The values of these variables can be modified. All local variables that are on the call stacks of the currently executing process or subprograms can also be modified.

<u>4. Driver handles:</u> *vhpiDriverK*

Drivers can have their current driving value read but and updated with respectively *vhpi_get_value* and *vhpi_put_value*. An additional function *vhpi_schedule_transaction* provides the functionality comparable to a VHDL simple signal assignment statement: the function can schedule a transaction with zero or non zero delay, a mode of transport or inertial and a optional pulse rejection limit.
The semantics of the update flags:

Deleted: not written
Formatted: Font: Italic
Formatted: Font: Italic
Formatted: Font: Italic
Deleted: . In order to change the current driving value of a driver, the mechanism is indirect through the use of the interface function *vhpi_schedule_transaction*.¶

The modification of variable values and returned values of function calls always takes immediate effect. For a variable, the flags vhpiDepositPropagate, and vhpiForcePropagate have the same effect as respectively the vhpiDeposit and vhpiForce flags. For a function call, vhpiDepositPropagate, vhpiForcePropagate and vhpiForce have the same effect as vhpiDeposit. vhpiRelease has no effect on function call handles. The behavior of *vhpi_put_value* with the vhpiForce flag is unspecified for shared variables of non protected types. For shared variables of protected type, any immediate update must be done through a *vhpi_protected_call*, an error should be generated if an immediate update is attempted outside of a *vhpi_protected_call.* The modification of signal and port values has special semantics and are described in the following paragraphs. The terms signal and port are used to denote the general class of objects which are either signal or port declarations or the out side of inout port declarations, or sub-element of these. All modes can be applied to update the value of a port or signal kind of object.

The interface supports four different modes of updating the values of signals and ports,

1. Depositing a value without propagation for the current cycle
2. Depositing a value with propagation for the current cycle
3. Forcing a value without propagation until release
4. Forcing a value with propagation until release

The following enumeration type defines the set of flags that can be used while updating the values of signals and ports

```
typedef enum
{
  vhpiDeposit,
  vhpiDepositPropagate,
  vhpiForce,
  vhpiForcePropagate,
  vhpiRelease,
  vhpiSizeConstraint

} vhpiPutValueModeT;
```

The following is a description of these flags,

vhpiDeposit

- The value is deposited but not placed on hold. There is no propagation through the signal network. If this happens at the beginning of a cycle or during network propagation (after a.1.0 but before d.0.1), the value could be overwritten through VHDL signal update or network propagation.
- If this happens after network propagation and before process execution (after c.1.0 but before d.1.0), all readers of this signal will see the new value for the current cycle but the signal network may be inconsistent.
- If the value is deposited during process execution (after d.0.1), there is no guarantee on whether all readers will see the same value in the current cycle. This last feature is non-portable.

vhpiDepositPropagate

The value is deposited with propagation. The value is not placed on hold. This form of update will be effective for the current cycle alone and can be done with the following semantics in the various phases of a given simulation cycle,

Beginning of a cycle (after a.1.0 but before b.1.0)

In this case the value is deposited and propagated in the same cycle.
Callback reasons that can be used to stop at the beginning of a cycle are :

- vhpiCbNextTimeStep (a.1.1)
- vhpiCbStartOfNextCycle (a.2.1)
- vhpiCbAfterDelay (a.2.2)

During or after network propagation but before VHDL process execution (after a.2.2 but before d.1.0)

In this case VHPI will introduce a new delta cycle, in which the value change with propagation takes effect. The value is not changed for the current delta cycle.

Callback reasons that can be used to get to this part of the simulation cycle :

- vhpiCbValueChange (after a.2.2 but before d.0.1)
- vhpiCbStartOfProcesses (foreign models execute) (d.0.1)

During non-postponed process execution (after d.0.1 but before f.1.0)

The interface introduces a new delta cycle, within which the value is updated with propagation. The value is not changed for the current cycle.
Callback reasons that can be used to get to this part of the simulation cycle :

- vhpiCbResume (after d.0.1 but before d.1.0)
- vhpiCbStmt (after d.0.1 and before d.1.0 but after vhpiCbResume)
- vhpiCbSuspend (during e.1.0 and before e.1.1)
- vhpiEndOfProcesses (e.1.1)
- vhpiCbLastKnownDeltaCycle (f.1.1)

<u>During and after postponed process execution</u> (after g.1.0 through the rest of the simulation cycle)
It is an error to deposit a value with propagation at this stage. This stage includes postponed process execution.
Callback reasons that can be used to get to this part of the simulation cycle :

- vhpiCbStartOfPostponed (g.1.1)
- vhpiCbEndOfTimeStep (h)
- vhpiCbQuiescence (after h)

vhpiForce

- The signal or port value is forced and placed on hold until release. There is no propagation. VHDL signal updates or network propagation cannot overwrite the value. The value will remain on hold, until released using *vhpi_put_value* with the vhpiRelease flag. Another force can be applied on an already forced value.
- If this happens before process execution (before d.1.0), all readers of the signals or ports will see the forced value.
- If this happens during process execution, not all readers of the signal or port are guaranteed to see the new value. This mode of usage will be non-portable.

vhpiForcePropagate

This flag can be used to achieve the same effect as vhpiDepositPropagate, with the added consequence of the value being placed on hold, until the user releases the hold using the vhpiRelease flag. VHDL signal updates or propagation will not overwrite the value as the value is put on hold with the force. Another force can be applied on an already forced value, which replaces the previously forced value

The semantics with respect to the various phases of a simulation cycle are precisely the same as with vhpiDepositPropagate.

Immediate update with vhpiDepositPropagate or vhpiDeposit during network propagation has an indeterminate result and is not portable. Immediate update with vhpiForce or vhpiForcePropagate have a determinate result for the signal that is forced but the network may be inconsistent with respect to the model.

- The integer property vhpiIsForcedP can be used to query if the object value is forced.

vhpiRelease

- This flag can be used to release the hold placed by a force. An object value value can be placed on hold using one of vhpiForce or vhpiForcePropagate. The hold can be released using vhpiRelease. After the value has been released, the object value can be updated through VHDL signal update or network propagation (it does not revert to the value prior to the force).
- vhpiRelease has only the effect of releasing the value of objects for which the vhpiProperty vhpiIsForcedP is TRUE.

- The pointer to the value structure valuep is not required when vhpi_put_value if called with the vhpiRelease flag. If a non null valuep pointer is provided, it will be ignored.

Formatted: Bullets and Numbering

Formatted: Indent: Left: 0.25"

vhpiSizeConstraint

- *vhpi_put_value* can be used with this mode to set the constraint of the reference handle if the type of the reference handle is unconstrained. A subsequent call to *vhpi_put_value* will update the value of the reference handle. An error should be generated if the size constraint indicated by the first call does not match the size of the value in the second *vhpi_put_value* call.

It should be noted that VHDL subprograms that have a VHPI foreign attribute can be executed at any point in a simulation cycle, which implies that the users will have access to values and can update values virtually at any point during signal update, network propagation and process execution. The semantics described under the various flags apply for these instances of value modification.

Further, any value change callbacks registered on signals, ports will be triggered when the user **updates their value and creates an event** using either vhpiDepositPropagate or vhpiForcePropagate. Using vhpiDeposit or vhpiForce on signal and ports will not involve triggering of any value change callbacks. Also, callbacks registered on signals and ports with reason vhpiCbForce will be triggered when the signal or port is forced using the interface. Similarly, all callbacks registered on signals and ports with reason vhpiCbRelease will be triggered when the signal or port is released using the vhpiRelease mode through the interface. Value change callbacks for variable trigger whenever the value of the variable has changed.

A deposit value has no effect if the object is forced. A forced signal can be forced to a new value, the last force takes effect. A release has no effect other than on a forced object.
vhpiCbForce or vhpiCbRelease callbacks trigger only if the force or release occurs.
informative note: a warning may be generated by an implementation if *vhpi_put_value* has no effect.

If *vhpi_put_value* detects a range constraint violation between the value and the target object, *vhpi_put_value* shall generate an error for range constraint violation if such a case is detected; that error can be checked immediately by calling vhpi_check_error.
informative note: the detection may occur later and be reported in an error cbk.

9.4 Scheduling transactions on signal drivers

```
(PLI_INT32T) vhpi_schedule_transaction(vhpiHandleT refHdl,
              vhpiValueT* valuep, PLI_INT32T numValues, vhpiTimeT*
delayp,
              PLI_UINT32 delayMode, vhpiTimeT* pulseRejp)
```

The interface provides a capability to schedule transactions on drivers. The function to use is *vhpi_schedule_transaction*, which has the above signature. The following are the parameters that should be passed to this function,

1. **refHdl**
   This is a VHPI handle that represents an object that supports value scheduling.
2. **valuep**
   This is a pointer to a single value structure or an array of value structures.
3. **numValues**
   This field is used to specify the number of values that are being passed through the value structure pointer, valuep.
4. **delayp**
   This is the delay, always with respect to the current simulation time, that will be used in processing the scheduling operation.

5. **delayMode**
   This represents the delay mode, which could be one of *vhpiInertial* or *vhpiTransport*.
6. **pulseRejp**
   This is a pulse rejection limit that can be specified. If the desire is to have an inertial delay without a pulse rejection limit, this field should be set to null.

The reference handle should be either a handle to a driver, of type *vhpiDriverK*, a handle to a VHPI driver collection, of type *vhpiDriverCollectionK*.

The drivers returned by the interface will always be drivers to basic signals. A basic signal is either a scalar signal or a resolved composite typed signal. A collection of drivers to basic signals can be constructed using the VHPI call *vhpi_create* and used to schedule updates collectively to all the drivers in the collection. A collection of drivers can be created only for the same unresolved composite typed signal. Drivers driving parts of different signals cannot be placed in the same driver collection.

For a driver to a scalar signal, a single value structure should be passed with the format set to indicate a scalar value. For collections of drivers or for a composite driver, the following rules, based on the type of the driven signal, should be used to associate value structures with driver transaction values for scheduling,

- When the type of a signal is an unresolved array of scalars, a collection of drivers can be created to one or more of the scalar sub-signals, when the type of a signal is a resolved array of scalars, a single composite driver handle should be used to schedule a transaction. In either cases, a single value structure should be passed with a format being a vector format, with an allocated value buffer having as many scalar values as the number of scalars. In the case of a driver collection, the *numElems* field of the value structure should be equal to the *numMembers* property of the collection and in the case of a composite driver, it should be equal to the number of scalars. The *numValues* parameter should always be 1.
- When the type of a signal is either an array of composites or a record, an array of value structures should be passed, whether we have a single composite driver or a collection of drivers. The value structures should be passed at the coarsest granularity possible. This implies that for all subsignals of the composite signal, which are arrays of scalars, the value has to be passed as a vector for all the scalars. For all other subsignals, the same rule should be used recursively, down the type hierarchy. All value structures passed can only be either values to scalars or arrays of scalars.

The following table describes the relationship between signal subtypes, drivers, and the value structures required to be passed.

Signal subtype	**Driver or Collection handle that can be used to schedule transactions.** Note: it is possible to not get any drivers if the signal is not driven	**Example**	**Number of value structure s**	**Description of value structures**
Unresolved scalar	0..1 scalar driver • A scalar driver handle.	BIT	1	Single value structure with the format set to an appropriate scalar format.

Resolved scalar	* scalar drivers. A scalar driver handle.	STD_LOGIC	1	Single value structure. • Scalar format and value for a driver handle
Unresolved composite Resolved composite	0..1 driver for each basic signal in the composite signal. • A driver handle to one of the basic signals • A collection of drivers to one or more basic signals. * composite drivers • A composite driver handle.	STD_LOGIC_VECTOR	1	Single value structure. • Scalar format for a single driver handle • Vector format for a collection of driver handles
		RECORD WITH N SCALAR FIELDS	N	N value structures with scalar formats, one per field
		RECORD WITH N SCALARS AND M VECTORS OF SCALARS	M+N	Array of value structures, • N value structures with scalar formats, one per scalar field. • M value structures with vector formats, one per vector field.

The layout of value structures passed adhere to the following rules:

1. For all composite drivers, the value structures passed in should be in the same order as the declaration order in the typemark of the resolved subtype.
2. For all collections of drivers, the value structures passed in should be in the same order as the sequence in which drivers were added to the collection, with Rule 1 applying, whenever we find a driver for a resolved composite in the collection. Can we have a collection of collection of drivers?

The vhpiCbTransaction callback reason is provided to be able to access the time/value pair of a transaction. The callback is registered on the driver or signal handle and triggers when a transaction matures.

A single value structure cannot be used to get the current driving value of a driver collection or a composite driver which is not an array of scalars. The only mechanism is to iterate over all the individual drivers and get their values.

A driver or a set of drivers can be disconnected with *vhpi_schedule_transaction()*, by posting a NULL transaction. The drivers involved in a request to disconnect using *vhpi_schedule_transaction* should be drivers corresponding to guarded signals assigned by sequential signal assignments. Using vhpi_schedule_transaction to post a NULL transaction to other type of drivers is undefined by the standard. Vendors may choose to extend the functionality defined by the standard by allowing scheduling of NULL transactions on drivers corresponding to guarded concurrent signal assignment statements to guarded signals. In this case the behavior of the driver on posting a NULL transaction will be specified by the vendor tool.
Issue on disconnecting guarded signals.

Any guarded signal which is disconnected using *vhpi_schedule_transaction* can be reconnected, either using *vhpi_schedule_transaction*, by posting a non-NULL transaction, as with a valid set of value structures, or through VHDL, when a guard expression becomes TRUE and a non-NULL signal assignment takes effect.

At a given point in time, there can be only one active disconnection scheduled for a driver through VHPI or VHDL?.

The disconnection of drivers can also be done using *vhpi_schedule_transaction* on procedure OUT mode signal parameters, as long as the actual is a guarded signal (why guarded signals). The same restrictions as noted above apply in this case as well.

A transaction scheduled with zero delay will take effect in the next delta cycle. Such zero delay transactions can be posted:
a) during non-postponed process execution
b) in a vhpi callback function which was registered for reasons:
  i) vhpiStartOfProcesses
  ii) vhpiCbStmt,
  iii) vhpiCbResume
  iv) vhpiCbSuspend
  v) vhpiCbSensitivity
  vi) vhpiCbTimeOut
  vii) vhpiCbRepTimeOut
  viii) vhpiEndOfProcesses
  ix) vhpiLastKnownDeltaCycle

It shall be an error to post zero delay transactions at any other time. *vhpi_schedule_transaction* for non zero delay transaction can be called at any of a) and b) and additionally during postponed process execution.

10. Utilities

10.1 Getting current simulation time

(PLI_VOID) vhpi_get_time(vhpiTimeT * time_p, long *cycles_p)

This function shall get the current simulation time. The caller must allocate a structure of type *vhpiTimeT* and pass a pointer to this structure to the function. The time will be returned in the simulation time precision. If the users require the conversion of the returned time value in other units, they should use the interface function *vhpi_format_value*. The simulator time unit precision can be retrieved with the property *vhpiPrecisionP*. This property returns the physical position of the unit of the standard TIME type which represents the minimum time unit precision of the tool (simulator). This property can be also used to interpret the unit in which time values are returned in callback time fields.

If the *time_p* parameter is set to NULL, then this function returns the absolute number of simulation cycles from start of simulation, through the second parameter.

If the *time_p* parameter is not NULL, then this function returns the current simulation time through *time_p* and the relative number of delta cycles that have been executed within the current simulation time step through the second parameter.

If the second parameter is NULL, then the relative or absolute delta cycle information is not returned. Further, the second parameter if non NULL, should be the address of an integer type at least 64 bits wide.

Procedural Interface References:

Use "vhpi_get_phys(vhpiPrecisionP, NULL)" to access the simulator precision of TIME values.
See vhpi_user.h file for the constant value definitions of the type and unit field.

10.2 Printing

10.2.1 Printing to the stdout, log files, displaying messages

```
(PLI_INT32T)vhpi_printf(const PLIBYTE8 * *format,…);
```

This function allows a VHPI application to send messages to the output files defined by the tool.

10.3 Error checking and handling

```
(PLI_INT32T) vhpi_check_error(vhpiErrorInfoT *errorp);
```

This function can be called to determine if the last previous VHPI function call had an error. It can either check if an error occurred or retrieve detailed error information. *vhpi_check_error()* takes an argument (pointer to error structure) and returns 0 if no error, or 1 if an error occurred.
If the parameter errorp is non null and if the previous VHPI call generated an error, the error information structure pointed out by errorp will be filled up on the return with the last error information. The error information structure must be allocated by the caller.
If 0 is returned, the error information structure field are meaningless. The message string is a static string buffer which contains the formatted error message. The error message is only valid until the the next VHPI function call.
File and line information are optional fields that can indicate for example the foreign model instance which caused the error, or the VHDL item from where an error originated. (see example in the functional reference). The error information is NOT persistent. The internal error information structure is static.

A tool flag could enable or disable the display of all VHPI errors to STDERR, STDOUT or LOG file. The configuration of which files to send the VHPI errors should be specified at the initialization (bootstrap/registration phase) of the VHPI interface. Need for a special function ? (Refer to section 8.4Printing messages).

Filtering of some errors can be performed via the registration of an error handler callback function. Such a callback must be registered for reason *vhpiCbPLIError*. (Refer to the callback chapter).

NOTE: Error messages may be printed to STDERR or LOG file. The number of errors displayed is not related to how many calls to *vhpi_check_error()* are in the code but to the number of errors generated by VHPI function calls that are not filtered out by the error handler.

vhpi_check_error() allows the user to check for VHPI errors and eventually take actions regarding certain types of error.

The error information structure is used for building the error message that can be printed to the screen and log file and is used by the *vhpi_check_error()* function.

Such an error message could look like:

VHPI: <vendor specific ERRCODE>: <message>
[<file> <line>]

10.4 Tool control

The VHPI functions *vhpi_assert(), vhpi_control()* can be called to affect the execution control flow.

11. Procedural Interface Reference manual

11.1 vhpi_assert()

vhpi_assert()			
Synopsis:	Raise an assertion		
Syntax:	vhpi_assert(severity, formatmsg, …)		
	Type	**Description**	
Returns:	int	0 on success, 1 on failure	
	Type	**Name**	**Description**
Arguments:	vhpiSeverityT	severity	The severity of the assertion
	const PLI_BYTE8 *	formatmsg	The assertion message

vhpi_assert() shall be equivalent to the execution of a VHDL report statement. The function shall return 0 on success and 1 on failure. The function takes a variable number of arguments. The first argument, *severity,* shall be a value corresponding to one of the VHDL severity levels: *vhpiNote, vhpiWarning, vhpiError, vhpiFailure.* The second argument, *formatmsg,* is the formatted assertion string that gets printed. The formatted string shall use the same formats as the C printf functions. It is possible that a *vhpi_assert()* call causes the simulation to stop in the same way that a VHDL assert statement would do.

Example:

```
int check_clock_signal(scopeHdl)
vhpiHandleT scopeHdl; /* a handle to a scope */
{
vhpiHandleT clkHdl;
vhpiValueT value;

/* look up for a VHDL object of name clk at the scope instance */
/* get a handle to the clk named object */

clkHdl = vhpi_handle_by_name(“clk”, scopeHdl);
if (!clkHdl) return 1;
value.format = vhpiLogicVal;
vhpi_get_value(clkHdl, &value);
if (value.logic == vhpiBit0) {
    vhpi_assert(vhpiError, “clock not high: %d”, value.logic);
    return 1;
}
return 0;
```

11.2 vhpi_check_error()

vhpi_check_error()			
Synopsis:	Retrieves information about a VHPI function error		
Syntax:	vhpi_check_error(errp)		
	Type	**Description**	
Returns:	int	0 if no error, non zero if an error had occurred.	
	Type	**Name**	**Description**
Arguments:	vhpiErrorInfoT *	errp	NULL or pointer to a structure containing error information
Related functions:	See each VHPI function for related standard or vendor specific error codes.		

vhpi_check_error () shall check if the last previous call to a VHPI function had caused an error. The function shall return 0 if no error occurred or non zero if an error had happened. If the error detailed information is not needed, NULL can be passed to the function. If the errp is non null, the error information structure to which it points to will be filled up with the error information. The error information structure must be allocated by the caller.

If 0 is returned (no error was detected), all field values are meaningless. The *severity* field indicates the severify level of the error. The *message* field is a pointer to a static buffer of the formatted error message. This error message string may be overwritten by subsequent calls to **vhpi_check_error().** The *str* field can be used for various purposes: either to put a mnemonic string abbreviation of the error, or the name of the vendor product where the error originated. The *file* and *line* fields are optional fields and respectively indicate the VHDL file name and line number corresponding to a VHPI handle from where the error originated. In the case where the file and line number cannot be provided, these fields should respectively be set to null and -1.

```
typedef struct vhpiErrorInfoS {

    vhpiSeverityT severity;/* the severity level of the error */
    char          *message; /* the error message string */
    char          *str;   /* vendor specific string */
    char          *file; /* Name of the VHDL file where the VHPI
                            error originated */
    int   line;        /* line number in the VHDL file of the
                          item related to the error */
} vhpiErrorInfoT;
```

```
typedef enum {
     vhpiNote = 1,
     vhpiWarning = 2,
     vhpiError = 3,
     vhpiFailure = 4,
     vhpiSystem = 5,
     vhpiInternal = 6
```

```
} vhpiSeverityT ;
```

Example 1:

```
vhpiErrorInfoT err;

if (!vhpi_check_error(&err))
        /* continue VHPI code */
else switch (err->severity)
    {
      case vhpiError:
      case vhpiFailure:
      case vhpiInternal:
           return;
      case vhpiSystem:
            if (errno == …)
                  return;
      default:
          /* examine and decide if need termination */
          ...
    }
```

Example 2:

```
1   entity top is
2   end top;
3   architecture my_vhdl of top is
4     constant val: integer:= 0;
5     signal s1, s2, s3: BIT;
6   begin
7     u1: C_and(s1, s2, s3);
8     process (s1)
9     variable va: integer:= val;
10    begin
11       va = myfunc(s1);
12    end process;
13  end my_vhdl;

/* hdl is a handle to the root instance */
void traverse_hierarchy(hdl)
vhpiHandleT hdl;
{
vhpiHandleT subHdl, itr, duHdl;
vhpiErrorInfoT err;

itr = vhpi_iterator(vhpiInternalRegions, hdl);
/* if error code is > 0 do not continue */
if (vhpi_check_error(NULL))  return;

if (itr)
while (subHdl = vhpi_scan(itr)) {
  duHdl = vhpi_handle(vhpiDesignUnit, subHdl)
  if (vhpi_check_error(&err) > 0)
  {
      if (err->severity > vhpiWarning)
          if (err->file != NULL)
             vhpi_printf("An error occurred during call to
                          traverse_hierarchy
                      at filename %s line %d\n",
                      err->file, err->line);
```

```
      else
         vhpi_printf("An error occurred during call to
                      traverse_hierarchy\n");
      return;
   }
  switch (vhpi_get(vhpiKindP, subHdl)) {
  ...
  }
}
}
```

The following error will be generated and displayed because the internal region of a process kind does not have a 1-to-1 method to the bound designUnit.

```
An error occurred during call to traverse_hierarchy at file myvhdl.vhd
line 8
```

11.3 vhpi_compare_handles()

vhpi_compare_handles()			
Synopsis:	Compare two handles to determine if they reference the same object		
Syntax:	vhpi_compare_handles(hdl1, hdl2)		
	Type	**Description**	
Returns:	int	1 if the two handles refer to the same object, 0 otherwise	
	Type	**Name**	**Description**
Arguments:	vhpiHandleT	hdl1	Handle to an object
	vhpiHandleT	hdl2	Handle to an object

vhpi_compare_handles () allows to determine if two handles are equivalent. Handle equivalence cannot be done with the C '==' comparison operator.

Example:

```
vhpiHandleT find_clock_signal(scopeHdl)
vhpiHandleT scopeHdl; /* a handle to a scope */
{
vhpiHandleT sigHdl, clkHdl, itrHdl;
int found = 0;

/* look up for a VHDL object of name clk at the scope instance */
/* get a handle to the clk named object */

clkHdl = vhpi_handle_by_name("clk", scopeHdl);
itrHdl = vhpi_iterate(vhpiSigDecl, scopeHdl);
while (sigHdl = vhpi_scan(itrHdl)) {
      if (vhpi_compare_handles(sigHdl, clkHdl))
      {
          found = 1;
          break;
      }
      else vhpi_release_handle(sigHdl);
}
vhpi_release_handle(itrHdl);
if found
   return(sigHdl);
else return(NULL);
```

11.4 vhpi_control()

vhpi_control()			
Synopsis:	Send a control request to the tool		
Syntax:	vhpi_control(command, …)		
	Type	**Description**	
Returns:	int	0 on success, 1 on failure	
	Type	**Name**	**Description**
Arguments:	int	command	The command request
	-	varargs	Variable number of command specific arguments

vhpi_control () provides some control capabilities over the tool. The argument ***command*** specifies the command request. The following command constant values are defined by the standard: ***vhpiStop, vhpiFinish*** and ***vhpiReset***. A tool may also define some specific commands and define additional command constants which require some command specific arguments specified as ***varargs***. Such commands can be used to pass information from VHPI user function to the tool.
All commands are queued and executed at a "safe point" by the tool.
I030: Francoise provide more precise definition of the safe point.
vhpiFinish should not cause the tool to return to the host environment but rather cause the tool to reach the vhpiEndOfSimulation point.

If multiple calls to vhpi_control are made in sequence, they should be all queued.
Further the standard commands reset, stop and finish are executed in the order they were queued.
Other vendors commands may be queued (some may not require queueing) and the order is unspecified in the order in which they are executed by the tool. (Basically it is left to the tool to decide what to do on its vendor specific commands).

If the argument value is set to *vhpiStop,* after returning from the VHPI user code, the simulator will stop.

If the argument value is set to *vhpiFinish,* it would request the simulation to finish the execution of all scheduled events in the current delta cycle but not to exit. The C user function continues execution even after the ***vhpi_control()*** until it returns control to the simulator.

If the argument value is set to *vhpiReset,* upon return of the VHPI user function,the simulator will execute the sequence of actions to go back to simulation time zero. The sequence of actions is defined in the standard draft section 7. The sequence of events gives the possibility for a foreign model to clean up its allocated memory and reinitialize its data structures for example in the callback function for reason vhpiCbStartOfReset). The design is not re-elaborated.

<u>Example:</u>

```
void user_app()
{

/* Application traverse hierarchy */
...
/* Application collect information */
...
vhpi_control(vhpiFinish);
```

}

11.5 vhpi_create()

vhpi_create()			
Synopsis:	Create a vhpiProcessStmtK or a vhpiDriverK for a vhpi foreign model or a vhpiDriverCollectionK or a vhpiAnyCollectionK		
Syntax:	vhpi_create(kind, refHdl, processHdl)		
	Type	**Description**	
Returns:	vhpiHandleT	A vhpiDriverK or vhpiProcessStmtK or vhpiDriverCollection handle on success, null on failure.	
	Type	**Name**	**Description**
Arguments:	vhpiClassKindT	kind	Class kind of the handle to be created
	vhpiHandleT	refHdl	Handle to a basic signal or to a foreign architecture
	vhpiHandleT	processHdl	Handle to a vhpiProcessStmtK or null
Related functions:			

vhpi_create () shall be used to create a vhpiDriverK of a VHDL basic signal to drive a VHDL signal from a VHPI foreign model, or to create a vhpiProcessStmtK within a vhpi foreign model, or to create a collection of drivers. This function should only be called during elaboration of a foreign model for creating drivers and processes. There is no restriction on when creation of collections can occur.The first argument, *kind,* specifies the kind of handle to be created (*vhpiDriverK, vhpiProcessStmtK* or *vhpiDriverCollectionK*). If ***kind*** is set to *vhpiDriverK*, the function creates and returns a driver for the basic signal/process pair respectively denoted by the handles ***refHdl*** and ***processHdl***. If ***kind*** is set to *vhpiProcessk*, the function creates and returns a vhpiProcessStmtK for the foreign architecture denoted by the handle ***refHdl***, the last parameter is set to null. For creation of collections, the reference handle must be null for the first time the collection handle is created or must be the collection handle for the subsequent calls when a driver handle is appended to the collection. The third parameter handle can either be a driver handle or a collection handle. The function returns a handle of the requested kind on success and null on failure.
Note: Interleaving addition of elements to the collection while iterating over the members has unspecified behaviour.

<u>Example:</u>

```
void create_vhpi_driver(archHdl)
vhpiHandleT archHdl; /* handle to a foreign architecture */
{
vhpiHandleT drivHdl, sigItr, sigHdl, processHdl;
vhpiHandleT arr_driv[MAX_DRIVERS];
int i = 0;

 if (!vhpi_get(vhpiIsForeignP, archHdl))
 return;
 /* create a VHPI process */
 processHdl = vhpi_create(vhpiProcessK, archHdl, NULL);
 /* iterate on the signals declared in the architecture and create a
    VHPI driver and process for each of them */
 sigItr = vhpi_iterator(vhpiSigDecls, archHdl);
 if (!sigItr) return;
 while (sigHdl = vhpi_scan(sigItr)) {
     drivHdl = vhpi_create(vhpiDriverK, sigHdl, processdl);
     arr_driv[i] = drivHdl;
    i++;
  }
```

}
Issue I 20: Francoise fix example below

```
void create_vhpi_collection(sigHdl)
vhpiHandleT sigHdl; /* handle to a signal */
{
vhpiHandleT itBasic, basicH, itDriver, driverH;
vhpiHandleT h = NULL;

  itBasic =  vhpi_iterate(vhpiBasicSignals, sigHdl);
  while (basicH = vhpi_scan(itBasic)) {
    itDriver = vhpi_iterator(vhpiDrivers, basicH)
    while (driverH = vhpi_scan(itDriver) {

    h = vhpi_create( (vhpiDriverCollectionK, h, driverH);
    }
  }
}
```

11.6 vhpi_disable_cb()

vhpi_disable_cb()			
Synopsis:	Disable a callback that was registered using vhpi_register_cb().		
Syntax:	vhpi_disable_cb(cbHdl)		
	Type	**Description**	
Returns:	int	0 on success, 1 on failure.	
	Type	**Name**	**Description**
Arguments:	vhpiHandleT	cbHdl	A callback handle of kind vhpiCallbackK
Related functions:	Use vhpi_enable_cb() to re-enable the callback		

vhpi_disable_cb () shall be used to disable a callback that was registered using *vhpi_register_cb()*. The argument to this function should be a handle to the callback. The function returns 0 on success and 1on failure.

11.7 vhpi_enable_cb()

vhpi_enable_cb()			
Synopsis:	Enable a callback that was registered using vhpi_register_cb().		
Syntax:	vhpi_enable_cb(cbHdl)		
	Type	**Description**	
Returns:	int	0 on success, 1 on failure.	
	Type	**Name**	**Description**
Arguments:	vhpiHandleT	cbHdl	A callback handle of kind vhpiCallbackK
Related functions:	Use vhpi_disable_cb() to disable a callback		

vhpi_enable_cb () shall be used to re-enable a callback that was disabled. The callback was disabled at the registration with the flags set to *vhpiDisable* or was disabled by a call to *vhpi_disable_cb ()*. The argument to this function should be a handle to the callback. The function returns 0 on success and 1 on failure.

Example:

```
void find_cbks()
{
vhpiHandleT cbHdl, cbItr;
vhpiStateT cbState;

/* iterate on the registered callbacks and re-enabled the disabled ones
*/

cbItr = vhpi_iterator(vhpiCallbacks, objHdl);
if (!cbItr) return;
while (cbHdl = vhpi_scan(cbItr)) {
     cbState = vhpi_get(vhpiStateP, cbHdl);
     if (cbState == vhpiDisable)
        vhpi_enable_cb(cbHdl);
}
}
```

11.8 vhpi_format_value

vhpi_format_value()			
Synopsis:	Format a value into another format representation		
Syntax:	vhpi_format_value(valueInP, valueOutp)		
	Type	**Description**	
Returns:	int	0 on success, non 0 on failure	
	Type	**Name**	**Description**
Arguments:	const vhpiValueT *	valuep	Pointer to the input value
	vhpiValueT *	valuep	Pointer to the output value
Related functions:	Use vhpi_get_value to get an object value Use vhpi_get_time() to get the current simulation time		

vhpi_format_value() shall change the representation of a value to a requested format. This function takes two value pointers, the first one is the original value, the second one is the output value. The first value contains the value in a given format, the second value structure should have:

- the format field set to the requested format,
- the bufSize field set to size of the user allocated buffer (if formatting values of array type),
- the unit field set to the physical unit position for time or physical type unit conversions if formatting time or physical values,
- the union field which corresponds to the requested format set to point the user allocated value.

Both input and output value and value buffers shall be allocated by the caller. This function shall be used to format value into non native format representations or to do unit conversions for time or physical types. The input value may have been obtained by vhpi_get_value, or be a value returned by a callback on value change.
The interface does not define the legal format conversions.

The function returns 0 on success and non zero on failure to get the value. In case the buffer size of the output value (valuep->bufSize) allocated by the user is insufficient, vhpi_format_value returns a positive value which indicates the required buffer size in bytes for the converted value. If a negative value is returned, an error occurred and can be checked immediately after the **vhpi_format_value()** call by calling **vhpi_check_error()**. An error is generated if the size of the buffer is not sufficient.

The following errors are possible:
- Either of the value pointers is NULL.
- *Bad format specified*: the given format is inappropriate for the format of the input value, or has not been set. For example, if the input format is vhpiCharVal, the requested format cannot be vhpiRealVal.
- *Overflow*: the value does not fit, for example requesting the value of a real as an integer may result in an overflow. The value is returned but truncated.

The value structure (see table in vhpi_get_value) must have been allocated by the user. The **format** field must have been set by the user. The space to hold the value should be allocated by the user and the **bufSize** field to the size of the allocated buffer..

On the successful operation, the corresponding union field of the output value will be set by **vhpi_format_value**.

For time or physical values the unit field must be set to the physical position of the unit of representation of the value (1 being the base unit for physical types).

Example:

```
  /* converting a real value to an integer value */
  struct vhpiValueS value, newValue;
  vhpiValueT *valuep, *newValuep;
  struct vhpiErrInfoS errInfo;

  valuep = &value;
  newValuep = &newValue;
  value.format = vhpiRealVal;

  if (vhpi_get_value(objHdl, valuep))
    vhpi_check_error(&errInfo);
  newValue.format = vhpiIntVal;
  if (vhpi_format_value(valueP, newValuep))
    vhpi_check_error(&errInfo);

/* converting a time value from vhpiFs unit(precision of the simulator)
to vhpiNs unit */

  value.format = vhpiTimeVal;
  vhpi_get_value(objHdl, valuep);

  newValue.unit = 1000000; /* physical position of ns */
  newValue.format = vhpiTimeVal;
  if (vhpi_format_value(valuep, newValuep))
     vhpi_check_error(&errInfo);
```

11.9 vhpi_get()

<table>
<tr><th colspan="4">vhpi_get()</th></tr>
<tr><td>Synopsis:</td><td colspan="3">Get the value of an integer based property of an object</td></tr>
<tr><td>Syntax:</td><td colspan="3">vhpi_get(prop, hdl)</td></tr>
<tr><td></td><td>Type</td><td colspan="2">Description</td></tr>
<tr><td>Returns:</td><td>vhpiIntT</td><td colspan="2">Value of the property</td></tr>
<tr><td></td><td>Type</td><td>Name</td><td>Description</td></tr>
<tr><td>Arguments:</td><td>vhpiIntPropertyT</td><td>prop</td><td>An enumerated integer property constant representing the property of an object for which to obtain the value</td></tr>
<tr><td></td><td>vhpiHandleT</td><td>hdl</td><td>Handle to an object</td></tr>
<tr><td>Related functions:</td><td colspan="3">Use vhpi_get_str() to get string valued properties.
Use vhpi_get_real to get real valued properties.</td></tr>
</table>

vhpi_get() shall return the value of an object property of type vhpiIntPropertyT. For properties that are boolean properties (have 2 possible values **1** for **true** and **0** for **false**. Some properties such as **vhpiSizeP** may return any integer, while some other properties return a defined value; for such properties (example **vhpiModeP**), the VHPI header file predefines the integer value to be returned.

11.10 vhpi_get_cb_info()

vhpi_get_cb_info()			
Synopsis:	Retrieve information about a callback registered with vhpi_register_cb()		
Syntax:	vhpi_get_cb_info(hdl, cbDatap)		
	Type	**Description**	
Returns:	PLI_INT32	0 on success, 1 on failure	
	Type	**Name**	**Description**
Arguments:	vhpiHandleT	hdl	Handle to the callback
	vhpiCbDataT	cbDatap	Pointer to a structure containing callback information.
Related functions:	Use vhpi_register_cb() to register a callback.		

vhpi_get_cb_info() shall be used to retrieve callback information for a registered callback. The information is returned in a data structure that shall be allocated by the caller. The caller only allocates the vhpiCbDataS structure.
The cbDatap argument should point to a vhpiCbDataT structure defined by the VHPI standard header file. The data structure fields are described in section 8.2.1 describing **vhpi_register_cb().** The cbData argument memory must be allocated by the caller. The first argument should be a handle to the callback of kind vhpiCallbackK. The callback information returned is equivalent to the callback data information that was passed at the registration of the callback. If the callback was a time callback and therefore time information was supplied, vhpi_get_cb_info must convey back this information by setting the time field of the vhpiCbDataS structure to point to a time structure allocated by the VHPI interface which contains similar information to the registration call.

```
typedef struct vhpiCbDataS {
    int reason;        /* the reason of the callback */
    (void) (*cbf)(const struct vhpiCbDataS *);/* the callback
                                                    function
                                            pointer */
    vhpiHandleT obj;   /* a handle to an object */
    vhpiTimeT *time;   /* a time */
    vhpiValueT *value; /* a value */

    void *user_data;   /* user data information */

} vhpiCbDataT, *vhpiCbDatap;
```

11.11 vhpi_get_data()

vhpi_get_data()			
Synopsis:	Retrieve data from a saved location.		
Syntax:	vhpi_get_data(int id, void *dataLoc, int numBytes)		
	Type	**Description**	
Returns:	PLI_INT32	the number of bytes read, 0 on error	
	Type	**Name**	**Description**
Arguments:	PLI_INT32	id	an identifier denoting a position in a saved location
	PLI_VOID *	dataLoc	the address in which to put the data read
	PLI_INT32	numBytes	the requested number of bytes to read
Related functions:	Use vhpi_put_data to store data in a saved location		

vhpi_get_data() shall be used to retrieve "*numBytes*" of data for a given "*id*" from a saved file. The data is placed at the address pointed to by "*dataLoc*". The memory pointed to by *dataLoc* must have been properly allocated by the caller. The first call for a given "*id*" will retrieve "*numBytes*" of data starting at was placed into the save location with the first call to **vhpi_put_data()** for the same "*id*". The returned value is the number of bytes that were read. The user is responsible to check if the number of bytes read in is equal to the number of bytes requested. Each subsequent call with the same *id* will start retrieving data where the last call left off.
It is acceptable for an application to read in less bytes than what was stored for a given id with **vhpi_put_data(),** a warning should be issued. In this case, the dataLoc address is filled up with the data left for the given id and the remaining bytes will be filled up with '\0'. The return value shall be the actual number of bytes retrieved.

vhpi_get_data() can only be called from a callback routine which was registered for reason *vhpiCbStartOfRestart* or *vhpiCbEndOfRestart*. Such callbacks must have been registered during the *vhpiCbStartOfSave* or *vhpiCbEndofSave* execution. The reason is that the restart callback information (in particular the ids) must be saved in the simulation save location. A good way to record the *id* of an application is to pass it in the *user_data* field of the callback data of reason *vhpiCbStartOfRestart* or *vhpiCbEndOfRestart*. The size of the user_data field is a pointer to char which is enough to contain an int.

<u>Example:</u> A consumer routine which retrieves stored data from a save location.
See also the example in **vhpi_put_data()** for how this restart callback was registered.

```
/* type definitions for private data structures to save used by the
foreign models or applications */
struct myStruct{
  struct myStruct *next;
  int d1;
  int d2;
}
void consumer_restart(vhpiCbDataT *cbDatap)
{
  int status;
  int cnt = 0;
  struct myStruct *wrk;
  int dataSize = 0;
```

```
   /* get the id for this restart callback */
   int id = (int) cbDatap->user_data;
   /* get the number of structures */
   status = vhpi_get_data(id, (char *)&cnt, sizeof(int));
   if (status != sizeof(int))
      vhpi_assert(vhpiError, "Data read is not an int %d\n", status);
   /* allocate memory to receive the data that is read in */
   firstWrk = calloc(cnt, sizeof(struct myStruct));

   /* retrieve the data for the first structure */
   dataSize = cnt * sizeof(struct myStruct);
   status = vhpi_get_data(id, (char *)wrk, dataSize);
   if (status != dataSize)
      vhpi_assert(vhpiError, " cannot read %d data structures\n", cnt );

   /* fix up the next pointers in the link list:
      recreate the linked list */
   for (wrk = firstWrk; cnt >0; cnt--)
   {
      wrk->next = wrk++;
      wrk = wrk->next;
   }
} /* end of consumer_restart */
```

11.12 vhpi_get_foreignf_info()

vhpi_get_foreignf_info()			
Synopsis:	Retrieve information about a foreign model or application functions		
Syntax:	vhpi_get_foreignf_info(hdl, foreignDatap)		
	Type	**Description**	
Returns:	PLI_INT32	0 on success, 1 on failure	
	Type	**Name**	**Description**
Arguments:	vhpiHandleT	hdl	Handle to the foreign architecture, procedure, function or application
	vhpiForeignDataT	foreignDatap	Pointer to a structure containing model information.
Related functions:	Use vhpi_register_foreignf() to register foreign model and application elaboration and execution functions. Use the iteration function vhpiForeignfs to get the foreign models and applications registered in a tool session. Use vhpi_get_cb_info() to retrieve information about a registered callback.		

vhpi_get_foreignf_info() shall be used to retrieve foreign function information for foreign models and applications. The information is returned in a data structure that shall be allocated by the caller. On the return, the foreignDatap data structure is populated with the function pointers of the foreign model if bound, the kind of foreign model (vhpiArchF, vhpiProcF, vhpiFuncF, or vhpiAppF) and the library and model string names. The populated information is read only and the lifetime of the strings is unspecified. The foreignDatap argument should point to a vhpiForeignDataT structure defined by the VHPI standard header file. The data structure fields are described in section 11.28 **vhpi_register_foreignf().** The foreignData argument memory must be allocated by the caller. The first argument should be a handle to a foreign model. The handle kind is vhpiForeignfK (refer to the foreign model class diagram). **vhpi_get_foreignf_info**() does not force any function binding; it shall return the function pointers if the model has been bound to its C behaviour or null if not bound yet. A warning should be issued that the foreign model has not been bound yet.

```
typedef struct vhpiForeignDataS {
    vhpiForeignT kind;/* the foreign model class kind:
                         vhpi[ArchF,ProcF,FuncF, AppF]K */
    char *libraryName;/* the library name that appears in the
                         VHDL foreign attribute string */
    char *modelName;  /* the model name that appears in the
                         VHDL foreign attribute string PA28*/
    void (*elabf)( const vhpiCbDataT *);
                      /* the callback function pointer for
                         elaboration of foreign architecture */
    void (*execf)( const vhpiCbDataT *);
                      /* the callback function pointer for
                         initialization/simulation execution of
                         foreign   architecture, procedure,
                         function or application */
```

```
} vhpiForeignDataT, *vhpiForeignDatap;
```

11.13 vhpi_get_next_time()

vhpi_get_next_time()			
Synopsis:	Retrieve the next simulation time when some activity is scheduled		
Syntax:	vhpi_get_next_time(time_p)		
	Type	**Description**	
Returns:	int	0 on success, vhpiNoActivity if nothing is scheduled, non zero on other errors	
	Type	**Name**	**Description**
Arguments:	vhpiTimeT *	time_p	A pointer to a time structure containing the next time information
Related functions:	Use vhpi_get_phys(vhpiPrecisionP, NULL) to get the TIME value precision. Use vhpi_get_phys(vhpiTimeUnitP, NULL) to get the simulator time unit. Use vhpi_get_time() to get the current simulation time.		

vhpi_get_next_time() shall return the next active time if this function is called during postponed process or end of time phase and the current time when called in any other phase. The function returns 0 when there is a next scheduled time, and the time argument provides the absolute time value for the next event. f no event is scheduled, the time low and high fields should be both set to represent the value of Time'HIGH and the function should return vhpiNoActivity (defined to be the constant 1). If there is any error during the execution of this function, the function returns a non zero value (other than vhpiNoActivity), the time value is unspecified. This function can be called at the end of time step or at end of initialization.

<u>Example:</u>

```
vhpiTimeT time;

switch (vhpi_get_next_time(&time))
{
  case vhpiNoActivity:
     vhpi_printf("simulation is over, %d %d\n");
     break;
  case 0:
    vhpi_printf(“time = %d %d\n”, time.high, time.low);
    break;
  default:
    vhpi_check_error(&errInfo);
    break;
}
```

11.14 vhpi_get_phys()

vhpi_get_phys()	
Synopsis:	Get the value of a physical property of an object
Syntax:	vhpi_get_phys(prop, hdl)

	Type		Description
Returns:	vhpiPhysT	Value of a physical property	
	Type	**Name**	**Description**
Arguments:	vhpiPhysPropertyT	prop	An enumerated physical property constant representing the property of an object for which to obtain the value
	vhpiHandleT	hdl	Handle to an object
Related functions:	Use vhpi_get() to get integer valued properties. Use vhpi_get_str to get string valued properties.		

vhpi_get_phys() shall return the value of an object property of type vhpiPhysPropertyT. The value is returned as a vhpiPhysT. The returned value is unspecified in case of an error.

Example:

```
vhpiHandleT type; /* a physical type declaration */;
vhpiHandleT range = vhpi_handle(vhpiConstraint, type);
vhpiPhysT phys = {0,0};

phys = vhpi_get_phys(vhpiPhysRightBoundP, range));
vhpi_printf(“ right bound of physical type is %d %d \n”, phys.low,
phys.high);
```

11.15 vhpi_get_real()

vhpi_get_real()			
Synopsis:	Get the value of a real property of an object		
Syntax:	vhpi_get_real(prop, hdl)		
	Type	**Description**	
Returns:	vhpiRealT	Value of a real property	
	Type	**Name**	**Description**
Arguments:	vhpiRealPropertyT	prop	An enumerated real property constant representing the property of an object for which to obtain the value
	vhpiHandleT	hdl	Handle to an object
Related functions:	Use vhpi_get() to get integer valued properties. Use vhpi_get_str to get string valued properties.		

vhpi_get_real() shall return the value of an object property of type vhpiRealPropertyT. The value is returned as a vhpiRealT. The return value is unspecified in case of an error.

Example:

```
vhpiHandleT type; /* a  float type declaration */;
vhpiHandleT range = vhpi_handle(vhpiConstraint, type);

vhpi_printf(" right bound of floating type is %f\n",
vhpi_get_real(vhpiFloatRightBoundP, range));
```

11.16 vhpi_get_str()

vhpi_get_str()			
Synopsis:	Get the value of a string property of an object		
Syntax:	vhpi_get_str(prop, hdl)		
	Type	**Description**	
Returns:	const PLI_BYTE8 *	Pointer to a character string that represents the property value	
	Type	**Name**	**Description**
Arguments:	vhpiStrPropertyT	prop	An enumerated string property constant representing the property of an object for which to obtain the value
	vhpiHandleT	hdl	Handle to an object
Related functions:	Use vhpi_get() to get integer valued properties. Use vhpi_get_real to get real valued properties.		

vhpi_get_str() shall return the value of an object property of type vhpiStrPropertyT. The next call to vhpi_get_str() may override the previous string value returned by the prior vhpi_get_str() call, therefore if the string is to be used after this call, the string should be copied by the user to another location. The function returns NULL on error.

Example:

```
char name[MAX_LENGTH];
vhpiHandleT inst = vhpi_handle_by_name(":u1", NULL);

strcpy(name, vhpi_get_str(vhpiDefNameP, inst));
vhpi_printf("instance u1 is a %s\n", name);
```

1

11.17 vhpi_get_time()

vhpi_get_time()			
Synopsis:	Retrieve the current simulation time		
Syntax:	vhpi_get_time(time_p, cycles_p)		
	Type		**Description**
Returns:	void		
	Type	**Name**	**Description**
Arguments:	vhpiTimeT *	time_p	A pointer to a time structure containing time information
	long *	cycles_p	The number of relative or absolute delta cycles.
Related functions:	Use vhpi_get_phys(vhpiPrecisionP, NULL) to get the simulator precision. Use vhpi_get_phys(vhpiSimTimeUnitP, NULL) to get the simulator time unit.		

vhpi_get_time() shall return the current simulation time. The time value is returned using in the format specified in the time structure. The caller must allocate the time structure. The time is returned in the base unit of type time. In order to get the time in an different unit or format the vhpi_format_value() function shall be used.

If time_p is not NULL and cycles_p is not NULL, the time_p argument is set to the current simulation time Tc, the cycles_p argument is set to the current number of delta cycles from the beginning of that time step . A delta cycle is counted even if not completed. If the time_p argument is NULL, the cycles_p argument is set to the the absolute number of simulation cycles executed from the beginning of simulation. Cycles_p should be a pointer to long. The time is the current time, and the current number of delta cycles even if the time step or current delta is not completed yet.

```
typedef struct vhpiTimeS {
    vhpiInt high;
    vhpiInt low;


} vhpiTimeT;
```

Issue Francoise: vhpiTimeS structure differs from header files.

Example:

```
vhpiTimeT time;

vhpi_get_time(&time, NULL);
vhpi_printf("time = %d %d\n", time.high, time.low);
```

1

11.18 vhpi_get_value()

vhpi_get_value()			
Synopsis:	Get the value of an object or name, driver or transaction		
Syntax:	vhpi_get_value(objHdl, valuep)		
	Type	**Description**	
Returns:	PLI_INT32	0 on sucess, non 0 on failure to get the value	
	Type	**Name**	**Description**
Arguments:	vhpiHandleT	objHdl	Handle to an object which has a value
	vhpiValueT *	valuep	Pointer to a value
Related functions:	Use vhpi_put_value() to set an object to a value. Use vhpi_schedule_transaction to update the waveform of a signal driver.		

vhpi_get_value() shall get the value of an object or expression wich possess a value. Classes of objects which have a value have the **vhpi_get_value** operation (see class diagrams). The function takes 2 arguments: ***objHdl***, handle to the object to get the value from, ***valuep,*** pointer to a value structure that contains information on how to format the value. The function returns 0 on success and non zero on failure to get the value. In case the buffer size of the value (in valuep->bufSize) allocated by the user is insufficient, vhpi_get_value returns a positive value which indicates the required buffer size in bytes for the value. If a negative value is returned, an error occurred and can be checked immediately after the **vhpi_get_value()** call by calling **vhpi_check_error()**. Also an error if a buffer is provided and the size of the buffer is not sufficient.

The following errors are possible:
- This is *not a valid object* to get a value from: the object does not carry a value or the object handle is NULL.
- *Bad format specified*: the given format is inappropriate for the subtype of the object, or has not been set.
- *Overflow*: the value does not fit, for example requesting the value of a real as an integer may result in an overflow. The value is returned but truncated.
- *Value is unavailable*: the simulator has made some performance optimizations that makes this object and value unaccessible.

The value structure (see table 4) must have been allocated by the user. The format field must have been set by the user. The space to hold the value should be allocated by the user and the bufSize field to the size of the allocated buffer..
On the successful operation, the corresponding union value field will be set by vhpi_get_value (see table 4).

Values of objects of physical type are returned in base units with the *unit* field set to 1. Values of physical literals are returned in the units of the literal.

In case where the format is set by the user to *vhpiObjTypeVal*, the returned value will be returned in the most appropriate format for the type of the object. On the return, the interface will set the format field. Table 5 specifies which format will be chosen for each VHDL basic type.

vhpi_get_value() can retrieve the value of VHDL scalar types, value of access types, and arrays of scalars. In order to get values of complex types which do not fall in these categories, VHPI provides methods to:

- iterate on the sub-element fields of an object of a record type with the method ***vhpiSelectedNames***.
- iterate on the sub-element index of an object of an array type with the method ***vhpiIndexedNames***.
- access the dereferenced object from a variable object of an access type with the one-to-one method ***vhpiDerefObj***.

Given a reference handle of the object of a composite type, it is possible to iterate through each record field or array element and for each handle of the iterator, call ***vhpi_get_value()*** to get the value of that sub-element. The iteration methods are respectively v***hpiSelectedNames***, and ***vhpiIndexedNames***.

vhpi_iterate(***vhpiSelectedNames***, compObjHdl)
vhpi_iterate(***vhpiIndexedNames***, compObjHdl)

These methods return an ordered list of elements.
In case of values of multi-dimensional arrays, each returned handle represents an element in that array. If the array has 3 dimensions r, c, t, each handle returned can be represented by the 3 indices in r_indx, c_indx and t_indx. The elements are returned by varying the index of the last index constraint first. For each index range, the variation starts by the left index to the right index, independently of the direction of the range.

<u>Example:</u>

```
type array (1 to 2), (4 to 6), (8 downto 7) of integers;
```

iterating on the array elements will return the elements represented by the following:

```
(1,4,8) (1,4,7) (1,5,8) (1,5,7) (1,6,8) (1,6,7)
(2,4,8) (2,4,7) (2,5,8) (2,5,7) (2,6,8) (2,6,7)
```

In the case of arrays of arrays, each returned handle represents an element in the base array.
<u>Example:</u>

```
type arr_of_arr (1 to 2) of bit_vector(1 to 16);
  signal s: arr_of_arr;
 vhpi_iterate(vhpiIndexedNames, sHdl); // sHdl is a handle to signal s;
```

The iteration function will return handles to s(1) and s(2) which are vectors of 16 bits each. The kind of these handles is *vhpiIndexedNameK*. The value of each handle will be the value of the 16 bit vector it refers to.

In the case of an object of a record type, iterating through the record fields should return the selected object fields in the order they are declared. The kind of these handles is *vhpiSelectedNameK*. The value of such a handle will be the value of the selected field of the object. The *vhpiSelectedNames* iteration method can only be called on an object which type is a record type. The *vhpiIndexedNames* iteration method can only be called on an object which type is an array type.
The reference handle kinds passed to the *vhpiSelectedNames* and *vhpiIndexedNames* iteration methods can either be members of the *objDecl* class or of the *name* class.
These 2 methods allow to walk through any complex composite VHDL object.

The 1-to-1 method *vhpiDerefObj* allows to access the dereference allocated object which is designated by the access value of the variable. The reference handle must denote a *vhpiVarDeclK* kind of handle or a sub-element of a variable which is of an access type. The handle returned by this method is of kind *vhpiDerefObjK*. This class is a sub-class of the class name and inherits the properties and methods of the class name. In particular, the *vhpiNameP* property, *vhpiSizeP* (size in scalars of the dereferenced object), the ***vhpi_get_value()*** operation. A dereference object has no simple name according to the LRM page 45, therefore the *vhpiNameP* should return NULL and an error should be generated (Issue I002). The dereference object has a subtype which is the subtype of the accessed value. As a consequence, from a derefObject which is of a composite, the iteration methods *vhpiIndexedNames* and *vhpiSelectedNames* may be allowed as well as the *vhpiDerefObj* method if the access value is again of an access type. In the same way, from an *vhpiIndexedNameK* or *vhpiSelectedNameK*, if the object designated is of an access type, the *vhpiDerefObj* method is allowed.

Handle kinds that have a value are:
1. all sub-classes of the objDecl class: *vhpiConstDeclK, vhpiSigDeclK, vhpiFileDeclK, vhpiVarDeclK, vhpiGenericDeclK, vhpiPortDeclK, vhpiSigParamDeclK, vhpiVarParamDeclK, vhpiConstParamDeclK, vhpiFileParamDeclK*

Their value can be fetched if the object declaration has been elaborated. If the value is fetched at the end of elaboration, the value is the default ('left) or initial value of the object provided in the declaration. The value of a generic after elaboration will be the value after generic propagation. During simulation, the value fetched is the value of the object at this particular time.

For a *vhpiFileDeclK*, the value is the logical name of the file. The value is of type string. The value is the string value supplied in the declaration if present or the logical name the file was associated with during a call to FILE_OPEN if the file was opened during simulation. If the file is not opened at the time of the query the value *str* field is set to NULL and an error is generated.

For a variable of an access type, the access value can be fetched with the format vhpiPtrVal: the access value is the address of the allocated object. The dereference value designated by the current access value of the variable can be fetched from a handle of the dereference object (*vhpiDerefObjK*) which is obtained after applying the *vhpiDerefObj* method to the variable handle. If the variable has an access value of 0 (null) it does not designate an allocated object. In that case it is an error to apply the *vhpiDerefObj* method (Dereferencing a null pointer); the *vhpiDerefObj* method should return a null handle and an error should be generated.
The access value designates the created object (just like a pointer). This is different from the value of the object which can be fetched by dereferencing the object. The value of a dereference name (xyz.ALL) will be the dereference value pointed by the object xyz. The handle kind of a dereference name is *vhpiDerefObjK*. The default value of a variable of an access type is NULL (NULL pointer). The initial access value of a variable with an initial expression will be the access value of the initial expression.
A dereference object (*vhpiDerefObjK*) has a value which can be obtained with vhpi_get_value and the format field set to a format applicable to the basic type corresponding to the dereference value. For example, for a dereference value of a variable of access to integer, the format should be set to vhpiIntVal, and the handle denoting the dereference object.
For foreign subprogram calls, it is possible to get the actual values associated with the formal parameters. Subprograms are dynamically elaborated therefore the values of their parameters or declared items can only be fetched when the program is currently executing or is suspended.
We provide a method to get the current equivalent process and a method to access the stack of an equivalent process (see subprogram call class diagram). The user can only fetch the values of parameters or declared items if the subprogram call is either the current executing process or is on the call stack of the current executing process or is a suspended process or on the stack of a suspended process.

Subprogram parameter values can only be fetched when the subprogram is executed because they come to existence dynamically. In order to get a value for either a parameter or a declared item within that subprogram, the static as well as dynamic info must be provided (stack frame level).

2. any sub-class of the class name has a value *(vhpiIndexedNameK, vhpiSliceNameK, vhpiSelectedNameK, vhpiAttrNameK).*
There is the possibility of getting a value for handles of any of the sub-classes of class literal using vhpi_get_value Another approach is to use the properties defined for each of the literal sub-classes, which do not require allocation of a value buffer. We defined different properties for each sub-classes: *vhpiIntValP* for class vhpiIntLiteralK, *vhpiRealValP* for class vhpiRealLiteralK, and *vhpiStrValP* for classes *vhpiStringLiteralK, vhpiBitStringLiteralK, vhpiCharLiteralK* and *vhpiEnumLiteralK*.

It is not possible to fetch directly the value of any other expression such as an aggregate, typeConv or function call for example.

3. Simulation objects:

A driver (vhpiDriverK) has a value which is its current driving value. This driving value can only be fetched after simulation initialization phase has been completed.
The value of a transaction object handle (vhpiTransactionK) can also be fetched during a simulation session.

Values can be fetched after simulation initialization has completed in a simulation session, or after elaboration in an elaboration session. (Francoise: Does this apply to simulation objects? Or to all objects which have a value?)

These class kinds have an operation vhpi_get_value() in the object class diagram.

Examples:The following function get_object_value() shows how to obtain value of array type, record and scalar types. As written, the function is able to get the value of the signals “bit3” and “rec1”. The default cases of the switch statements need to be completed to be able to get values of objects of any type.

```
type bit3 is array (1 to 3) of std_logic;

type myrecord is record
 i: integer;
 r: real;
 b3: bit3;
end record;

-- array of records: my_recarray
type my_recarray is array (0 to 2) of myrecord;
-- multi-dimension: array 2 dimensions of time values
type my_2dim is array (0 to 2, 1 to 3 ) of time;

type word is array of (1 to 8) of bits;
-- array of array
type mem is array (1 to 4) of word;

signal bits : bit3 := (std0, std1, stdu);
signal rec1 : myrecord := (34, 2.0, (STD0, STD0, STD0));
signal rec2 : my_recarray := ((34, 2.0, (others =>STD0)), (35, 3.0,
(others => STD1)));
signal arr2dim : my_2dim := ((0 ns, 1 ns, 2 ns), (3 ns, 4 ns, 5 ns), (6
ns, 7 ns, 8 ns));
signal arrofarr : mem := ("0000_0000", "0000_0001", "0000_0010",
"0000_0011");


void get_object_value(vhpiHandleT sigHdl)
{
vhpiHandleT elemHdl, baseHdl;
struct vhpiValueS value;
vhpiValueT * valuep = &value;
int size;
char *buffer = NULL;

/* access the value of signal sigHdl*/

value.bufSize = 0
value->value = NULL;
size = vhpi_get_value(sigHdl, valuep);
```

```
if (size > value.bufSize) {
  buffer = malloc (sizeof(size));
  value.bufsize = size;
}

baseHdl = vhpi_handle(sigHld, vhpiBaseType);
switch (vhpi_get(vhpiKindP, baseHdl)) {
case vhpiArrayTypeDeclK:
{
  elemHdl = vhpi_handle(vhpiElemType, baseHdl);
  switch (vhpi_get(vhpiKindP, elemHdl))
  {
     case vhpiIntTypeDeclK:
       value.format = vhpiIntVecVal;
       value.intgs = buffer;
      vhpi_get_value(sigHdl, &value);
       for (value.elemScalars, value.intgs; value.elemScalars=0;
            value.elemScalars--, (value.intgs)++)
            vhpi_printf("%s [ %d] = %d \n",
                         vhpi_get_str(sigHdl, vhpiName),
                         value.elemScalars, value.intgs);
       break;
     case vhpiEnumTypeDeclK:
       if (!strcmp(vhpi_get_str(vhpiCaseNameP, elemHdl) == "CHARACTER"))
       {

         value.format = vhpiStrVal;
         value.str = buffer;
         vhpi_get_value(sigHdl, &value);
         vhpi_printf("%s = %s\n", vhpi_get_str(vhpiName, SigHdl),
value.str);
       }
       else {
         value.format = vhpiEnumVecVal;
         value.enumvs = buffer;
         vhpi_get_value(sigHdl, &value);
         for (value.elemScalars, value.enumvs; value.elemScalars=0;
            value.elemScalars--, (value.enumvs)++)
            vhpi_printf("%s [ %d] = %d \n",
                         vhpi_get_str(sigHdl, vhpiName),
                         value.elemScalars, value.enumvs);
       }
       break;
     case vhpiPhysTypeDeclK:

       if (!strcmp(vhpi_get_str(vhpiCaseNameP, elemHdl) == "TIME"))
       {
         value.format = vhpiTimeVecVal;
         value.times = buffer;
         vhpi_get_value(sigHdl, &value);

         for (value.elemScalars, value.times; value.elemScalars=0;
              value.elemScalars--, (value.times)++)
              vhpi_printf("%s [ %d] = %d  %d\n",
                           vhpi_get_str(sigHdl, vhpiName),
                           value.elemScalars, (value.times).high,
                           (value.times).low);

       }
       else {
```

```
        value.format = vhpiPhysVecVal;
        value.physs = buffer;
        vhpi_get_value(sigHdl, &value);

         for (value.elemScalars, value.physs; value.elemScalars=0;
              value.elemScalars--, (value.physs)++)
              vhpi_printf("%s [ %d] = %d  %d\n",
                          vhpi_get_str(sigHdl, vhpiName),
                          value.elemScalars,
                          (value.physs).high,
                          (value.physs).low);
      }

      break;
      case vhpiFloatTypeDeclK:
      {
        value.format = vhpRealVecVal;
        value.reals = buffer;
        vhpi_get_value(sigHdl, &value);

        for (value.elemScalars, value.reals; value.elemScalars=0;
             value.elemScalars--, (value.reals)++)
           vhpi_printf("%s [ %d] = %f\n",
                       vhpi_get_str(sigHdl, vhpiName),
                       value.elemScalars, value.reals);
      }

      break;
      default:
        vhpi_printf("need to decompose the element subtype:
                     array of %s\n",
                     vhpi_get_str(vhpiKindStrP, elemHdl));;

   } /* end switch on elemHdl */
   }
   break;
   case vhpiRecordTypeDeclK:
   {
     vhpiHandleT memberH, iterH;
     iterH = vhpi_iterator(vhpiMembers, sigHdl);
     while (memberH = vhpi_scan(iterH))
     {
       get_object_value(memberH);
     }
   }
   break;
   case vhpiIntTypeDeclK:
   {
      value.format = vhpiIntVal;
      vhpi_get_value(sigHdl, valuep);
      vhpi_printf("%s = %d \n",
                       vhpi_get_str(sigHdl, vhpiName),
                       value.intg);
   }
   break;
   case vhpiEnumTypeDeclK:
   {
      value.format = vhpiEnumVal;
      vhpi_get_value(sigHdl, valuep);
      vhpi_printf("%s = %d \n",
```

```
                            vhpi_get_str(sigHdl, vhpiName),
                            value.enumv);
      }

      break;
      case vhpiFloatTypeDeclK:
      {
         value.format = vhpiRealVal;
         vhpi_get_value(sigHdl, valuep);
         vhpi_printf("%s = %d \n",
                            vhpi_get_str(sigHdl, vhpiName),
                            value.real);
      }

      break;
      default:
         vhpi_printf("not implemented: %s\n",
                         vhpi_get_str(vhpiKindStrP, baseHdl));
    } /* end switch on baseHdl */
} /* end get_object_value */
```

We have a specialized time structure to be used for time values.
The physical structure should be used for any other physical types.

```
/* time value structure */

typedef struct vhpiTimeS {
  vhpiInt high;
  vhpiInt low;

} vhpiTimeT;
```

Table 2: Time value structure

```
/* time unit values */

#define vhpiFS  -15   /* femto second */
#define vhpiPS  -12   /* pico second */
#define vhpiNS  -9    /* nano second */
#define vhpiUS  -6    /* micro second */
#define vhpiMS  -3    /* milli second */
#define vhpiS   0     /* second */
#define vhpiMN  1     /* minute */
#define vhpiHR  2     /* hour */
```

```
/* physical value structure */

typedef struct vhpiPhysS {
  vhpiInt high;
  vhpiInt low;
} vhpiPhysT;
```

Table 3: Physical value structure

```
/* value structure */
typedef struct vhpiValueS
{
  vhpiFormatT format;     /* IN/OUT: (depending on format) value
                                    format */
  int bufSize;            /* IN: size in bytes of the buffer */
  vhpiIntT numElems;      /* OUT: number of array elements in the
value,
                                    undefined value for scalars */
  vhpiPhysT unit;      /* IN/OUT: physical position of the unit
representation for the value */

  union
  {
    vhpiEnumT enumv, *enumvs;  /* OUT: enumeration */
    vhpiIntT intg, *intgs;     /* OUT: integer */
    vhpiRealT real, *reals;    /* OUT: floating point */
    vhpiPhysT phys, *physs;    /* OUT: physical */
    vhpiTimeT time, *times;    /* OUT: time */
    char ch, *str;             /* OUT:character or string */
    void* ptr, ptrs;           /* OUT: simulator representation value
                                         or access value */
  } value;

} vhpiValueT;
```

Table 4: Value structure

```
/* value formats */
#define vhpiBinStrVal          1
#define vhpiOctStrVal          2
#define vhpiDecStrVal          3
#define vhpiHexStrVal          4
#define vhpiEnumVal            5
#define vhpiIntVal             6
#define vhpiLogicVal           7
#define vhpiRealVal            8
#define vhpiStrVal             9
#define vhpiCharVal           10
#define vhpiTimeVal           11
#define vhpiPhysVal           12
#define vhpiObjTypeVal        13
#define vhpiPtrVal            14
#define vhpiEnumVecVal        15
#define vhpiIntVecVal         16
#define vhpiLogicVecVal       17
#define vhpiRealVecVal        18
#define vhpiTimeVecVal        19
#define vhpiPhysVecVal        20
#define vhpiPtrVecVal         21
#define vhpiRawDataVal        22

/* IEEE STD_LOGIC and STD_U_LOGIC values */
#define vhpiU                  0   /* uninitialized */
```

```
#define vhpiX              1   /* unknown */
#define vhpi0              2   /* forcing 0 */
#define vhpi1              3   /* forcing 1 */
#define vhpiZ              4   /* high impedance */
#define vhpiW              5   /* weak unknown */
#define vhpiL              6   /* weak 0 */
#define vhpiH              7   /* weak 1 */
#define vhpiDontCare       8   /* don't care */

/* std BIT values */
#define vhpiBit0           0
#define vhpiBit1           1
```

VHDL base type	**Format of returned value**
INTEGER	vhpiIntVal
ENUMERATION	vhpiEnumVal
CHARACTER	vhpiCharVal
BIT	vhpiEnumVal
BOOLEAN	vhpiEnumVal
SEVERITY_LEVEL	vhpiEnumVal
FILE_OPEN_KIND	vhpiEnumVal
FILE_OPEN_STATUS	vhpiEnumVal
STD_LOGIC STD_U_LOGIC	vhpiLogicVal
PHYSICAL	vhpiPhysVal
TIME	vhpiTimeVal
STRING	vhpiStrVal
CHARACTER	vhpiCharVal
FLOATING POINT	vhpiRealVal
REAL	vhpiRealVal
ACCESS	vhpiPtrVal
ARRAY of *	vhpi*VecVal
RECORD	vhpiRawDataVal

Table 5: vhpiObjTypeVal returned formats

11.19 vhpi_handle()

vhpi_handle()			
Synopsis:	Return the destination handle of a one-to-one relationship		
Syntax:	vhpi_handle(oneRel, refHdl)		
	Type	**Description**	
Returns:	vhpiHandleT	returns the target handle or NULL.	
	Type	**Name**	**Description**
Arguments:	vhpiOneToOneT	oneRel	An integer constant denoting the one-to-one relationship to traverse
	vhpiHandleT	refHdl	Handle to the reference object of the relationship

vhpi_handle () shall be used to traverse either:

1. a one-to-one unamed directed relationship that exists between an object denoted by *refHdl* and an object of type *oneRel*.
2. a one-to-one tagged directed relationship that exists between two classes.

If the multiplicity of the association is 0..1, it is possible to not obtain a handle by traversing that relationship and the function *vhpi_handle()* will return NULL.
If the multiplicity of the association is 1, the traversal of the relationship should always return a handle.

Example: The following code shows the traversal of several one-to-one relationships. The relationships are illustrated by the class scope diagram.

```
vhpiHandleT get_instance_info(scopeHdl)
vhpiHandleT scopeHdl; /* a handle to a scope */
{
vhpiHandleT upScopeHdl, duHdl;

  /* escalade the hierarchy one level */
  /* traverse a tagged relationship */
  upScopeHdl = vhpi_handle(vhpiUpperRegion, scopeHdl);
  if (vhpi_get(vhpiKindP, upScopeHdl) == vhpiCompInstStmtK)
  {   /* traverse a unamed relationship */
      duHdl = vhpi_handle(vhpiDesignUnit, upScopeHdl);
      return(duHdl);
  }
  else return(NULL);
}/* end get_instance_info() */
```

11.20 vhpi_handle_by_index()

vhpi_handle_by_index()			
Synopsis:	Get a handle to an object using an index position in a parent object.		
Syntax:	vhpi_handle_by_index(itRel, parentHdl, index)		
	Type	**Description**	
Returns:	vhpiHandleT	A handle to an object.	
	Type	**Name**	**Description**
Arguments:	vhpiOneToManyT	itRel	an ordered iteration relationship tag
	vhpiHandleT	parentHdl	the handle to the object from which to obtain the indexed handle.
	PLI_INT32	index	index position of the object for which to obtain a handle for.
Related functions:	Use vhpi_iterator and vhpi_scan() to get each element of the parent handle		

vhpi_handle_by_index() shall be used to get a handle to an object based on the index number of that object within a parent object. The parent object must bear an ordered iteration relationship to the indexed object (vhpiIndexedNames) or an integer property denoting the number of elements in the iteration (called vhpiNumParamsP for vhpiParamDecls iteration, vhpiNumGensP for the vhpiGenDecls iteration, vhpiNumPorts for the vhpiPortDecls iteration for example. Such ordered relationships exist for example between an object of type array and its sub-elements, a subprogram call and its formal parameters, a object of type record and its fields, a variable or a derefobj of an access type to an array or record and its sub-Dereference objects (I0023: do we allow shortcut iterations on indexednames and selected names from a variable of an access type?). The first argument must denote an ordered iteration relationship. These ordered iteration relationships are marked “ordered” on the information model. The ordered iterations can also be traversed with the *vhpi_iterator()* and *vhpi_scan()* functions.
vhpi_handle_by_index(itRel, parentHdl, index) returns the handle that would have been returned by creating an iterator of the relationship denoted by *itRel,* and scanning for *index* + 1 times.

<u>Example 1: (I024: fix example subtype/basetype changes)</u>

```
vhpiHandleT find_indexed_constraint(parentHdl, index)
vhpiHandleT parentHdl; /* a handle to a object */
int index; /* the index position of the object to obtain */

{
    vhpiHandleT subtypeHdl,typeHdl, subHdl;

    subtypeHdl = vhpi_handle(vhpiType, parentHdl);
    typeHdl = vhpi_handle(vhpiBaseType, subtypeHdl);
    if (vhpi_get(vhpiIsCompositeP, typeHdl))
    {  /* get the given indexed array element or indexed record field
          of the parent object */
      if (vhpi_get(vhpiKindP, typeHdl) == vhpiArrayTypeDeclK))
       subHdl = vhpi_handle_by_index(vhpiConstraints,parentHdl,
                                     index);
      else if (vhpi_get(vhpiKindP, typeHdl) == vhpiRecordTypeDeclK))
       subHdl = vhpi_handle_by_index(vhpiRecordElems,parentHdl,
```

```
                                       index);

        return subHdl;
     }
     else return NULL;
}
```

Example 2:
This example shows how to access a formal parameter by index position from a subprogram call handle. This is useful for accessing values of VHDL formal parameters from a foreign subprogram C function implementation. Formal parameter declarations define an order in the interface parameter list.

```
void exec_proc(cbDatap)
vhpiCbDataT cbDatap; /* the call-data structure of the C foreign
                         function implementation of a VHDL
subprogram
                         behavior.*/

{

    vhpiHandleT subpCallHdl, formal1, formalIt;
    int val = 0;
    vhpiValueT value;

    value.format = vhpiIntVal;
    value.value->integer = &val;
    procCallHdl = cbDatap->obj;

    /* get a handle to the first formal parameter of the
       subprogram call */
    formal1 = vhpi_handle_by_index(vhpiParamDecls, subpCallHdl, 0);

    switch(vhpi_get(vhpiModeP, formal1))
    {
        case vhpiIN:
            vhpi_get_value(formal1, &value);
            break;
        case vhpiOUT:
            vhpi_put_value(formal1, &value);
            break;
        default:
            break;
    }
```

Example 3: (I025: fix example)
This example shows how to get a handle to a sub-object of a composite type.

```
type myrecord is record
  I: integer;
  B: bit;
  AR: array (2 to 4);
end record;
type myrecord_ptr is access to myrecord;
type mybit_vector_ptr is access to bit_vector;

variable A: array (2 to 5) of bit := ('1', '0', '1', '0');
variable M: array ((2 to 5), (3 to 5)) of integer:= (1, 2, 3, 4,
                                                     5, 6, 7, 8,
                                                     9,10,11,12);
```

```
variable R: myrecord := (9, '0', B"111");
variable R_p: myrecord_ptr;
variable BV_p: mybit_vector_ptr;

/* if Ahdl is an handle to variable A */
hdl = vhpi_handle_by_index(vhpiIndexedNames, Ahdl, 0)
/* returns a handle to A(2) */
/* if Mhdl is an handle to variable M */
hdl = vhpi_handle_by_index(vhpiIndexedNames, Mhdl, 0)
/* returns a handle to M(2,3) */
/* if Rhdl is an handle to variable R */
hdl = vhpi_handle_by_index(vhpiSelectedNames, Rhdl, 0)
/* returns a handle to R.I */
/* if Rhdl is an handle to variable R */
subeltHdl = vhpi_handle_by_index(vhpiSelectedNames, Rhdl, 2)
/* subeltHdl is a handle to R.AR */
hdl = vhpi_handle_by_index(vhpiIndexedNames, subeltHdl, 2)
/* hdl is a handle to R.AR(4} */
/* if BV_phdl is an handle to variable BV_p */
hdl = vhpi_handle_by_index(vhpiIndexedNames, BV_phdl, 0)
/* returns a handle to BV_p(0)*/

/* if R_phdl is an handle to variable R_p */
hdl = vhpi_handle_by_index(vhpiSelectedNames, R_phdl, 0)
/* returns a handle to R_p.I */
```

11.21 vhpi_handle_by_name()

vhpi_handle_by_name()			
Synopsis:	Returns a handle to the named item if found in the search scope.		
Syntax:	vhpi_handle_by_name(name, refHdl)		
	Type	**Description**	
Returns:	vhpiHandleT	A handle on success, NULL if no objects of the given name exists.	
	Type	**Name**	**Description**
Arguments:	const PLI_BYTE8 *	name	A character string or a pointer to a string containing the full, partial or simple name of an object.
	vhpiHandleT	refHdl	Handle to a reference search region or scope handle or NULL
Related functions:	Use vhpi_get_str() to get the name (vhpiNameP) or full name (vhpiFullNameP) of an object		

vhpi_handle_by_name () shall return a handle to an object that matches the given name. This function can be only applied to objects having the *vhpiFullNameP* property. Francoise: why not the name property ? Is there any object which has the fullname property but not the name property?. The *name* can be the full hierarchical name or a partial hierarchical name of an elaborated object or uninstantiated object. If *refHdl* is NULL, the *name* shall be searched for from the top level of the hierarchy (*vhpiRootInstK)* or from the packages instantiated in the design (*vhpiPackInstK* handles) or from the library context if the vhpiFullNameP denotes a full uninstantiated name (vhpiFullNameP is the same as vhpiDefNameP in the uninstantiated context). If *refHdl* is not NULL, *name* shall be searched from the declarative instantiated region or uninstantiated scope designated by the *refHdl* handle. The function cannot be applied to handles of anonymous types (do we have handles to anonymous types?). For overloaded subprograms or enumeration literals, the name must include the parameter result profile. In this case the name must follow the following syntax:
<subp_name> | <enum_literal> ({param_type, } [:return_type])
The return type is only necessary for functions and enumeration literals.
vhpi_handle_by_name() will return null if the name is ambiguous or if it cannot find the object of that name.

<u>Example:</u>
This function finds a signal handle given the simple signal name.

```
vhpiHandleT findsignal(sigName)

char *sigName;/* the signal name */
{
  vhpiHandleT subitr, hdl, subhdl, sigHdl;
  /* first search the signal in the design hierarchy, starting at the
root instance level  and recursively descending into the sub-instances
*/
  itr = vhpi_handle(vhpiRootInst, NULL);
  if (itr) {

     sigHdl = vhpi_handle_by_name(sigName, hdl);
     if (sigHdl)
```

```
        return sigHdl;
        else {
          subitr = vhpi_iterator(vhpiInternalRegions, hdl);
          if (subitr)
          while (subhdl = vhpi_scan(subitr)) {
              sigHdl = vhpi_handle_by_name(sigName, subhdl);
              if (sigHdl)
              return sigHdl;
          }
        }
     }
     itr = vhpi_iterator(vhpiPackInsts, NULL);
     if (itr)
     while (hdl = vhpi_scan(itr)) {
        sigHdl = vhpi_handle_by_name(sigName, hdl));
        if (sigHdl)
        return sigHdl;
     }
     return NULL;
   }
```

11.22 vhpi_iterator()

vhpi_iterator()			
Synopsis:	Create an iterator handle to the reference handle which has a a one-to-many relationship and initialize it to point to the first element of the iteration		
Syntax:	vhpi_iterator(iterType, refHdl)		
	Type	**Description**	
Returns:	vhpiHandleT	An initialized iterator handle on success, NULL if no objects of type iterType exists.	
	Type	**Name**	**Description**
Arguments:	vhpiOneToManyT	iterType	An integer constant representing the iteration type
	vhpiHandleT	refHdl	Handle to the reference handle
Related functions:	Use vhpi_scan() to get each element of the iteration		

vhpi_iterator () shall be used to traverse one-to-many relationships which are indicated by arrows with a multiplicity of * or 1..* in the information model. This function creates and initializes an iterator handle, whose type is **vhpiIteratorK** and which can be used by the *vhpi_scan()* function to scan through each object of type *type* which is in a multiple association with the reference object *refHdl*. If there are no objects of type *type* associated with the reference handle, then the **vhpi_iterator()** shall return NULL. A NULL can be expected for one-to-many relationships that are marked with a multiplicity of * (zero or more) in the information model. The iterator handle is automatically released at the end of an iteration when there is no more elements to be returned by *vhpi_scan*. Reference to the iterator handle after the end of an iteration is erroneous. The iterator handle should be explicitly released with *vhpi_release_handle* if the iteration has not been exhausted in order to avoid a memory leak.

Example:

```
void find_signals(scopeHdl)
vhpiHandleT scopeHdl; /* a handle to a scope */
{
vhpiHandleT sigHdl,itrHdl;

/* find all signals in that scope and print their names */

itrHdl = vhpi_iterator(vhpiSigDecl, scopeHdl);
if (!itrHdl) return;
while (sigHdl = vhpi_scan(itrHdl)) {
      vhpi_printf("Found signal %s\n", vhpi_get_str(vhpiNameP, sigHdl));
}
```

Deleted: ¶

Deleted: `vhpiHandleT`

Deleted: `int found = 0;`¶

11.23 vhpi_protected_call()

vhpi_protected_call()			
Synopsis:	Executes an operation on a shared variable of a protected type		
Syntax:	vhpi_protected_call(varHdl, userFct, userData)		
	Type	**Description**	
Returns:	PLI_INT32	Returns 0 on success, non zero on failure.	
	Type	**Name**	**Description**
Arguments:	vhpiHandleT	varHdl	A handle to a shared variable declaration of a protected type
	vhpiUserFctT	userFct	The user function pointer to be called during the protected access
	PLI_VOID *	userData	The user data to be passed to the user function
Related functions:	typedef PLI_INT32 (*vhpiUserFctT)(); The prototype of the user function should be: PLI_INT32 userFct(vhpiHandleT varHdl, PLI_VOID *userData) varHdl: reference handle of the vhpi_protected_call function. userData: user data to be passed to the userFct, can be NULL.		

vhpi_protected_call() shall be used to perform an operation to a shared variable of a protected type. The function guarantees atomicity of the operation by performing a lock on the variable passed as a reference handle.

vhpi_protected_call() should acquire a lock on the shared protected type variable passed as a reference handle, ***varHdl***, execute the user function passed as a second argument, ***userFct***, then release the lock on the protected variable handle. **vhpi_protected call()** returns 0 on success and non zero on failure.

The lock performed is equivalent to the lock defined in the VHDL LRM 1076-2001. The kind of the reference handle that is passed to **vhpi_protected_call** should only be a handle denoting a shared variable declaration of a protected type. Handles and access to local objects defined in the protected type body associated with a shared variable can be performed within the user function where a variable lock has been acquired. If vhpi_protected_call fails to obtain a lock, it shall return a status of failure. vhpi_protected_call allows correct locking semantics of shared variable. If read or write access to a shared variable of a protected type is not made within a call to **vhpi_protected_call()**, the results may be erroneous.

Francoise: redo the example to access a private data declaration of the variable.

Example:

```
#include <stdio.h>
#include vhpi_user.h

/* user function which is called on the protected variable handle */
int Myfunc( vhpiHandleT protectedVarDeclHdl, void* ClientData )
{
  #define FAIL -1;
  int status=0;
```

```
  vhpiHandleT resultH;

  MyData* Data=(MyData*)ClientData;
/* result is a private variable declaration for the protected type */
  resultH = vhpi_handle_by_name("result", protectedVarDeclHdl);
  if (!resultH)
     return(FAIL);

  /* access the current value of result */
  status = vhpi_get_value( result,  Data->Value );
  if (status)
  {
    vhpi_printf("error in reading protected variable\n");
    return (status);
  }
  switch(Data->Op)
  {
     case op1 : op1CB( Data->Value);break;
     case ....
     case
     default: Bombout();
  }
  /* set result to a new value */
  status = vhpi_put_value( resultH, Data->Value, vhpiDeposit );

  /* do some more error checking */
  if (status)
     vhpi_printf("error in writing to protected variable\n");

  return status;
}

/* the proposed function for controlling protected variables access
This function is implemented by the VHDL simulator and VHPI interface */

int vhpi_protected_call(
        vhpiHandleT protectedVarDeclHdl,
        int (*Myfunc)(vhpiHandleT protectedVarDeclHdl,void* ClientData),
        void* ClientData )
{

   int status;

   /* acquire the lock on the protected variable */
   int Lock = internal_getlock(protectedVarDeclHdl);
   /* do some error checking to determine if the lock was */
   /* obtained ok */

   /* then executes the user function passed in */
   status = Myfunc( protectedVarDeclHdl, ClientData );

   /* release up the lock */
   internal_releaselock(Lock);

   /* return the user function status */
   return status;

}
```

```
/* in user code */

int op1CB( int value )
{
      …
}

main (argc, argv)
{
/* get a handle to the protected variable declaration named "Foo" */
vhpiHandleT protectedvarDeclHdl = vhpi_handle_by_name( "Foo", NULL);

MyData Data;
int status = 0;

   Data.Op = op1;
   Data.Size = 100;
   bzero(Data.Value, Data.Size );

  status = vhpi_protected_call(protectedVarDeclHdl,Myfunc,Data);

  if (status)
      vhpi_printf("Unable to perform operation op1
                  with protected variable Foo\n");
}
```

11.24 vhpi_printf()

vhpi_printf()			
Synopsis:	Write to whatever files were defined by the tool to be the message display files (for example stdout, simulator log files)		
Syntax:	vhpi_printf(format, …)		
	Type	**Description**	
Returns:	PLI_INT32	The number of characters written	
	Type	**Name**	**Description**
Arguments:	const PLI_BYTE8 *	format	A format string
	…	args	Arguments for the formatted string
Related functions:	VHPI_GET_PRINTABLESTRCODE() vhpi_is_printable_char()		

vhpi_printf shall write to the files that were defined by the tool to receive output messages. Such files could be stdout, the tool log file for example… The format string shall use the same formats as the C printf. The function shall return the number of characters printed or EOF (-1) if an error occurred.
In order to print the VHDL non graphic characters that can be found in VHDL string literals or value of type VHDL string, VHPI provides a macro **VHPI_GET_PRINTABLE_STRINGCODE** to get the string corresponding to the enumerated character value defined in the standard VHDL character type set. A function to test if a character is a graphic or non graphic character is also provided***vhpi_is_printable()***.
The example below shows how to print a VHDL string which may contain non graphic characters using the macro and table look up included in the vhpi_user.h file.

Example:
From vhpi_user.h

```
#include <stdio.h>

static PLI_BYTE8* VHPICharCodes[256]={
"NUL",   "SOH", "STX", "ETX", "EOT", "ENQ","ACK",  "BEL" ,
"BS",    "HT",  "LF",  "VT",  "FF", "CR", "SO" , "SI",
"DLE",  "DC1", "DC2", "DC3", "DC4","NAK", "SYN" , "ETB",
 "CAN",   "EM", "SUB", "ESC", "FSP", "GSP", "RSP" , "USP",
 " ","!","\"","#","$","%","&","",
"(",")","*","+",",","-",".","/",
"0","1","2","3","4","5","6","7",
"8","9",":",";","<","=",">","?",
"@","A","B","C","D","E","F","G",
"H","I","J","K","L","M","N","O",
"P","Q","R","S","T","U","V","W",
"X","Y","Z","[","\\","]","^","_",
"`","a","b","c","d","e","f","g",
"h","i","j","k","l","m","n","o",
"p","q","r","s","t","u","v","w",
"x","y","z","{","|","}","~","DEL",
"C128", "C129", "C130","C131","C132","C133","C134","C135",
"C136", "C137", "C138","C139","C140","C141","C142","C143",
"C144", "C145", "C146","C147","C148","C149","C150","C151",
```

```
"C152", "C153", "C154","C155","C156","C157","C158","C159",
" ","¡","¢","£","¤","¥","¦","§",
"¨","©","ª","«","¬","-","®","¯",
"°","±","²","³","´","µ","¶","·",
"¸","¹","º","»","¼","½","¾","¿",
"À","Á","Â","Ã","Ä","Å","Æ","Ç",
"È","É","Ê","Ë","Ì","Í","Î","Ï",
"Ð","Ñ","Ò","Ó","Ô","Õ","Ö","×",
"Ø","Ù","Ú","Û","Ü","Ý","Þ","ß",
"à","á","â","ã","ä","å","æ","ç",
"è","é","ê","ë","ì","í","î","ï",
"ð","ñ","ò","ó","ô","õ","ö","÷",
"ø","ù","ú","û","ü","ý","þ","ÿ" };

#define VHPI_GET_PRINTABLE_STRINGCODE( ch )  VHPICharCodes[PLI_UBYTE8 ch]

PLI_INT32 vhpi_is_printable( PLI_BYTE8 ch )
{
unsigned char uch = (unsigned char)ch;

    if (uch < 31) return 0;
      if (uch < 127) return 1;
      if (uch == 127) return 0;
      if (uch < 160) return 0;
      return 1;
}
```

User code:

```
int PrintMyNastyVHDLString( char* VHDLString, int Length )
{
int i;
unsigned char ch;
int needcomma=0;
  for (i=0; i<Length; i++)
  {
         ch = (unsigned char)VHDLString[i];
             if (vhpi_is_printable(ch))
             {
               vhpi_printf("%c", ch );
                     needcomma=1;
             }
             else
             {
               if (needcomma) printf(",");
                     vhpi_printf("%s",
VHPI_GET_PRINTABLE_STRINGCODE(ch)  );
                     if (i!=(Length-1)) vhpi_printf(",");
                     needcomma=0;
             }
  }
  return 0;
}
```

The output of that program to the screen for the following input string literal:

HELLO & NUL & C128 & DEL

is

HELLO,NUL,C128,DEL

11.25 vhpi_put_data()

vhpi_put_data()			
Synopsis:	Save data to a simulation save location.		
Syntax:	vhpi_put_data(PLI_INT32 id, PLI_VOID *dataLoc, PLI_INT32 numBytes)		
	Type	**Description**	
Returns:	PLI_INT32	the number of bytes saved or 0 on error	
	Type	**Name**	**Description**
Arguments:	PLI_INT32	id	an identifier denoting a position in a saved location
	PLI_VOID *	dataLoc	the address of the data to be saved.
	PLI_INT32	numBytes	the number of bytes to write out.
Related functions:	Use vhpi_get_data() to read data from a saved location Use vhpi_get(vhpiIdP, NULL) to get a new unique id.		

vhpi_put_data() shall be used to store “numBytes” of data located at “dataLoc” into a simulation saved location. The return value will be the number of bytes successfully saved. Id is a unique identifier for the simulation session that denotes a reserved area in the simulation save location. **vhpi_get(vhpiIdP, NULL)** shall be used to obtain a new unique identifier. The returned unique id denotes an index position that is used to refer to a reserved area in a simulation save location. This function shall be called during the save operation. The id returned is a non null integer. Each call to vhpi_get(vhpiIdP, NULL) will generate a new id.

There is no restriction on:

* how many times **vhpi_put_data()** can be called with the same “id”,
* how many “id”s a foreign model or an application creates,
* the order foreign models or applications store data using different “id”s.

The data from multiple calls to **vhpi_put_data()** with the same “id” must be stored by the simulator in a way that the opposite routine **vhpi_get_data(),** for the <u>same id</u>, will retrieve data in the order it was put in the save location . This allows the **vhpi_get_data()** function to pull the data stored in the save file from different “ids” corresponding to different save location index positions.

vhpi_put_data() can only be called from a callback routine which was registered for reason *vhpiCbStartOfSave* or *vhpiCbEndOfSave*.

<u>Example:</u> A consumer routine which saves data to a simulation save location and registers callback to restore the data.

See also the example in **vhpi_get_data()** for the description of the restart callback function.

```
/* type definitions for private data structures to save used by the
foreign models or applications */
struct myStruct{
  struct myStruct *next;
  int d1;
  int d2;
}
void consumer_save(vhpiCbDataT *cbDatap)
{
  char *data;
  vhpiCbDataS cbData; /* a cbData structure */
```

```
   int cnt = 0;
   struct myStruct *wrk;
   vhpiHandleT cbHdl; /* a callback handle */
   int id =0;
   int savedBytesCount = 0;
   /* get the number of structures */
   wrk = firstWrk;
   while (wrk)
   {
      cnt++;
      wrk = wrk->next;
   }
   /* request an id */
   id = vhpi_get(vhpiIdP, NULL);
   /* save the number of data structures */
   savedBytesCount = vhpi_put_data(id, (char*)&cnt, sizeof(int);
   /* reinitialize wrk pointer to point to the first structure */
   /* save the different data structures, the restart routine will have
      to fix the pointers */
   while (wrk)
   {
     savedBytesCount += = vhpi_put_data(id, (char *)wrk, sizeof(struct
 myStruct));
     wrk = wrk->next;
   }
   /* check if everything has been saved */
   assert(savedBytesCount == 4 + cnt * (sizeof(struct myStruct)));
   /* now register the callback for restart and pass the id to retrieve
      the data, the user_data field of the callback data structure is
      one easy way to pass the id to the restart operation */
   cbData.user_data = (void *)id;
   cbData.reason = vhpiCbStartOfRestart;
   cbData.cb_rtn = consumer_restart; /* see example in vhpi_get_data()
                                      * for the description of
                                      * consumer_restart
                                      */
    vhpi_register_cb(&cbData, vhpiNoReturn);
 } /* end of consumer_save */
```

11.26 vhpi_put_value()

vhpi_put_value()			
Synopsis:	Update the value of an object, name or foreign function returned value		
Syntax:	vhpi_put_value(objHdl, valuep, flags)		
	Type	**Description**	
Returns:	PLI_INT32	0 on sucess, non 0 on failure to change the value	
	Type	**Name**	**Description**
Arguments:	vhpiHandleT	objHdl	Handle to an object of which the value can be changed
	vhpiValueT *	valuep	Pointer to a value
	PLI_UINT32	flags	Flags values defined in the enumeration type vhpiPutValueModeT
Related functions:	Use vhpi_get_value() to get an object to a value. Use vhpi_schedule_transaction to update the waveform of a signal driver.		

vhpi_put_value() shall update the value of an object. Classes to which this operation can be applied are either sub-classes of the class ObjDecl, Name or foreign function call. The function takes 3 arguments: ***objHdl***, handle to the object to which the value update will be applied, ***valuep,*** pointer to a value structure that describes the value. The function returns 0 on success and non zero on failure to apply the value. The update can be done in several ways, the third parameter, **flags**, indicate which kind of immediate update is requested.

If the **flag** parameter is set to:

vhpiDeposit: the value is immediately applied with no force, no propagation, no event creation
no signal value change callbacks trigger; variable value change callbacks trigger
only readers of that object will see the new value.

vhpiDepositPropagate:
value is immediately applied, propagated only for this cycle.
may create an event or remove an event, signal value change callback may trigger if signal effective value is changed; variable value change callbacks trigger.

vhpiForce: (until release)
no propagation, no VHDL net can overwritethat value readers can see the value, no value change callbacks trigger.
Force callbacks trigger if the object value is forced.

vhpiForcePropagate
value is forced, propagated, can create or remove an event
Signal value change callbacks trigger if an event occurs on the signal.
Force callbacks trigger if the object value is forced.

vhpiRelease
The previous forced value is released, if the object was not forced, this has no effect. The value of

Deleted:

Formatted: Indent: Left: 0.5"

Deleted:

the object is then left to the network updation. The pointer to the value structure valuep is not required when vhpi_put_value if called with the vhpiRelease flag. If a non null valuep pointer is provided, it will be ignored.

Release callbacks trigger if the object changed from a state of forced to released.

Formatted: Indent: First line: 0.5"

The property *vhpiIsForcedP* is true for an object that is being forced with **vhpi_put_value()** and the flag parameter was *vhpiForce* or *vhpiForcePropagate*.

The semantics related to modes and classes of the VHDL formal parameter interface declarations are carried on by foreign subprograms. Runtime errors should be generated if the VHDL rules are violated by the foreign C code; for example, one should not call *vhpi_put_value()* on a handle to a IN formal parameter, or try to get the value of an OUT formal parameter by calling *vhpi_get_value()*.

vhpiSizeConstraint

vhpi_put_value can be used to set the returned value of a foreign function call. If the function return type is unconstrained, a preliminary call to **vhpi_put_value** with a flag argument set to *vhpiSizeConstraint* will set the constraint of the returned value, the *numElems* field of the value parameter shall be set to the size of the constraint. The first argument must be a handle to the function call. The second call to **vhpi_put_value** will actually pass the value to be returned. The flag argument should be set to *vhpiDeposit*. If **vhpi_put_value** is used to provide the value of an unconstrained type object, it must first be called with *vhpiSizeConstraint* prior to setting the value.

11.27 vhpi_register_cb()

vhpi_register_cb()			
Synopsis:	Register a callback function for a specific reason		
Syntax:	vhpi_register_cb(cbDatap, flags)		
	Type	**Description**	
Returns:	vhpiHandleT	A handle to the callback object or NULL	
	Type	**Name**	**Description**
Arguments:	vhpiCbDataT *	cbDatap	Pointer to a structure with data about which and when callback should occur and data to be passed.
	PLI_UINT32	flags	defined constant value flags: vhpiDisableCb and vhpiReturnCb
Related functions:	Use vhpi_remove_cb() to remove the callback. Use vhpi_get_cb_info to get information about the given callback handle		

vhpi_register_cb() shall be used to register a callback for a specific reason. The reason is the condition of occurrence of the callback.
The cbDatap argument should point to a vhpiCbDataS structure that is defined in the VHPI standard header file. This data structure is allocated by the caller and should contain information about the callback to be registered. Depending on the reason, some fields of the vhpiCbDataS structure must be provided (refer to chapter 8). The *obj* field of cbDatap may be set to a handle; the client code is free to release that handle after the callback has been registered with no impact on the callback registration. If the flags field is set to *vhpiReturnCb*, the registration function shall return a callback handle, otherwise it shall return NULL. The callback can be set to a disabled state at the registration if the flags field is set to *vhpiDisableCb*.

Note: It is useful for a client application which disables the callback at the registration to also set the flags field to *vhpiReturnCb* so that it can retain the callback handle to enable it at a future time. Alternatively the client application can use the *vhpiCallbacks* iteration method in conjunction with the query property vhpiStateP to find the disabled callbacks.

When the callback function cb_rtn(const struct vhpiCbDataS *cbDatap) triggers, the cbDatap is allocated by the VHPI server and is a read only data structure. The contents of the cbDatap passed in the callback function are equivalent to the content of the cbDatap structure which was passed at the callback registration to vhpi_register_cb(). Specifically if the obj field is set to a handle, it may not be the same handle which was set at the time of the registration. The entire cbDatap structure and its contents (including the handle) are owned by the VHPI server and are not guaranteed to be valid outside the callback function scope. In particular the client code shall not release the handle pointed by the obj field.

```
typedef struct vhpiCbDataS {
    int reason;        /* the reason of the callback */
    (void) (*cb_rtn)(const struct vhpiCbDataS *); /* the
                        callbackfunction pointer */
    vhpiHandleT obj;   /* a handle to an object */
    vhpiTimeT *time;   /* a time */
```

```
    vhpiValueT *value; /* a value */
    void *user_data;   /* user data information */

} vhpiCbDataT, *vhpiCbDatap;
```

Example 1: Register value change callbacks on all signals in the design

```
/* the callback function */
void vcl_trigger(cbDatap)
const vhpiCbDataT *cbDatap;
{
char *sigName;
int toggleCount = (int)(cbDatap->user_data);

    cbDatap->user_data = (char *)(++toggleCount);
    sigName= vhpi_get_str(vhpiFullNameP, cbDatap->obj);
    vhpi_printf("Signal %s changed value %d, at time %d\n", sigName,
                 cbDatap->value.int, cbDatap->time.low);
    return;
}
static void monitorSignals(instHdl); /* this is the name of the
                                        function which registers signal
                                        value change callbacks */
vhpiHandleT instHdl; /* a handle to an instance */
{
   /* monitors all signals in this instance */
static vhpiCbDataT  cbData;
vhpiValueT value;
vhpiTimeT time;
int flags;

  value.format = vhpiIntVal;
  cbData.reason = vhpiCbValueChange;
  cbData.cb_rtn = vcl_trigger;
  cbData.value = &value;  cbData.time = &time;
  cbData.user_data = 0;
  flags = 0; /* do not return a callback handle and do not disable the
callback at registration */
/* register the callback function */
  sigIt = vhpi_iterator(vhpiSigDecls, instHdl);
  if(!sigIt) return;
  while(sigHdl = vhpi_scan(sigIt))
  {
     cbData.obj = sigHdl;
     vhpi_register_cb(&cbData, flags);
  }

}
```

11.28 vhpi_register_foreignf()

vhpi_register_foreignf()			
Synopsis:	Register foreign architecture/procedure/function/application related functions		
Syntax:	vhpi_register_foreignf(foreignDatap)		
	Type	**Description**	
Returns:	vhpiHandleT	A handle to the callback object.	
	Type	**Name**	**Description**
Arguments:	vhpiForeignDataT *	foreignDatap	Pointer to a structure with data about which and when elab and execution functions should occur and data to be passed.
Related functions:	Use vhpi_register_cb() to register other reason callbacks for simulation events. Use vhpi_get_foreignf_info() to get information about which functions were registered for a particular model.		

vhpi_register_foreignf () shall be used to register foreign C functions for foreign architecture elaboration and initialization, procedure, function or application execution. The functions registered correspond to the C behaviour to invoke when a foreign architecture is encountered during elaboration of the VHDL code or when a foreign architecture, procedure, function or application is executed during simulation of the VHDL design.

The foreignDatap argument should point to a vhpiForeignDataT structure that is defined in the VHPI standard header file. This data structure contains information about the elaboration or execution functions.

```
typedef struct vhpiForeignDataS {
    vhpiForeignT kind;/* the foreign model class kind:
                         vhpi[Arch,Proc,Func]F */
    char *libraryName;/* the library name that appears in the
                         VHDL foreign attribute string */
    char *modelName;  /* the model name that appears in the
                         VHDL foreign attribute string or
                         application name PA 28*/
    void (*elabf)( const vhpiCbDataT *);
                      /* the callback function pointer for
                         elaboration of foreign architecture */
    void (*execf)( const vhpiCbDataT *);
                      /* the callback function pointer for
                         initialization/simulation execution of
                         foreign   architecture, procedure,
                         function or application */
} vhpiForeignDataT, *vhpiForeignDatap;
```

This data structure contains the mapping between a given foreign model and the C functions which implement the foreign model behaviour. The data is described below:

The *kind* field should register the foreign model to be an architecture, a procedure, a function or an application. The *kind* field value should be one of the following enumeration constants: **vhpiArchF**, **vhpiProcF**, **vhpiFuncF, vhpiAppF**.

The *libraryName* and *modelName* are respectively the library logical name and model name that are found in the VHPI foreign attribute string for foreign architecture and subprograms and the library and application names for a foreign application.

The *elabf* and *execf* fields should be pointers to the user-defined functions.
The function pointed by the *elabf* field will be invoked during elaboration of the foreign model. The function pointed by the *execf* field will be invoked during simulation initialization and/or execution of the foreign model or application. For foreign architectures, the *execf* function call occurs **once** during simulation initialization. For foreign procedures or functions, the *execf* function call occurs **each time** the VHDL corresponding procedure or function is invoked during simulation. For a foreign application, the *execf* function is called once before vhpiCbStartOfTool if the tool has determined that this application must be activated in this session. The determination of which applications must be activated is tool specific.

vhpi_register_foreignf() returns a handle to the model registered functions. The handle is of type **vhpiForeignfK.** The function **vhpi_get_foreignf_info()** shall be used to retrieve the foreign model information that was registered for a given foreign model or application. The VHPI method **vhpiForeignfs** can be used to iterate over all the registered foreign models and applications in a given tool session; the reference handle passed in should be null.

The following example illustrates how a user can dynamically link foreign model function callbacks.

<u>Example:</u>

```
void dynlink(foreignName, libName)
char * foreignName; /* name of the foreign model to link in */
char * libName;     /* logical name of the C dynamic library where the
                       model resides */
{
  static vhpiForeignDataT archData = {vhpiArchF};
  char dynLibName[MAX_STR_LENGTH];
  char platform[6];
  char extension[3];
  char fname[MAX_STR_LENGTH];
  char elabfname[MAX_STR_LENGTH];
  char execfname[MAX_STR_LENGTH];



  sprintf(platform, getenv("SYSTYPE"));
  if (!strcmp(platform, "SUNOS"))
      strcpy(extension, "so");
  else if (!strcmp(platform, "HP-UX"))
      strcpy(extension, "sl");

  sprintf(dynLibName, "%s.%s", libName, extension);
  sprintf(fname, "%s", foreignName);
  sprintf(elabfname, "elab_%s", foreignName);
  sprintf(execfname, "sim_%s", foreignName);
  archData->libraryName = libname;
  archData->modelName = fName;
  /* find the function pointer addresses */
  archData->elabf = (void(*)()) dynlookup(dynLibName, elabfName);
  archData->execf = (void(*)()) dynlookup(dynLibName, execfName);

 vhpi_register_foreignf(&archData);
}
```

This next example illustrates how to write a bootstrap function for a library of models. This bootstrap function can be called just after the VHDL tool (elaborator or simulator) has been invoked. It registers all the models defined in the C library at once. One way of writing this bootstrap function is to have an internal library table of **vhpiForeignDataS** structures. An entry in that table corresponds to a C model in the library, then the bootstrap function just needs to iterate through the entries in that table, and for each entry, call **vhpi_register_foreignf()**. The developer of the library has also the possibility to separate the registration of his models into several bootstrap functions.

Example 2:

```
extern void register_my_C_models(); /* this is the name of the bootstrap
                                       function that must be the ONLY
                                       visible symbol of the C library.
                                    */
void register_my_C_models()
{
   static vhpiForeignDataT foreignDataArray[] = {
      {vhpiArchF, "lib1", “C_AND_gate”, “elab_and”, “sim_and”},
      {vhpiFuncF, "lib1", “addbits”, 0, “ADD”},
      {vhpiProcF, "lib1", “verify”, 0, “verify”},
      0
   };
/* start by the first entry in the array of the foreign data structures
*/
vhpiForeignDatap  foreignDatap = &(foreignDataArray[0]);

/* iterate and register every entry in the table */
while (*foreignDatap)
    vhpi_register_foreignf(foreignDatap++);
}
```

Errors:

The registration of a foreign model fails:

- if the required information is not present,
- if the library name/model name pair is not unique.

11.29 vhpi_release_handle()

vhpi_release_handle()			
Synopsis:	Release handle reference, free any memory allocated for this handle		
Syntax:	vhpi_release_handle(hdl)		
	Type	**Description**	
Returns:	PLI_INT32	0 on success, 1 on failure	
	Type	**Name**	**Description**
Arguments:	vhpiHandleT	hdl	Handle to an object

vhpi_release_handle() can be used by an application to tell the VHPI interface that it does not intend to reference and use the passed handle any more. Some implementations may free the memory that they had allocated to construct this handle in the case where the handle does not refer to an internal simulation or elaboration internal object. This function can be used for handles obtained from the navigation functions, callback registration, transaction scheduling. The function returns 0 on success and 1on failure.

Example:

```
vhpiHandleT rootHdl, itrHdl;

rootHdl = vhpi_handle(vhpiRootInst, null);
itrHdl = vhpi_iterator(vhpiInternalRegions, rootHdl);
if (itrHdl) {
  while (instHdl = vhpi_scan(itrHdl)) {
        vhpi_printf("found sub-scope %s\n",
                    vhpi_get_str (vhpiName, instHdl));
  }
}
itrHdl = vhpi_iterator(vhpiInternalRegions, rootHdl);
if (itrHdl) {
  while (instHdl = vhpi_scan(itrHdl)) {
        if (vhpi_get(vhpiKindP, instHdl) == vhpiBlockStmtK)
            break;
        /* free this instance handle */
        vhpi_release_handle(instHdl);
  }
}
```

11.30 vhpi_remove_cb()

vhpi_remove_cb()			
Synopsis:	Remove a callback that was registered using vhpi_register_cb().		
Syntax:	vhpi_remove_cb(cbHdl)		
	Type	**Description**	
Returns:	PLI_INT32	0 on success, 1 on failure.	
	Type	**Name**	**Description**
Arguments:	vhpiHandleT	cbHdl	A callback handle of kind vhpiCallbackK
Related functions:	Use vhpi_register_cb() to register a callback		

vhpi_remove_cb () shall be used to remove a callback that was registered using *vhpi_register_cb()*. The argument to this function should be a handle to the callback. The function returns 0 on success and 1 on failure. After the callback has been removed, the callback handle becomes invalid (the interface implicitly free the memory that was allocated for the callback including the callback handle).

Example:

```
int find_cbk(objHdl)
vhpiHandleT objHdl; /* a handle to an object */
{
vhpiHandleT cbHdl, cbItr;
vhpiCbDataT cbdata;
int found = 0;

/* find a specific callback on value change that was registered for that
object and remove it */

cbItr = vhpi_iterator(vhpiCallbacks, objHdl);
if (!cbItr) return;
while (cbHdl = vhpi_scan(cbItr)) {
      vhpi_get_cb_info(cbHdl, &cbdata);
     if (cbdata.user_data == 2) {
        vhpi_remove_cb(cbHdl);
        found = 1;
        vhpi_release_handle(cbItr); /* free the iterator */
        break;
     }
}
return(found);
}
```

11.31 *vhpi_scan()*

vhpi_scan()			
Synopsis:	Scan the VHDL model for objects in a one-to-many relationship with the reference handle indicated by the iterator handle		
Syntax:	vhpi_scan(iterHdl)		
	Type	**Description**	
Returns:	vhpiHandleT	A handle on success, NULL if no objects of the type and reference handle indicated by the iterator exists.	
	Type	**Name**	**Description**
Arguments:	vhpiHandleT	iterHdl	An iterator handle created by vhpi_iterator()
Related functions:	Use vhpi_iterator() to get an iterator handle		

vhpi_scan () shall be used to obtain handles to objects that are in a one-to-many relationship with the reference handle indicated by the iterator handle passed in. The **vhpi_scan()** function returns NULL when there is no more handle that comply to the iterator. The iterator handle is automatically released at the end of the iteration. References to the iterator after the end of an iteration are erroneous. If the iteration is not exhausted, the user should explicitly release the iterator to avoid a memory leak.

Example:

```
void find_signals(scopeHdl)
vhpiHandleT scopeHdl; /* a handle to a scope */
{
vhpiHandleT sigHdl,itrHdl;

/* find all signals in that scope and print their names */

itrHdl = vhpi_iterator(vhpiSigDecl, scopeHdl);
if (!itrHdl) return;
    while (sigHdl = vhpi_scan(itrHdl)) {
       vhpi_printf("Found signal %s\n", vhpi_get_str(vhpiNameP, sigHdl));
       /* done with handle */;
       vhpi_release_handle(sigHdl);
    }
}
```

Deleted: hpiHandleT

Deleted: int found = 0;¶

11.32 vhpi_schedule_transaction()

vhpi_schedule_transaction()			
Synopsis:	Schedule transactions on drivers		
Syntax:	vhpi_schedule_transaction(hdl, valuep, numValues, delayp, delayMode, pulseRefp)		
	Type	**Description**	
Returns:	PLI_INT32	0 on success, non zero on failure.	
	Type	**Name**	**Description**
Arguments:	vhpiHandleT	hdl	A handle to a driver or a drivercollection if routine is used to schedule a transaction
	vhpiValueT **	valuep	Array of pointer to values for the driver transaction
	PLI_UINT32	numValues	Number of values in valuep
	vhpiTimeT *	delayp	Relative time delay for the transaction
	PLI_UINT32	delayMode	Delay mode to apply
	vhpiTimeT *	pulseRejp	Pulse rejection limit for inertial mode
Related functions:	Use vhpi_put_value to change the effective value of a signal.		

vhpi_schedule_transaction() shall be used to schedule a transaction on a driver. The function can only be called during process execution phase. The transaction can be scheduled with zero or non-zero delay and with inertial or transport mode. To schedule a value on a driver requires to get a handle to a driver, ***hdl,*** of the signal, specify a value pointer, ***valuep***, the number of values, ***numValues***, a delay, ***delayp*** and delay mode, ***delayMode***.
The delay modes that are supported are ***vhpiInertial*** and ***vhpiTransport***. In the case of inertial delays, a user could specify an additional optional parameter, the pulse rejection limit that should be passed in ***pulseRej*** argument.
The value pointer could be
1. a pointer to a single value in the case of a scalar driver or a composite driver
2. a pointer to several values in the case of a composite driver.

There is no correspondance between the number of drivers in the driverCollection and the number of values structures passed in.
Note: scheduling transaction for drivers of composite type not resolved at the composite level, such as bit vectors, standard logic vectors, record types must be done by scheduling individual transactions on each scalar driver (see example 1) or by scheduling a transaction on a collection composed of these drivers.

Transaction for driver of signals resolved at the composite level must be scheduled at once; the user must allocate as many as value structures that are necessary for scheduling a transaction on the composite. The format and corresponding union value field must be set for each value structure. The space for holding each individual value (value union field of each individual vhpiValueS structure) must be allocated by the user in accordance with the chosen format. The next argument, ***numValues***, indicates how many values are provided by ***valuep***.
In general there will not be any VHPI check that all values have been actually allocated and set but the simulator may issue an error for case 2 if the composite driver is not assigned as a whole.
There will be a check that the specified format is allowed.
The array of values follows the order defined by the LRM for array or record aggregates.

The format of the specified value must be appropriate with the VHDL type of the driver as defined in chapter 9. For example, if the driver type is standard logic, the format cannot be vhpiRealVal. Runtime type errors shall occur if the provided value is not within the range of the driver subtype. The execution of a ***vhpi_schedule_transaction*** involves runtime constraint type checking but does not guarantee that the operation had no effect.

The pulse rejection limit is specified by a time structure. If the simulator is VHDL’87 compliant, it may ignore the pulse rejection limit. The VHPI interface should warn the application that this pulse time rejection is being ignored.

NULL transactions can be posted by setting ***valuep*** to a NULL pointer.

The specified delay is a relative delay and is specified by using the time structure ***vhpiTimeS*** to providing a 64 bits value.
Francoise: Since all time value are always expressed in terms of the resolution limit, how can it be possible to indicate a time value smaller than the resolution limit?The delay value is truncated if smaller than the VHDL simulator resolution limit. The delay value must be legal as if it were the delay value of a VHDL signal assignment.
Zero delay transactions are specified by setting the low and high fields of the time structure to 0.

Scheduling a 0 delay transaction is only allowed during non postponed process execution and any callback which occur during the non postponed process execution phase:

i) vhpiCb(Rep)StartOfProcesses
ii) vhpiCb(Rep)TimeOut
iii) vhpiCbSensitivity
iv) vhpiCbStmt,
v) vhpiCbResume
vi) vhpiCbSuspend
vii) vhpiStartOfSubpCall
viii) vhpiEndOfSubpCall
ix) vhpiCb(Rep)LastKnownDeltaCycle
x) vhpiCb(Rep)EndOfProcesses

Non-zero delay transactions can be scheduled during non postponed and postponed process execution and any callback which occur during the non postponed and postponed process execution phase:

i) vhpiCb(Rep)StartOfProcesses
ii) vhpiCb(Rep)EndOfProcesses
iii) vhpiCb(Rep)TimeOut
iv) vhpiCbSensitivity
v) vhpiCbStmt,
vi) vhpiCbResume
vii) vhpiCbSuspend
viii) vhpiStartOfSubpCall
ix) vhpiEndOfSubpCall
x) vhpiCb(Rep)LastKnownDeltaCycle
xi) vhpiCb(Rep)StartOfPostponed
x) vhpiCb(Rep)EndOfPostponed

Non-zero delay transaction scheduling must occur before vhpiCbEndOfTimeStep, effectively before the next time is computed. Scheduling a transaction at any other phase has no effect and may generate an error. All zero delay transactions will be scheduled for the next delta cycle and all other delay transactions will be scheduled for the time that corresponds to the current simulation time added to the specified relative delay.

Example 1: Recursive function which schedules a transaction on a signal.

The code of this function is incomplete: the case describes un unresolved record with a bit and a bit vector members.

.

Iteration on vhpiSelectedNames obtain a handle to each record member. For the scalar bit field, schedule a transaction on the single scalar driver. For the bit vector field, iterate on vhpiIndexedNames to get a handle to each bit element of the bit vector field, then use the driver iteration to obtain a driver of each bit element. Create a collection of the drivers and then schedule a bit vector value transaction on the collection.

```
Type R is Record
  B : BIT;
  BARR : BIT_VECTOR (0 to 7);
end record;

signal S : R;

int schedule_transaction_value(sigHdl)
vhpiHandleT sigHdl; /* a handle to a signal */

{
    vhpiHandleT baseTypeHdl, driverIt, driverHdl;

    char *name;
    vhpiValueS value;
    vhpiTimeS delay;

    delay.low = 1000;/* delay is 1 ns */
    delay.high = 0;

    baseTypeHdl = vhpi_handle(vhpiBaseType, sigHdl);

    /* check the signal type */
    switch (vhpi_get(vhpiKindP, baseTypeHdl))
    {
    casevhpiRecordTypeDeclK :
    {
       vhpiHandleT itsel, selh;
       if (!vhpi_get(vhpiIsResolved, sigHdl)
       { /* signal not resolved at the composite level */
         itsel = vhpi_iterator(vhpiSelectedNames, sigHdl);
         while (selh = vhpi_scan(itsel))
            schedule_transaction_value(selh);
       }
       else
       {
          vhpi_printf("unimplemented\n");
          return -1;
       }
     }
     break;

     case vhpiArrayTypeDeclK:
     {  /* get the element subtype */
        vhpiHandleT eltSubtypeHdl, bitIt, bitHdl,
        vhpiHandleT colHdl = NULL;
        int countdrivs = 0;

       if (vhpi_get(vhpiIsResolved, sigHdl))
       {
```

```
            vhpi_printf("unimplemented\n");
            return -1;
         }
         /* signal not resolved at the composite level */
         elemSubtypeHdl = vhpi_handle(vhpiElemType, baseTypeDecl);
         baseTypeHdl = vhpiHandle(vhpiBaseType, elemSubtypeHdl);
         name = vhpi_get_str(vhpiNameP, baseTypeHdl);
         if (!strncmp(name, "BIT"))
         {
            bitIt = vhpi_iterator(vhpiIndexedNames, sigHdl);
            while (bitHdl = vhpi_scan(bitIt))
            {
               assert (vhpi_get(vhpiIsBasicP, bitHdl) == vhpiTrue);
               driverIt = vhpi_iterator(vhpiDrivers, bitHdl);
               while (driverHdl = vhpi_scan(driverIt))
               {
                  countdrivs++;
                  colHdl = vhpi_create(vhpiDriverCollectionK, colHdl,
driverHdl);
               }
             }
             value.format = vhpiLogicVecVal;
             value.numElems = countDrivs;
             while (countdrivs)
             {
                value.value.logics++ = vhpiBit0;
                countdrivs--;
             }
             vhpi_schedule_transaction(colHdl, &value, 1,
                                    &delay, vhpiInertial, 0);

          }
          else
          {
             vhpi_printf("unimplemented\n");
             return -1;
          }
        }
        break;
        case vhpiEnumTypeDeclK:
        {
           name = vhpi_get_str(vhpiNameP, baseTypeHdl);
           if (!strncmp(name, "BIT"))
           {
              value.format = vhpiLogicVal;
              value.logic = vhpiBit0;

              assert (vhpi_get(vhpiIsBasicP, sigHdl) == vhpiTrue);
              driverIt = vhpi_iterator(vhpiDrivers, sigHdl);
              while (driverHdl = vhpi_scan(driverIt))
                  countdrivs++;
              assert (countDrivs == 1);
              vhpi_schedule_transaction(driverHdl, &value, 1,
                                   &delay, vhpiInertial, 0);

           }
           else
           {
              vhpi_printf("unimplemented\n");
              return -1;
```

```
            }
        }
        break;
        default:
            vhpi_printf("unimplemented\n");
            return (-1);
        break;
    }
}
```

The VHPI interface should report an error if:

1. A negative relative delay is provided

 Issue: if the times structure only contains UINT, it is not possible to specify a negative value.

2. The value given is not compatible with the signal base type.
3. The driver handle used to schedule an update is NULL.
4. The pulse rejection limit is greater than the inertial delay that is provided.
5. A zero delay transaction is attempted to be scheduled within the postponed process execution phase.
6. The handle passed in does not denote a driver.

12. Interoperability between VPI and VHPI

Function Category	Function Purpose	VHPI function	VPI function
Utilities	Checks/returns error info	vhpi_check_error()	vpi_chk_error()
	Send control ommands to the simulator	vhpi_control	vpi_control
	Compares handles	vhpi_compare_handles()	vpi_compare_objects()
	Deallocates handle	vhpi_release_handle()	vpi_free_object()
	Get current simulation time	vhpi_get_time()	vpi_get_time()
	Returns invocation information	NA (see tool class)	vpi_get_vlog_info()
	Closes mcd channels	NA	vpi_mcd_close()
	Open mcd channels	NA	vpi_mcd_open()
	Flush	NA	vpi_flush()
	Flush mcd channels	NA	vpi_mcd_flush()
	Performs like C printf	vhpi_printf()	vpi_printf()
Navigation access	Follows a singular relationship	vhpi_handle()	vpi_handle()
	Follows an iteration relationship	vhpi_iterator()	vpi_iterate()
	Gets next handle iteration element	vhpi_scan()	vpi_scan()
	Obtains a handle from a name reference	vhpi_handle_by_name()	vpi_handle_by_name()
	Obtains a handle to an indexed element	vhpi_handle_by_index()	vpi_handle_by_index()
	Obtains a handle to a multi-index element	vhpi_handle_by_index()	vpi_handle_by_multi_index()
	Obtains a handle to multi handles	vhpi_handle_multi()	vpi_handle_multi()
Property access	Returns value of integer property	vhpi_get()	vpi_get()
	Returns value of string property	vhpi_get_str()	vpi_get_str()
	Returns value of a physical property	vhpi_get_phys	NA
	Returns value of real property	vhpi_get_real()	NA
Value access and modifications	Gets the value of an object	vhpi_get_value()	vpi_get_value()
	Forces or schedules a zero delay value on an object	vhpi_put_value()	vpi_put_value()
	Schedule a future or zero delay transaction	vhpi_schedule_transaction( )	vpi_put_value()
	Get a delay value	NA	vpi_get_delays()
	Protected type access	vhpi_protected_call()	NA
	Modify a delay value	NA	vpi_put_delays()
Callbacks	Registers a callback	vhpi_register_cb()	vpi_register_cb()
	Removes a callback	vhpi_remove_cb	vpi_remove_cb()
	Enables a callback	vhpi_enable_cb()	vpi_enable_cb()
	Disables a callback	vhpi_disable_cb()	vpi_disable_cb()
	Gets callback info	vhpi_get_cb_info()	vpi_get_cb_info()
C Foreign function	Registers a foreign function	vhpi_register_foreignf()	vpi_register_systf()
	Gets foreign function info	vhpi_get_foreignf_info()	vpi_get_systf_info()
	Create a new elaborated object	vhpi_create()	vpi_create()
	Save data in a file	vhpi_put_data()	vpi_put_userdata()
	Restore data from file	vhpi_get_data()()	vpi_put_userdata()

Table 1: Correspondence between VHPI and VPI functions

13. Annex A (normative) VHPI header file

The header file should be included by any application intending to use VHPI. This header file should be provided by tool vendors supporting the VHPI interface.
The range of values from of 1000 to 2000 are RESERVED by the standard.
Vendors are allowed to provide additional functionality (other than the one defined by the standard) and incorporate it in the header. This can be done by defining the following macros.

Note: INT_AMS * macros are place holders for VHPI ams extensions.

```
/* define internal macros for VHPI internal functionality */
#ifndef VHPI_INTERNAL_H
#define INT_CLASSES
#define INT_AMS_CLASSES
#define INT_ONE_METHODS
#define INT_AMS_ONE_METHODS
#define INT_MANY_METHODS
#define INT_AMS_MANY_METHODS
#define INT_INT_PROPERTIES
#define INT_AMS_INT_PROPERTIES
#define INT_STR_PROPERTIES
#define INT_AMS_STR_PROPERTIES
#define INT_REAL_PROPERTIES
#define INT_AMS_REAL_PROPERTIES
#define INT_PHYS_PROPERTIES
#define INT_AMS_PHYS_PROPERTIES
#define INT_VAL_FORMATS
#define INT_AMS_VAL_FORMATS
#define INT_ATTR
#define INT_AMS_ATTR
#define INT_PROTOTYPES
#endif

/*
 * |---------------------------------------------------------------|
 * |                                                               |
 * | This is the VHPI header file                                  |
 * |                                                               |
 * |                                                               |
 * |                                                               |
 * | FOR CONFORMANCE WITH THE VHPI STANDARD, A VHPI APPLICATION    |
 * | OR PROGRAM MUST REFERENCE THIS HEADER FILE                    |
 * | Its contents can be modified to include vendor extensions.    |
 * |                                                               |
 * |---------------------------------------------------------------|
 */

/*** File vhpi_user.h ***/
/*** This file describe the procedural interface to access VHDL
     compiled, instantiated and run-time data. It is derived from
     the UML model. ***/

#ifndef VHPI_USER_H
#define VHPI_USER_H

#ifdef  __cplusplus
extern "C" {
#endif
```

```
/*----------------------------------------------------------------------
-----*/
/*--------------------------- Portability Help -------------------------
-----*/
/*----------------------------------------------------------------------
-----*/

/* Sized variables */
#ifndef PLI_TYPES
#define PLI_TYPES
typedef int             PLI_INT32;
typedef unsigned int    PLI_UINT32;
typedef short           PLI_INT16;
typedef unsigned short  PLI_UINT16;
typedef char            PLI_BYTE8;
typedef unsigned char   PLI_UBYTE8;
#endif

typedef void            PLI_VOID;

/* Use to export a symbol */
#if WIN32
#ifndef PLI_DLLISPEC
#define PLI_DLLISPEC __declspec(dllimport)
#define VHPI_USER_DEFINED_DLLISPEC 1
#endif
#else
#ifndef PLI_DLLISPEC
#define PLI_DLLISPEC
#endif
#endif

/* Use to import a symbol */
#if WIN32
#ifndef PLI_DLLESPEC
#define PLI_DLLESPEC __declspec(dllexport)
#define VHPI_USER_DEFINED_DLLESPEC 1
#endif
#else
#ifndef PLI_DLLESPEC
#define PLI_DLLESPEC
#endif
#endif

/* Use to mark a function as external */
#ifndef PLI_EXTERN
#define PLI_EXTERN
#endif

/* Use to mark a variable as external */
#ifndef PLI_VEXTERN
#define PLI_VEXTERN extern
#endif

#ifndef PLI_PROTOTYPES
#define PLI_PROTOTYPES
#define PROTO_PARAMS(params) params
/* object is defined imported by the application */
#define XXTERN PLI_EXTERN PLI_DLLISPEC
/* object is exported by the application */
```

```
#define EETERN PLI_EXTERN PLI_DLLESPEC
#endif

/* define internal macros for VHPI internal functionality */
#ifndef VHPI_INTERNAL_H
#define INT_CLASSES
#define INT_AMS_CLASSES
#define INT_ONE_METHODS
#define INT_AMS_ONE_METHODS
#define INT_MANY_METHODS
#define INT_AMS_MANY_METHODS
#define INT_INT_PROPERTIES
#define INT_AMS_INT_PROPERTIES
#define INT_STR_PROPERTIES
#define INT_AMS_STR_PROPERTIES
#define INT_REAL_PROPERTIES
#define INT_AMS_REAL_PROPERTIES
#define INT_PHYS_PROPERTIES
#define INT_AMS_PHYS_PROPERTIES
#define INT_VAL_FORMATS
#define INT_AMS_VAL_FORMATS
#define INT_ATTR
#define INT_AMS_ATTR
#define INT_PROTOTYPES
#endif

/* basic typedefs */
#ifndef VHPI_TYPES
#define VHPI_TYPES
typedef PLI_UINT32 *vhpiHandleT;
typedef PLI_UINT32 vhpiEnumT;
typedef PLI_UINT32 vhpiIntT;
typedef char vhpiCharT;
typedef double vhpiRealT;
typedef struct vhpiPhysS
{
  PLI_INT32 high;
  PLI_UINT32 low;
} vhpiPhysT;

/********************** time structure ****************************/
typedef struct vhpiTimeS
{
  PLI_UINT32 high;
  PLI_UINT32 low;
} vhpiTimeT;

/********************** value structure **************************/

/* value formats */
typedef enum {
  vhpiBinStrVal        = 1, /* do not move */
  vhpiOctStrVal        = 2, /* do not move */
  vhpiDecStrVal        = 3, /* do not move */
  vhpiHexStrVal        = 4, /* do not move */
  vhpiEnumVal          = 5,
  vhpiIntVal           = 6,
  vhpiLogicVal         = 7,
  vhpiRealVal          = 8,
  vhpiStrVal           = 9,
```

```
  vhpiCharVal          = 10,
  vhpiTimeVal          = 11,
  vhpiPhysVal          = 12 ,
  vhpiObjTypeVal       = 13,
  vhpiPtrVal           = 14,
  vhpiEnumVecVal       = 15,
  vhpiIntVecVal        = 16,
  vhpiLogicVecVal      = 17,
  vhpiRealVecVal       = 18,
  vhpiTimeVecVal       = 19,
  vhpiPhysVecVal       = 20,
  vhpiPtrVecVal        = 21,
  vhpiRawDataVal       = 22

  INT_VAL_FORMATS
  INT_AMS_VAL_FORMATS

} vhpiFormatT;

/* value structure */
typedef struct vhpiValueS
{
  vhpiFormatT format;   /* vhpi[Char,[Bin,Oct,Dec,Hex]Str,
                                Enum, Logic,Int,Real,Phys,Time,Ptr,
                                EnumVec,LogicVec,IntVect,RealVec,PhysVec
,TimeVec,
                                PtrVec,ObjType,RawData]Val */
  PLI_UINT32 bufSize;  /* the size in bytes of the value buffer; this is
set
                          by the user */
  PLI_INT32 numElems;
  /* different meanings depending on the format:
     vhpiStrVal, vhpi{Bin...}StrVal: size of string
     array type values: number of array elements
     scalar type values: undefined
  */

  vhpiPhysT unit;     /* changed to vhpiPhysT in charles */
  union
    {
      vhpiEnumT enumv, *enumvs;
      vhpiIntT  intg, *intgs;
      vhpiRealT real, *reals;
      vhpiPhysT phys, *physs;
      vhpiTimeT time, *times;
      vhpiCharT ch, *str;
      void *ptr, **ptrs;
    } value;
} vhpiValueT;

#endif

/* Following are the constant definitions. They are divided into
   three major areas:

 1) object types

 2) access methods

 3) properties
```

```
*/
#define vhpiUndefined 1000

/*************** OBJECT KINDS *******************/
typedef enum {
        vhpiAccessTypeDeclK = 1001,
        vhpiAggregateK = 1002,
        vhpiAliasDeclK = 1003,
        vhpiAllK = 1004,
        vhpiAllocatorK = 1005,
        vhpiAnyCollectionK = 1006,
        vhpiArchBodyK = 1007,
        vhpiArgvK = 1008,
        vhpiArrayTypeDeclK = 1009,
        vhpiAssertStmtK = 1010,
        vhpiAssocElemK = 1011,
        vhpiAttrDeclK = 1012,
        vhpiAttrSpecK = 1013,
        vhpiBinaryExprK = 1014,
        vhpiBitStringLiteralK = 1015,
        vhpiBlockConfigK = 1016,
        vhpiBlockStmtK = 1017,
        vhpiBranchK = 1018,
        vhpiCallbackK = 1019,
        vhpiCaseStmtK = 1020,
        vhpiCharLiteralK = 1021,
        vhpiCompConfigK = 1022,
        vhpiCompDeclK = 1023,
        vhpiCompInstStmtK = 1024,
        vhpiCondSigAssignStmtK = 1025,
        vhpiCondWaveformK = 1026,
        vhpiConfigDeclK = 1027,
        vhpiConstDeclK = 1028,
        vhpiConstParamDeclK = 1029,
        vhpiConvFuncK = 1030,
        vhpiDerefObjK = 1031,
        vhpiDisconnectSpecK = 1032,
        vhpiDriverK = 1033,
        vhpiDriverCollectionK = 1034,
        vhpiElemAssocK = 1035,
        vhpiElemDeclK = 1036,
        vhpiEntityClassEntryK = 1037,
        vhpiEntityDeclK = 1038,
        vhpiEnumLiteralK = 1039,
        vhpiEnumRangeK = 1040,     /* new in ldv40 */
        vhpiEnumTypeDeclK = 1041,
        vhpiExitStmtK = 1042,
        vhpiFileDeclK = 1043,
        vhpiFileParamDeclK = 1044,
        vhpiFileTypeDeclK = 1045,
        vhpiFloatRangeK = 1046,
        vhpiFloatTypeDeclK = 1047,
        vhpiForGenerateK = 1048,
        vhpiForLoopK = 1049,
        vhpiForeignfK = 1050,
        vhpiFuncCallK = 1051,
        vhpiFuncDeclK = 1052,
        vhpiGenericDeclK = 1053,
        vhpiGroupDeclK = 1054,
```

```
        vhpiGroupTempDeclK = 1055,
        vhpiIfGenerateK = 1056,
        vhpiIfStmtK = 1057,
        vhpiInPortK = 1058,
        vhpiIndexedNameK = 1059,
        vhpiIntLiteralK = 1060,
        vhpiIntRangeK = 1061,
        vhpiIntTypeDeclK = 1062,
        vhpiIteratorK = 1063,
        vhpiLibraryDeclK = 1064,
        vhpiLoopStmtK = 1065,
        vhpiNextStmtK = 1066,
        vhpiNullLiteralK = 1067,
        vhpiNullStmtK = 1068,
        vhpiOperatorK = 1069,
        vhpiOthersK = 1070,
        vhpiOutPortK = 1071,
        vhpiPackBodyK = 1072,
        vhpiPackDeclK = 1073,
        vhpiPackInstK = 1074,
        vhpiParamAttrNameK = 1075,
        vhpiPhysLiteralK = 1076,
        vhpiPhysRangeK = 1077,
        vhpiPhysTypeDeclK = 1078,
        vhpiPortDeclK = 1079,
        vhpiProcCallStmtK = 1080,
        vhpiProcDeclK = 1081,
        vhpiProcessStmtK = 1082,
        vhpiProtectedTypeK = 1083,
        vhpiProtectedTypeBodyK = 1084,
        vhpiProtectedTypeDeclK = 1085,
        vhpiRealLiteralK = 1086,
        vhpiRecordTypeDeclK = 1087,
        vhpiReportStmtK = 1088,
        vhpiReturnStmtK = 1089,
        vhpiRootInstK = 1090,
        vhpiSelectSigAssignStmtK = 1091,
        vhpiSelectWaveformK = 1092,
        vhpiSelectedNameK = 1093,
        vhpiSigDeclK = 1094,
        vhpiSigParamDeclK = 1095,
        vhpiSimpAttrNameK = 1096,
        vhpiSimpleSigAssignStmtK = 1097,
        vhpiSliceNameK = 1098,
        vhpiStringLiteralK = 1099,
        vhpiSubpBodyK = 1100,
        vhpiSubtypeDeclK = 1101,
        vhpiSubtypeIndicK = 1102,
        vhpiToolK = 1103,
        vhpiTransactionK = 1104,
        vhpiTypeConvK = 1105,
        vhpiUnaryExprK = 1106,
        vhpiUnitDeclK = 1107,
        vhpiUserAttrNameK = 1108,
        vhpiVarAssignStmtK = 1109,
        vhpiVarDeclK = 1110,
        vhpiVarParamDeclK = 1111,
        vhpiWaitStmtK = 1112,
        vhpiWaveformElemK = 1113,
        vhpiWhileLoopK = 1114
```

```
        INT_CLASSES
        INT_AMS_CLASSES
} vhpiClassKindT;

/************** methods used to traverse 1 to 1 relationships
*******************/
typedef enum {
        vhpiAbstractLiteral = 1301,
        vhpiActual = 1302,
        vhpiAll = 1303,
        vhpiAttrDecl = 1304,
        vhpiAttrSpec = 1305,
        vhpiBaseType = 1306,
        vhpiBaseUnit = 1307,
        vhpiBasicSignal = 1308,
        vhpiBlockConfig = 1309,
        vhpiCaseExpr = 1310,
        vhpiCondExpr = 1311,
        vhpiConfigDecl = 1312,
        vhpiConfigSpec = 1313,
        vhpiConstraint = 1314,
        vhpiContributor = 1315,
        vhpiCurCallback = 1316,
        vhpiCurEqProcess = 1317,
        vhpiCurStackFrame = 1318,
        vhpiDerefObj = 1319,
        vhpiDecl = 1320,
        vhpiDesignUnit = 1321,
        vhpiDownStack = 1322,
        vhpiElemSubtype = 1323,
        vhpiEntityAspect = 1324,
        vhpiEntityDecl = 1325,
        vhpiEqProcessStmt = 1326,
        vhpiExpr = 1327,
        vhpiFormal = 1328,
        vhpiFuncDecl = 1329,
        vhpiGroupTempDecl = 1330,
        vhpiGuardExpr = 1331,
        vhpiGuardSig = 1332,
        vhpiImmRegion = 1333,
        vhpiInPort = 1334,
        vhpiInitExpr = 1335,
        vhpiIterScheme = 1336,
        vhpiLeftExpr = 1337,
        vhpiLexicalScope = 1338,
        vhpiLhsExpr = 1339,
        vhpiLocal = 1340,
        vhpiLogicalExpr = 1341,
        vhpiName = 1342,
        vhpiOperator = 1343,
        vhpiOthers = 1344,
        vhpiOutPort = 1345,
        vhpiParamDecl = 1346,
        vhpiParamExpr = 1347,
        vhpiParent = 1348,
        vhpiPhysLiteral = 1349,
        vhpiPrefix = 1350,
        vhpiPrimaryUnit = 1351,
        vhpiProtectedTypeBody = 1352,
```

```
        vhpiProtectedTypeDecl = 1353,
        vhpiRejectTime = 1354,
        vhpiReportExpr = 1355,
        vhpiResolFunc = 1356,
        vhpiReturnExpr = 1357,
        vhpiReturnTypeMark = 1358,
        vhpiRhsExpr = 1359,
        vhpiRightExpr = 1360,
        vhpiRootInst = 1361,
        vhpiSelectExpr = 1362,
        vhpiSeverityExpr = 1363,
        vhpiSimpleName = 1364,
        vhpiSubpBody = 1365,
        vhpiSubpDecl = 1366,
        vhpiSubtype = 1367,
        vhpiSuffix = 1368,
        vhpiTimeExpr = 1369,
        vhpiTimeOutExpr = 1370,
        vhpiTool = 1371,
        vhpiType = 1372
        vhpiTypeMark = 1373,
        vhpiUnitDecl = 1374,
        vhpiUpStack = 1375,
        vhpiUpperRegion = 1376,
        vhpiUse = 1377,
        vhpiValExpr = 1378,
        vhpiValSubtype = 1379

        INT_ONE_METHODS
        INT_AMS_ONE_METHODS

} vhpiOneToOneT;

/************** methods used to traverse 1 to many relationships
*******************/
typedef enum {
        vhpiAliasDecls = 1501,
        vhpiArgvs = 1502,
        vhpiAttrDecls = 1503,
        vhpiAttrSpecs = 1504,
        vhpiBasicSignals = 1505,
        vhpiBlockStmts = 1506,
        vhpiBranchs = 1507,
        vhpiCallbacks = 1508,
        vhpiChoices = 1509,
        vhpiCompInstStmts = 1510,
        vhpiCondExprs = 1511,
        vhpiCondWaveforms = 1512,
        vhpiConfigItems = 1513,
        vhpiConfigSpecs = 1514,
        vhpiConstDecls = 1515,
        vhpiConstraints = 1516,
        vhpiContributors = 1517,
        vhpiCurRegions = 1518, Issue I019
        vhpiDecls = 1519,
        vhpiDepUnits = 1520,
        vhpiDesignUnits = 1521,
        vhpiDrivenSigs = 1522,
        vhpiDrivers = 1523,
        vhpiElemAssocs = 1524,
```

```
        vhpiEntityClassEntrys = 1525,
        vhpiEntityDesignators = 1526,
        vhpiEnumLiterals = 1527,
        vhpiForeignfs = 1528,
        vhpiGenericAssocs = 1529,
        vhpiGenericDecls = 1530,
        vhpiIndexExprs = 1531,
        vhpiIndexedNames = 1532,
        vhpiInternalRegions = 1533,
        vhpiMembers = 1534,
        vhpiPackInsts = 1535,
        vhpiParamAssocs = 1536,
        vhpiParamDecls = 1537,
        vhpiPortAssocs = 1538,
        vhpiPortDecls = 1539,
        vhpiRecordElems = 1540,
        vhpiSelectWaveforms = 1541,
        vhpiSelectedNames = 1542,
        vhpiSensitivitys = 1543,
        vhpiSeqStmts = 1544,
        vhpiSigAttrs = 1545,
        vhpiSigDecls = 1546,
        vhpiSigNames = 1547,
        vhpiSignals = 1548,
        vhpiSpecNames = 1549,
        vhpiSpecs = 1550,
        vhpiStmts = 1551, /* vhpiTargets removed in 4.0 */
        vhpiTransactions = 1552,
        vhpiTypeMarks = 1553,
        vhpiUnitDecls = 1554,
        vhpiUses = 1555,
        vhpiVarDecls = 1556,
        vhpiWaveformElems = 1557

        INT_MANY_METHODS
        INT_AMS_MANY_METHODS

} vhpiOneToManyT;

/****************** PROPERTIES *******************/
/******* INTEGER or BOOLEAN PROPERTIES **********/
typedef enum {
        vhpiAccessP = 1001,
        vhpiArgcP = 1002,
        vhpiAttrKindP = 1003,
        vhpiBaseIndexP = 1004,
        vhpiBeginLineNoP = 1005,
        vhpiEndLineNoP = 1006,
        vhpiEntityClassP = 1007,
        vhpiForeignKindP = 1008,
        vhpiFrameLevelP = 1009,
        vhpiGenerateIndexP = 1010,
        vhpiIntValP = 1011,
        vhpiIsAnonymousP = 1012,
        vhpiIsBasicP = 1013,
        vhpiIsCompositeP = 1014,
        vhpiIsDefaultP = 1015,
        vhpiIsDeferredP = 1016,
        vhpiIsDiscreteP = 1017,
        vhpiIsForcedP = 1018,
```

```
        vhpiIsForeignP = 1019,
        vhpiIsGuardedP = 1020,
        vhpiIsImplicitDeclP = 1021,
        vhpiIsInvalidP = 1022,
        vhpiIsLocalP = 1023,
        vhpiIsNamedP = 1024,
        vhpiIsNullP = 1025,
        vhpiIsOpenP = 1026,
        vhpiIsPLIP = 1027,
        vhpiIsPassiveP = 1028,
        vhpiIsPostponedP = 1029,
        vhpiIsProtectedTypeP = 1030,
        vhpiIsPureP = 1031,
        vhpiIsResolvedP = 1032,
        vhpiIsScalarP = 1033,
        vhpiIsSeqStmtP = 1034,
        vhpiIsSharedP = 1035,
        vhpiIsTransportP = 1036,
        vhpiIsUnaffectedP = 1037,
        vhpiIsUnconstrainedP = 1038,
        vhpiIsUninstantiatedP = 1039,
        vhpiIsUpP = 1040,
        vhpiIsVitalP = 1041,
        vhpiIteratorTypeP = 1042,
        vhpiKindP = 1042,  -- change all numbers
        vhpiLeftBoundP = 1043,
        vhpiLevelP = 1044,
        vhpiLineNoP = 1045,
        vhpiLineOffsetP = 1046,
        vhpiLoopIndexP = 1047,
        vhpiModeP = 1048,
        vhpiNumDimensionsP = 1049,
        vhpiNumFieldsP = 1050,
        vhpiNumGensP = 1051,
        vhpiNumLiteralsP = 1052,
        vhpiNumMembersP = 1053,
        vhpiNumParamsP = 1054,
        vhpiNumPortsP = 1055,
        vhpiOpenModeP = 1056,
        vhpiPhaseP = 1057,
        vhpiPositionP = 1058,
        vhpiPredefAttrP = 1059,
        vhpiProtectedLevelP = 1060,
        vhpiReasonP = 1061,
        vhpiRightBoundP = 1062,
        vhpiSigKindP = 1063,
        vhpiSizeP = 1064,
        vhpiStartLineNoP = 1065,
        vhpiStateP = 1066,
        vhpiStaticnessP = 1067,
        vhpiVHDLversionP = 1068,
        vhpiIdP = 1069,

        /* MIXED_LANG_PROPERTY */
        vhpiLanguageP = 1200
        INT_INT_PROPERTIES
        INT_AMS_INT_PROPERTIES

} vhpiIntPropertyT;
```

```
/******* STRING PROPERTIES **********/
typedef enum {
        vhpiCaseNameP = 1301,
        vhpiCompNameP = 1302,
        vhpiDefNameP = 1303,
        vhpiFileNameP = 1304,
        vhpiFullCaseNameP = 1305,
        vhpiFullNameP = 1306,
        vhpiKindStrP = 1307,
        vhpiLabelNameP = 1308,
        vhpiLibLogicalNameP = 1309,
        vhpiLibPhysicalNameP = 1310,
        vhpiLogicalNameP = 1311,
        vhpiLoopLabelNameP = 1312,
        vhpiNameP = 1313,
        vhpiOpNameP = 1314,
        vhpiStrValP = 1315,
        vhpiToolVersionP = 1316,
        vhpiUnitNameP = 1317,
        vhpiSaveRestartLocationP = 1318,

        /* MIXED LANG PROPERTIES */
        vhpiFullVlogNameP = 1500,
        vhpiFullVHDLNameP = 1501,
        vhpiFullLSNameP = 1502,
        vhpiFullLSCaseNameP = 1503

        INT_STR_PROPERTIES
        INT_AMS_STR_PROPERTIES

} vhpiStrPropertyT;
/******* REAL PROPERTIES **********/
typedef enum {
        vhpiFloatLeftBoundP = 1601,
        vhpiFloatRightBoundP = 1602,
        vhpiRealValP = 1603

        INT_REAL_PROPERTIES
        INT_AMS_REAL_PROPERTIES

} vhpiRealPropertyT;

/******* PHYSICAL PROPERTIES **********/
typedef enum {
        vhpiPhysLeftBoundP = 1651,
        vhpiPhysPositionP = 1652,
        vhpiPhysRightBoundP = 1653,
        vhpiPhysValP = 1654,
        vhpiPrecisionP = 1655,
        vhpiSimTimeUnitP = 1656

        INT_PHYS_PROPERTIES
        INT_AMS_PHYS_PROPERTIES

} vhpiPhysPropertyT;

/****************** PROPERTY VALUES ************************/

/* vhpiOpenModeP */
#define vhpiInOpen                1001
```

```
#define vhpiOutOpen                 1002
#define vhpiReadOpen                1003
#define vhpiWriteOpen               1004
#define vhpiAppendOpen              1005

/* vhpiModeP */
#define vhpiInMode                  1001
#define vhpiOutMode                 1002
#define vhpiInoutMode               1003
#define vhpiBufferMode              1004
#define vhpiLinkageMode             1005

/* vhpiSigKindP */
#define vhpiRegister                1001
#define vhpiBus                     1002
#define vhpiNormal                  1003

/* vhpiStaticnessP */
#define vhpiLocallyStatic           1001
#define vhpiGloballyStatic          1002
#define vhpiDynamic                 1003

/* vhpiPredefAttrP */
#define vhpiActivePA                1001
#define vhpiAscendingPA             1002
#define vhpiBasePA                  1003
#define vhpiDelayedPA               1004
#define vhpiDrivingPA               1005
#define vhpiDriving_valuePA         1006
#define vhpiEventPA                 1007
#define vhpiHighPA                  1008
#define vhpiImagePA                 1009
#define vhpiInstance_namePA         1010
#define vhpiLast_activePA           1011
#define vhpiLast_eventPA            1012
#define vhpiLast_valuePA            1013
#define vhpiLeftPA                  1014
#define vhpiLeftofPA                1015
#define vhpiLengthPA                1016
#define vhpiLowPA                   1017
#define vhpiPath_namePA             1018
#define vhpiPosPA                   1019
#define vhpiPredPA                  1020
#define vhpiQuietPA                 1021
#define vhpiRangePA                 1022
#define vhpiReverse_rangePA          1023
#define vhpiRightPA                 1024
#define vhpiRightofPA               1025
#define vhpiSimple_namePA            1026
#define vhpiStablePA                1027
#define vhpiSuccPA                  1028
#define vhpiTransactionPA           1029
#define vhpiValPA                   1030
#define vhpiValuePA                 1031

/* vhpiAttrKindP */
typedef enum {
  vhpiFunctionAK    = 1,
  vhpiRangeAK       = 2,
  vhpiSignalAK      = 3,
```

```
  vhpiTypeAK        = 4,
  vhpiValueAK       = 5
  INT_ATTR
  INT_AMS_ATTR
} vhpiAttrKindT;

/* vhpiEntityClassP */
#define vhpiEntityEC              1001
#define vhpiArchitectureEC        1002
#define vhpiConfigurationEC       1003
#define vhpiProcedureEC           1004
#define vhpiFunctionEC            1005
#define vhpiPackageEC             1006
#define vhpiTypeEC                1007
#define vhpiSubtypeEC             1008
#define vhpiConstantEC            1009
#define vhpiSignalEC              1010
#define vhpiVariableEC            1011
#define vhpiComponentEC           1012
#define vhpiLabelEC               1013
#define vhpiLiteralEC             1014
#define vhpiUnitsEC               1015
#define vhpiFileEC                1016
#define vhpiGroupEC               1017

/* vhpiAccessP */
#define vhpiRead                  1
#define vhpiWrite                 2
#define vhpiConnectivity          4
#define vhpiNoAccess              8

/* value for vhpiStateP property for callbacks */
typedef enum {
        vhpiEnable,
        vhpiDisable,
        vhpiMature /* callback has occured */
} vhpiStateT;

/* MIXED LANGUAGE PROPERTY VALUES */
/* vhpiLanguageP */
#define vhpiVHDL                  1001
#define vhpiVerilog               1002

/* the following enumeration types are used only for vhpiSimTimeUnitP
and vhpiPrecisionP property and for setting the unit field of the value
structure; they represent the physical position of a given
VHDL time unit */
/* time unit physical position values {high, low} */
PLI_VEXTERN PLI_DLLISPEC const vhpiPhysT  vhpiFS;
PLI_VEXTERN PLI_DLLISPEC const vhpiPhysT  vhpiPS;
PLI_VEXTERN PLI_DLLISPEC const vhpiPhysT  vhpiNS;
PLI_VEXTERN PLI_DLLISPEC const vhpiPhysT  vhpiUS;
PLI_VEXTERN PLI_DLLISPEC const vhpiPhysT  vhpiMS;
PLI_VEXTERN PLI_DLLISPEC const vhpiPhysT  vhpiS;
PLI_VEXTERN PLI_DLLISPEC const vhpiPhysT  vhpiMN;
PLI_VEXTERN PLI_DLLISPEC const vhpiPhysT  vhpiHR;

/********************* delay structures *************************/
/* removed ; postponed to next version of the standard */
```

```
/* IEEE std_logic values */
#define vhpiU                  0   /* uninitialized */
#define vhpiX                  1   /* unknown */
#define vhpi0                  2   /* forcing 0 */
#define vhpi1                  3   /* forcing 1 */
#define vhpiZ                  4   /* high impedance */
#define vhpiW                  5   /* weak unknown */
#define vhpiL                  6   /* weak 0 */
#define vhpiH                  7   /* weak 1 */
#define vhpiDontCare           8   /* don't care */

/* IEEE std bit values */
#define vhpibit0                0   /* bit 0 */
#define vhpibit1                1   /* bit 1 */

/* IEEE std boolean values */
#define vhpiFalse              0   /* false */
#define vhpiTrue               1   /* true */

/************** vhpiPhaseP property values *************/
typedef enum {
       vhpiRegistrationPhase   = 1,
       vhpiAnalysisPhase       = 2,
       vhpiElaborationPhase    = 3,
       vhpiInitializationPhase = 4,
       vhpiSimulationPhase     = 5,
       vhpiTerminationPhase    = 6,
       vhpiSavePhase           = 7,
       vhpiRestartPhase        = 8,
       vhpiResetPhase          = 9
} vhpiPhaseT ;
/**************** PLI error information structure ****************/

typedef enum {
        vhpiNote        = 1, /* same as vpiNotice */
        vhpiWarning     = 2, /* same as vpiWarning */
        vhpiError       = 3, /* same as vpiError */
        vhpiFailure     = 6, /* keep it like that for interoperability
with VPI */
        vhpiSystem      = 4, /* same as vpiSystem */
        vhpiInternal    = 5 /* same as vpiInternal */
} vhpiSeverityT;

typedef struct vhpiErrorInfoS
{
  vhpiSeverityT    severity;
  PLI_BYTE8       *message;
  PLI_BYTE8       *str;
  PLI_BYTE8       *file; /* Name of the VHDL file where the VHPI error
                            originated */
  PLI_INT32        line; /* Line number in the VHDL file */
} vhpiErrorInfoT;

/******************** callback structures ***********************/
/* callback user data structure */

typedef struct vhpiCbDataS
{
  PLI_INT32 reason;             /* callback reason */
  PLI_VOID (*cb_rtn) (const struct vhpiCbDataS *);  /* call routine */
```

```
  vhpiHandleT obj;             /* trigger object */
  vhpiTimeT *time;             /* callback time */
  vhpiValueT *value;           /* trigger object value */
  PLI_VOID *user_data;         /* pointer to user data to be passed to
the
                                  callback function */
} vhpiCbDataT;

/**************************** CALLBACK REASONS
****************************/
/*************************** Simulation object related
***********************/
/* These are repetitive callbacks */
#define vhpiCbValueChange          1001
#define vhpiCbForce                1002
#define vhpiCbRelease              1003
#define vhpiCbTransaction          1004 /* optional callback reason */

/****************************** Statement related
****************************/
/* These are repetitive callbacks */
#define vhpiCbStmt                 1005
#define vhpiCbResume               1006  /* ganges */
#define vhpiCbSuspend              1007  /* ganges */
/* issue: one time or repetitive callbacks */
#define vhpiCbStartOfSubpCall      1008  /* ganges */
#define vhpiCbEndOfSubpCall        1009  /* ganges */

/****************************** Time related
******************************/
/* the Rep callback reasons are the repeated versions of the callbacks
*/

#define vhpiCbAfterDelay           1010
#define vhpiCbRepAfterDelay        1011
/*************************** Simulation cycle phase  related
*****************/
#define vhpiCbNextTimeStep         1012
#define vhpiCbRepNextTimeStep      1013
#define vhpiCbStartOfNextCycle     1014
#define vhpiCbRepStartOfNextCycle  1015
#define vhpiCbStartOfProcesses     1016 /* new in charles */
#define vhpiCbRepStartOfProcesses  1017 /* new in ldv 3.3 */
#define vhpiCbEndOfProcesses       1018 /* ganges */
#define vhpiCbRepEndOfProcesses    1019 /* new ldv 3.3 */
#define vhpiCbLastKnownDeltaCycle  1020 /* new in ldv 3.3 */
#define vhpiCbRepLastKnownDeltaCycle 1021 /* new in ldv 3.3 */
#define vhpiCbStartOfPostponed     1022 /* ganges */
#define vhpiCbRepStartOfPostponed  1023 /* new in ldv 3.3 */
#define vhpiCbEndOfTimeStep        1024 /* ganges */
#define vhpiCbRepEndOfTimeStep     1025 /* new in ldv 3.3 */

/***************************** Action related
*****************************/
/* these are one time callback unless otherwise noted */
#define vhpiCbStartOfTool            1026 /* new in charles */
#define vhpiCbEndOfTool              1027 /* new in charles */
#define vhpiCbStartOfAnalysis        1028 /* new in charles */
#define vhpiCbEndOfAnalysis          1029 /* new in charles */
```

```
#define vhpiCbStartOfElaboration   1030 /* new in charles */
#define vhpiCbEndOfElaboration     1031 /* name was cbkEndOfElaboration
in tiber */
#define vhpiCbStartOfInitialization 1032 /* Name change in charles */
#define vhpiCbEndOfInitialization  1033 /* new in ldv 3.3 */
#define vhpiCbStartOfSimulation    1034
#define vhpiCbEndOfSimulation      1035
#define vhpiCbQuiescense           1036 /* repetitive */
#define vhpiCbPLIError             1037 /* repetitive */
#define vhpiCbStartOfSave          1038
#define vhpiCbkEndOfSave           1039
#define vhpiCbStartOfRestart       1040
#define vhpiCbEndOfRestart         1041
#define vhpiCbStartOfReset         1042
#define vhpiCbEndOfReset           1043
#define vhpiCbEnterInteractive     1044 /* repetitive */
#define vhpiCbExitInteractive      1045 /* repetitive */
#define vhpiCbSigInterrupt         1046 /* repetitive */

/* Foreign model callbacks */
#define vhpiCbTimeOut              1047 /* non repetitive */
#define vhpiCbRepTimeOut           1048 /* repetitive */
#define vhpiCbSensitivity          1049 /* repetitive */

/*************************** CALLBACK FLAGS
******************************/
#define vhpiReturnCb  0x00000001
#define vhpiDisableCb 0x00000010

/************** vhpiAutomaticRestoreP property values *************/
typedef enum {
       vhpiRestoreAll   = 1,
       vhpiRestoreUserData       = 2,
       vhpiRestoreHandles    = 4,
       vhpiRestoreCallbacks = 8,
} vhpiAutomaticRestoreT ;


/******************** FUNCTION DECLARATIONS *********************/
XXTERN PLI_INT32 vhpi_assert PROTO_PARAMS((vhpiSeverityT severity,
PLI_BYTE8 *formatmsg,…));
/* callback related */

XXTERN vhpiHandleT  vhpi_register_cb     PROTO_PARAMS((vhpiCbDataT
*cb_data_p, PLI_UINT32 flags));
XXTERN PLI_INT32          vhpi_remove_cb       PROTO_PARAMS((vhpiHandleT
cb_obj));
XXTERN PLI_INT32          vhpi_disable_cb      PROTO_PARAMS((vhpiHandleT
cb_obj));
XXTERN PLI_INT32          vhpi_enable_cb       PROTO_PARAMS((vhpiHandleT
cb_obj));
XXTERN PLI_INT32          vhpi_get_cb_info     PROTO_PARAMS((vhpiHandleT
object, vhpiCbDataT *cb_data_p));

/* for obtaining handles */
XXTERN vhpiHandleT   vhpi_handle_by_name  PROTO_PARAMS((const PLI_BYTE8
*name, vhpiHandleT scope));

XXTERN vhpiHandleT   vhpi_handle_by_index PROTO_PARAMS((vhpiOneToManyT
itRel, vhpiHandleT parent, PLI_INT32 indx));
```

```
/* for traversing relationships */
XXTERN vhpiHandleT   vhpi_handle          PROTO_PARAMS((vhpiOneToOneT
type, vhpiHandleT referenceHandle));
XXTERN vhpiHandleT   vhpi_iterator         PROTO_PARAMS((vhpiOneToManyT
type, vhpiHandleT referenceHandle));
XXTERN vhpiHandleT   vhpi_scan            PROTO_PARAMS((vhpiHandleT
iterator));

/* for processsing properties */
XXTERN
PLI_INT32          vhpi_get             PROTO_PARAMS((vhpiIntPropertyT
property, vhpiHandleT object));
XXTERN const PLI_BYTE8
*       vhpi_get_str         PROTO_PARAMS((vhpiStrPropertyT property,
vhpiHandleT object));
XXTERN
vhpiRealT       vhpi_get_real        PROTO_PARAMS((vhpiRealPropertyT
property, vhpiHandleT object));
/* vhpi_get_phys new in charles */
XXTERN
vhpiPhysT       vhpi_get_phys        PROTO_PARAMS((vhpiPhysPropertyT
property, vhpiHandleT object));

/* for access to protected types: new in ganges  */
typedef int (*vhpiUserFctT)();
XXTERN PLI_INT32 vhpi_protected_call  PROTO_PARAMS((vhpiHandleT varHdl,
vhpiUserFctT userFct, PLI_VOID *userData));

/* value processing */
XXTERN PLI_INT32          vhpi_get_value       PROTO_PARAMS((vhpiHandleT
expr, vhpiValueT *value_p));
XXTERN PLI_INT32  vhpi_put_value       PROTO_PARAMS((vhpiHandleT object,
vhpiValueT *value_p, PLI_UINT32 flags));

/* vhpi_put_value flags */
typedef enum {
  vhpiDeposit,
  vhpiDepositPropagate,
  vhpiForce,
  vhpiForcePropagate,
  vhpiRelease,
  vhpiSizeConstraint
} vhpiPutValueModeT;

XXTERN PLI_INT32  vhpi_schedule_transaction PROTO_PARAMS((vhpiHandleT
drivHdl, vhpiValueT *value_p, PLI_INT32 numValues, vhpiTimeT
*delayp,PLI_UINT32 delayMode, vhpiTimeT * pulseRejp));
typedef enum {
  vhpiInertial,
  vhpiTransport
} vhpidelayModeT;

XXTERN PLI_INT32          vhpi_format_value     PROTO_PARAMS((const
vhpiValueT *in_value_p, vhpiValueT *out_value_p));

/* time processing */
/* the current simulation time is retrieved */
```

```
XXTERN
void         vhpi_get_time        PROTO_PARAMS((vhpiTimeT  *time_p, long
*cycles));
/* The next active time */
#define vhpiNoActivity -1
XXTERN
PLI_INT32         vhpi_get_next_time        PROTO_PARAMS((vhpiTimeT  *ti
me_p));

/* simulation control */
typedef enum {
  vhpiStop,
  vhpiFinish,
  vhpiReset
} vhpiSimControlT;

XXTERN PLI_INT32   vhpi_sim_control PROTO_PARAMS((vhpiSimControlT
command,
                                                 ...));
/* I/O routine */
XXTERN PLI_INT32          vhpi_printf          PROTO_PARAMS((const
PLI_BYTE8 *format,...));

/* utilities to print VHDL strings */

static PLI_INT32 vhpi_is_printable( PLI_BYTE8 ch )
{
unsigned char uch = (unsigned char)ch;

    if (uch < 31) return 0;
      if (uch < 127) return 1;
      if (uch == 127) return 0;
      if (uch < 160) return 0;
      return 1;
}


static const PLI_UBYTE8* VHPICharCodes[256]={
"NUL",   "SOH", "STX", "ETX", "EOT", "ENQ","ACK",  "BEL" ,
"BS",    "HT",  "LF",  "VT",  "FF", "CR", "SO" , "SI",
"DLE",  "DC1", "DC2", "DC3", "DC4","NAK", "SYN" , "ETB",
 "CAN",   "EM", "SUB", "ESC", "FSP", "GSP", "RSP" , "USP",
 " ","!","\"","#","$","%","&","",
"(",")","*","+",",","-",".","/",
"0","1","2","3","4","5","6","7",
"8","9",":",";","<","=",">","?",
"@","A","B","C","D","E","F","G",
"H","I","J","K","L","M","N","O",
"P","Q","R","S","T","U","V","W",
"X","Y","Z","[","\\","]","^","_",
"`","a","b","c","d","e","f","g",
"h","i","j","k","l","m","n","o",
"p","q","r","s","t","u","v","w",
"x","y","z","{","|","}","~","DEL",
"C128", "C129", "C130","C131","C132","C133","C134","C135",
"C136", "C137", "C138","C139","C140","C141","C142","C143",
"C144", "C145", "C146","C147","C148","C149","C150","C151",
"C152", "C153", "C154","C155","C156","C157","C158","C159",
" ","¡","¢","£","¤","¥","¦","§",
"¨","©","ª","«","¬","-","®","¯",
```

```
"°","±","²","³","´","µ","¶","·",
"¸","¹","º","»","¼","½","¾","¿",
"À","Á","Â","Ã","Ä","Å","Æ","Ç",
"È","É","Ê","Ë","Ì","Í","Î","Ï",
"Ð","Ñ","Ò","Ó","Ô","Õ","Ö","×",
"Ø","Ù","Ú","Û","Ü","Ý","Þ","ß",
"à","á","â","ã","ä","å","æ","ç",
"è","é","ê","ë","ì","í","î","ï",
"ð","ñ","ò","ó","ô","õ","ö","÷",
"ø","ù","ú","û","ü","ý","þ","ÿ" };

#define VHPI_GET_PRINTABLE_STRINGCODE( ch )  VHPICharCodes[PLI_UBYTE8 ch]


/* utility routines */
XXTERN PLI_INT32          vhpi_compare_handles PROTO_PARAMS((vhpiHandleT
handle1, vhpiHandleT handle2));
XXTERN
PLI_INT32          vhpi_check_error       PROTO_PARAMS((vhpiErrorInfoT
*error_info_p));
/* name change was vhpi_free_handle in charles */
XXTERN
PLI_INT32          vhpi_release_handle     PROTO_PARAMS((vhpiHandleT
object));

/* creation functions */
XXTERN vhpiHandleT vhpi_create PROTO_PARAMS((vhpiClassKindT kind,
vhpiHandleT handle1, vhpiHandleT handle2));

/* Foreign model data structures and functions */
typedef enum {
   vhpiArchF,
   vhpiFuncF,
   vhpiProcF,
   vhpiLibF,
   vhpiAppF
} vhpiForeignT;

typedef struct vhpiForeignDataS {
    vhpiForeignT kind;
    PLI_BYTE8 * libraryName;
    PLI_BYTE8 * modelName;
    PLI_VOID (*elabf)(const struct vhpiCbDataS *cb_data_p);
    PLI_VOID (*execf)(const struct vhpiCbDataS *cb_data_p);
} vhpiForeignDataT;

XXTERN vhpiHandleT vhpi_register_foreignf PROTO_PARAMS((vhpiForeignDataT
*foreignDatap));
XXTERN int vhpi_get_foreign_info PROTO_PARAMS((vhpiHandleT hdl,
vhpiForeignDataT *foreignDatap));
/* for saving and restoring foreign models data */
XXTERN PLI_INT32 vhpi_get_data PROTO_PARAMS((PLI_INT32 id, PLI_VOID *
dataLoc, PLI_INT32 numBytes));
XXTERN PLI_INT32 vhpi_put_data PROTO_PARAMS((PLI_INT32 id, PLI_VOID *
dataLoc, PLI_INT32 numBytes));

/* internal function prototypes from vhpi_internal.h */
INT_PROTOTYPES
```

```
/**************************** GLOBAL VARIABLES
****************************/
typedef PLI_VOID (*vhpiBootstrapFctT)();

extern PLI_DLLESPEC vhpiBootstrapFctT *vhpiBootstrapArray[]; /* array of
function pointers, */
                                                       /* last pointer
should be null */

#undef PLI_EXTERN
#undef PLI_VEXTERN

#ifdef VHPI_USER_DEFINED_DLLISPEC
#undef VHPI_USER_DEFINED_DLLISPEC
#undef PLI_DLLISPEC
#endif
#ifdef VHPI_USER_DEFINED_DLLESPEC
#undef VHPI_USER_DEFINED_DLLESPEC
#undef PLI_DLLESPEC
#endif

#ifdef PLI_PROTOTYPES
#undef PLI_PROTOTYPES
#undef PROTO_PARAMS
#undef XXTERN
#undef EETERN
#endif

#ifdef  __cplusplus
}
#endif

#endif /* VHPI_USER_H */
```

14. ANNEX B: Description of properties

14.1 Integer properties

vhpiLineNoP:
returns the line number in the source file of the item designated by the handle.
For a vhpiRootInstK: the line number of the architecture identifier.
For a vhpiPackInst, the line number of the package identifier of the package body.
For all the other region derived classes, the line number of the first word which starts the region instance.
For the decl class, the line number of the declared identifier

vhpiLineOffsetP:
returns the character offset from the beginning of the line of the designated item
Note: Not all classes that have the line number property (vhpiLineNoP) have the line offset property

vhpiStartLineNoP: Applies only to the design unit class, returns the line number in the source where the design unit starts (includes the line number of the library and use clauses)

vhpiEndLineNoP: returns the line number of the end keyword in the VHDL text source.

vhpiBeginLineNoP: returns the line number of the begin keyword in the VHDL text source.

14.2 String properties

vhpiNameP:
This property returns the name of the designated object in unspecified case for basic identifiers (VHDL is case insensitive) or case preserved for extended identifiers:

for a declared item, the declared identifier,

for a component instance statement, block stmt: the label instance name,

for a generate statement, the <label_name>(generate index)

for a process stmt, the label of the process if specified or a created process label name (see rule below),

for a VHDL name (see name class diagram) the VHDL string name as it appears in the VHDL text source with or without case preserved depending if there is reference to an extended identifier within the name.

Example: selected name: "f.a", indexed name: "r(j)"

Naming of unlabelled equivalent processes:
the VHPI would generate an equivalent process label name which starts by the concatenation of the "_P" or "_p" string and an integer which denotes the sequence appearance number of the equivalent process in the VHDL source text of the declared region. The numbering starts at 0 and increments by 1. For example the auto-generated vhpiNameP of the first equivalent process statement in an entity declaration would be "_p0". Numbering of equivalent processes in the architecture follows the numbering sequence used for the entity. The number used for the first process in the architecture will be either 0 if the entity did not contain any unlabelled processes or n+1 where n is the number used for naming the last unlabelled equivalent process of the entity. Numbering of processes is reset for each internal region (block, generate or component statement).

example: "_2" is the vhpiNameP of the second occurring process for this entity/architecture pair.
note: _2 is not a legal VHDL identifier (should be escaped) this ensures that this identifier is not used in the rest of the design.

vhpiFullNameP:
The vhpiFullNameP property should return the concatenation of the vhpiNameP strings of each instance or item found on the hierarchy path.
The character : is used between 2 successive names returned by vhpiNameP.
The vhpiFullNameP property provides a non-ambiguous, simplest and easiest way to name a "named entity" (in the VHDL sense) which belongs to a VHDL design.
The vhpiFullNameP property returns a string that is different from the strings returned by the standard attributes 'PATH_NAME and 'INSTANCE_NAME.

The vhpiFullNameP of the root instance is ":". This is sufficient to refer to a unique root, VHDL 93 and 98 only allow one top level design unit. If multiple top level units were going to be defined in future versions of the language, we could allow the hierarchical name to specify which tree of the design hierarchy the name has to be searched from.

example of full names when multiple roots are allowed
@<entity_name>(<arch_name>):<instance_name>...
Notes:
1) both the entity and architecture names are necessary to identify the root.
2) entity_name(arch_name) is optional if the design has only one top design unit but required if multiple roots.

Construction of full names for items declared in packages is defined as follow:
@<lib_logical_name>:<pack_name>:<declared_item_name>

Since VHDL is case insensitive, the case of the vhpiFullNameP string is not specified unless there is an extended identifier.
Note: vhpiFullCaseName should be used to retrieve the hierarchical name with case preserved characters for the declared items.

vhpiCaseNameP:
This property returns the case preserved string of the item declaration. The string returned will reflect lower or upper case characters used in the identifier declaration. Note that for extended identifiers, or unlabelled equivalent processes, the vhpiCaseNameP string will be exactly the same as the vhpiNameP string.

vhpiFullCaseNameP:
The string returned is formed by the concatenation of each single vhpiCaseNameP string on the hierarchical path to the designated object. The ':'character is the delimitor between each simple case name.

Note: all these properties vhpiNameP, vhpiCaseNameP, vhpiFullNameP and vhpiFullCaseNameP apply to the name class (see primary expression diagram).

vhpiDefNameP:
The returned string name identifies the path to the declared thing in the library.
This property only applies to the region class and returns in unspecified case unless it is an extended identifier a string which identifies the library path name of the declared item which is bound to the designated region.

for block, generate or component instance statement
<lib_logical_name>:<entity_name>(<arch_name>)
for a package instance:
<lib_logical_name>:<pack_name>
for a subprogram call depending where the subprogram definition is
<lib_logical_name>:<entity_name>(<arch_name>):<subprogram_name>
or
<lib_logical_name>:<pack_name>:<subprogram_name>

for a process stmt declared in an entity:
<lib_logical_name>:<entity_name>:<process_label_name>
for a process declared in an architecture:
<lib_logical_name>:<entity_name>(arch_name):<process_label_name>

Note: If the process is unlabelled, a process_label_name is constructed according to the rule mentioned in the vhpiNameP description.

vhpiUnitName:
The name of the declared design unit in the VHDL source. This property is ONLY applicable to the designUnit class.
The name is returned in unspecified case for basic identifiers or case-preserved for extended identifiers.
The unitName of a design unit of the following class is:
EntityDecl: lib_name.entity_name
Arch body: lib_name.entity_name:arch_name
PackDecl: lib_name.pack_name
Pack Body: lib_name.pack_name:BODY
note: all variations of upper and lower case letters for BODY are allowed.
Config: lib_name.config_name

vhpiFileNameP
Physical file system path name of the VHDL source file where the item designated by the handle appears. Property is applicable for every VHPI class kind that has a vhpiLineNoP (line number property). Among these are declared items, design units for example.

14.3 Real properties

15. Annex : issues and resolutions

15.1 Creation of signal attributes with vhpi_create

Resolution: Cannot create signal attributes with vhpi_create

15.2 What does a foreign function is allowed to do (callback, vhpi_schedule_transaction …)

Resolution: No restriction on which callbacks can a foreign subprogram register.

15.3 Modeling for wait in subprograms

1) Open issue on restriction of foreign procedures to be non blocking, with dependency on elaboration of declarative parts of subprograms. There is a problem with foreign subprograms registering callbacks. Issue to be resolved in conjunction with elaboration of foreign subprograms declaration.part
Action: John: document analysis of problems
Francoise: Investigate what current interfaces allow :OMI, NCSIM-CIF, FLI/ do we know of any models that needs this functionality, what are the current limitations.

   Resolution: None of the current VHDL procedural interfaces support this functionality; nobody has requested it. Seems a high cost for specification and implementation.

15.4 Default process implied by a foreign architecture

Resolution: no default implicit process.

15.5 cancelling transaction and returning handle to a transaction bundle.

Resolution: removed from the standard, no use

15.6 Can vhpi_put_value be called during initialization cycle?

15.7 Are vhpiStartOfSubpCall and vhpiEndOfSubpCall repetitive callbacks

Resolved 1/18/01 YES repetitive and for a specific instance of the subprogram call

15.8 Are save/restart and reset callbacks repetitive?

Resolution: Save are repetitive until the end of simulation or a reset or restart command. Restart are not. Reset are repetitive. Contradiction with the fact we say that at the reset all callbacks are removed.

15.9 Representation of real physical literals 2.5 ns

Resolved: physical literal diagram

15.10 VhpiLogicVal and vhpiLogicVecval standard formats for logic types

Resolved: added to the standard

15.11 When a signal or a port is forced, what should vhpiContributors and vhpiDrivers return?

Resolved: should still return all the contributors and drivers

15.12 Restart sequence

Do we need to call again all the bootstrap functions? Including the application bootstrap? (probably not since they should have register a restart callback which will be called at restart. Unless requested explicitly at the tool invocation, the bootstrap functions or dynamically loaded library will be processed.

Resolution: application should be saved/restart aware (no need to bootstrap them again). Restart sequence has been approved

15.13 Reset sequence

Which callbacks are removed?

15.14 CbAfterDelay callback

callback cbAfterDelay what is the behaviour is delay is 0 ?

15.15 CbStartOfPostponed callback

callback cbStartOfPostponed is a phase callback and should occur whether or not there is any postponed process waking up?

15.16 vhpiDecl inheritance class

Is missing vhpiDesignUnit. vhpiUses returns declarations refered by the use clause.

15.17 Can vhpi_put_value *be called during initialization cycle?*

Vish: Yes. We in fact use it during initialization. There is a problem with this though. We could discuss this at our meeting. The problem is that the deposit should take effect immediately, as there is no concept of the next delta cycle. What we do is that we do signal propagation only for all such updates. This is a very useful feature, that we use with a flag that we call vhpiForceImmediate (which I think should really be vhpiDepositImmediate, that does just the propagation without queuing processes for execution)

15.18 Access to the component declaration from a component instance statement

Currently only the component declaration name is available for a component instance statement. It is useful to be able to get to the real component declaration as this component declaration contain local ports and generics with modes and intiial expressions… A lint HDL tool may want to process some checks on the component declaration. This component declaration may be declared in a package or in the architecture which contains the component instance statement, this makes it difficult for an application to find the matching the component declaration which was used for the component instance statement. I propose we add a VHPI one to one method to return the component declaration.

15.19 Access to the subprogram body from a subprogram declaration

Currently the VHPI information model allows to go from a subpCall to its subpBody and from the subpbody to its supDecl; this is not sufficient. It is not possible to go from a subpDecl to its subpBody. Francoise to check the LRM to see if multiple subpBody can be associated with one subpDecl.

15.20 When can you apply vhpi_schedule transaction

Issue: We need to define when in the simulation cycle the cbAfterDelay and cbNextTimeStep callback occurs: if these calback occur after transaction have been processed, when will the vhpi_schedule_transaction take effect. Should vhpi_schedule transaction only allowed during process execution? Any other phase would be undefined behaviour
Discussion:
cancelling transaction does not seem to be useful, it will be removed from the standard. Same effect can be obtained by calling vhpi_schedule_transaction (and doing transaction preemption).

Vish pointed out that allowing 0 delay transaction at NextTimeStep or CbAfterDelay callbacks may be non portable across simulators. Some simulators may schedule the transaction in the current delta cycle, some may create a new delta cycle.
Resolutrion: We decided to only allow to schedule a 0 delay transaction during process executation and cbLastKnownDeltaCycle, which will cause a new delta cycle to be created if the transaction generates an event. non-zero delay transactions can be scheduled at any time before cbEndOfTimeStep, effectively before the next time is computed.

15.21 Collection of drivers

Can you create collection of drivers for drivers of same signal , different processes

15.22 What happen to Mature callbacks

Mature callbacks should be handled consistent with the principlesof resource ownership for VHPI clients.
When a callback matures, there is no value to it except for query. It cannot be re-enabled, and it cannot be discovered via traversal of the information model. It should be deleted by the VHPI server,unless the client(user) has previously obtained a handle to the transaction. If the client has a handle, he has ownership, albeit to something of marginal value. He can query it or just waste the memory resource. It follows that, after all such handles are released, the mature transaction should be deleted. The VHPI server is free to waste resources itself, but the point is, it has ownership of the transaction.

15.23 Uninstantiated access: expanded names;

The following statements were not approved by the committee.

The uninstantiated model should store an additional property, *IsExpanded*, on the *SelectedName* class indicating whether it is an expanded name in the source file.

ISSUE: How to support this? FM is of the opinion that we should have a different class (OutOfModuleRef) with Name and LineNo and a method vhpi_oomr_decl() that will get you to the object's declaration by crossing the design unit. JB: It needs to be a derived object off name since any object, operator, procedure name, type name, etc. can be an out-of-module reference.

After discussion, the resolution was to treat expanded names to declarations in other design unit the same way as names declared in the current design unit. This has the drawback to not being able for decompilation applications to exactly produce the original source. This is more efficient for synthesis oriented applications and more inline with the information retained by analyzers.

15.24 vhpi_handle_by_name returning collections

Description:A conceivable extension of handle_by_name is the support of regular expressions returning collections of handles.
Rationale: This is a powerful convenience function that can be built from current VHPI capabilities. It sets the requirements for compliance too high for the first version of the spec.
Resolution: This is out of the scope of VHPI.

15.25 Associating Errors with VHPI Clients

Description: There are methods of handling errors that occur during the use of VHPI, but there are situations where VHPI cannot determine which of multiple VHPI client applications or models caused a particular error. If one registers a callback on error, for example, VHPI will trigger it when an error occurs, regardless of what caused it. A desirable improvement is to call only the client that both caused the error and registered for such a callback. The problem is, there is no method to associate errors with a particular client, nor does VHPI maintain an association between client and its callbacks that would support this type of improvement.

There is a related problem, a corrolary that says if you obtain a handle to a callback by navigating the information model, there is no straightforward way to examine it to see if it was your application that registered it. Both these problems are barriers to friendly VHPI applications that can peacefully coexist with each other.

Solution proposed:
One proposal to resolve it is to provide a mechanism that defines a unique client identity that can be associated with callbacks and, in general, with executing VHPI code. Since callbacks are the principle means by which a VHPI client's code gains control, VHPI has a means to track which client appears to be executing when it encounters an error. This is not a solid proposal yet, and questions remain whether this mechanism will fail to have a correct association in some important cases.

The basic idea is to have a vhpi_client_registration function that takes a string argument representing the name of the client and return a unique integer id each time it is called. This id is then provided with any callback registration made by the client. First, given a callback handle, the client can get the callback info and recognize its own unique id. Secondly, when VHPI dispatches a callback, it can "know" which client is running. Using that information, it can provide it as another part of the error structure when a errors are checked, it can choose to only call error callbacks that have been registered by that client, etc. We can certainly allow the notion of an anonymous client with a well known id (say "0"), and the unknown client whose id does not have a name associated with it. We can even allow an error callback to register for "any"
client in a meaningful way.

Something to note is that it is not mandatory that a client have unique client id and this is not a means of securely isolating one application from another. It is meant as a practical way of writing more friendly VHPI applications. In order to enforce the use of a unique client id, one would have to require it to call any API function except the registration call itself. Worse, it may difficult in some simulator architectures for VHPI to know which client's code is executing in some cases. It is undesirable to require that VHPI clearly know which client caused a particular error. It is only generally required to know that the last vhpi

function call was responsible at this time. If there is a circumstance where the error is not detectable until after possibly many VHPI calls have occured, the simulator may only know that some prior VHPI action led to the error condition.
Such cases are not intended to be covered by this proposal.

Resolution: At a minimum, the difficulty with using the error callback mechanism and possibly identifying your own callbacks must be stated in the spec. There is no desire to make a complete VHPI client server model or require client ids as arguments to each vhpi function.

15.26 vhpiFullName same as 'path_name predefined attribute string?

Issue: The vhpiFullNameP property is not returning the same string as either 'path_name or 'instance_name

The vhpiFullNameP property is intended to be an improvement on `path_name attribute of the language. It is meant to be minimal in string length. The idea behind minimizing the string length includes conserving real estate in user interfaces, printed reports. Choices like using `path_name vs. `instance_name as a starting point and eliminating the rootInst entity name derive from this goal.

It is reasonable to expect that a user will obtain names using both VHPI properties and the predefined VHDL attributes and provide them as input to VHPI-based applications or functions. With the vhpiFullNameP property, well-defined standard behavior can be expected of vhpi_handle_by_name. With the predefined attributes, under most conditions, a well-defined behavior will also occur **if** vhpiFullNameP is defined to be as consistent as possible to `path_name.
Not Handling Redundant Information In Lookups

VHPI could support another name property that is analogous to X`INSTANCE_NAME. The additional information (i.e., the @e(a)) is redundant information for the search and increases its cost. We propose that the information be accepted and verified in the search algorithm.
It is forward looking to consider this in the future, as VHDL requires that the instantiation label be unique within a scope but not all HDLs do.

Resolution: vhpiFullNameP property will have a different string than the predefined attributes to resolve ambiguities and to minimize the string length to be used for looking up the object of that name with vhpi_handle_by_name. We would provide two additional properties: vhpiInstanceNameP and vhpiPathNameP so that foreign models and applications can provide similar output similar as to the simulator using the 'instance_name and'path_name attributes.

Issue: *Pathological Cases of Ambiguity*
VHDL syntax leaves room in its particular choice of namespaces and keywords for certain pathological problems. One of them is that the root entity name and a library logical name may be the same in some pathologically difficult elaborated design. Entity names like work, IEEE, etc. pose no conflict, even though those are also well known logical library names. The impact of this is that the first part of 'path_name may refer to the root instance entity name or a logical library and you cannot distinguish between them.

I regard this as a pathological (vs. practical) problem in which a number of solutions are rationale.

1) You can define a search order and return the first one found (solution hides names in a predictable manner).

2) You can also search the entire space, verify the name is a duplicate, and diagnose the problem as an ambiguous reference.

3) You can allow or require an extended syntax to qualify whether you mean the packInst or the root instance. For example, you could require:e(a) to disambiguate and always attempt to search the packInsts before the root inst. Probably this is acceptable

Resolution: always precede a library name or package name by a @

There is another minor pathological problem, that of multiple logical libraries referring to the same physical library. VHPI should not make any statement that referring to an object through 2 or more logical library names in any way preserves that this is the same object.

15.27 Creation of foreign drivers

Requirement: Testbench tools need to be able to participate to the resolved value of signals, for that they need to be able to create their own drivers of the signal they are interestd in. Testbench tools often apply several different test sequence to a design, and may want to driver different signals each time. The test pattern if often generated after inspection and analysis of the design.

Issue: Elaboration phase creation of foreign drivers is too late.
When and how can we create foreign drivers?

16. ANNEX C: Formal textual definition of the VHPI information model

```
Class abstractLiteral

Superclasses: literal

{
 intLiteral
 realLiteral
}

-----------------------------------------------------------

Class accessTypeDecl

Superclasses: typeDecl

one-to-one relationships:
    mult 1 subtype<-ValSubtype
-----------------------------------------------------------

Class aggregate

Superclasses: primaryExpr

Iteration relationships:
it  mult 1..* elemAssoc<-elemAssoc
-----------------------------------------------------------

Class aliasDecl
```

Description:
This class represents alias declarations.
In case of non alias objects which are not character literals, the vhpiSizeP property will return -1; the operation vhpi_get_value will return 0 because the alias has no value.

Superclasses: decl, simpleName

Attributes:
p int   -Size
  size in scalars of a value of the object

one-to-one relationships:
    mult 0..1 subtypeIndic<-Subtype
    mult 1 name<-name returns the aliased object or range. Name can itself be an alias.
Operations:
#vhpi_get_value(handle: vhpiHandle, value: vhpi_value_p) : int
  get the value of the object designated by the alias

Class allLiteral

Superclasses: entityDesignator

Class allocator

Description:
an allocator
new subtypeIndication [expr]

Superclasses: primaryExpr

one-to-one relationships:
    mult 1 subtypeIndic<-ValSubtype the subtype indication for the value accessed;
this subtype indication should be the subtype used for the allocation (new subtype_indication)
    mult 0..1 expr<-InitExpr the initial expression if specified by the allocator operation
----

Class anyCollection

Description:
A collection of any handle. No constraints on the members of the collection.

Superclasses: collection, base

Class archBody

Superclasses: secondaryUnit, entityAspect

Attributes:

p bool   -IsForeign

Iteration relationships:
it mult * stmt<-stmt
it mult * configItem<-ConfigSpecs

Class argv

Description:
A command line argument string separated by white spaces passed
to the tool.
This class mimics an element of the array of argv parameters that would
be passed to a C main routine.
A vendor is not required to give access to all the arguments passed on the
command line of the tool.

Superclasses: base

Attributes:
p string   -StrVal
 The string value of the argument as found on the command line
p bool   -IsPLI
 True if the argument is an argument which concerns VHPI. This is either for
 VHPI interface to process and take some action or for a VHPI application to
 process.
 This property allows an application to test if itshould process this command
 line argument. Command line arguments have no special syntax for VHPI.

p int   -Argc
 number of argvs

Operations:
-vhpi_handle_by_index(itRel: vhpiOneToManyT, handle: vhpiHandleT, index: int) : vhpiHandleT

Class arrayTypeDecl

Superclasses: compositeTypeDecl

Attributes:
p int   -NumDimensions
 number of the dimensions of the array
p bool   -IsAnonymous
 anonymous types have a simple name of $anonymous

one-to-one relationships:
  mult 1 subtype<-ElemSubtype
Iteration relationships:
it mult 1..* constraint<-constraint

Class assertStmt

Superclasses: seqStmt, eqProcessStmt

```
one-to-one relationships:
    mult 1 expr<-CondExpr
    mult 0..1 expr<-ReportExpr
    mult 0..1 expr<-SeverityExpr
----------------------------------------------------------

Class assocElem

Superclasses: base

Attributes:
p int   -Position
  position of the formal in the interface list
p bool   -IsOpen
  association has the OPEN keyword
p bool   -IsNamed
  returns True if this is a named association, false if
  positional

one-to-one relationships:
    mult 0..1 interfaceElt<-Local
    mult 0..1 interfaceElt<-Actual
    mult 0..1 interfaceElt<-Formal
----------------------------------------------------------

Class attrDecl

Superclasses: decl

one-to-one relationships:
    mult 1 typeMark<-Subtype
----------------------------------------------------------

Class attrName

Superclasses: name

{
 userAttrName
 predefAttrName
}

one-to-one relationships:
    mult 1 entityDesignator<-Prefix
----------------------------------------------------------

Class attrSpec

Superclasses: spec

one-to-one relationships:
    mult 1 expr<-expr
    mult 1 attrDecl<-attrDecl
Iteration relationships:
it  mult * pragma<-pragma internal
```

```
it  mult 1..* entityDesignator<-entityDesignator
-----------------------------------------------------------

Class base

Description:
Base class,
all other classes are derived from the base class.
all derived classes inherit the kind attribute

{
 region
 entityDesignator
 entityClassEntry
 subtype
 range
 protectedTypeBody
 stackFrame
 assocElem
 branch
 choice
 waveformElem
 condWaveform
 selectWaveform
 iterScheme
 transaction
 contributor
 interfaceElt
 basicSignal
 inPort
 outPort
 prefix
 elemAssoc
 callback
 foreignf
 entityAspect
 pragma
 tool
 argv
 reference
 spec
 iterator
 collection
 driverCollection
 anyCollection
}

Attributes:
p int   -Kind
  associates an integer constant (identifier) to each leaf class
p string   -KindStr
p bool   -IsInvalid
  True if the handle is invalid: the object which the handle refers to ceases to
  exist either because the object was dynamically elaborated,
  or by virtue of a user action (removing a callback or transaction or because
  the transaction matures.
```

If a handle is invalid, this is the only property that can be accessed. No other access is possible.

p bool -IsUninstantiated

If this property is TRUE then the handle represent uninstantiated VHDL data. This means that the data represented by this handle is pre elaboration (post-analysis) and does not contain any post elaboration information. In particular it is not possible to walk the instantiated design hierarchy from this handle or to get the value of this handle if that value can only be determined after elaboration. (may have a full name to be defined later in the uninstantiated access spec).

If this property is FALSE, the handle represent post-elaboration VHDL data and full access to the VHPI instantiated (post-elaboration) model is allowed.

Class basicSignal

Description:

a basic signal according to the LRM definition page 165,

The basic property is true for the classes derived from a basicSignal

The vhpiEntityClassP property shall return vhpiSignalEC for a basic signal handle kind.

Superclasses: base

{
sigDecl
portDecl
selectedName
indexedName
sliceName
}

Attributes:

p bool -IsBasic

is it a basic signal?

Explicit Guard signals can be basic signals; implicit guard signals cannot be.

a slice is basic only if it denotes a an indexedName basic signal (slice is size 1) or if it denotes the entire slice of a resolved composite basic signal.

p bool -IsResolved

The basic signal is resolved if if a resolution function is associated with the declaration of that signal or in the declaration of the subtype of that signal (page 27 VHDL lrm 1076-93). A signal can be resolved at the sub-element subtype level.

Iteration relationships:

it mult * driver<-driver returns the drivers for the basic signal.

note: a signal attribute is not a basic signal therefore you cannot iterate on drivers from a signal attribute. VHPI should generate an error.

it mult * contributor<-contributor

it mult * contributor<-contributor

Class binaryExpr

Superclasses: expr

one-to-one relationships:
mult 1 operator
<-operator

mult 1 expr<-LeftExpr
mult 1 expr<-RightExpr

Class bitStringLiteral

Superclasses: literal

Attributes:
p string -StrVal
The string value of the literal as it appears in the VHDL

Class blockConfig

Superclasses: configItem, lexicalScope

Iteration relationships:
it mult * decl<-Uses The uses clauses within the blockConfig
it mult * configItem<-configItem list of the configuration items within the blockConfig

----

Class blockStmt

Description:
a block statement instance

Superclasses: concStmt, lexicalScope, region

Attributes:
p int -BeginLineNo
the line number of the begin keyword
p int -EndLineNo
the linenumber of the end keyword
p bool -IsGuarded
p int -NumGens
number of generic declarations
p int -NumPorts
number of port declarations

one-to-one relationships:
mult 0..1 sigDecl<-GuardSig if the block is guarded, returns the GUARD signal declaration (implicit or explicit)
mult 0..1 expr<-GuardExpr
Iteration relationships:

it mult * spec<-spec The specifications defined in the block declarative region (may return attribute, disconnection or configuration specifications)
it mult * sigDecl<-sigDecl
it mult * portDecl<-portDecl
it mult * genericDecl<-genericDecl
it mult * constDecl<-constDecl
it mult * compInstStmt<-compInstStmt
it mult * compInstStmt<-compInstStmt
it mult * blockStmt<-blockStmt
it mult * attrSpec<-attrSpec
it mult * attrDecl<-attrDecl
it mult * attrDecl<-attrDecl
it mult * assocElem<-PortAssocs
it mult * assocElem<-GenericAssocs
it mult * aliasDecl<-aliasDecl
it mult * aliasDecl<-aliasDecl
Operations:
-vhpi_handle_by_index(itRel: vhpiOneToManyT, handle: vhpiHandleT, index: int) : vhpiHandleT

Class branch

Superclasses: base

Attributes:
p int -LineNo
line number of the branch
p int -LineOffset
p string -FileName

Iteration relationships:
it mult * seqStmt<-seqStmt
it mult * choice<-CondExpr

Class callback

Superclasses: base

Attributes:
p int -Reason
p int -State
either vhpiDisable, vhpiEnable, vhpiMature

Operations:
-vhpi_get_cb_info(cbHdl: handle) : vhpiCbDataT*;
-vhpi_remove_cb(cbHdl: handle) : int;
-vhpi_enable_cb(cbHdl: handle) : int;
-vhpi_disable_cb(cbHdl: handle) : int;

Class caseStmt

Superclasses: seqStmt

one-to-one relationships:
   mult 1 expr<-CaseExpr
Iteration relationships:
it mult 1..* branch<-branch

Class charLiteral

Description:
This is a character literal of the standard CHARACTER type or
one of its subtype.

Superclasses: literal, decl

Attributes:
p int   -Position
p string   -StrVal
  The string value of the literal as it appears in the VHDL:
  examples: '0' for literal of type char
                  NUL for literal nul

Class choice

Description:
A choice can either be an expression or a range denoted by a predefined
range attribute or a integer/enumerated range

Superclasses: base

{
 constraint
 expr
 othersLiteral
}

Class collection

Description:
a user-defined heterogeneous collection of objects

Superclasses: base

{
 uniformCollection
 anyCollection
}

Attributes:
p int   -NumMembers
  number of members in the collection

Iteration relationships:
it mult * base<-Members iteration method returns the element of the collection
Operations:
-vhpi_handle_by_index(itRel: vhpiOneToManyT, handle: vhpiHandleT, index: int) : vhpiHandleT
-vhpi_create(handleKind: classKind, refHdl: vhpiHandle, hdltoadd: vhpiHandle) : vhpiHandle
  used to return an ordered collection of handles composed of the refHdl and the hdltoadd

----

Class compConfig

Superclasses: configItem, lexicalScope, spec

Attributes:
p bool   -IsOpen
  The component configuration is opened: the entity aspect is "use OPEN" no port map or generic map should be provided
p bool   -IsDefault
  the component configuration uses default binding
p string   -CompName
  the component declaration name it applies to

one-to-one relationships:
    mult 0..1 blockConfig<-blockConfig
Iteration relationships:
it mult * assocElem<-PortAssocs the port map aspect
it mult * assocElem<-GenericAssocs the generic map aspect

Class compDecl

Superclasses: decl, lexicalScope

Attributes:
p int   -NumGens
p int   -NumPorts

Iteration relationships:
it mult * portDecl<-portDecl
it mult * genericDecl<-genericDecl
it mult * attrSpec<-attrSpec

Class compInstStmt

Description:
a component instance statement instance

Superclasses: designInstUnit, concStmt

Attributes:
p bool   -IsOpen
  the binding of the component instance is opened which means that the

component instance is not bound to an architecture/entity pair.

p bool -IsDefault

The binding of the component instance uses default binding.

p string -CompName

the component specification name or null if direct instantiation was used

p int -NumGens

number of generic declarations

p int -NumPorts

number of port declarations

one-to-one relationships:

mult 0..1 configItem<-ConfigSpec The optional configuration specification for that component instance which may be specified in the architecture body, or the config spec information if the component instance is a direct instantiation.

mult 0..1 compDecl<-compDecl internal return the component declaration or NULL if direct instantiation

Iteration relationships:

it mult * varDecl<-varDecl
it mult * sigDecl<-sigDecl
it mult * portDecl<-portDecl
it mult * genericDecl<-genericDecl
it mult * constDecl<-constDecl
it mult * compInstStmt<-compInstStmt
it mult * blockStmt<-blockStmt
it mult * assocElem<-PortAssocs
it mult * assocElem<-GenericAssocs

Operations:

-vhpi_handle_by_index(itRel: vhpiOneToManyT, handle: vhpiHandleT, index: int) : vhpiHandleT

Class compositeTypeDecl

Superclasses: typeDecl

{
arrayTypeDecl
recordTypeDecl
}

Class concStmt

Superclasses: stmt

{
generateStmt
blockStmt
compInstStmt
eqProcessStmt
}

Class condSigAssignStmt

Superclasses: sigAssignStmt

Iteration relationships:
it mult 1..* condWaveform<-condWaveform

Class condWaveform

Superclasses: base

Attributes:
p int   -LineNo
  line number of the waveform
p string   -FileName

one-to-one relationships:
    mult 1 expr<-CondExpr
Iteration relationships:
it mult 1..* waveformElem<-waveformElem

Class configDecl

Superclasses: primaryUnit, entityAspect

one-to-one relationships:
    mult 1 entityDecl<-entityDecl the entity Declaration this configuration refers to
    mult 1 blockConfig<-blockConfig

Class configItem

{
 blockConfig
 compConfig
}

one-to-one relationships:
    mult 1 entityAspect<-entityAspect
Iteration relationships:
it mult * entityDesignator<-SpecNames The iteration specNames will return null when the config item is for a direct component instance statement.

Class constDecl

Description:
a constant declaration

Superclasses: objDecl, constant

Attributes:
p bool   -IsDeferred

one-to-one relationships:
mult 0..1 expr<-InitExpr
Iteration relationships:
it mult * selectedName<-selectedName
it mult * indexedName<-indexedName

Class constParamDecl

Superclasses: paramDecl, constant

Attributes:
p modeT -Mode
mode can only be vhpiIn

Class constant

Superclasses: interfaceElt

{
constDecl
constParamDecl
selectedName
indexedName
}

Class constraint

Description:
a constrait can either be range,or a subtype indication or the 'range or 'reverse_range attribute

Superclasses: choice

{
subtypeIndic
range
predefAttrName
}

Class contributor

Description:
a contributor to the value of a signal. A contributor can be a driver, a source (port of a block with which the signal is associated), a conversion function applied to a port which is connected to the signal, a type conversion, a static expression, or a composite collection of sources.
Contributors to the IN value or to the OUT value of a port can be acquired.
note: a contributor can be a signal attribute 'active, 'quiet or 'transaction or 'delayed.

note: Iteration on contributors from the signal attributes handle should return NULL and an error.
iteration on contributor from an implicit GUARD signal should return the guard expression. Iteration on contributors from an explicit GUARD should return the contributor of that GUARD.

Superclasses: base

```
{
 driver
 port
 signal
 convFunc
 expr
}
```

Class convFunc

Superclasses: interfaceElt, contributor

one-to-one relationships:
    mult 1 interfaceElt<-Actual
    mult 1 funcDecl<-funcDecl
    mult 1 contributor<-contributor

Class derefObj

Description:
A vhpiDerefObjK handle represents the object designated by an access value of a variable. A vhpiDerefObjK handle kind can be obtained by:
- applying the vhpiDerefObj method to a variable of an access type if the access value of that variable is not null
- by accessing an expression which is a name denoting an object designated by an access value, such names have a suffix of .ALL
- by applying the vhpiDerefObj method to an object of kind vhpiAllocatorK

Superclasses: prefixedName

Operations:
-vhpi_put_value(handle: vhpiHandleT, value: vhpiValueT *, flags: vhpiPutValueModeT) : int
#vhpi_get_value(handle: vhpiHandle, value: vhpi_value_p) : int

----

metaclass Class decl

Description:
a declaration

Superclasses: entityDesignator, reference

{

objDecl
aliasDecl
attrDecl
groupDecl
compDecl
groupTempDecl
libraryDecl
typeDecl
subtypeDecl
unitDecl
elemDecl
subpBody
subpDecl
enumLiteral
charLiteral
}

Attributes:
p string -Name
the declaration name, unspecified case if basic identifier or case - preserved if extended identifier
If the declaration denotes an anonymous type then the vhpiNameP property returns "$anonymous"
p string -CaseName
The case sensitive name of the declared item, Same restrictions as for vhpiNameP.
p string -FullName
full hierarchical from the top of the hierarchy.The path name is given in unspecified case for basic identifiers and case-preserved for extended identifiers.
FullName properties does not apply to libraryDecl class
Note: a local port or generic does not have a fullName.
Issue: do subpdecl and subpbody have fullName
p string -FullCaseName
Case preserved full name
FullCaseNameP property does not apply to libraryDecl class.
note: a local component port or generic does not have a full name
p string -FileName
pathname of the source file where that declaration appears
p int -LineNo
line number where the declared identifier for the declared item appears in the source
p int -LineOffset
The character offset in the source file of the definition name of the declaration
p bool -IsImplicitDecl
Returns true for implicit constant, signals, functions and procedure declarations.
For example:
this property returns true for GUARD signals of blocks, for loop parameter and generate parameter, for implicit functions such as OPEN, NEW, ...

one-to-one relationships:
mult 0..1 region<-ImmRegion vhpiDecls returns all declarations of class vhpiDecl in the

instance
vhpiImmRegion for a local ports/generics should return null as the component declaration is not a region.
mult 0..1 lexicalScope<-lexicalScope The lexical scope of a library declaration should return null. For all other declarations, it should return the immediate scope where the delcaration is defined.
Iteration relationships:
it mult * pragma<-pragma internal

Class designInstUnit

Superclasses: region

{
rootInst
packInst
compInstStmt
}

one-to-one relationships:
mult 1 designUnit<-designUnit
Iteration relationships:
it mult * attrSpec<-attrSpec
it mult * attrDecl<-attrDecl
it mult * attrDecl<-attrDecl
it mult * aliasDecl<-aliasDecl
it mult * aliasDecl<-aliasDecl

Class designUnit

Description:
an analyzed library unit (primary or secondary unit)

Superclasses: entityDesignator, lexicalScope

{
primaryUnit
secondaryUnit
}

Attributes:
p string -LibLogicalName
the library logical name where that design unit can be found
p string -LibPhysicalName
the physical name of the library where that design unit has been compiled
p int -StartLineNo
the line number in the source where that library unit starts (includes the line number of the library and use clauses)
p int -EndLineNo
the line number in the source where that described library unit ends
p string -FileName
pathName of the source filename where that library unit was described

p string -UnitName
name of the declared design unit in the VHDL source
name is unspecified case for basic identifiers or case-preserved for extended identifiers The unitName of a design unit of the following class is:
EntityDecl: lib_name.entity_name
Arch body: lib_name.entity_name:arch_name
PackDecl: lib_name.pack_name
Pack Body: lib_name.pack_name:BODY
Config: lib_name.config_name
p string -Name
The identifier name of the declared design unit
p protectKindT -ProtectedLevel
the level of protection of that design unit, 0 for complete visibility, 1 for interface cell visibility (ports, generics, cell name, ...), 2 ...

p string -CaseName

Iteration relationships:
it mult * spec<-spec The specifications defined in the design unit
For an archBody, there could be attrSpecs, disconnections or config specs)
for an entity or package declaration , it could be attribute or disconecttion specs.
For a configDecl the specifications defined in the configuration declarative part (only attribute specifications)
For a packBody, no specifications at all
it mult * pragma<-pragma internal
it mult 1..* designUnit<-DepUnits returns the dependent design units
it mult * decl<-Uses returns the declarations imported by a use clause that are referenced by the design unit
it mult * decl<-decl the declarations (all kinds of the vhpiDecl class) within the library unit not including the design unit itself
----

Class disconnectSpec

Superclasses: spec

one-to-one relationships:
mult 1 typeMark<-typeMark
mult 0..1 othersLiteral<-othersLiteral
mult 1 expr<-TimeExpr
mult 0..1 allLiteral<-allLiteral
Iteration relationships:
it mult * signal<-Signals
it mult * pragma<-pragma internal

Class driver

Superclasses: contributor

Attributes:
p int -State
the driver state for the current simulation cycle: active, or disconnected, or quiet, undefined.
Can be queried any time during a delta cycle before or after the driver

transaction matures.
This is an advanced property

p bool -IsPLI
is this a PLI created driver?

p int -Size
driver size in bytes?

p accessT -Access
read, write or no access at all to the driver

one-to-one relationships:
mult 1 eqProcessStmt<-eqProcessStmt
mult 1 basicSignal<-basicSignal returns the drivers for the basic signal.

note: a signal attribute is not a basic signal therefore you cannot iterate on drivers from a signal attribute. VHPI should generate an error.

Iteration relationships:
it mult * transaction<-transaction

Operations:
-vhpi_schedule_transaction(drvhdl: vhpiHandleT, value: vhpiValueT *, delay: vhpiTimeT *, delayMode: int, pulseRej: vhpiTimeT *, flags: int) : vhpiHandleT
-vhpi_get_value(handle: vhpiHandle, value: vhpi_value_p) : int
returns the current value of the driver
-vhpi_create(kind: int, basicSignal: vhpiHandle, process: vhpiHandle) : vhpiHandle
-vhpi_register_cb(cbdatap: vhpiCbDataT *, int flags: <unnamed>) : vhpiHandleT;

Class driverCollection

Description:
A collection of driver handles.
All drivers must belong to the same signal and the same process.

Superclasses: uniformCollection, base

Operations:
-vhpi_schedule_transaction(hdl: vhpiHandle, values: vhpiValueT *, delay: vhpiTimeT, pulserej: vhpiTimeT, flags: int) : vhpiHandle
The value is scheduled on the drivers of the collection.The value is interpreted with respect to the order of the drivers in the collection.

Class elemAssoc

Superclasses: base

Attributes:
p bool -IsNamed
True if it is named association, false if positional

one-to-one relationships:
mult 1 expr<-expr

Iteration relationships:
it mult * choice<-choice

Class elemDecl

Superclasses: decl

Attributes:
p int -Position
position number of the declaration in the records, starts at 0

one-to-one relationships:
mult 1 subtype<-ElemSubtype

Class entityAspect

Superclasses: base

{
archBody
entityDecl
configDecl
}

Class entityClassEntry

Superclasses: base

Attributes:
p vhpiEntityClassT -EntityClass
p bool -IsUnconstrained
true if range is unconstrained, false otherwise

Class entityDecl

Superclasses: primaryUnit, entityAspect

Attributes:
p int -NumGens
p int -NumPorts

Iteration relationships:
it mult * stmt<-stmt
it mult * portDecl<-portDecl Should this be an ordered iteration for uninstantiated access?
it mult * genericDecl<-genericDecl Should this be an ordered iteration for uninstantiated access?

Class entityDesignator

Superclasses: base

{
decl
designUnit

stmt
name
literal
othersLiteral
allLiteral
}

Attributes:
p vhpiEntityClassT -EntityClass
the entity class enumeration values can be:
vhpi[Entity,Architecture, Configuration,Procedure,
Function,Package,Type,Subtype,Constant,Signal,Variable,Literal,Units,File,
Group, Component, Label]EC
If the entity designator is the others or all literal, the entity class will be the entity class of the entities denoted by others or all. In case of an entity designator for a disconnection specification, the entityClass is vhpiSignalEC.
For name class, property returns the class of the name as defined by the LRM page 71 A name entity class can either be a procedure, signal, variable, group, function, variable, literal, file, constant.
For unlabelled statements, return vhpiUndefined.

Class enumLiteral

Superclasses: literal, decl

Attributes:
p int -Position
p string -StrVal
The string value of the literal as it appears in the VHDL

Class enumRange

Description:
An enumeration range

Superclasses: range

Attributes:
p int -LeftBound
The left bound value of the range or -1 if unconstrained
p int -RightBound
the right bound value of the range or -1 if unconstrained

Class enumTypeDecl

Superclasses: scalarTypeDecl

Attributes:

```
p int   -NumLiterals
  number of enumeration literals

Iteration relationships:
it  mult 1..* enumLiteral<-enumLiteral
-----------------------------------------------------------

Class eqProcessStmt

Description:
an equivalent process statement instance

Superclasses: concStmt, stackFrame, region

{
 procCallStmt
 processStmt
 assertStmt
 sigAssignStmt
}

Attributes:
p bool   -IsPostponed
  returns 1 if this is a postponed equivalent process 0 otherwise

one-to-one relationships:
    mult 1 stackFrame<-CurStackFrame returns the current executing or suspended stack frame,
could be the process itself
Iteration relationships:
it  mult * name<-Sensitivity
it  mult * driver<-driver
-----------------------------------------------------------

Class exitStmt

Superclasses: seqStmt

Attributes:
p string   -LoopLabelName
  The name of the loop label

one-to-one relationships:
    mult 0..1 expr<-CondExpr
-----------------------------------------------------------

Class expr

Superclasses: choice, interfaceElt, contributor

{
 primaryExpr
 binaryExpr
 unaryExpr
}

one-to-one relationships:
```

mult 1 typeDecl<-typeDecl internal returns the type of an expression
mult 1 subtype<-subtype returns the subtype of the expression.

the returned subtype can either be a subtype indication or a typeMark this allows VHPI to not create unnecessary handles for subtype indications when the subtype indication is the same as the type declaration.

Class file

Superclasses: interfaceElt

{
fileDecl
fileParamDecl
}

Class fileDecl

Superclasses: objDecl, file

Attributes:
p openModeT -OpenMode
For VHDL 93:
-> In instantiated mode, the open mode of the file declaration either WRITE_MORE, READ_MODE, APPEND_MODE.
-> In uninstantiated mode the open mode property may return vhpiUndefined (if not open mode is specified or if the open mode expression is not a locally static expression).

In VHDL 87, in instantiated access, the OPEN_MODE is either vhpiInMode or vhpiOutMode.
in vhdl 87, in uninstantiated access the vhpiOpenModeP property may return vhpiInMode or vhpiOutMode. The default mode if the open mode is unspecified is vhpiInMode.
p string -LogicalName
The file logical name if opened, or null if not opened.

one-to-one relationships:
mult 0..1 expr<-LogicalExpr Returns the expression providing the logical name of the file declaration if the open information is provided

----

Class fileParamDecl

Superclasses: file, paramDecl

Class fileTypeDecl

Superclasses: typeDecl

one-to-one relationships:
mult 1 subtype<-ValSubtype

Class floatRange

Superclasses: range

Attributes:
p real   -FloatLeftBound
  the float left bound of the range
p real   -FloatRightBound
  the float right bound of the range

Class floatTypeDecl

Superclasses: scalarTypeDecl

one-to-one relationships:
    mult 0..1 constraint<-constraint

Class forGenerate

Superclasses: generateStmt

Attributes:
p int   -GenerateIndex

one-to-one relationships:
    mult 1 constraint<-constraint the range of the generate parameter either a predefined attribute range or an integer/enumerated range, or a subtype indication
    mult 1 constDecl<-ParamDecl
Iteration relationships:
it mult * attrSpec<-attrSpec

Class forLoop

Description:
A for loop statement

Superclasses: iterScheme

Attributes:
p int   -LoopIndex
  the loop index integer value (or position of the enumeration literal if enumerated type) if the loop is executing, -1 if the region has not been elaborated (is not executing).

one-to-one relationships:
    mult 1 constraint<-constraint The range of the for loop either a predefined attribute range, or an integer range.
    mult 1 constDecl<-ParamDecl returns the parameter implicit declaration, or NULL if the loop has not been elaborated

----

Class foreignf

Superclasses: base

Attributes:
p vhpiForeignT; -ForeignKind
returns the kind of foreign model one of: vhpiArchF, vhpiProcF or vhpiFuncF

Operations:
-vhpi_get_foreignf_info(vhpiHandleT: hdl, vhpiForeignDataT*: foreigndatap) : int

----

Class funcCall

Superclasses: subpCall, primaryExpr, prefix

one-to-one relationships:
mult 0..1 derefObj<-DerefObj
Iteration relationships:
it mult * selectedName<-selectedName
it mult * indexedName<-indexedName
Operations:
-vhpi_put_value(handle: vhpiHandleT, value: vhpiValueT *, flags: vhpiPutValueModeT) : int
Mandatory for foreign function call, will set the return value of the foreign function call,
legitimate vendor extension for VHDL function, not specified by the standard
If the function return type is a composite type which is not an array of scalars, then individual vhpi_put_value class must be made to set each of the sub-elements.

Class funcDecl

Description:
a function declaration

Superclasses: subpDecl

Attributes:
p bool -IsPure

one-to-one relationships:
mult 1 typeMark<-ReturnTypeMark

Class generateStmt

Description:
a generate statement instance

Superclasses: concStmt, region

{
 ifGenerate
 forGenerate
}

Attributes:
p int   -BeginLineNo
  the line number of the begin keyword
p int   -EndLineNo
  the linenumber of the end keyword

Iteration relationships:
it mult * varDecl<-varDecl
it mult * spec<-spec The specifications defined in the generate stmt declarative region (may return attribute, disconnection, configuration specifications)
it mult * sigDecl<-sigDecl
it mult * constDecl<-constDecl
it mult * compInstStmt<-compInstStmt
it mult * blockStmt<-blockStmt
it mult * aliasDecl<-aliasDecl

Class generic

Superclasses: interfaceElt

{
 genericDecl
 selectedName
 indexedName
}

Class genericDecl

Description:
The following methods/properties are not allowed on local generics:
vhpiFullNameP?

Superclasses: interfaceDecl, generic

Attributes:
p bool   -IsVital
p modeT   -Mode
  mode can only be vhpiIn
p bool   -IsLocal
  true if this is local component generic declaration

Class groupDecl

Superclasses: decl

one-to-one relationships:

mult 1 groupTempDecl<-groupTempDecl
Iteration relationships:
it mult 1..* entityDesignator<-entityDesignator
it mult * attrSpec<-attrSpec

Class groupTempDecl

Superclasses: decl

Iteration relationships:
it mult * entityClassEntry<-entityClassEntry

Class ifGenerate

Superclasses: generateStmt

one-to-one relationships:
mult 1 expr<-CondExpr

Class ifStmt

Superclasses: seqStmt

Iteration relationships:
it mult 1..* branch<-branch

Class inPort

Superclasses: base

Iteration relationships:
it mult * basicSignal<-basicSignal
Operations:
#vhpi_get_value(handle: vhpiHandle, value: vhpi_value_p) : int
get the IN value of the port (effective value)
-vhpi_register_cb(cbdatap: vhpiCbDataT *, int flags: <unnamed>) : vhpiHandleT;

Class indexedName

Superclasses: prefixedName, port, basicSignal, variable, signal, generic, constant

Attributes:
p int -BaseIndex
returns the offset of the indexedname to the base of the entire declaration.
The first indexedname of the declared object has an offset of 0. The returned
value of this property can be passed to vhpi_handle_by_index to create the
same indexedname handle.

Iteration relationships:
it mult * expr<-IndexExprs This iteration should be supported for handles which denote real

VHDL references which appear in the source code.
This iteration returns NULL in other cases.
Operations:
#vhpi_get_value(handle: vhpiHandle, value: vhpi_value_p) : bool
#vhpi_put_value(handle: vhpiHandle, value: vhpi_value_p, flags: vhpiPutValueModeT) : bool

----

Class intLiteral

Superclasses: abstractLiteral

Attributes:
p int   -IntVal
  The integer value of the literal

Class intRange

Description:
an integer bounded range

Superclasses: range

Attributes:
p int   -LeftBound
  The left value of the range, or -1 if range null or unconstrained
p int   -RightBound
  the right value of the range or -1 if range null or unconstrained

----

Class intTypeDecl

Superclasses: scalarTypeDecl

one-to-one relationships:
    mult 0..1 constraint<-constraint

Class interfaceDecl

Description:
an interface declaration

Superclasses: objDecl

{
 genericDecl
 portDecl
 paramDecl
}

Attributes:
p int   -Position

the position of the interface declaration in the interface list, index starts at 0.

one-to-one relationships:
mult 0..1 expr<-InitExpr returns the signal attributes which have been referenced in the VHDL source or which may have been created some other way (gui command or vhpi_create function)

Class interfaceElt

Superclasses: base

{
variable
generic
constant
file
port
signal
convFunc
expr
}

Class iterScheme

Superclasses: base

{
forLoop
whileLoop
}

Class iterator

Superclasses: base

Class lexicalScope

{
designUnit
blockStmt
compDecl
recordTypeDecl
protectedTypeDecl
protectedTypeBody
subpBody
subpDecl
loopStmt
blockConfig
compConfig

}

Class libraryDecl

Description:
a library declaration. A library ony has a name, line, offset properties

Superclasses: decl

Iteration relationships:
it mult * designUnit<-designUnit

Class literal

Superclasses: primaryExpr, entityDesignator

{
 enumLiteral
 physLiteral
 stringLiteral
 bitStringLiteral
 charLiteral
 nullLiteral
 abstractLiteral
}

Iteration relationships:
it mult * attrSpec<-attrSpec
Operations:
vhpi_get_value(hdl: vhpiHandleT, value: vhpiValueT *) : int
  get the current value of the literal.
  The value of a physical literal is retrieved with the unit field set to the
  physical position of the unit in which the physical literal is expressed.

Class loopStmt

Superclasses: seqStmt, region, lexicalScope

one-to-one relationships:
    mult 0..1 iterScheme<-iterScheme
Iteration relationships:
it mult * attrSpec<-attrSpec

Class name

Superclasses: primaryExpr, reference, prefix, entityDesignator

{
 prefixedName

attrName
simpleName
}

Attributes:
p string   -Name
p string   -FullName
p string   -CaseName
p string   -FullCaseName
p int   -Size
  size in scalars of the name
  This property should be supported for locally static names. An implementation may optionally support globally static names.
  size of scalar variables of access types is 1.
  size of the null literal which represents the null access value is 1.

one-to-one relationships:
    mult 1 derefObj<-DerefObj This relationship is not allowed from a sub-class of the class predefAttrName
Iteration relationships:
it mult * selectedName<-selectedName This relationship is not allowed from a sub-class of the class predefAttrName.
it mult * indexedName<-indexedName This relationship is not allowed from a sub-class of the class predefAttrName
Operations:
vhpi_get_value(hdl: vhpiHandleT, value: vhpiValueT *) : bool
  get the current value of the named thing. The vhpi_get_value should be supported for locally static names. Implementations may provide support for globally static names as well.

Class nextStmt

Superclasses: seqStmt

Attributes:
p string   -LoopLabelName
  the name of the loop label

one-to-one relationships:
    mult 0..1 expr<-CondExpr

Class null

Description:
This represents a null handle. This is not a class.

one-to-one relationships:
    mult 1 tool<-tool
    mult 1 rootInst<-rootInst
    mult 1 eqProcessStmt<-CurEqProcess
    mult 0..1 callback<-CurCallback Returns the currenly executing callback if any
Iteration relationships:

it mult * vpidesign<-vpidesign internal
it mult * region<-CurRegions the currently executing region instances
it mult * packInst<-packInst Iteration on package instances will return all package body instances used in the design. It also returns the standard package declaration.
it mult * foreignf<-foreignf returns the foreign models which have been registered for this tool session
it mult * callback<-callback returns all callbacks (including the disabled, freed but not removed, matured).
Operations:
-vhpi_get_time(timep: vhpiTimeT *, cyclesp: vhpiInt64T *) : void
  computes the current simulation time and relative or absolute delta cycles
-vhpi_get_next_time(timep: vhpiTimeT *) : vhpiIntT
  computes the next simulatrion time when some activity is scheduled,

----

Class nullLiteral

Description:
The literal represented by the VHDL keyword "null".

Superclasses: literal

Class nullStmt

Superclasses: seqStmt

Class objDecl

Description:
an object declaration

Superclasses: decl, simpleName

{
 sigDecl
 varDecl
 interfaceDecl
 fileDecl
 constDecl
}

Attributes:
p int   -Size
  size in scalars of a value of the object
p accessT   -Access
  The access of the object: vhpiNoAccess, vhpiRead, vhpiWrite, vhpiConnectivity (must be defined as bit flags).
  If the file is not opened, vhpiNo Access.
  If the file i sopened for read mode, vhpiRead
  If the file is opened for write or append mode, vhpiWrite

one-to-one relationships:
mult 1 typeDecl<-BaseType
mult 1 subtypeIndic<-Subtype
Iteration relationships:
it mult * attrSpec<-attrSpec
Operations:
#vhpi_get_value(handle: vhpiHandle, value: vhpi_value_p) : int
get the current value of the object
The value of a file declaration should be the logical name of the opened file.
The value should be nul if the file is not opened.

Class operator

Superclasses: primaryExpr

Attributes:
p string -OpName

one-to-one relationships:
mult 1 decl<-decl

Class othersLiteral

Superclasses: choice, entityDesignator

Class outPort

Superclasses: base

Iteration relationships:
it mult * basicSignal<-basicSignal
Operations:
#vhpi_get_value(handle: vhpiHandle, value: vhpi_value_p) : int
get the OUT value of the port (driving value)
#vhpi_put_value(handle: vhpiHandleT, value: vhpiValueT *, flags: vhpiPutValueModeT) : int
-vhpi_register_cb(cbdatap: vhpiCbDataT *, int flags: <unnamed>) : vhpiHandleT;

Class packBody

Superclasses: secondaryUnit

Class packDecl

Superclasses: primaryUnit

Class packInst

Description:
represent an elaborated package that we call package instance (usually package declaration/body pair) exceptions are for example the package standard that only has a package declaration

Superclasses: designInstUnit

one-to-one relationships:
   mult 1 vpidesign<-vpidesign internal
   mult 1 null<-UpperRegion Iteration on package instances will return all package body instances used in the design. It also returns the standard package declaration.
Iteration relationships:
it mult * varDecl<-varDecl
it mult * sigDecl<-sigDecl
it mult * constDecl<-constDecl

Class paramAttrName

Superclasses: predefAttrName

one-to-one relationships:
   mult 0..1 expr<-ParamExpr

Class paramDecl

Description:
a sub-program formal parameter declaration

Superclasses: interfaceDecl

{
 fileParamDecl
 sigParamDecl
 varParamDecl
 constParamDecl
}

Class physLiteral

Superclasses: literal

Attributes:
p phys   -PhysVal
  The physical value of the physical literal expressed in the base unit of its physical type
p phys   -PhysPosition
  The position number of the physical literal as defined by the LRM page 37 line 175

one-to-one relationships:
mult 1 unitDecl<-unitDecl
mult 1 abstractLiteral<-abstractLiteral

Class physRange

Superclasses: range

Attributes:
p phys -PhysLeftBound
The left bound of the physical range
p phys -PhysRightBound

Class physTypeDecl

Superclasses: scalarTypeDecl

one-to-one relationships:
mult 1 unitDecl<-BaseUnit
mult 0..1 constraint<-constraint
Iteration relationships:
it mult 1..* unitDecl<-unitDecl

Class port

Description:
A port is either a port declaration or sub-part thereof or a predefined implicit signal attribute ('delayed, 'quiet, 'stable, 'transaction)

Superclasses: interfaceElt, contributor

{
portDecl
selectedName
indexedName
predefAttrName
}

Attributes:
p bool -IsForced
true if the object has been externally forced by either
vhpi_put_value or some other way.
false otherwise

one-to-one relationships:
mult 0..1 outPort<-outPort
mult 0..1 interfaceElt<-Actual
mult 0..1 inPort<-inPort
Iteration relationships:
it mult * callback<-callback
it mult * basicSignal<-basicSignal

Class portDecl

Description:
The following methods/properties are not allowed on a local port:

Superclasses: basicSignal, port, interfaceDecl

Attributes:
p sigKindT -SigKind
  vhpiBus or vhpiNormal
p bool -IsGuarded
p bool -IsOpen
p modeT -Mode
  mode is either vhpiIn, vhpiOut, vhpiInout, vhpiBuffer or vhpiLinkage
  only signal/port class can be buffer or linkage mode.
  A buffer signal can only have one source
p bool -IsLocal
  true if this a local component port

one-to-one relationships:
    mult 0..1 funcDecl<-ResolFunc returns a handle to the resolution function if if a resolution function is associated with the declaration of that port or in the declaration of the port of that signal (page 27 VHDL lrm 1076-93)

Iteration relationships:
it mult * selectedName<-selectedName
it mult * predefAttrName<-SigAttrs returns the signal attributes which are referenced in the VHDL source or which have been created some other way (gui or vhpi_create function)
it mult * indexedName<-indexedName
it mult * basicSignal<-basicSignal
Operations:
-vhpi_put_value(handle: vhpiHandleT; value: vhpiValueT *; flags: vhpiPutValueModeT) : int
-vhpi_register_cb(cbdatap: vhpiCbDataT *, int flags: <unnamed>) : vhpiHandleT;

----

internal Class pragma

Superclasses: base

Attributes:
p string -Name
  the pragma name
  ex " translate_on", "resolution_method" etc... as it appears in the VHDL source
p string -StrVal
  returns the pragma string value or null if no string is supplied after the pragma name.
  For example a pragma value string would be null for "translate_on", or would be the entity name, or port name following the pragmas "map_to_entity" or "return_port_name"
  Pragmas which have a value string are pragmas of resolution function,built in functions or subprograms.

Class predefAttrName

Superclasses: attrName, constraint, signal, port

{
 paramAttrName
 simpAttrName
}

Attributes:
p int -PredefAttr
 The predefined attribute (one of the values
 {vhpiStablePA, vhpiQuietPA...)
p int -AttrKind
 the attribute kind either value, function, type, range or signal attribute
 vhpiValueAK, vhpiFunctionAK, vhpiTypeAK, vhpiRangeAK

Class prefix

Superclasses: base

{
 funcCall
 name
}

Class prefixedName

Superclasses: name

{
 derefObj
 selectedName
 indexedName
 sliceName
}

one-to-one relationships:
 mult 1 simpleName<-simpleName I think this method returns the declared name of the prefixed name
 mult 1 prefix<-prefix The vhpiPrefix method should be supported for handles which denote real VHDL references encountered in the VHDL source, not handles created by iteration
such as indexedNames, selectednames, basicSignals,
contributors etc... In the case of fake handles, the vhpiPrefix method should return NULL and a vhpi error.

Class primaryExpr

Superclasses: expr

{
 funcCall
 name
 operator

 literal
 aggregate
 typeConv
 allocator
}

Attributes:
p int   -Staticness
  returns vhpiLocallyStatic, vhpiGloballyStatic or vhpiDynamic

----

Class primaryUnit

Superclasses: designUnit

{
 entityDecl
 packDecl
 configDecl
}

Class procCallStmt

Description:
a procedure call statement

Superclasses: seqStmt, subpCall, eqProcessStmt

Attributes:
p bool   -IsPassive
  true if no signal assignments appear in the procedure body

Iteration relationships:
it mult * sigDecl<-DrivenSigs

Class procDecl

Description:
a procedure declaration

Superclasses: subpDecl

Class processStmt

Description:
a process statement

Superclasses: eqProcessStmt

Attributes:
p bool   -IsPassive
  process is passive: does not contain any signal assignments
p int   -BeginLineNo
  line where the begin keyword appears
p int   -EndLineNo
  line number where the end keyword appears
p bool   -IsForeign

Iteration relationships:
it mult * varDecl<-varDecl
it mult * spec<-spec returns the specifications (only attribute specifications) defined in the process declarative region
it mult * constDecl<-constDecl
it mult * attrSpec<-attrSpec
it mult * attrDecl<-attrDecl
it mult * attrDecl<-attrDecl
it mult * aliasDecl<-aliasDecl
it mult * aliasDecl<-aliasDecl

Class protectedType

Description:
region formed by both the protected type declaration and body

Superclasses: region

one-to-one relationships:
    mult 1 protectedTypeDecl<-protectedTypeDecl

Class protectedTypeBody

Superclasses: base, lexicalScope

Attributes:
p string   -Name
  the protected body name (same as the protected type declaration name
p string   -CaseName
  the case preserved name of the protected body
p int   -LineNo
  the line number where the protected type body name appears
p int   -LineOffset
  the line offset for the first character of the protected body identifier name

one-to-one relationships:
    mult 1 protectedTypeDecl<-protectedTypeDecl

Class protectedTypeDecl

Superclasses: typeDecl, lexicalScope

Attributes:
p int   -EndLineNo

one-to-one relationships:
    mult 1 protectedTypeBody<-protectedTypeBody

Class range

Description:
a range either integer, float range or a predefined attribute denoting a range ('range or 'reverse_range attributes)

Superclasses: base, constraint

{
 intRange
 floatRange
 physRange
 enumRange
}

Attributes:
p bool   -IsDiscrete
p bool   -IsUp
p bool   -IsNull
  it is a null range
p bool   -IsUnconstrained
  true if range is unconstrained, false otherwise
p int   -Staticness
  returns vhpiLocallyStatic, vhpiGloballyStatic, or vhpiDynamic

one-to-one relationships:
    mult 0..1 expr<-LeftExpr returns the leftExpr or NULL if range is unconstrained also generates an error if range is unconstrained
    mult 0..1 expr<-RightExpr return the right expression of the range or null if range is unconstrained; generates an error if range is unconstrained
----

Class realLiteral

Superclasses: abstractLiteral

Attributes:
p real   -RealVal
  The real value of the literal

Class recordTypeDecl

Superclasses: compositeTypeDecl, lexicalScope

Attributes:
p int -NumFields
number of fields in the record

Iteration relationships:
it mult 1..* elemDecl<-RecordElems
----

internal Class reference

Description:
The referenced item

Superclasses: base

{
decl
region
region
subpBody
subpCall
name
}

Class region

Description:
Class representing a VHDL scope in an elaborated model
A component instance statement that is unbound is still considered as a scope. A for generate for which the range is NULL or an if generate for which the condition is false is not considered as a scope

Superclasses: base, reference, reference

{
designInstUnit
generateStmt
blockStmt
protectedType
subpCall
eqProcessStmt
loopStmt
}

Attributes:
p string -FullName
full hierarchical statically instantiated name of the scope.
A for loop stmt is a dynamically elaborated region which has no fullname.
A sequential subprogram call is not a region and has no fullname.
A concurrent subprogram call is a region and has a fullname.
A function call is an expression and has no full name.
The name is returned in unspecified case characters unless it applies to names

of extended identifiers for which the case is preserved.
It is the absolute path to the scope containing
instance name (but no binding information as defined
in the instanceName VHDL attribute)
This property can only aply to objects living in a VHDL scope.

It starts by a ":" which denotes the top of the hierarchy followed by all the
different scope instances to that scope. Each scope is separated by a ":".
A scope name is either:
a package body name,
an instance name (instance label, generate index),
a concurrent statement implicit or explicit label..

p string -FullCaseName
Full hierarchical instantiated case sensitive name of the region. Same
restrictions as for vhpiFullNameP.
The case is the case of the identifier declaring the region.

p string -Name
simple name of the static region instance
A for loop stmt is a dynamically elaborated region and has no name. A
sequential subprogram call is not a region and has no name. a concurrent
procedure call
is a region and has a name: the explicit or implicit label name. a function
call is an expression and has no name.
This is the name (unspecified case) for basic region identifiers or the
case-preserved name for extended identifiers.
This name is:
for a component instance, the instance label name (ex. u1)
for a generate instance, the generate label name with the index (ex. g(1))
for a block instance, the block instance label (ex. b_lab)
for a process instance,a created process label name (ex: _P<num> or _p<num>
where num is an integer corresponding to the sequence number of the equivalent
process appearing in the VHDL text source. num starts at 0 and increments by
1. Numbering of unlabelled equivalent processes starts at the entity.
for a rootInst: :
for a package instance, the package body name
(ex: pack1)
for a subprogram call instance, the subprogram declaration name (ex: fp)

p string -CaseName
The case sensitive name of the declared region. Same as the vhpiNameP for
extended identifiers or unlabelled eqprocesses. Same restrictions as for
vhpiNameP.

p string -DefName
name which identifies the path to the declared thing
in the library that is bound to this scope.
For a designInstUnit, it is the
lib_name:entity_name(arch_name),
for a subprogram, it is:
lib_name:entity_name(arch_name):subprogram_name
if the subprogram is declared in an architecture
lib_name:pack_body_name:subprogram_name
if the subprogram body is declared in a package body
for a processInst it is:
lib_name:entity_name(arch_name):eq_process_label_name
Each of the names is either unspecified for basic
identifiers or case-preserved for extended identifiers.
For unlabelled processes , it is the name generated by the VHPI interface

"_P<int>" or "_p<int>".

p string -FileName
pathname of the source file where that scope instance was found during analysis
p int -LineNo
line number where the scope instance starts:
for a subpCall, the line number of the subprogram call,
for a rootInst, the line number of the definition of the architecture body it is bound to,
for a packInst, the line number of the package body
for a compInst, the line number of the component instance,
for a generate instance, the line number of the generate statement,
for a block instance, the line number where the block statement starts in the source
for a eqProcessStmt, the line number of the equivalent process statement.
For all kinds of regions, the line number should be the line number for that region.
p int -LineOffset
The character offset in the source file of the definition
name of the scope (architecture name or package body name)
p : domainT #Domain
returns whether or not the region is digital, analog or mixed (vhpiDigital, vhpiAnalog or vhpiMixedSignal)

one-to-one relationships:
mult 1 region<-UpperRegion Internal regions return the elaborated regions including:
- equivalent processes,
- structural concurrent statements.
- dynamically elaborated regions (for loop stmts and subpCall).
However support of dynamically elaborated regions is not included in the compliance level of hierarchy traversal.

UpperRegion: returns the higher enclosing structural instance or NULL if the reference handle is a rootinst or a packinst.
Iteration relationships:
it mult * stmt<-stmt
it mult * region<-InternalRegions Internal regions return the elaborated regions including:
- equivalent processes,
- structural concurrent statements.
- dynamically elaborated regions (for loop stmts and subpCall).
However support of dynamically elaborated regions is not included in the compliance level of hierarchy traversal.

UpperRegion: returns the higher enclosing structural instance or NULL if the reference handle is a rootinst or a packinst.
it mult * objDecl<-objDecl internal
it mult * decl<-decl vhpiDecls returns all declarations of clas vhpiDecl in the instance
vhpiImmRegion for a local ports/generics should return null as the component declaration is not a region.
it mult * attrSpec<-attrSpec the attribute specifications attributing that region (not the ones defined within that region)

it mult * attrDecl<-attrDecl
it mult * aliasDecl<-aliasDecl

Class reportStmt

Superclasses: seqStmt

one-to-one relationships:
  mult 0..1 expr<-SeverityExpr
  mult 1 expr<-ReportExpr

Class returnStmt

Superclasses: seqStmt

one-to-one relationships:
  mult 0..1 expr<-ReturnExpr

Class rootInst

Description:
represents the root of the instantiated design hierarchy (top level instance)

Superclasses: designInstUnit

Attributes:
p int   -NumGens
 number of generic declarations
p int   -NumPorts
 number of port declarations

one-to-one relationships:
  mult 1 vpidesign<-vpidesign internal
  mult 1 null<-UpperRegion
  mult 0..1 configDecl<-configDecl returns a null handle if default configuration applied
Iteration relationships:
it mult * varDecl<-varDecl
it mult * sigDecl<-sigDecl
it mult * portDecl<-portDecl
it mult * genericDecl<-genericDecl
it mult * constDecl<-constDecl
it mult * compInstStmt<-compInstStmt
it mult * blockStmt<-blockStmt
it mult * assocElem<-PortAssocs
it mult * assocElem<-GenericAssocs
Operations:
-vhpi_handle_by_index(itRel: vhpiOneToManyT, handle: vhpiHandleT, index: int) : vhpiHandleT

Class scalarTypeDecl

Superclasses: typeDecl

```
{
 enumTypeDecl
 intTypeDecl
 floatTypeDecl
 physTypeDecl
}

Attributes:
p bool   -IsAnonymous
  Anonymous types have a simple name of $anonymous

---------------------------------------------------------

Class secondaryUnit

Superclasses: designUnit

{
 archBody
 packBody
}

one-to-one relationships:
    mult 1 primaryUnit<-primaryUnit
---------------------------------------------------------

Class selectSigAssignStmt

Superclasses: sigAssignStmt

one-to-one relationships:
    mult 1 expr<-SelectExpr
Iteration relationships:
it  mult 1..* selectWaveform<-selectWaveform
---------------------------------------------------------

Class selectWaveform

Superclasses: base

Attributes:
p int   -LineNo
  line number of the waveform
p string   -FileName

Iteration relationships:
it  mult 1..* waveformElem<-waveformElem
it  mult 1..* choice<-choice
---------------------------------------------------------

Class selectedName

Superclasses: prefixedName, port, basicSignal, variable, signal, generic, constant

one-to-one relationships:
```

mult 1 simpleName<-Suffix
Operations:
#vhpi_get_value(handle: vhpiHandle, value: vhpi_value_p) : int
#vhpi_put_value(handle: vhpiHandle, value: vhpiValueT *, flags: vhpiPutValueModeT) : int

----

Class seqStmt

Superclasses: stmt

{
procCallStmt
assertStmt
waitStmt
reportStmt
ifStmt
caseStmt
loopStmt
nextStmt
returnStmt
exitStmt
nullStmt
varAssignStmt
simpleSigAssignStmt
}

one-to-one relationships:
mult 1 stmt<-Parent
mult 1 region<-ImmRegion

Class sigAssignStmt

Superclasses: eqProcessStmt

{
condSigAssignStmt
selectSigAssignStmt
simpleSigAssignStmt
}

Attributes:
p bool -IsTransport
p bool -IsGuarded

one-to-one relationships:
mult 0..1 sigDecl<-GuardSig If the signal assign statement is guarded by a guard signal, then returns the explicit or implicit guard signal declaration, otherwise returns null
mult 1 expr<-LhsExpr
mult 0..1 expr<-RejectTime

Class sigDecl

Description:

a signal declaration

Superclasses: objDecl, basicSignal, signal

Attributes:
p bool -IsGuarded
the signal declares a guard signal of the signal kind indicated in the declaration
p sigKindT -SigKind
signal kind register, bus or normal for signals, bus or normal for ports.

one-to-one relationships:
mult 0..1 funcDecl<-ResolFunc returns the resolution function handle if a resolution function is used to calculate the effective value of that signal: A resolution function handle will be returned if the signal declaration contains a resolution function or of the subtype declaration of that signal contains a resolution function.
mult 0..1 expr<-InitExpr
Iteration relationships:
it mult * selectedName<-selectedName
it mult * predefAttrName<-SigAttrs
it mult * indexedName<-indexedName
it mult * basicSignal<-basicSignal returns the basic signal as defined page 165 of the VHDL LRM lines 485- 490.
Note: an implicit GUARD signal is not a basic signal.
an explicit GUARD may have at most one basic signal.
signal attributes are not basic signals.
Operations:
-vhpi_put_value(handle: vhpiHandleT value: vhpiValueT * flags: int) : int
deposit a value as a force for this cycle or force until release, creates an event if requested
-vhpi_register_cb(cbdatap: vhpiCbDataT *, int flags: <unnamed>) : vhpiHandleT;

Class sigParamDecl

Superclasses: paramDecl, signal

Attributes:
p bool -IsGuarded
p bool -IsResolved
p modeT -Mode
mode is either vhpiIn, vhpiOut, vhpiInout, vhpiBuffer or vhpiLinkage
only signal class can be buffer or linkage mode.
A buffer signal can only have one source

one-to-one relationships:
mult 0..1 funcDecl<-ResolFunc
Operations:
-vhpi_put_value(handle: vhpiHandleT, value: vhpiValueT *, flags: vhpiPutValueModeT) : int

----

Class signal

Description:
A signal can either be:
a signal declaration or sub-part thereof, a predefined implicit signal attribute ('delayed, 'stable, 'quiet, 'transaction) or a signal parameter declaration.

Superclasses: contributor, interfaceElt

{
 sigDecl
 sigParamDecl
 selectedName
 indexedName
 predefAttrName
}

Attributes:
p bool   -IsForced
  true if the object has been externally forced by either
  vhpi_put_value or some other way.
  false otherwise

Iteration relationships:
it mult * callback<-callback
it mult * basicSignal<-basicSignal

Class simpAttrName

Superclasses: predefAttrName

Class simpleName

Superclasses: name

{
 objDecl
 aliasDecl
}

Class simpleSigAssignStmt

Superclasses: seqStmt, sigAssignStmt

Iteration relationships:
it mult 1..* waveformElem<-waveformElem

Class sliceName

Superclasses: prefixedName, basicSignal

one-to-one relationships:
    mult 1 constraint<-constraint A slice range can be a discrete subtype indication or a range
----

Class spec

Superclasses: base

{
 compConfig
 disconnectSpec
 attrSpec
}

one-to-one relationships:
    mult 1 lexicalScope<-lexicalScope

Class stackFrame

Superclasses: base

{
 subpCall
 eqProcessStmt
}

Attributes:
p int   -FrameLevel
  returns the stack frame level of this subprogram call.
  0 for lowest, -1 if unknown (subprogram not
  dynamically executing or inlined).

one-to-one relationships:
    mult 0..1 subpCall<-DownStack moving up and down stack frames,
vhpiDownStack returns null if no more stack frame
vhpiUpStack returns null if this no more stack frame

Class stmt

Description:
a sequential or concurrent statement

Superclasses: entityDesignator

{
 concStmt
 seqStmt
}

Attributes:
p int   -LineNo
  the line number of the concurrent or sequential statement.
p string   -FileName

p string   -LabelName
  The optional label name of the statement, null string if none.
  vhpiLabelNameP property for a for generate statement does not include the index of the for generate).

p bool   -IsSeqStmt
  returns true if the stmt is a sequential stmt, false otherwise

Iteration relationships:
it mult * pragma<-pragma internal
it mult * callback<-callback
it mult * attrSpec<-attrSpec The attribute specifications which are associated with the label of that statement.

Class stringLiteral

Superclasses: literal

Attributes:
p string   -StrVal
  The string value of the literal as it appears in the VHDL

Class subpBody

Description:
A subprogram body with its subprogram specification

Superclasses: reference, decl, lexicalScope

Attributes:
p bool   -IsForeign
p int   -BeginLineNo
  the line number of the begin keyword
p int   -EndLineNo
  the line number of the end keyword

one-to-one relationships:
    mult 1 subpDecl<-subpDecl returns the subprogram specification of the subprogram body
Iteration relationships:
it mult * stmt<-stmt returns explicit sequential statements from the subpDecl
it mult * spec<-spec returns the specification defined in the subprogram declaration (only Attribute specifications)
it mult * decl<-decl returns explicit declarations from the subpDecl
it mult * attrSpec<-attrSpec Returns the attribute specifications for that subprogram declaration
----

Class subpCall

Superclasses: reference, region, stackFrame

{
 funcCall

procCallStmt
}

Attributes:
p int -NumParams

one-to-one relationships:
mult 1 subpBody<-subpBody
mult 1 stackFrame<-CurStackFrame returns the current executing or suspended stack frame or null.
mult 0..1 stackFrame<-UpStack moving up and down stack frames,
vhpiDownStack returns null if no more stack frame
vhpiUpStack returns null if this no more stack frame
Iteration relationships:
it mult * varDecl<-varDecl
it mult * paramDecl<-paramDecl iteration over the dynamically elaborated formals (instantiated data) different from the paramDecl obtained from a subpDecl handle
it mult * constDecl<-constDecl
it mult * assocElem<-ParamAssocs iteration over the parameter association as they appear in the VHDL source it is not an ordered iteration
Operations:
-vhpi_handle_by_index(itRel: vhpiOneToManyT, handle: vhpiHandleT, index: int) : vhpiHandleT

Class subpDecl

Description:
The subprogram declaration either found alone as a declaration or as the subprogram specification of a subprogram body.

Superclasses: decl, lexicalScope

{
funcDecl
procDecl
}

Attributes:
p int -NumParams
the number of formal parameters in the subprogram declaration includes the return parameter for function declarations

one-to-one relationships:
mult 1 subpBody<-subpBody internal
Iteration relationships:
it mult * typeMark<-typeMark The signature of the subprogram either implicit or explicit
it mult * paramDecl<-paramDecl
it mult * paramDecl<-paramDecl iteration over uninstantiated parameter declarations
Operations:
-vhpi_handle_by_index(itRel: vhpiOneToManyT, handle: vhpiHandleT, index: int) : vhpiHandleT
returns the handle which corresponds to the index for the given iteration for the reference handle

----

Class subtype

Description:
a created subtype

Superclasses: base

{
subtypeIndic
typeMark
}

Attributes:
p bool -IsUnconstrained
The subtype is unconstrained

one-to-one relationships:
mult 1 typeDecl<-BaseType

Class subtypeDecl

Description:
a subtype declaration

Superclasses: decl, typeMark

Attributes:
p bool #IsResolved

one-to-one relationships:
mult 1 typeMark<-typeMark
mult 0..1 funcDecl<-ResolFunc returns a resolution function handle only if a resolution function is present in the subtype declaration.
Iteration relationships:
it mult * constraint<-constraint
it mult * attrSpec<-attrSpec

Class subtypeIndic

Superclasses: subtype, constraint

Attributes:
p bool #IsResolved
if the subtype indication has a resolution function associated with it.

one-to-one relationships:
mult 1 typeMark<-typeMark
mult 0..1 funcDecl<-ResolFunc returns a resolution function handle only if a resolution function is present in the subtype indication
Iteration relationships:
it mult * constraint<-constraint

Class tool

Description:
This represents the tool with which the VHPI application or models are interacting; such a tool is an elaborator or a simulator which provides the VHPI interface.

Superclasses: base

Attributes:
p string   -Name
  The tool vendor name: executable name which implements the VHPI interface.
p int   -Level
  the VHDL conformance level:
   0 : no VHPI support available
   1: VHPI level 1
   2: VHPI level 2
   3: advanced VHPI capabilities
p int   -VHDLversion
  The language VHDL version the tool is complaint with: 87, 93 or 99
p physT   -SimTimeUnit
  The simulator tool time unit
p physT   -Precision
  The simulator precision for representing TIME values.
p PhaseT   -Phase
  the phase : vhpiRegistrationPhase, vhpiAnalysisPhase, vhpiInitializationPhase, vhpiElaborationPhase vhpiSimulationPhase, vhpiTerminationPhase, vhpiSavePhase, vhpiRestartPhase, vhpiResetPhase
p string   -ToolVersion
  The tool release version number

Iteration relationships:
it mult * argv<-argv Iteration returns the argv[] command line arguments passed to the tool invocation in the order they were passed.

Class transaction

Superclasses: base

Attributes:
p bool   -IsNull

Operations:
-vhpi_get_value(handle: vhpiHandle, value: vhpi_value_p) : int
-vhpi_get_time(handle: vhpiHandle, time: vhpi_time_p) : bool

Class typeConv

Superclasses: primaryExpr

one-to-one relationships:

mult 1 expr<-expr returns the expression that is the object of the conversion
----

Class typeDecl

Superclasses: typeMark, decl

{
 fileTypeDecl
 scalarTypeDecl
 compositeTypeDecl
 accessTypeDecl
 protectedTypeDecl
}

Attributes:
p bool   #IsScalar
p bool   #IsComposite

Iteration relationships:
it mult * attrSpec<-attrSpec

Class typeMark

Description:
A type mark name that originated from a subtype declaration or type declaration

Superclasses: subtype

{
 typeDecl
 subtypeDecl
}

Class unaryExpr

Superclasses: expr

one-to-one relationships:
    mult 1 operator
<-operator

    mult 1 expr<-expr

Class uniformCollection

Description:
an homogeneous collection of handles. All handles gathered in this collection are of the same kind, or a collection of handles of this kind.

Superclasses: collection

{
 driverCollection
}

Class unitDecl

Description:
a unit declaration

Superclasses: decl

Attributes:
p phys   -PhysPosition
  the position number of this unit

one-to-one relationships:
    mult 1 physLiteral<-physLiteral

Class userAttrName

Superclasses: attrName

one-to-one relationships:
    mult 1 attrSpec<-attrSpec returns the attribute specification which defines the value of the userAttrName.

Class varAssignStmt

Superclasses: seqStmt

one-to-one relationships:
    mult 1 expr<-LhsExpr
    mult 1 expr<-RhsExpr

Class varDecl

Description:
a variable declaration

Superclasses: objDecl, variable

Attributes:
p bool   -IsShared
p bool   -IsProtectedType
  The variable has a protected type

one-to-one relationships:
    mult 0..1 expr<-InitExpr
    mult 1 derefObj<-DerefObj method only from a variable of an access type

Iteration relationships:
it mult * selectedName<-selectedName
it mult * indexedName<-indexedName
Operations:
-vhpi_put_value(handle: vhpiHandleT, value: vhpiValueT *, flags: vhpiPutValueModeT) : int
-vhpi_protected_call(varHdl: vhpiHandleT, userfct: vhpiUserFctT, user_data: void *)
-vhpi_register_cb(cbdatap: vhpiCbDataT *, int flags: <unnamed>) : vhpiHandleT;

Class varParamDecl

Superclasses: paramDecl, variable

Attributes:
p modeT   -Mode
  mode can be vhpiIn, vhpiOut, vhpiInOut

Operations:
-vhpi_put_value(handle: vhpiHandle, value: vhpi_value_p, flags: vhpiPutValueModeT) : int

----

Class variable

Superclasses: interfaceElt

{
 varDecl
 varParamDecl
 selectedName
 indexedName
}

Attributes:
p bool   -IsForced
  true if the object has been externally forced by either
  vhpi_put_value or some other way.
  false otherwise

Iteration relationships:
it mult * callback<-callback

Class waitStmt

Superclasses: seqStmt

one-to-one relationships:
    mult 0..1 expr<-CondExpr
    mult 0..1 expr<-TimeOutExpr
Iteration relationships:
it mult * name<-SigNames

Class waveformElem

Superclasses: base

Attributes:
p bool -IsUnaffected
true if the waveforms is the unaffected keyword.
if true then valExpr and timeExpr methods return null;
always false for sequential signal assignments

one-to-one relationships:
mult 0..1 expr<-TimeExpr
mult 0..1 expr<-ValExpr

Class whileLoop

Superclasses: iterScheme

one-to-one relationships:
mult 1 expr<-CondExpr
