vpiAutomaticScheme indicates the object is allocated as part of a frame or thread and has the lifetime of that frame or thread. **vpiDynamicScheme** indicates the object was allocated in dynamic memory and may be a class object or part thereof. For all other objects, **vpiAllocScheme** shall return **vpiOtherScheme**.

37.3.8 Managing transient objects

One may obtain a handle to an object during its lifetime, and it remains valid only as long as the object exists. For a static object, one may therefore keep its handle indefinitely. For a transient object, one may release its handle after use or expect that handle to be released and become invalid when the object ceases to exist.

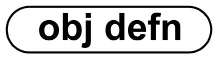
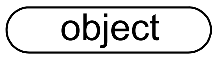
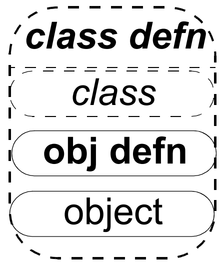
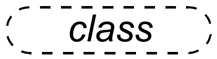
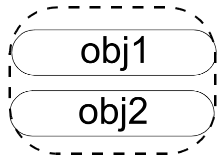
The life of a transient object may be tracked through various callbacks, depending on the specific type of object. The callbacks are described on the object model diagrams and/or the function reference for **vpi_register_cb()**, as appropriate. The relevant callbacks are as follows:

cbCreateObj, cbReclaimObj, cbStartofFrame, cbEndOfFrame, cbStartOfThread, cbEndOfThread, and cbEndOfObject.

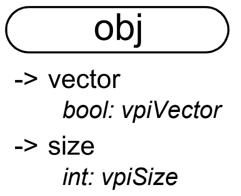
37.4 Key to data model diagrams

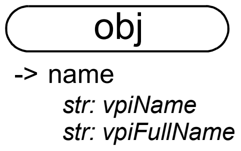
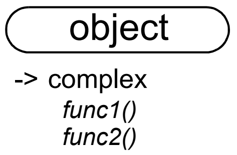
This subclause contains the keys to the symbols used in the data model diagrams. Keys are provided for objects and classes, traversing relationships, and accessing properties.

37.4.1 Diagram key for objects and classes

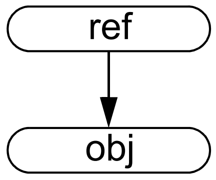
	Object definition: **Bold** letters in a solid enclosure indicate an object definition. The properties of the object are defined in this location.
	Object reference: Normal letters in a solid enclosure indicate an object reference.
	Class definition: ***Bold italic*** letters in a dotted enclosure indicate a class definition, where the class groups other objects and classes. Properties of the class are defined in this location. The class definition can contain an object definition.
	Class reference: *Italic* letters in a dotted enclosure indicate a class reference.
	Unnamed class: A dotted enclosure with no name is an unnamed class. It is sometimes convenient to group objects although they shall not be referenced as a group elsewhere; therefore, a name is not indicated.

37.4.2 Diagram key for accessing properties

 obj
-> vector
bool: vpiVector
-> size
int: vpiSize
	Integer and Boolean properties are accessed with the routine **vpi_get()**. These properties are of type `PLI_INT32`.
	
	For example: Given handle `obj_h` to an object of type **vpiObj**, test if the object is a vector, and get the size of the object.
	`PLI_INT32 vect_flag = vpi_get(vpiVector, obj_h);`
	`PLI_INT32 size = vpi_get(vpiSize, obj_h);`

	String properties are accessed with routine **vpi_get_str()**. String properties are of type `PLI_BYTE8 *`. For example: `PLI_BYTE8 *name = vpi_get_str(vpiName, obj_h);`
	Complex properties for time and logic value are accessed with the indicated routines. See the descriptions of the routines for usage.

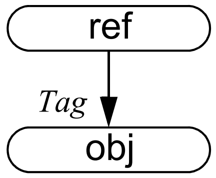
37.4.3 Diagram key for traversing relationships



A single arrow indicates a one-to-one relationship accessed with the routine **vpi_handle()**.

For example: Given **vpiHandle** variable `ref_h` of type `ref`, access `obj_h` of type *Obj*:

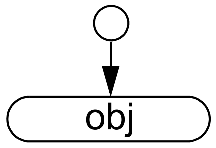
```
    obj_h = vpi_handle(Obj, ref_h);
```



A tagged one-to-one relationship is traversed similarly, using *Tag* instead of *Obj*.

For example:

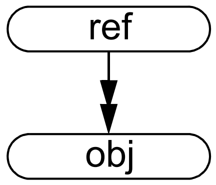
```
    obj_h = vpi_handle(Tag, ref_h);
```



A one-to-one relationship that originates from a circle is traversed using `NULL` for the `ref_h`.

For example:

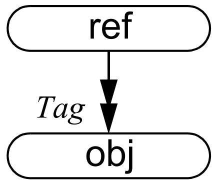
```
    obj_h = vpi_handle(Obj, NULL);
```



A double arrow indicates a one-to-many relationship accessed with the routine **vpi_scan()**.

For example: Given **vpiHandle** variable `ref_h` of type `ref`, scan objects of type *Obj*:

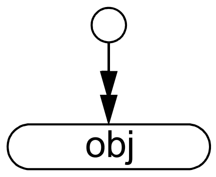
```
    itr = vpi_iterate(Obj, ref_h);
    while (obj_h = vpi_scan(itr) )
      /* process 'obj_h' */
```



A tagged one-to-many relationship is traversed similarly, using *Tag* instead of *Obj*.

For example:

```
    itr = vpi_iterate(Tag, ref_h);
    while (obj_h = vpi_scan(itr) )
      /* process 'obj_h' */
```



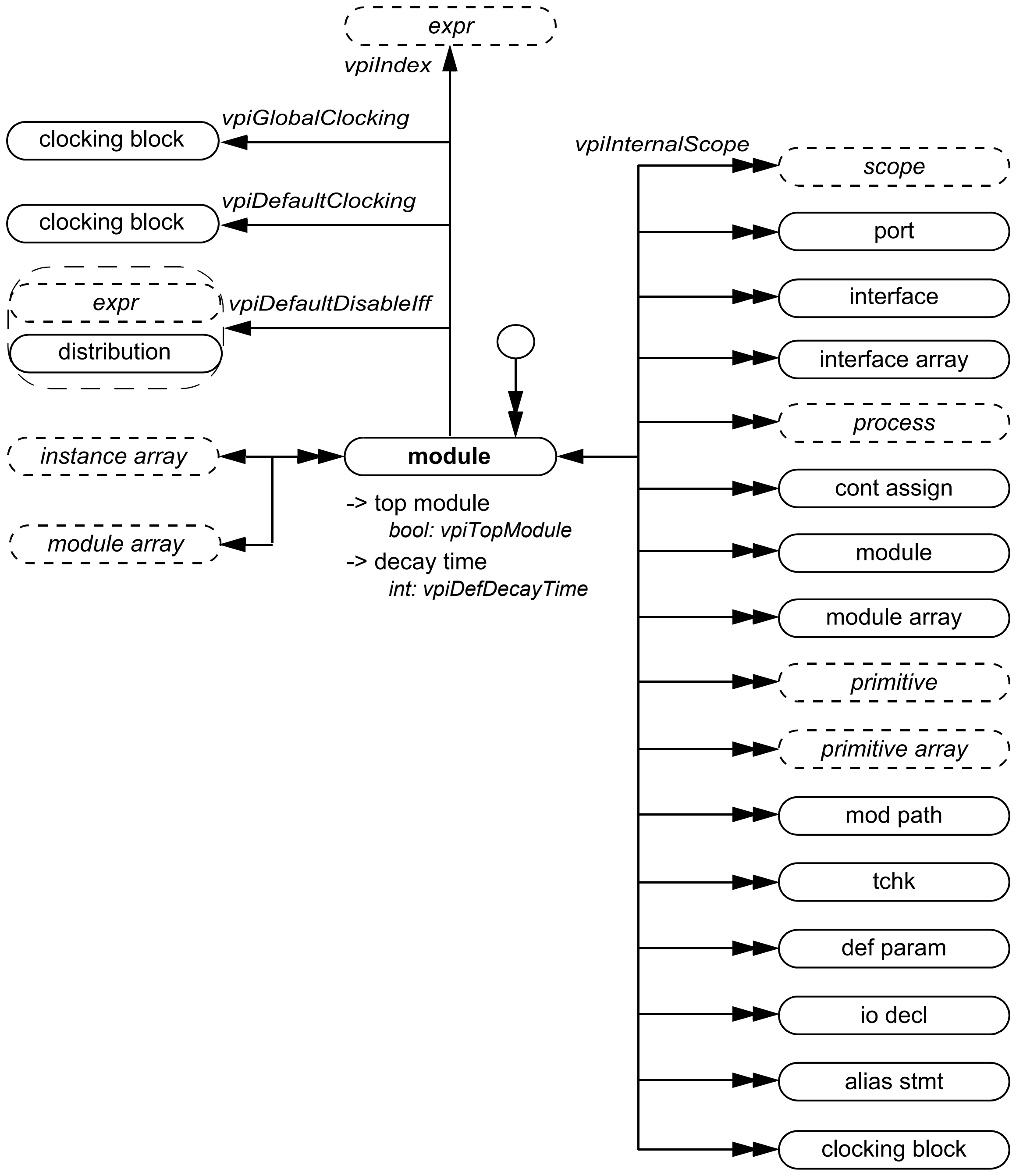
A one-to-many relationship that originates from a circle is traversed using `NULL` for the `ref_h`.

For example:

```
    itr = vpi_iterate(Obj, NULL);
    while (obj_h = vpi_scan(itr) )
      /* process 'obj_h' */
```

For relationships that do not have a tag, the type used for access is determined by adding “vpi” to the beginning of the word within the enclosure, with each word’s first letter being a capital. See 37.3 for more details on VPI access to constant names.

37.5 Module

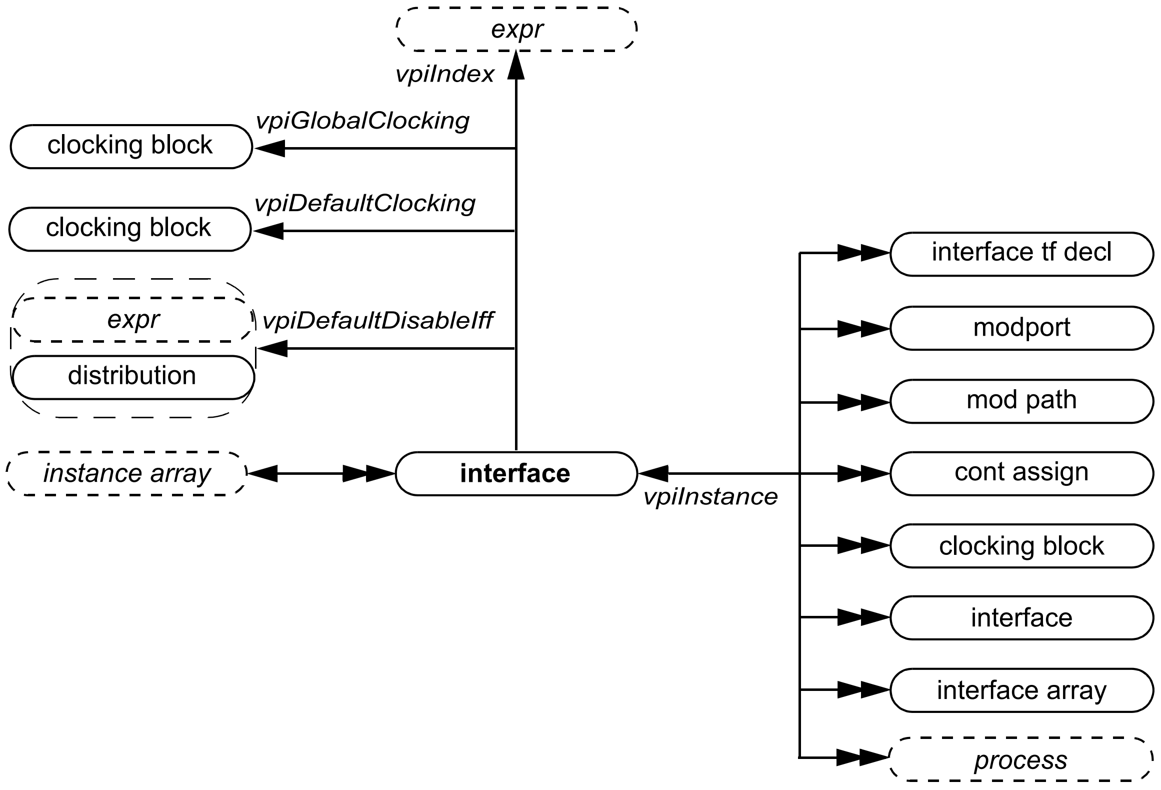


Details:

1) Top-level modules shall be accessed using **vpi_iterate**() with a `NULL` reference object.

2) If a module is an element within a module array, the **vpiIndex** transition is used to access the index within the array. If a module is not part of a module array, this transition shall return `NULL`.

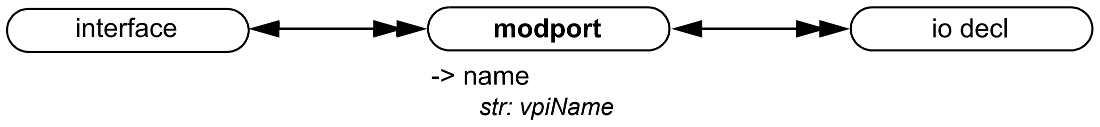
37.6 Interface



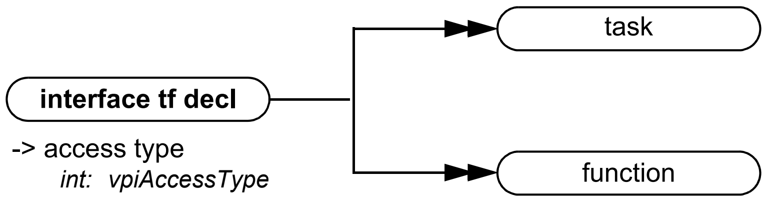
Details:

1) If an interface is an element within an instance array, the **vpiIndex** transition is used to access the index within the array. If an interface is not part of an instance array, this transition shall return NULL.

37.7 Modport



37.8 Interface task or function declaration

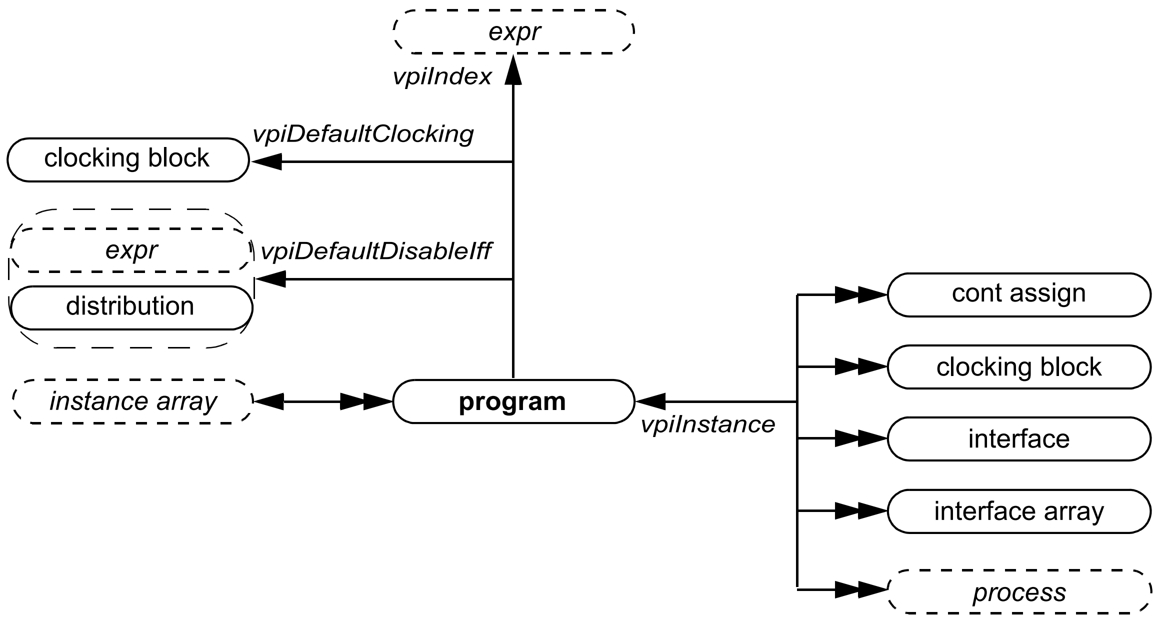


Details:

1) **vpi_iterate()** can return more than one task or function declaration for modport tasks or functions with an access type of **vpiForkJoinAcc**, because the task or function can be imported from multiple module instances.

2) Possible return values for the **vpiAccessType** property for an interface tf decl are **vpiForkJoinAcc** and **vpiExternAcc**.

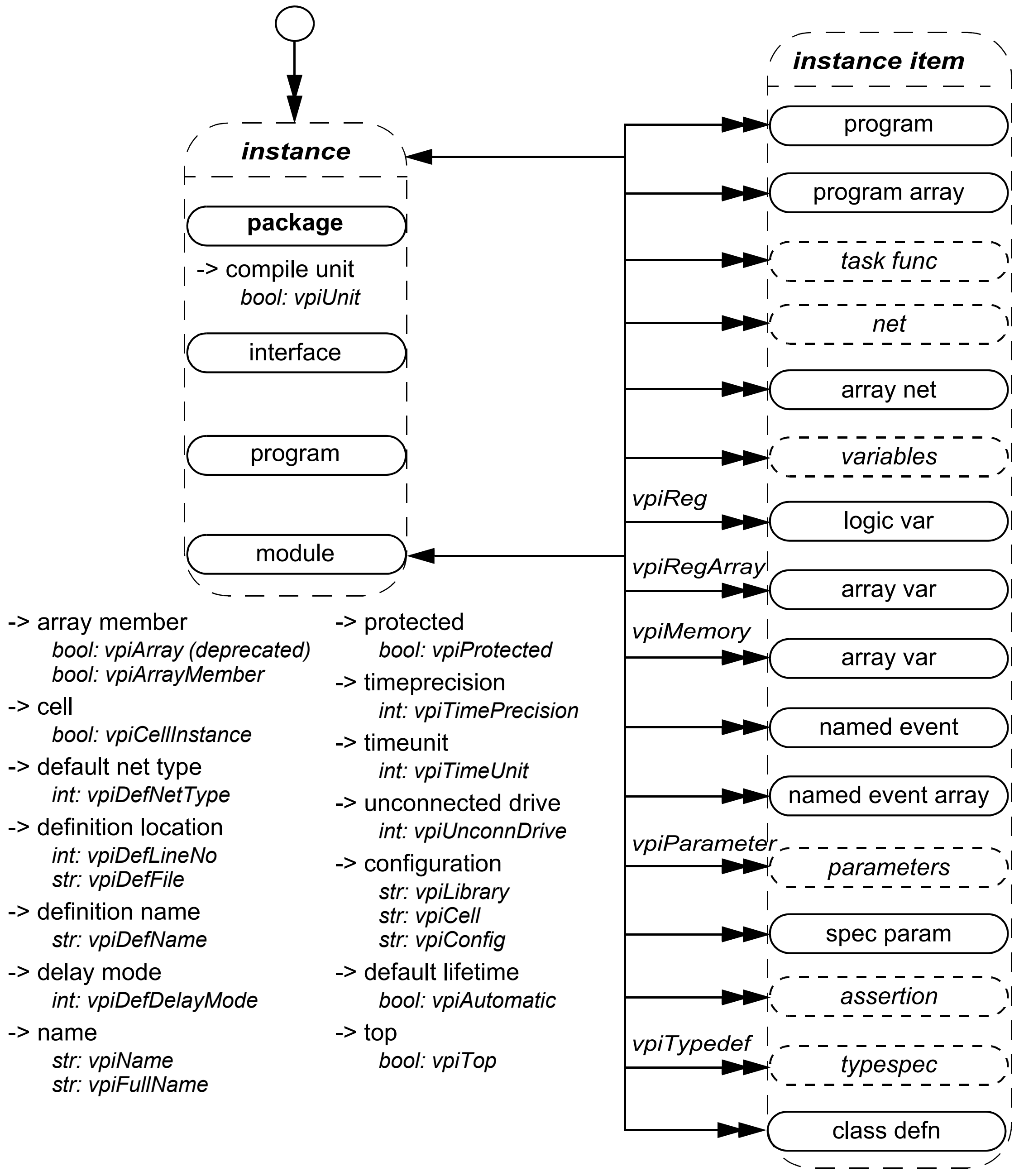
37.9 Program



Details:

1) If a program is an element within an instance array, the **vpiIndex** transition is used to access the index within the array. If a program is not part of an instance array, this transition shall return NULL.

37.10 Instance



Details:

1) The **vpiTypedef** iteration shall return the user-defined typespecs that have typedefs explicitly declared in the instance.
2) **vpiModule** shall return a module if the object is inside a module instance, otherwise it shall return `NULL`.
3) **vpiInstance** shall always return the immediate instance (package, module, interface, or program) in which the object is instantiated.
4) **vpiMemory** shall return array variable objects rather than **vpiMemory** objects.

5) **vpiFullName** for objects that exist within a compilation unit shall begin with "`$unit::`". As a result, the full name for objects within a compilation unit may be ambiguous. **vpiFullName** for a package shall be the name of the package and should end with "`::`"; this syntax disambiguates between a module and a package of the same name. **vpiFullName** for objects that exist in a package shall begin with the name of the package followed by "`::`". The separator `::` shall appear between the package name and the immediately following name component. The "`.`" separator shall be used in all cases except package and class defn.

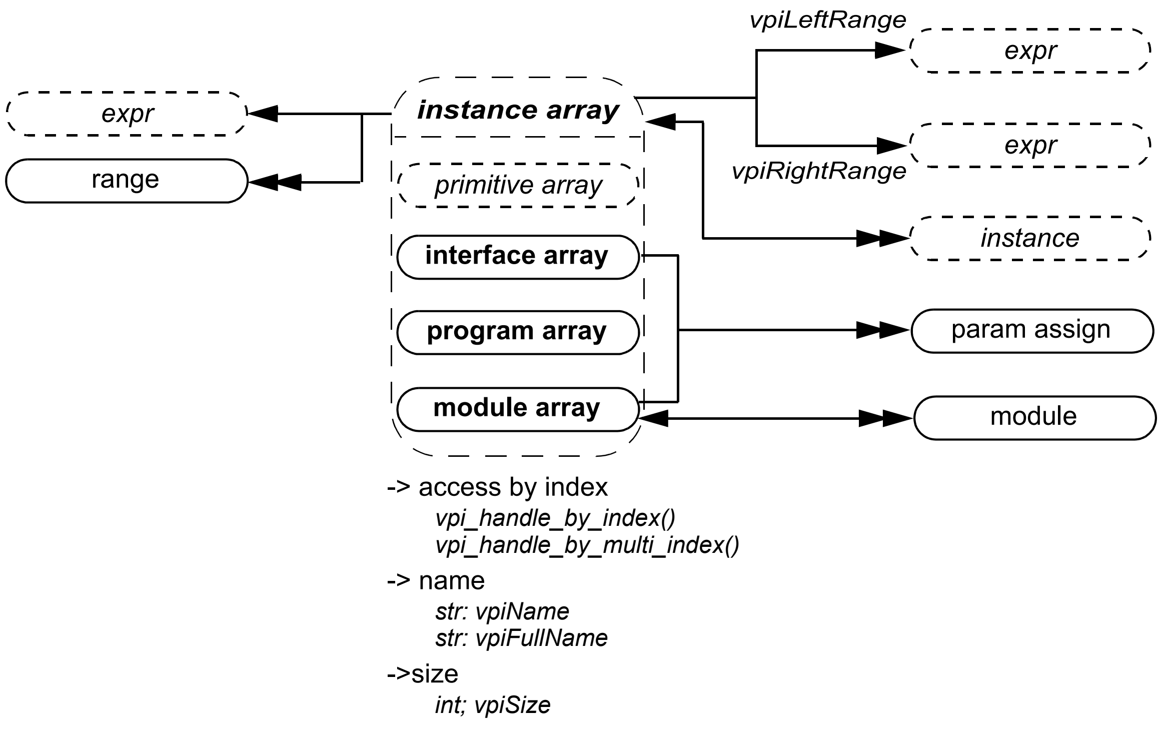
6) The following items shall not be accessible via `vpi_handle_by_name()`:
   - Imported items
   - Objects that exist within a compilation unit

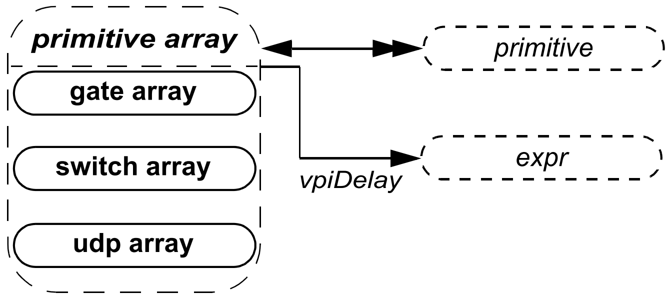
7) Passing a `NULL` handle to **vpi_get**() with properties **vpiTimePrecision** or **vpiTimeUnit** shall return the smallest time precision of all modules in the instantiated design.

8) The properties **vpiDefLineNo** and **vpiDefFile** can be affected by the `` `line `` compiler directive. See 22.12 for more details on the `` `line `` directive.

9) For details on lifetime and memory allocation properties, see 37.3.7.

37.11 Instance arrays



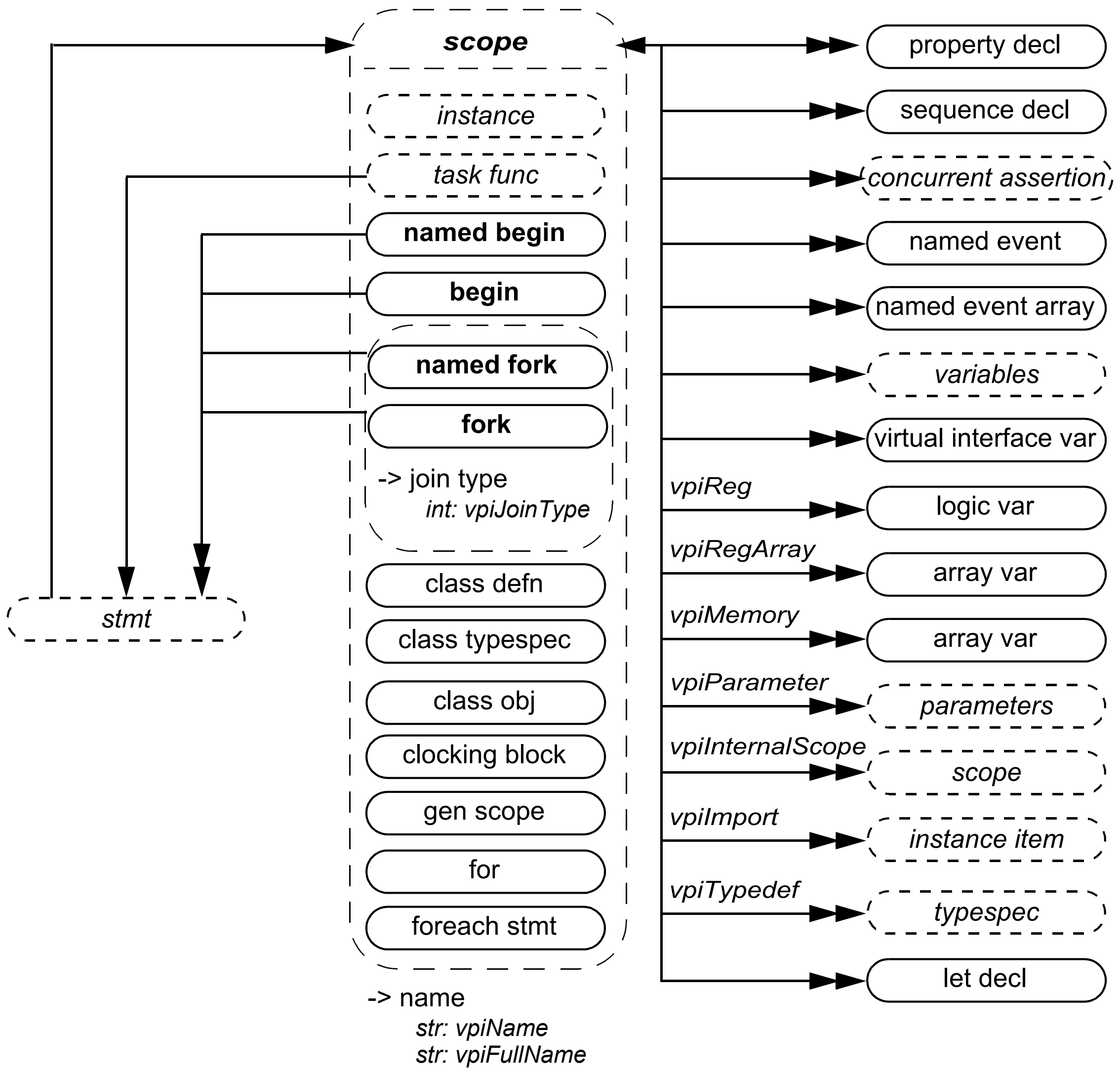


Details:

1) Traversing from the instance array to expr shall return a simple expression object of type **vpiOperation** with a **vpiOpType** of **vpiListOp**. This expression can be used to access the actual list of connections to the instance array in the SystemVerilog source code

2) **vpi_iterate(vpiRange, instance_array_handle)** shall return the set of instance array ranges beginning with the leftmost range of the array declaration and iterating through the rightmost range. Using the **vpiLeftRange**/ **vpiRightRange** properties returns the bounds of the leftmost dimension of a multidimensional array.

37.12 Scope



Details:

1) An unnamed begin or unnamed fork shall be a scope if, and only if, it directly contains a block item declaration such as a variable declaration or type declaration. A named begin or named fork shall always be a scope.

Example:

```
begin
   begin : BLK
      var logic v; // This declaration is not local to the unnamed begin
      v = 1'b1;
   end
end
```

In this example, the block BLK is a scope, but the unnamed begin is not a scope because it does not directly contain a block item declaration.

2) A for statement shall be a scope if, and only if, the **vpiLocalVarDecls** property returns TRUE. In this case, the scope of each loop control variable shall be the for statement.

3) The scope of each loop control variable in a foreach stmt shall be the foreach stmt.

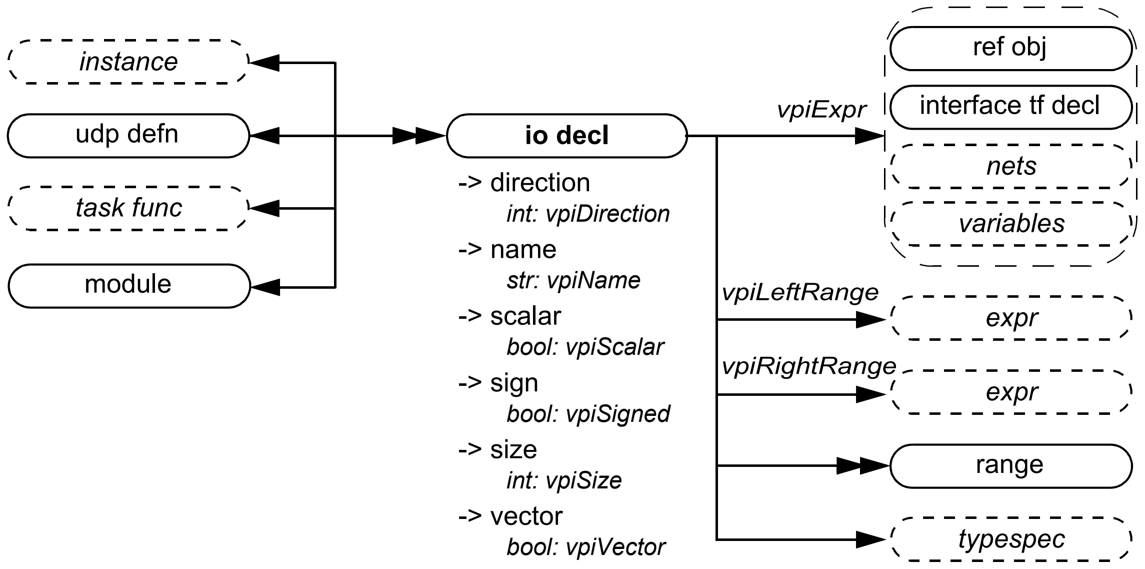
4) The **vpiImport** iterator shall return all objects imported into the current scope via import declarations. Only objects actually referenced through the import shall be returned, rather than items potentially made visible as a result of the import. Refer to 26.3 for more details.

5) A task func can have zero or more statements (see 13.3, 13.4). If the number of statements is greater than 1, the **vpiStmt** relation shall return an unnamed **`begin`** that contains the statements of the task or function. If the number of statements is zero, the **vpiStmt** relation shall return `NULL`.

6) The **vpiJoinType** property indicates what type of join statement terminates the fork-join block. It shall return one of the values **vpiJoin**, **vpiJoinNone**, or **vpiJoinAny**.

7) The **vpiVirtualInterfaceVar** iteration is supported only within elaborated contexts and is not supported within lexical contexts such as class defns (see 37.29). If the scope declares an array of virtual interfaces, the **vpiVirtualInterfaceVar** iteration shall return each element of the array separately. However, the **vpiVariables** iteration shall return the array declaration as a single **vpiArrayVar**.

37.13 IO declaration



Details:

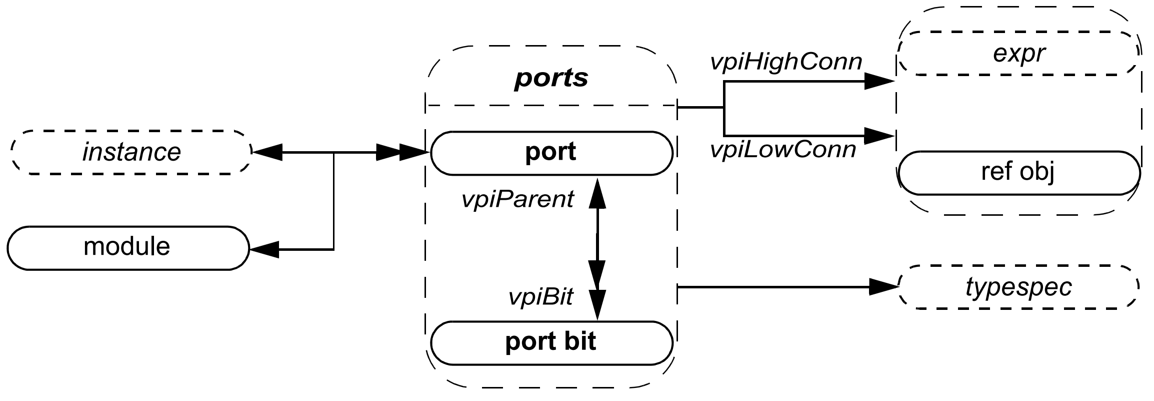
1) **vpiDirection** returns **vpiRef** for pass by **`ref`** ports or arguments.

2) A ref obj type handle shall be returned for the **vpiExpr** of an io decl if it is passed by reference or if the io decl is an interface or a modport. If the io decl is a virtual interface, **vpiExpr** shall return a **vpiVirtualInterfaceVar**.

3) If the **vpiExpr** of an io decl is a ref obj and if the **vpiActual** of the ref obj is an interface or modport declaration, then the **vpiDirection** of the io decl shall be undefined. The **vpiDirection** shall also be undefined if the **vpiExpr** is a virtual interface var.

4) The **vpiRange**, **vpiLeftRange**, and **vpiRightRange** relations for an io decl shall be the same as for the corresponding typespec (see 37.23).

37.14 Ports



-> access by index
  vpi_handle_by_index()
  vpi_handle_by_multi_index()
->connected by name
  bool: vpiConnByName
-> delay (mipd)
  vpi_get_delays()
  vpi_put_delays()
-> direction
  int: vpiDirection
-> explicitly named
  bool: vpiExplicitName

-> index
  int: vpiPortIndex
-> name
  str: vpiName
-> port type
  int: vpiPortType
-> scalar
  bool: vpiScalar
-> size
  int: vpiSize
-> vector
  bool: vpiVector

Details:

1) **vpiPortType** shall be one of the following three types: **vpiPort**, **vpiInterfacePort**, or **vpiModportPort.** Port type depends on the formal, not on the actual.

2) **vpi_get_delays()** and **vpi_put_delays()** delays shall not be applicable for **vpiInterfacePort.**

3) **vpiHighConn** shall indicate the hierarchically higher (closer to the top module) port connection.

4) **vpiLowConn** shall indicate the lower (further from the top module) port connection.

5) **vpiLowConn** of a **vpiInterfacePort** shall always be **vpiRefObj**.

6) Properties **vpiScalar** and **vpiVector** shall indicate if the port is 1 bit or more than 1 bit. They shall not indicate anything about what is connected to the port.

7) Properties **vpiIndex** and **vpiName** shall not apply for port bits.

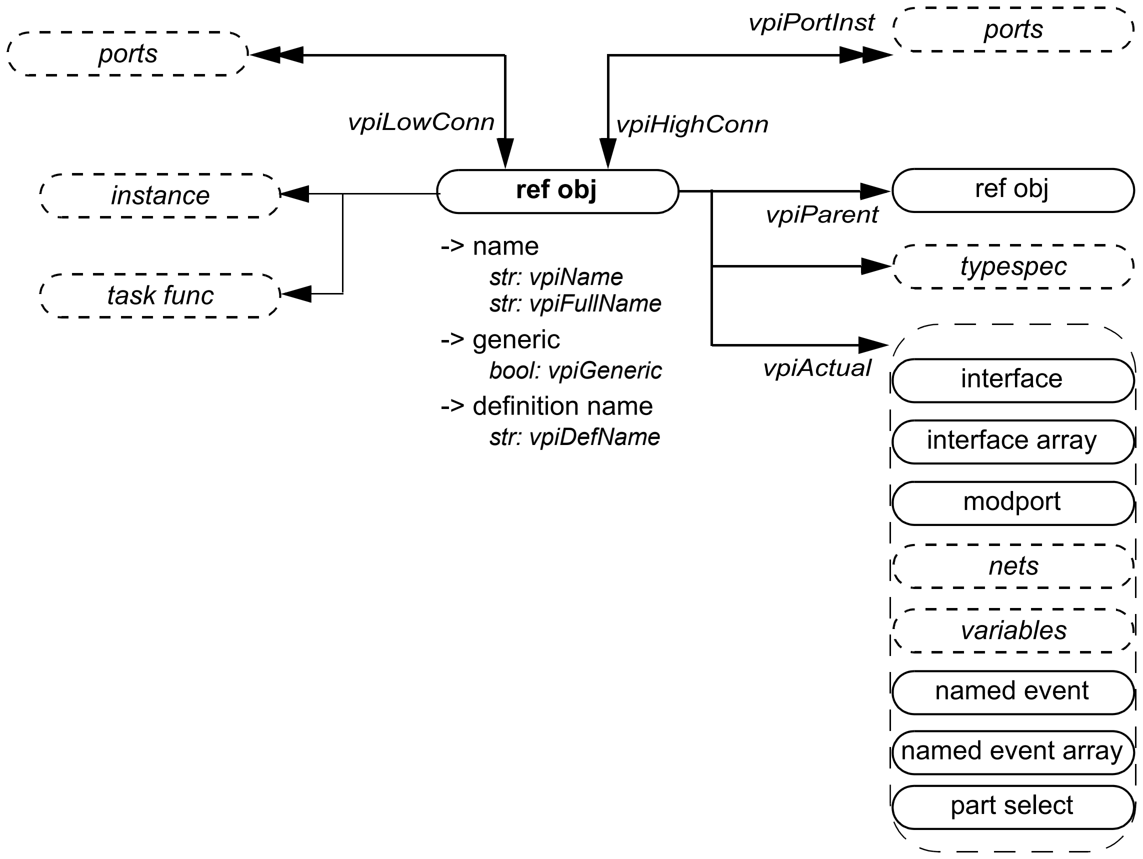
8) If a port is explicitly named, then the explicit name shall be returned. If not, and a name exists, then that name shall be returned. Otherwise, `NULL` shall be returned.

9) **vpiPortIndex** can be used to determine the port order. The first port has a port index of zero.

10) **vpiLowConn** shall return `NULL` if the module or interface or program port is a null port (e.g., "`module M();`"). **vpiHighConn** shall return `NULL` if the instance of the module, interface, or program does not have a connection to the port.

11) **vpiSize** for a null port shall return 0.

37.15 Reference objects



Details:

1) A ref obj represents a declared object or subelement of that object that is a reference to an actual instantiated object. A ref obj exists for ports with **ref** direction, for an interface port, a modport port, or for formal task function ref arguments. The specific cases for a ref obj are as follows:
   - — A variable, named event, named event array that is the lowconn of a ref port
   - — Any subelement expression of the above
   - — A local declaration of an interface or modport passed through a port or any net, variable, named event, named event array of those
   - — A ref formal argument of a task or function, or subelement expression of it
2) A ref obj may be obtained when walking port connections (lowConn, highConn), when traversing an expression that is a use of such ref obj, or when accessing the io decl of an instance or task or function.
3) The name of ref obj can be different at every instance level it is being declared. The **vpiActual** relationship always returns the actual instantiated object if the ref obj is bound to an actual object at the time of the query.
4) The **vpiParent** relationship allows the traversal of a ref obj that is a subelement of a ref obj. In the following example, `r[0]` is a ref obj whose parent is the ref obj `r`. The **vpiActual** for the ref obj `r[0]` would return the var bit `a[0]`, and the **vpiActual** of the ref obj `r` would return the variable `a`.

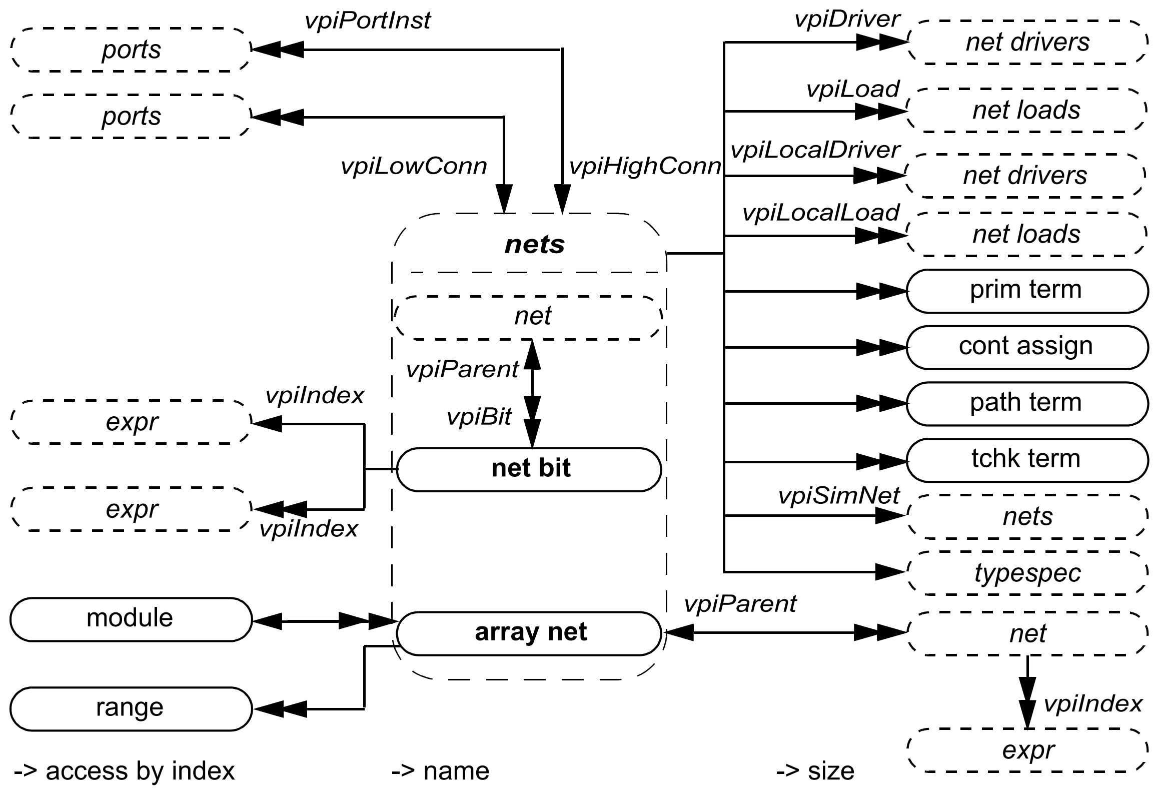
```
module top;
   logic [2:0] a;
   m u1 (a);
endmodule
module m (ref [2:0] r);
   initial
      r[0] = 1'b0;
endmodule
```

5) The **vpiGeneric** property shall return `TRUE` if the ref obj is a reference to a generic interface and `FALSE` if the ref obj is a reference to an interface that is not a generic interface. The **vpiGeneric** property shall return **vpiUndefined** for all other kinds of ref obj.
6) The **vpiDefName** property when applied to a ref obj that is an actual of an interface or modport shall return the interface definition name or modport name.
7) The **vpiTypespec** relation returns `NULL` for a ref obj that **vpiActual** is a not a net, variable, or part select.

Example: Passing an interface or modport through a port:

```
interface simple ();
   logic req, gnt;
   modport slave (input req, output gnt);
   modport master (input gnt, output req);
endinterface

module top();

   interface simple i;

   child1 i1(i);
   child2 i2(i.master);
endmodule

/**********************************
for the port of i1,
   the vpiHighConn relationship returns a handle of type vpiRefObj. The
   vpiActual relationship applied to the ref obj returns a handle of type
   vpiInterface.

for the port of i2 ,
   the vpiHighConn relationship returns a handle of type vpiRefObj. The
   vpiActual relationship applied to the ref obj returns a handle of type
   vpiModport.
**************************************/

module child1(interface simple s);
   c1 c_1(s);
   c1 c_2(s.master);
endmodule

/***************************
for the port of module child1,
   the vpiLowConn relationship returns a handle of type vpiRefObj. The
   vpiActual relationship applied to the ref obj returns a handle of type
   vpiInterface.
for that refObj,
   the vpiPort relationship returns the port of child1.
   the vpiPortInst iteration returns handles to s, s.master.
   the vpiActual relationship returns a handle to i.
for the port of instance c_1 :
   vpiHighConn returns a handle of type vpiRefObj. The vpiActual relationship
   applied to the ref obj handle returns a handle of type vpiInterface.
for the port of instance c_2 :
   vpiHighConn returns a handle of type vpiRefObj. The vpiActual relationship
   applied to the ref obj handle returns a handle of type vpiModport.
**************************************/
```

37.16 Nets



-> access by index
vpi_handle_by_index()
vpi_handle_by_multi_index()

-> array member
bool: vpiArray (deprecated)
bool: vpiArrayMember

-> constant selection
bool: vpiConstantSelect

-> delay
vpi_get_delays()

-> expanded
bool: vpiExpanded

-> implicitly declared
bool: vpiImplicitDecl

-> name
str: vpiName
str: vpiFullName

-> net decl assign
bool: vpiNetDeclAssign

-> net type
int: vpiNetType
int: vpiResolvedNetType

-> scalar
bool: vpiScalar

-> scalared declaration
bool: vpiExplicitScalared

-> sign
bool: vpiSigned

-> size
int: vpiSize

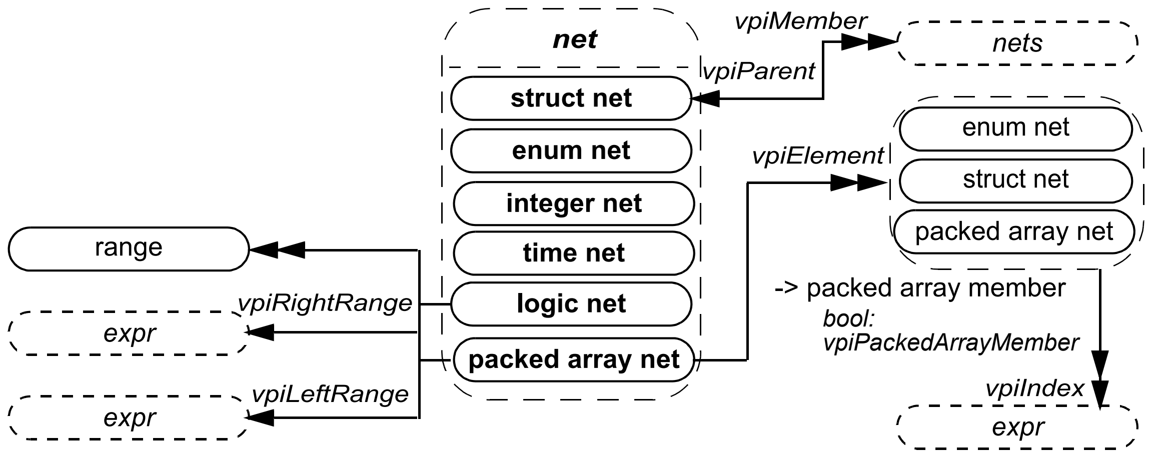
-> strength
int: vpiStrength0
int: vpiStrength1
int: vpiChargeStrength

-> value
vpi_get_value()
vpi_put_value()

-> vector
bool: vpiVector

-> vectored declaration
bool: vpiExplicitVectored

->member
bool: vpiStructUnionMember



Details:

1) Any net declared as an array with one or more unpacked ranges is an array net. Any packed struct net or enum net declared with one or more explicit packed ranges is a packed array net. The range iterator for a packed array net returns only the explicit packed ranges for such a net. It shall not return the implicit range of packed struct net elements themselves, nor shall it return the range (explicit or implicit) for the base type of enum net elements. For example:

```
// a 34-bit-wide struct net (range iteration not allowed)
wire struct packed { logic [1:0]vec1; integer i1; } psnet;

// a packed array net (ranges [3:0] and [2:1] returned by range iteration)
wire struct packed { logic [1:0]vec1; integer i1; } [3:0][2:1] panet;

// an array net (ranges [5:4] and [6:8] returned by range iteration)
wire struct packed { logic [1:0]vec1; integer i1; } [3:0][2:1] anet
[5:4][6:8];
```

2) The Boolean property **vpiArray** is deprecated in this standard. The **vpiArrayMember** property shall be TRUE for a net that is an element of an array net. It shall be `FALSE` otherwise. The **vpiPackedArrayMember** property shall be TRUE for a packed struct net, an enum net, or a packed array net that is an element of a packed array net.

3) For logic nets, net bits shall be available regardless of vector expansion.

4) Continuous assignments and primitive terminals shall be accessed regardless of hierarchical boundaries.

5) Continuous assignments and primitive terminals shall only be accessed from scalar nets or bit-selects.

6) For **vpiPorts**, if the reference handle is a net bit, then port bits shall be returned. If it is an entire net or array net, then a handle to the entire port shall be returned.

7) For **vpiPortInst**, if the reference handle is a bit or scalar, then port bits or scalar ports shall be returned, unless the highconn for the port is a complex expression where the bit index cannot be determined. If this is the case, then the entire port shall be returned. If the reference handle is an entire net or array net, then the entire port shall be returned.

8) For **vpiPortInst**, it is possible for the reference handle to be part of the highconn expression, but not connected to any of the bits of the port. This may occur if there is a size mismatch. In this situation, the port shall not qualify as a member for that iteration.

9) For implicit nets, **vpiLineNo** shall return 0, and **vpiFile** shall return the file name where the implicit net is first referenced.

10) **vpi_handle(vpiIndex, net_bit_handle)** shall return the bit index for the net bit. **vpi_iterate(vpiIndex, net_bit_handle)** shall return the set of indices for a multidimensional net array bit-select, starting with the index for the net bit and working outward.

11) Only active forces and assign statements shall be returned for **vpiLoad**.

12) Only active forces shall be returned for **vpiDriver**.

13) **vpiDriver** shall also return ports that are driven by objects other than nets and net bits.

14) **vpiLocalLoad** and **vpiLocalDriver** return only the loads or drivers that are local, i.e., contained by the module instance that contains the net, including any ports connected to the net (output and inout ports are loads, input and inout ports are drivers).

15) For **vpiLoad**, **vpiLocalLoad**, **vpiDriver**, and **vpiLocalDriver** iterators, if the object is a vector net (an enum net, integer net, time net, packed array net, or a logic net or struct net for which **vpiVector** is `TRUE)`, then all loads or drivers are returned exactly once as the loading or driving object. That is, if a part-select loads or drives only some bits, the load or driver returned is the part-select. If a driver is repeated, it is only returned once. To trace exact bit-by-bit connectivity, pass a **vpiNetBit** object to **vpi_iterate**.

16) An iteration on loads or drivers for a variable bit-select shall return the set of loads or drivers for whatever bit to which the bit-select is referring to at the beginning of the iteration.

17) **vpiSimNet** shall return a unique net if an implementation collapses nets across hierarchy (refer to 23.3.3.7 for the definition of simulated net and collapsed net).

18) The property **vpiExpanded** on an object of type **vpiNetBit** shall return the property's value for the parent.

19) The loads and drivers returned from **(vpiLoad, obj_handle)** and **vpi_iterate(vpiDriver, obj_handle)** may not be the same in different implementations, due to allowable net collapsing 23.3.3.7. The loads and drivers returned from **vpi_iterate(vpiLocalLoad, obj_handle)** and **vpi_iterate(vpiLocalDriver, obj_handle)** shall be the same for all implementations.

20) The Boolean property **vpiConstantSelect** shall return `TRUE` for a net or net bit if it has no parent (the **vpiParent** relation returns `NULL`) or if both of the following are true of the "select" part of the equivalent primary expression (see A.8.4):

— Every index expression in the select is an elaboration time constant expression.

— Every element within the select denotes either a member of a struct net or a member of a packed or unpacked array with static bounds.

Otherwise, **vpiConstantSelect** shall return FALSE.

NOTE—If **vpiConstantSelect** is TRUE, then if the handle refers to a valid underlying simulation object at the beginning of simulation (or at any point in the simulation), it refers to the same object at all points in the simulation. Moreover, if any index expression is in or out of bounds at the beginning of simulation, it is in or out of bounds at all subsequent simulation times as well.

21) **vpiSize** for an array net shall return the number of nets in the array. For unpacked structures, the size returned indicates the number of members in the structure. For an enum net, integer net, logic net, time net, packed struct net, or packed array net, **vpiSize** shall return the size of the net in bits. For a net bit, **vpiSize** shall return 1.

22) **vpi_iterate(vpiIndex, net_handle)** shall return the set of indices for a net within an array net, starting with the index for the net and working outward. If the net is not part of an array (the **vpiArrayMember** property is `FALSE`), a `NULL` shall be returned. The **vpiIndex** iterator shall work similarly for packed array net elements (packed struct nets, enum nets, or packed array nets whose **vpiPackedArrayMember** property is TRUE). The indices returned shall start with the index of the element and work outward until the **vpiParent** packed array net is reached (see detail 28). The indices retrieved for packed array net elements shall be the same as those shown in the example for detail 29 for each of the subelements returned by **vpiElement**. The indices will be retrieved in right-to-left order as they appear in the text.

23) For an array net, **vpi_iterate(vpiRange, handle)** shall return the set of array range declarations beginning with the leftmost unpacked range of the array declaration and iterating through the rightmost unpacked range. For a packed array (logic net), the iteration shall return the set of ranges beginning with the leftmost packed range and iterating through the rightmost packed range. For a logic net, the **vpiLeftRange** and **vpiRightRange** relations shall return the bounds of the leftmost packed dimension.

24) **vpiArrayNet** is #defined the same as **vpiNetArray** for backward compatibility. A call to **vpi_get_str(vpiType, <array_net_handle>)** may return either "vpiArrayNet" or "vpiNetArray".

25) A logic net without a packed dimension defined is a scalar; and for that object, the property **vpiScalar** shall return `TRUE` and the property **vpiVector** shall return `FALSE`. A logic net with one or more packed dimensions defined is a vector, and the property **vpiVector** shall return `TRUE` (**vpiScalar** shall return `FALSE`). Packed struct nets and packed array nets are vectors, and the property **vpiVector** shall return `TRUE` (**vpiScalar** shall return `FALSE`). A net bit is a scalar, and the property **vpiScalar** shall return `TRUE` (**vpiVector** shall return `FALSE`). The properties **vpiScalar** and **vpiVector** when queried on a handle to an enum net shall return the value of the respective property for an object for which the typespec is the same as the base typespec of the typespec of the enum net. For an integer net or a time net, the property **vpiVector** shall return `TRUE` (**vpiScalar** shall return `FALSE`). For an array net, the **vpiScalar** and **vpiVector** properties shall return the values of the respective properties for an array element. The **vpiScalar** and **vpiVector** properties shall return `FALSE` for all other net objects.

26) **vpiLogicNet** is #defined the same as **vpiNet** for backward compatibility. A call to **vpi_get_str(vpiType, <logic_net_handle>)** may return either "vpiLogicNet" or "vpiNet".

27) Neither an array net nor an unpacked struct net has a value property.

28) The **vpiParent** transition shall be allowed on all net objects. It shall return one of the following types of objects listed, representing one of its prefix objects (field select prefix or indexing select prefix as described in 11.5.3), or NULL, depending on whether certain criteria are met. For purposes of defining **vpiParent**, a prefix object is the object obtained from successively removing the rightmost index or identifier from a compound or indexed/multidimensional object name.

Consider the following **vpiArrayNet** objects:

```
wire logic [1:0][2:3] mda [4:6][6:8];
wire struct { int i1; logic[1:0][2:3]bvec[4:5]; } spa [9:11][12:13];
```

`mda[6][8][1][3]` is a **vpiLogicNet**, `mda[6][8][1]` is its first prefix object (a 2-bit **vpiLogicNet** vector), and `mda[6][8]` is its second prefix object (a 2 × 2 packed array **vpiLogicNet**), etc. The `spa[9][12].bvec[4]` object is a **vpiLogicNet** (a 2 × 2 packed array **vpiLogicNet**), and `spa[9][12].bvec` is its first prefix object (a **vpiArrayNet** struct member), and `spa[9][12]` is the second prefix object (the **vpiStructNet** containing the `bvec` member), etc.

For a net object with prefix objects, the **vpiParent** transition shall return one of the following prefix objects, whichever comes first in prefix order (rightmost to leftmost):
— Struct or union net
— Struct or union member net
— The largest containing packed array net object
— The largest containing unpacked array net object

If there is no prefix object, or no prefix object meets at least one of the above criteria, **vpiParent** shall return NULL.

Using the preceding declarations, the **vpiParent** of `mda[6][8][1][3]` is `mda[6][8]`, the **vpiLogicNet** representing the largest containing packed array prefix; the **vpiParent** of `mda[6][8]` is `mda`, the **vpiArrayNet** representing the largest containing unpacked array net prefix. Likewise, the **vpiParent** of `spa[9][12].bvec[4][0]` is `spa[9][12].bvec[4]` (the largest containing packed array net); the **vpiParent** of `spa[9][12].bvec[4]` is `spa[9][12].bvec` (struct member), and applying **vpiParent** again yields `spa[9][12]`, the struct net for member `bvec`. The **vpiParent** of `spa[9][12]` is `spa`, the largest containing unpacked array of the struct net; **vpiParent** of `spa` (or `mda`) would return NULL.

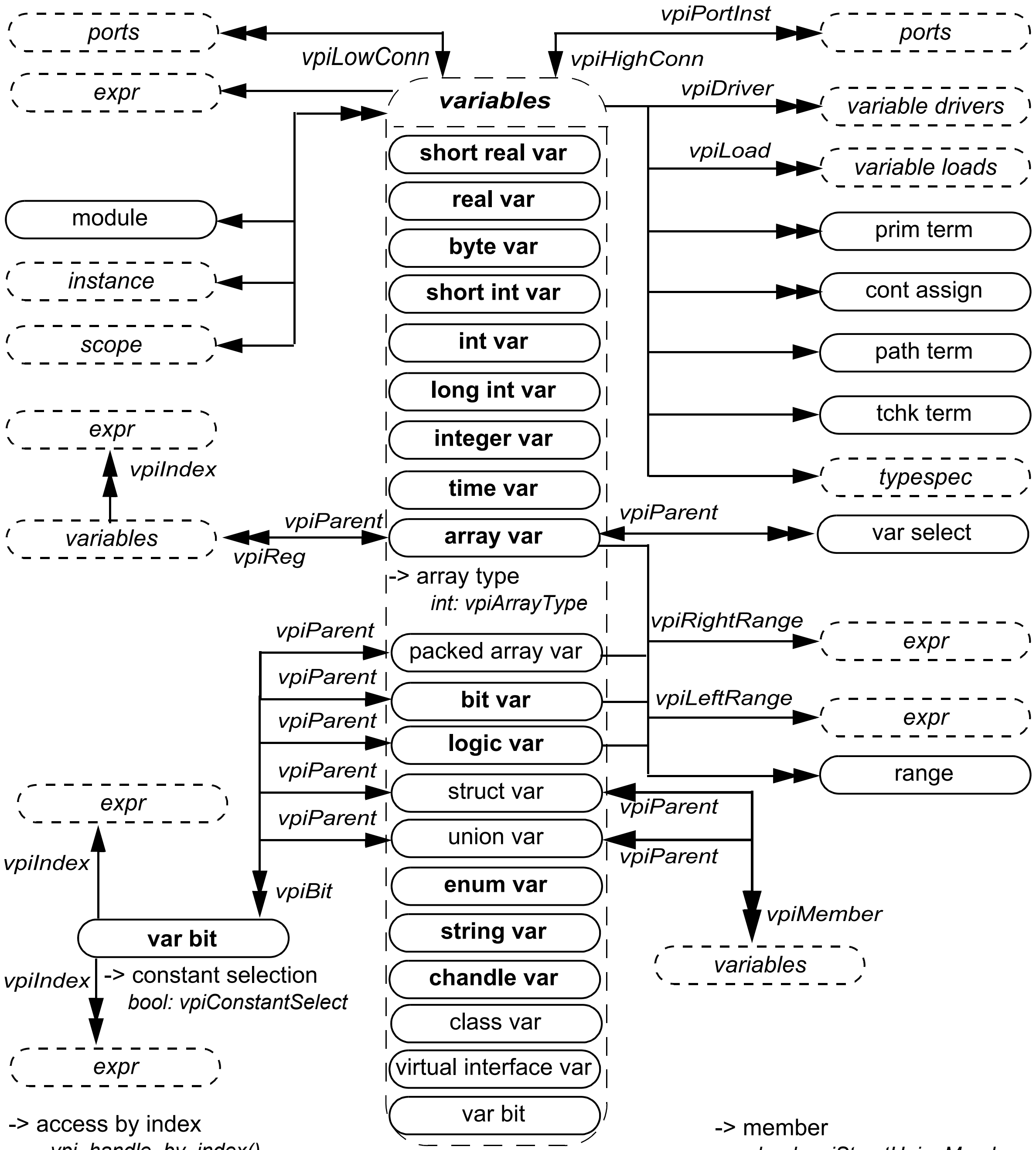
29) The **vpiElement** transition shall be used to iterate over the subelements of packed array nets. Unlike **vpiNet** iterations for **vpiArrayNet** objects, **vpiElement** shall retrieve elements for only one dimension level at a time. This means that for multidimensioned packed array nets, **vpiElement** shall retrieve elements that are themselves also **vpiPackedArrayNet** objects. **vpiElement** can then be used to iterate over the subelements of these objects and so on, until the leaf level struct nets or enum nets are returned. In other words, the data type of each element retrieved by **vpiElement** is equivalent to the original **vpiPackedArrayNet** object's data type with one leftmost packed range removed. For example, consider the following **vpiPackedArrayNet** object:

```
typedef struct packed { integer i1; logic [1:0][2:3]bvec; } pavartype;
wire pavartype [0:2][6:3] panet1;
```

The **vpiElement** transition applied to `panet1` shall return 3 **vpiPackedArrayNet** objects: `panet1[0]`, `panet1[1]`, and `panet1[2]`. The **vpiElement** transition applied to **vpiPackedArrayNet** `panet1[0]` in turn shall retrieve **vpiStructNet** objects `panet1[0][6]`, `panet1[0][5]`, `panet1[0][4]`, and `panet1[0][3]`, respectively. Also, the **vpiParent** transition for all the above-mentioned subelements of `panet1` shall return `panet1` (as per detail 28), since `panet1` is "the largest containing packed array net object."

30) The **vpiStructUnionMember** property shall be TRUE for any enum net, integer net, time net, struct net, packed array net, or array net that is a direct member of a struct net, i.e., whose **vpiParent** is a struct net (see detail 28). This property shall be FALSE for any net, array net, or net bit whose **vpiParent** is not a struct net.

37.17 Variables



-> access by index
  vpi_handle_by_index()
  vpi_handle_by_multi_index()
-> array member
  bool: vpiArray (deprecated)
  bool: vpiArrayMember
-> name
  str: vpiName
  str: vpiFullName
-> sign
  bool: vpiSigned
-> size
  int: vpiSize

-> lifetime
  bool: vpiAutomatic
-> memory allocation
  int: vpiAllocScheme
-> constant variable
  bool: vpiConstantVariable
-> determine random availability
  bool: vpiIsRandomized
-> randomization type
  int: vpiRandType

-> member
  bool: vpiStructUnionMember
->value
  vpi_get_value()
  vpi_put_value()
-> scalar
  bool: vpiScalar
-> visibility
  int: vpiVisibility
-> vector
  bool: vpiVector

Details:

1) Any variable declared as an array with one or more unpacked ranges is an array var.

2) The Boolean property **vpiArray** is deprecated in this standard. The Boolean property **vpiArrayMember** shall be TRUE if the referenced variable is a member of an array variable. It shall be FALSE otherwise.

3) To obtain the members of a union and structure, see the relations in 37.24.

4) For an array var, **vpi_iterate(vpiRange, handle)** shall return the set of array range declarations beginning with the leftmost unpacked range and iterating through the rightmost unpacked range. If any dimension of the unpacked array other than the first dimension is a dynamic array or queue dimension, the iteration shall return an empty range (see 37.22) for that dimension. The iteration shall also return an empty range for any dimension that is an associative array dimension. For a packed array, the iteration shall return the set of ranges beginning with the leftmost packed range and iterating through the rightmost packed range. The ranges returned for a packed array shall not include the implicit range for packed struct or union var elements themselves, or the range (explicit or implicit) for the base type of enum var elements.

5) **vpi_handle (vpiIndex, var_select_handle)** shall return the index of a var select in a one-dimensional array. **vpi_iterate (vpiIndex, var_select_handle)** shall return the set of indices for a var select in a multidimensional array, starting with the index for the var select and working outward.

6) The **vpiLeftRange** and **vpiRightRange** relations shall return the bounds of the leftmost packed dimension for a packed array and of the leftmost unpacked dimension for an unpacked array. If the unpacked array has no members,or the leftmost range corresponds to an empty range (see 37.22), **vpiLeftRange** and **vpiRightRange** shall return NULL.

7) A var select is an element selected from an array var.

8) If the variable has an initialization expression, the expression can be obtained from **vpi_handle(vpiExpr, var_handle)**.

9) **vpiSize** for a variable array shall return the number of variables in the array. For variables belonging to an integer data type (see 6.11), for enum vars, and for packed struct and union variables, **vpiSize** shall return the size of the variable in bits. For a string var, it shall return the number of characters that the variable currently contains. For unpacked structures and unions, the size returned indicates the number of fields in the structure or union. For a var bit, **vpiSize** shall return 1. For all other variables, the behavior of the **vpiSize** property is not defined.

10) **vpiSize** for a var select shall return the number of bits in the var select. This applies only for packed var select.

11) Variables of type **vpiArrayVar**, **vpiClassVar** or **vpiVirtualInterfaceVar** do not have a value property. Struct var and union var variables for which the **vpiVector** property is FALSE do not have a value property.

12) **vpiBit** iterator applies only for logic, bit, packed struct, packed union, and packed array variables.

13) **vpi_handle(vpiIndex, var_bit_handle)** shall return the bit index for the variable bit. **vpi_iterate(vpiIndex, var_bit_handle)** shall return the set of indices for a multidimensional variable bit select, starting with the index for the bit and working outwards.

14) **cbSizeChange** shall be applicable only for dynamic and associative arrays, for queues, and for string vars. If both value and size change, the size change callback shall be invoked first. This callback fires after the size change occurs and before any value changes for that variable. The value in the callback is the new size of the array.

15) The property **vpiRandType** returns the current randomization type for the variable, which can be one of **vpiRand**, **vpiRandC**, or **vpiNotRand**.

16) **vpiIsRandomized** is a property to determine whether a random variable is currently active for randomization.

17) When the **vpiStructUnionMember** property is TRUE, it indicates that the variable is a member of a parent struct or union variable. See also the relations in 37.24 and 37.18 detail 5.

18) If a variable is an element of an array (the **vpiArrayMember** property is TRUE), the **vpiIndex** iterator shall return the indexing expressions that select that specific variable out of the array. See 37.18 (and detail 6) for similar functionality available for elements of packed array vars.

19) In the preceding diagram:

    logic var == reg
    var bit == reg bit
    array var == reg array

    vpiVarBit is #defined the same as **vpiRegBit** for backward compatibility. However, a **vpiVarBit** can be an element of a **vpiBitVar** (2-state) or a **vpiLogicVar** (4-state), whereas **vpiRegBit** could only be an element of a **vpiReg** (4-state).

SystemVerilog treats **`reg`** and **`logic`** variables as equivalent in all respects. To allow for backward compatibility, a call to **vpi_get_str(vpiType, <logic_var_handle>)** may return either "vpiLogicVar" or "vpiReg". Similarly, **vpi_get_str(vpiType, <var_bit_handle>)** may return either "vpiVarBit" or "vpiRegBit", while **vpi_get_str(vpiType, <array_var_handle>)** may return either "vpiArrayVar" or "vpiRegArray".

20) A bit var or logic var, without a packed dimension defined, is a scalar and for those objects, the property **vpiScalar** shall return `TRUE`, and the property **vpiVector** shall return `FALSE`. A bit var or logic var, with one or more packed dimensions defined, is a vector, and the property **vpiVector** shall return `TRUE` (**vpiScalar** shall return `FALSE`). A packed struct var, a packed union var, and packed array var are vectors, and the property **vpiVector** shall return `TRUE` (**vpiScalar** shall return `FALSE`). A var bit is a scalar, and the property **vpiScalar** shall return `TRUE` (**vpiVector** shall return `FALSE`). The properties **vpiScalar** and **vpiVector** when queried on a handle to an enum var shall return the value of the respective property for an object for which the typespec is the same as the base typespec of the typespec of the enum var. For an integer var, time var, short int var, int var, long int var, and byte var, the property vpiVector shall return `TRUE` (**vpiScalar** shall return `FALSE`). For an array var, the **vpiScalar** and **vpiVector** properties shall return the values of the respective properties for an array element. The **vpiScalar** and **vpiVector** properties shall return `FALSE` for all other var objects.

21) **vpiArrayType** can be one of **vpiStaticArray**, **vpiDynamicArray**, **vpiAssocArray**, or **vpiQueue**.

22) **vpiRandType** can be one of **vpiRand**, **vpiRandC**, or **vpiNotRand**.

23) For details on lifetime and memory allocation properties, see 37.3.7.

24) **vpiVisibility** denotes the visibility (**`local`**, **`protected`**, or default) of a variable that is a class member. **vpiVisibility** shall return **vpiPublicVis** for a class member that is not **`local`** or **`protected`**, or for a variable that is not a class member.

25) A non-static data member of a class var does not have a **vpiFullName** property. The static data member of a class, referenced either via a class var or a class defn, has the **vpiFullName** property. It shall return a full name string representing the hierarchical path of the static variable through "class defn". For example:

```
module top;
   class Packet ;
      static integer Id ;
      ....
   endclass
   Packet p;
   c = p.Id;
   ....
```

The **vpiFullName** for `p.Id` is "`top.Packet::Id`".

26) The **vpiParent** transition shall be allowed on all variable objects. It shall return one of the following types of objects, representing one of its prefix objects (similar to the field select prefix or indexing select prefix as described in 11.5.3), or NULL, depending on whether certain criteria are met. For purposes of defining **vpiParent**, a prefix object is the object obtained from successively removing the rightmost index or identifier from a compound or indexed/multidimensional object name (excluding scope identifiers).

Consider the following **vpiArrayVar** objects:

```
logic [1:0][2:3] mda [4:6][6:8];
struct { int i1; bit [1:0][2:3]bvec[4:5]; } spa [9:11][12:13];
```

`mda[6][8][1][3]` is a **vpiVarBit**, `mda[6][8][1]` is its first prefix object (a 2-bit **vpiLogicVar** vector), and `mda[6][8]` is its second prefix object (a 2 x 2 **vpiLogicVar** packed array), etc. The `spa[9][12].bvec[4]` object is a **vpiBitVar** (a 2 x 2 **vpiBitVar** packed array), and `spa[9][12].bvec` is its first prefix object (a **vpiArrayVar** struct member), and `spa[9][12]` is the second prefix object (the **vpiStructVar** containing the `bvec` member). etc.

For a variable object with prefix objects, the **vpiParent** transition shall return one of the following prefix objects, whichever comes first in prefix order (rightmost to leftmost):
— Struct, union, or class variable
— Struct or union member variable, or class variable data member
— The largest containing packed array object

— The largest containing unpacked array object

If there is no prefix object, or no prefix object meets at least one of the above criteria, **vpiParent** shall return NULL.

Using the preceding declarations, the **vpiParent** of `mda[6][8][1][3]` is `mda[6][8]`, the **vpiLogicVar** representing the largest containing packed array prefix; the **vpiParent** of `mda[6][8]` is `mda`, the **vpiArrayVar** representing the largest containing unpacked array prefix. Likewise, the **vpiParent** of `spa[9][12].bvec[4][0]` is `spa[9][12].bvec[4]` (the largest containing packed array); the **vpiParent** of `spa[9][12].bvec[4]` is `spa[9][12].bvec` (struct member), and applying **vpiParent** again yields `spa[9][12]`, the struct variable for member `bvec`. The **vpiParent** of `spa[9][12]` is `spa`, the largest containing unpacked array of the struct variable; **vpiParent** of `spa` (or `mda`) would return NULL.

Class variables (as previously mentioned in the prefix object types) shall be returned as parent objects only when they are explicitly used to reference corresponding class data members in the design. A VPI handle to a data member that does not correspond to such an explicit reference in the design (e.g., a VPI handle to a data member derived from iterations on its **vpiClassObj** or **vpiClassDefn**) shall have a NULL parent.

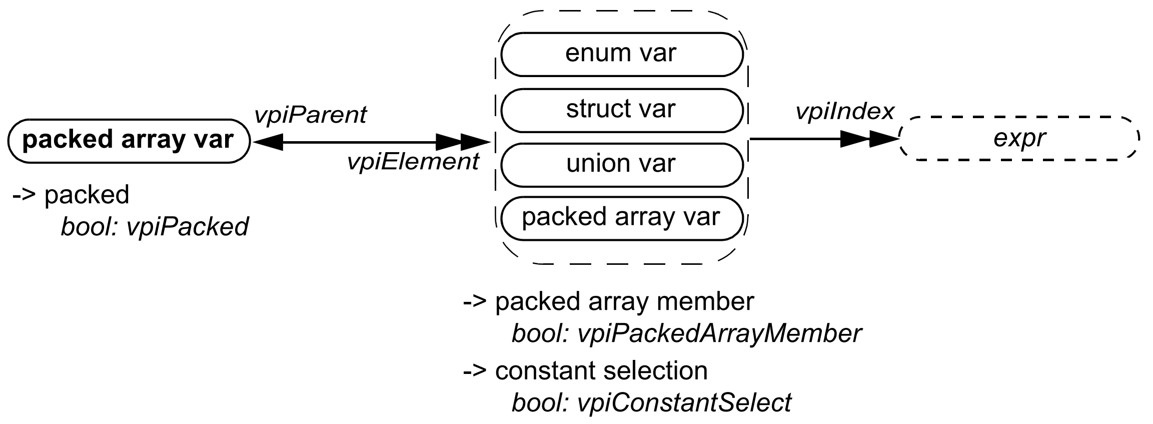
27) The property **vpiConstantSelect** shall return TRUE for a var bit or other variable if it has a static lifetime and has no parent (the **vpiParent** relation returns `NULL`) or if both of the following are true of the “select” part of the equivalent primary expression (see A.8.4):
   - — Every index expression in the select is an elaboration time constant expression.
   - — Every element within the select denotes either a member of a struct or union variable or a member of a packed or unpacked array with static bounds.

   Otherwise, **vpiConstantSelect** shall return FALSE.

   NOTE 1—The final (non-prefix) element of the select may be an unindexed member identifier belonging to any VPI variable type. It may, for example, be the name of a class variable or dynamic array. However, it must not be a member of a class variable if the member has an automatic lifetime, and it must not be an element of a dynamically allocated array.

   NOTE 2—If **vpiConstantSelect** is TRUE, then if the handle refers to a valid underlying simulation object at the beginning of simulation (or at any point in the simulation), it refers to the same object at all points in the simulation. Moreover, if any index expression is in or out of bounds at the beginning of simulation, it is in or out of bounds at all subsequent simulation times as well.

37.18 Packed array variables



Details:

1) **vpiPackedArrayVar** objects shall represent packed arrays of packed struct var, union var, or enum var objects. The properties **vpiVector** and **vpiPacked** for these objects and their underlying struct var, union var, or enum var elements shall always be TRUE (see 37.17).

2) For consistency with other variable-width vector objects, the **vpiSize** property for **vpiPackedArrayVar** objects shall be the number of bits in the packed array, not the number of struct, enum, or union var elements. The total

number of struct var, enum var, or union var elements for a packed array var can be obtained by computing the product of the **vpiSize** property for all of its packed ranges.

3) The **vpiElement** transition shall be used to iterate over the subelements of packed array variables. Unlike **vpiVarSelect** or **vpiReg** transitions for **vpiArrayVar** objects, **vpiElement** shall retrieve elements for only one dimension level at a time. This means that for multi-dimensioned packed arrays, **vpiElement** shall retrieve elements that are themselves also **vpiPackedArrayVar** objects. **vpiElement** can then be used to iterate over the subelements of these objects and so on, until the leaf level struct, enum, or union vars are returned. In other words, the data type of each element retrieved by **vpiElement** is equivalent to the original **vpiPackedArrayVar** object's data type with the leftmost packed range removed. For example, consider the following **vpiPackedArrayVar** object:

```
typedef struct packed { int i1; bit [1:0][2:3] bvec; } pavartype;
pavartype [0:2][6:3] pavar1;
```

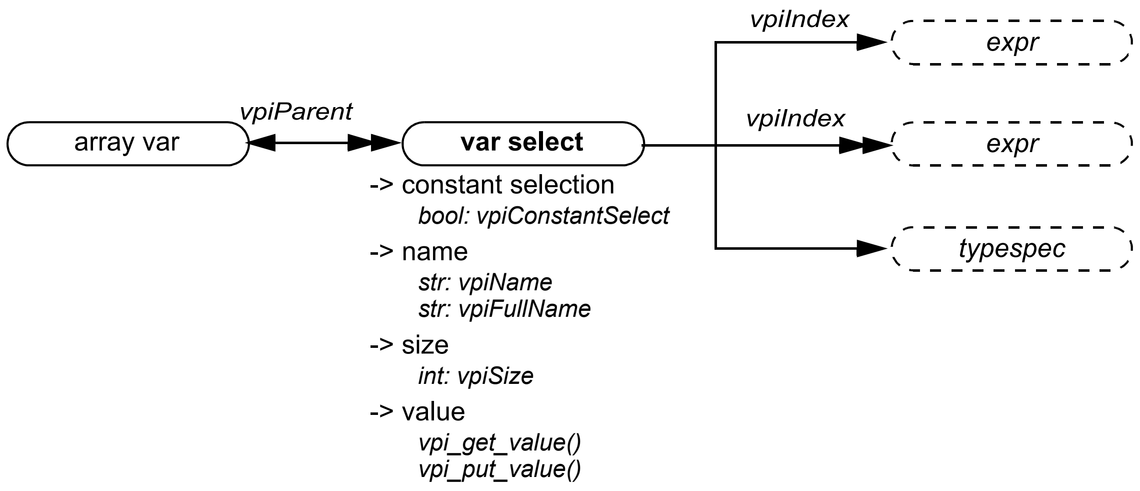
The **vpiElement** transition applied to `pavar1` shall return 3 **vpiPackedArrayVar** objects: `pavar1[0]`, `pavar1[1]`, and `pavar1[2]`. The **vpiElement** transition applied to **vpiPackedArrayVar** `pavar1[0]` in turn shall retrieve **vpiStructVar** objects `pavar1[0][6]`, `pavar1[0][5]`, `pavar1[0][4]`, and `pavar1[0][3]`, respectively. Also, the **vpiParent** transition for all the above-mentioned subelements of `pavar1` shall return `pavar1` (as per detail 26 of 37.17, since `pavar1` is "the largest containing packed array object").

4) The **vpiPackedArrayMember** property shall be TRUE for any struct var, union var, enum var, or packed array var whose **vpiParent** is a packed array var (see detail 26 of 37.17).

5) The **vpiStructUnionMember** property shall be TRUE only for packed array vars that are direct members of struct or union vars, i.e., whose **vpiParent** is a struct or union var (see detail 26 of 37.17). This property shall be FALSE for all subelements (as returned by the **vpiElement** iterator) of such packed array vars.

6) **vpi_iterate(vpiIndex, packed_array_var_handle)** shall return the set of indices for a subelement of a packed array variable (relative to its **vpiParent**), starting with the index for the subelement and working outwards. The indices retrieved shall be the same as those shown in the example for detail 3 for each of the subelements returned by **vpiElement**. The indices will be retrieved in right-to-left order as they appear in the text.

37.19 Variable select



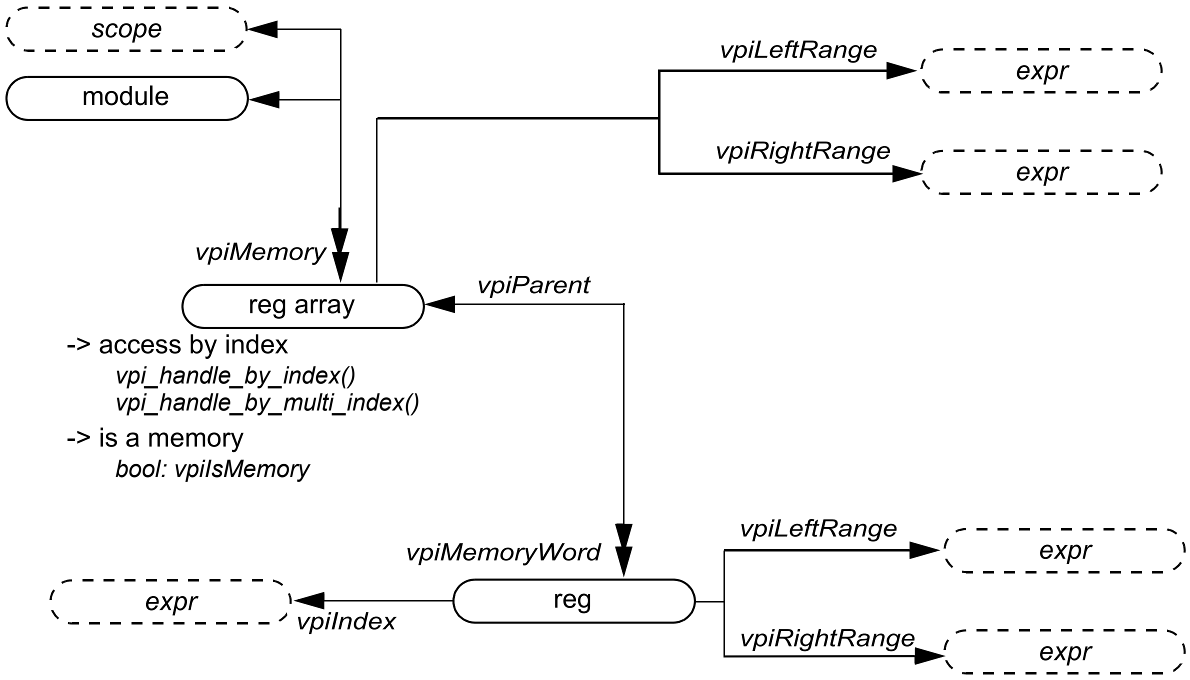
Details:

1) The property **vpiConstantSelect** shall return TRUE for a var select if
   — every associated index expression is an elaboration time constant expression, and
   — the parent of the var select is an unpacked array with static bounds, and
   — **vpiConstantSelect** returns TRUE for the parent of the var select.

   Otherwise, **vpiConstantSelect** shall return FALSE.

NOTE—If **vpiConstantSelect** is TRUE, then if the handle refers to a valid underlying simulation object at the beginning of simulation (or at any point in the simulation), it refers to the same object at all points in the simulation. Moreover, if an index expression of the var select or of any of its parents is in or out of bounds at the beginning of simulation, it is in or out of bounds at all subsequent simulation times as well.

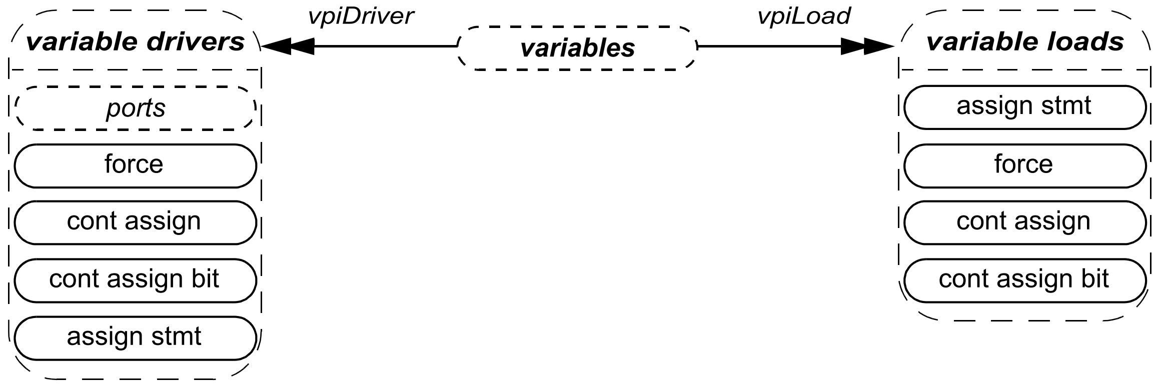
37.20 Memory



Details:

1) The objects **vpiMemory** and **vpiMemoryWord** have been generalized with the addition of arrays of variables. To preserve backwards compatibility, they have been converted into methods that will return objects of type **vpiRegArray** and **vpiReg**, respectively. See 37.17 for the definitions of variables and variable arrays.

37.21 Variable drivers and loads



Details:

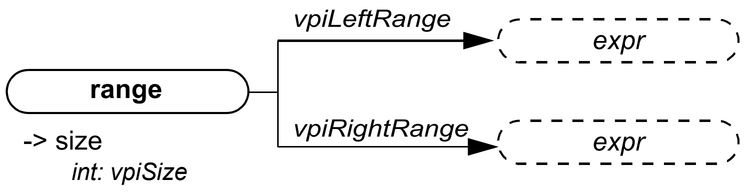
1) **vpiDrivers/Loads** for a structure, union, or class variable shall include the following:
    — Driver/Load for the whole variable

— Driver/Load for any bit-select or part-select of that variable

— Driver/Load of any member nested inside that variable

2) **vpiDrivers/Loads** for any variable array should include driver/load for entire array/vector or any portion of an array/vector to which a handle can be obtained.

37.22 Object Range



Details:

1) An empty range is a range that has no elements. An empty range shall be used to represent:
   - any range corresponding to an associative array dimension (see 37.17, detail 4)
   - a range corresponding to an empty dynamic array or queue
   - any range obtained from a typespec corresponding to a dynamic array, queue, or associative array dimension

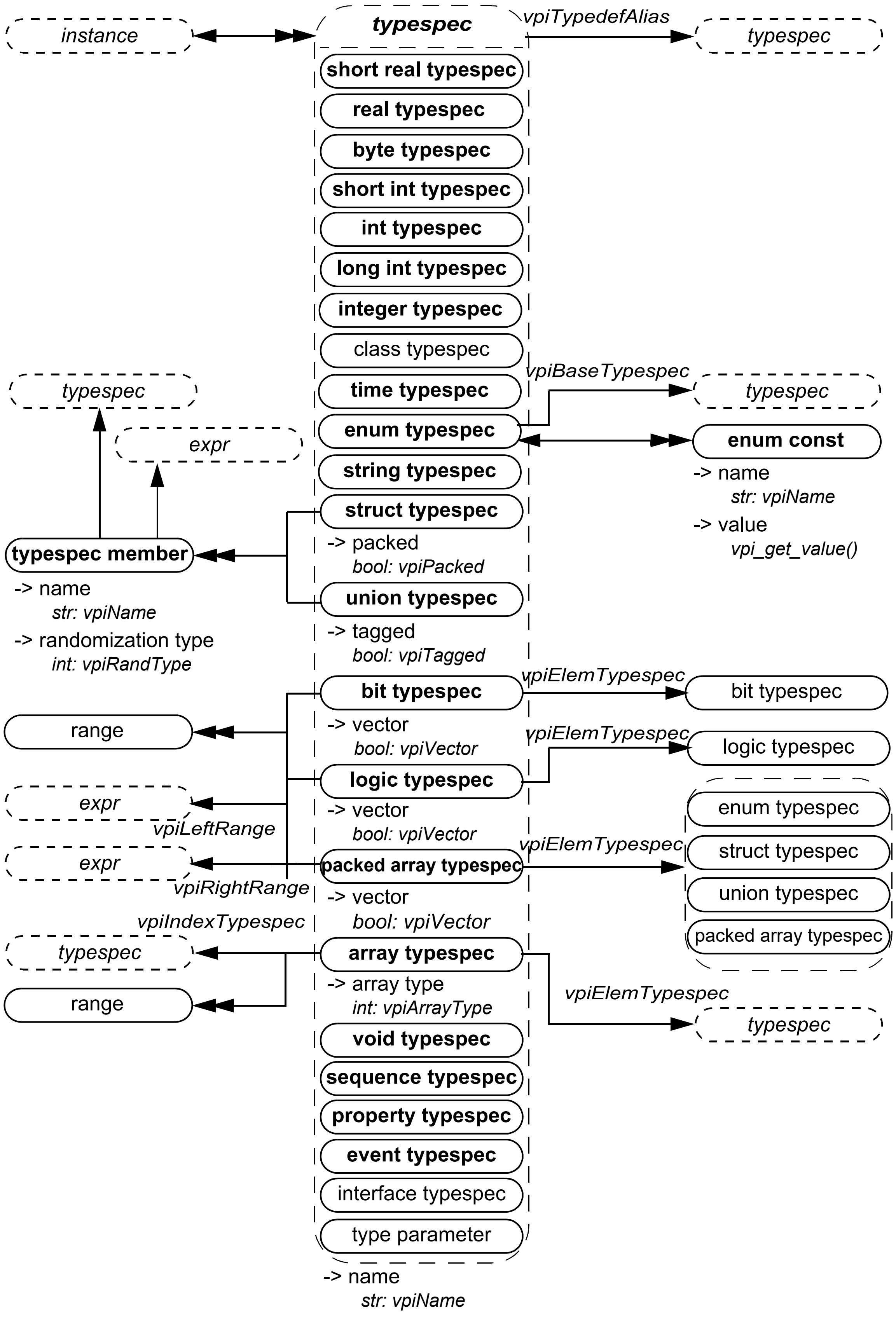
   For example:

```
int arr1 [][string];
initial
   begin
      #1 arr1 = new[2];
      #1 arr1[0]["hello"] = 5;
   end
```

   All ranges obtained from the typespec handle of `arr1` are empty. Also, ranges obtained from the `arr1` object itself at simulation time 0 are all empty, since the array is not sized yet. At times 1 and 2, the first range of `arr1` is `[0:1]` and the second is empty since it corresponds to an associative array dimension.

2) For an empty range, **vpiSize** shall return 0, while the **vpiLeftRange** and **vpiRightRange** relations shall each return `NULL`.

37.23 Typespec



Details:

1) If a typespec denotes a type that has a user-defined typedef, the **vpiName** property shall return the name of that type; otherwise, except in the case of a class typespec (see 37.30), the **vpiName** property shall return `NULL`. Consequently the **vpiName** property returns `NULL` for any SystemVerilog built-in type. If the typespec denotes a type with a typedef that creates an alias of another typedef, then the **vpiTypedefAlias** of the typespec shall return a non-null handle, which represents the handle to the aliased typedef. For example:

```
typedef enum bit [0:2] {red, yellow, blue} primary_colors;
typedef primary_colors colors;
```

If "`h1`" is a handle to the typespec `colors`, its **vpiType** shall return **vpiEnumTypespec**, the **vpiName** property shall return "colors," **vpiTypedefAlias** shall return a handle "`h2`" to the typespec "`primary_colors`" of **vpiType vpiEnumTypespec**. The **vpiName** property for "`h2`" shall return "`primary_colors`", and its **vpiTypedefAlias** shall return `NULL`.

2) **vpiIndexTypespec** relation is present only on associative array typespecs and returns the type that is used as the key into the associative array. For the wildcard index type (see 7.8.1), **vpiIndexTypespec** shall return `NULL`.

3) If the value of the property **vpiType** of a typespec is **vpiStructTypesec** or **vpiUnionTypespec**, then it is possible to iterate over **vpiTypespecMember** to obtain the structure of the user-defined type. For each typespec member, the typespec relation indicates the type of the member.

4) The property **vpiName** of a typespec member returns the name of the corresponding member, rather than the name (if any) of the associated typespec.

5) The name of a **typedef** may be the empty string if the typespec denotes typedef field defined in-line rather than via a typedef declaration. For example:

```
typedef struct {
   struct
      int a;
   } B
} C;
```

The typespec representing the typedef C is a struct typespec; it has a single typespec member named B. The typespec relation for B returns another struct typespec that has no name and has a single typespec member named "a". The typespec relation for "a" returns an int typespec.

6) If a type is defined as an alias of another type, it inherits the **vpiType** of this other type. For example:

```
typedef time my_time;
my_time t;
```

The **vpiTypespec** of the variable named "`t`" shall return a handle `h1` to the typespec "`my_time`" whose **vpiType** shall be a **vpiTimeTypespec**. The **vpiTypedefAlias** applied to handle `h1` shall return a typespec handle `h2` to the predefined type "**time**".

7) The expr associated with a typespec member shall represent the explicit default member value, if any, of the corresponding member of an unpacked structure data type (See 7.2).

8) The **vpiElemTypespec** transition shall be used to unwind the typespec of an unpacked array (array typespec) or a packed array (packed array typespec, or a bit or logic typespec with one or more dimensions), one dimension level at a time. This means that for a multidimensional array typespec (a typespec with more than one unpacked range), **vpi_handle(vpiElemTypespec, array_typespec_handle)** shall initially retrieve a **vpiArrayTypespec** equivalent to the original typespec with its leftmost unpacked range removed. Subsequent calls to the **vpiElemTypespec** method continue the unwinding until a typespec object is retrieved that has no unpacked ranges remaining. Similarly, when the **vpiElemTypespec** is applied to a typespec of a multidimensional packed array object, a **vpiPackedArrayTypespec** (or **vpiBitTypespec** or **vpiLogicTypespec**) is retrieved that is equivalent to the original typespec with its leftmost packed range removed, and so on, until a typespec without an explicit packed range is retrieved. When the **vpiElemTypespec** relation is applied to a **vpiStructTypespec**, **vpiUnionTypespec**, **vpiEnumTypespec**, or a **vpiBitTypespec** or **vpiLogicTypespec** with no ranges present, it shall return `NULL`. This allows packed or unpacked array typespecs constructed with multiple typedefs to be unwound without losing name information. Consider the complex array typespec defined below for `arr`:

```
typedef struct packed { int i1; bit bvec; } [1:3] parrtype;
typedef parrtype [2:1] parrtype2;
typedef parrtype2 unparrtype [6:4];
unparrtype arr [3:0];
```

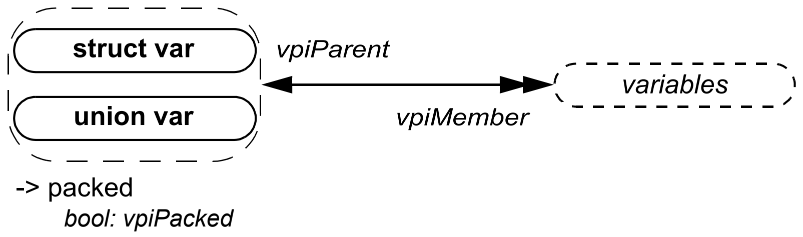
The typespec of the object `arr` is an unpacked 4 × 3 array typespec with a `NULL` **vpiName** property. The typespec retrieved by applying **vpiElemTypespec** to this is a 3-element unpacked array typespec with a **vpiName** property of "`unparrtype`". The typespec retrieved by using **vpiElemTypespec** on this in turn yields a 2 × 3 packed array typespec (of packed struct objects) with a **vpiName** property of "`parrtype2`". Using **vpiElemTypespec** again in turn yields another packed array typespec (of 3 packed struct objects) with a **vpiName** property of "`parrtype`". One more application of **vpiElemTypespec** to this result yields a struct typespec, a non-array typespec for which no further array subelements exist (the unwinding is done).

9) If a logic typespec, bit typespec, or packed array typespec has more than one packed dimension, **vpiLeftRange** and **vpiRightRange** shall return the bounds of the leftmost packed dimension. If an array typespec has more than one unpacked dimension, **vpiLeftRange** and **vpiRightRange** shall return the bounds of the leftmost unpacked dimension, unless that dimension corresponds to an empty range (see 37.22), in which case they shall return NULL.

10) For an array typespec, **vpi_iterate(vpiRange, handle)** shall return the set of array range declarations beginning with the leftmost unpacked range and iterating through the rightmost unpacked range. If any dimension of the array typespec corresponds to a dynamic array, associative array, or queue, the iteration shall return an empty range (see 37.22) for that dimension. For a logic typespec or bit typespec that has an associated range, the iteration shall return the set of ranges beginning with the leftmost packed range and iterating through the rightmost packed range.

11) In a context (such as a class defn) in which a type parameter has not been resolved, the type parameter itself shall act as a typespec.

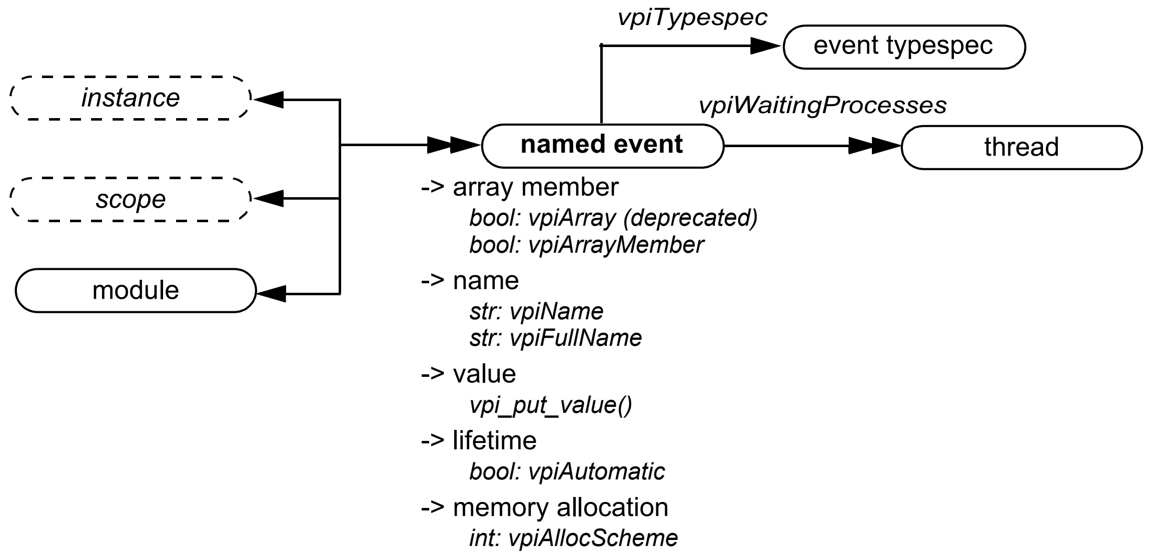
37.24 Structures and unions



Details:

1) **vpi_get_value()/vpi_put_value()** cannot be used to access values of entire unpacked structures and unpacked unions.

37.25 Named events

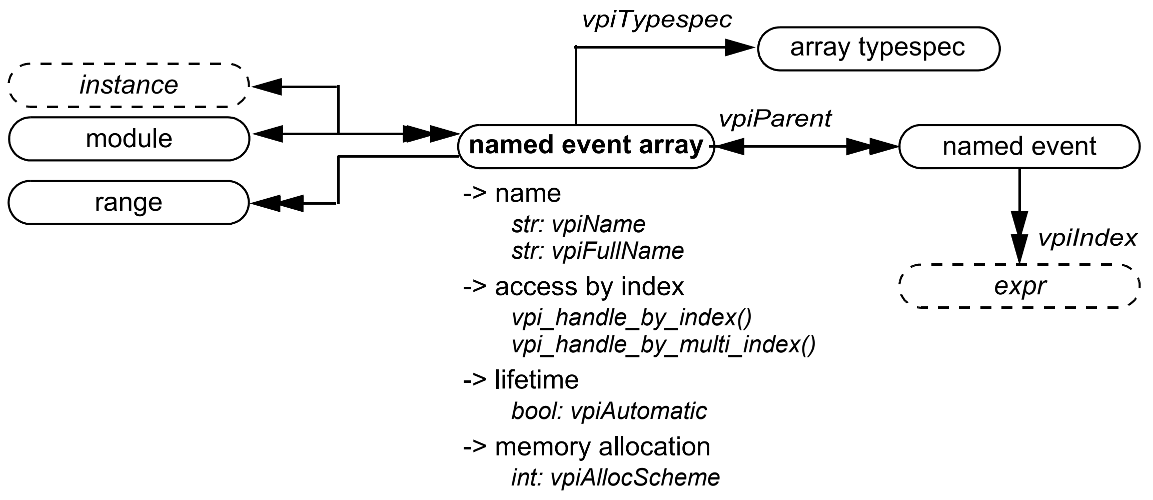


Details:

1) The **vpiWaitingProcesses** iterator returns all waiting processes, static or dynamic, identified by their thread, for that named event.

2) **vpi_iterate(vpiRange, named_event_array_handle)** shall return the set of array range declarations beginning with the leftmost unpacked range and iterating through the rightmost unpacked range.

3) For details on lifetime and memory allocation properties, see 37.3.7.

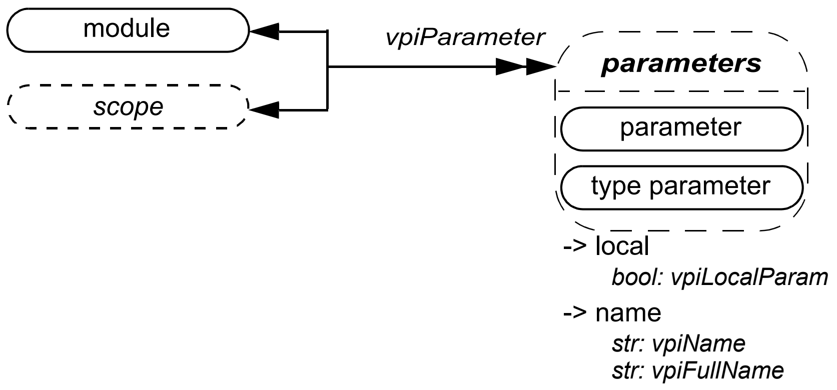


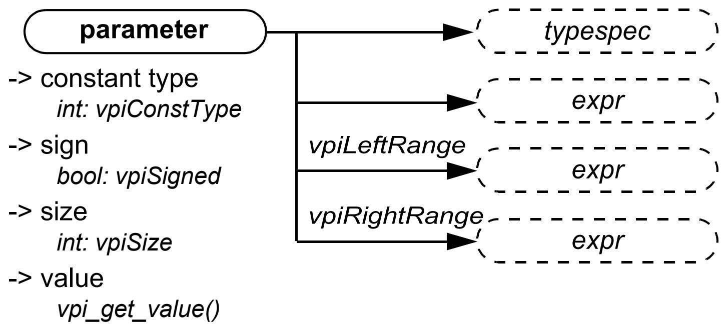
Details:

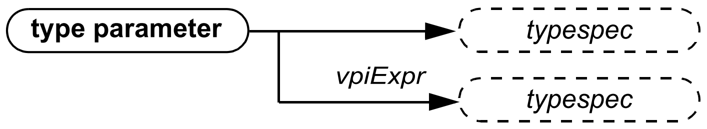
1) **vpi_iterate(vpiIndex, named_event_handle)** shall return the set of indices for a named event within an array, starting with the index for the named event and working outward. If the named event is not part of an array, a `NULL` shall be returned.

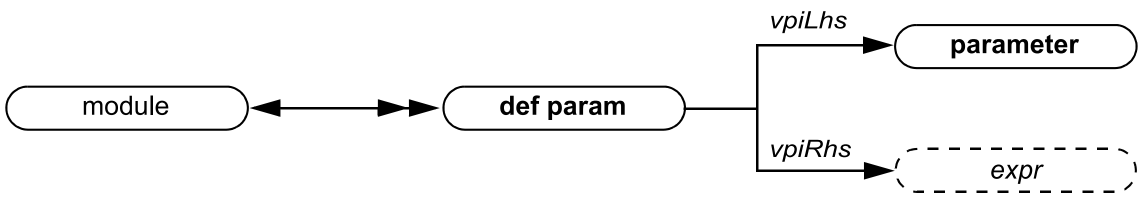
2) For details on lifetime and memory allocation properties, see 37.3.7.

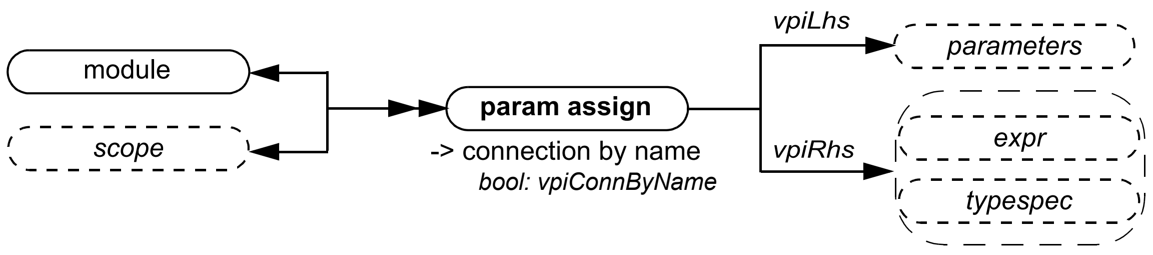
37.26 Parameter, spec param, def param, param assign











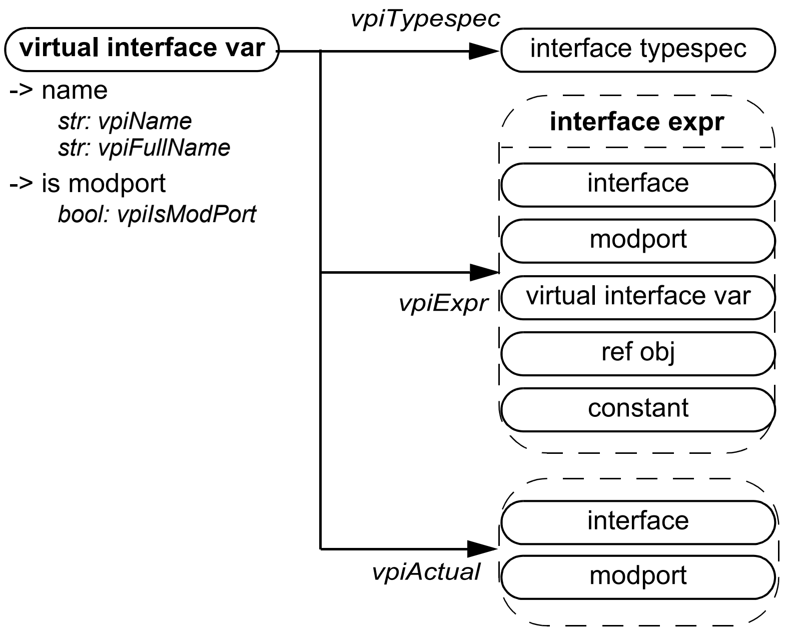
Details:

1) For a value parameter, **vpi_get_value()** shall return the value that the parameter has at the end of elaboration.

2) The **vpiTypespec** of a type parameter shall return the typespec that the type parameter has at the end of elaboration, but without resolving typedef aliases.

3) The **vpiExpr** relation of a value parameter shall return the default expr, while the **vpiExpr** relation of a type parameter shall return the default typespec.
4) **vpiLhs** from a param assign object shall return a handle to the overridden value parameter or type parameter.
5) If a value parameter does not have an explicitly defined range, **vpiLeftRange** and **vpiRightRange** shall return a `NULL` handle.

37.27 Virtual interface



Details:

1) The **vpiExpr** relation shall return the interface instance assigned to the virtual interface in its declaration, if any; otherwise, **vpiExpr** shall return NULL.
2) A ref obj may be an interface expr only if it is a local declaration of an interface or modport passed through a port. A constant may be an interface expr only if it has a **vpiConstType** of **vpiNullConst**.

Example 1: Passing an interface or modport through a port:

```
interface SBus #(parameter WIDTH=8);
   logic req, grant;
   logic [WIDTH-1:0] addr, data;
   modport phy(input addr, inout data);
endinterface

module top;

   parameter SIZE = 4;

   virtual SBus#(16) V16;
   virtual SBus#(32).phy V32_Array [1:SIZE];
   ...
endmodule
```

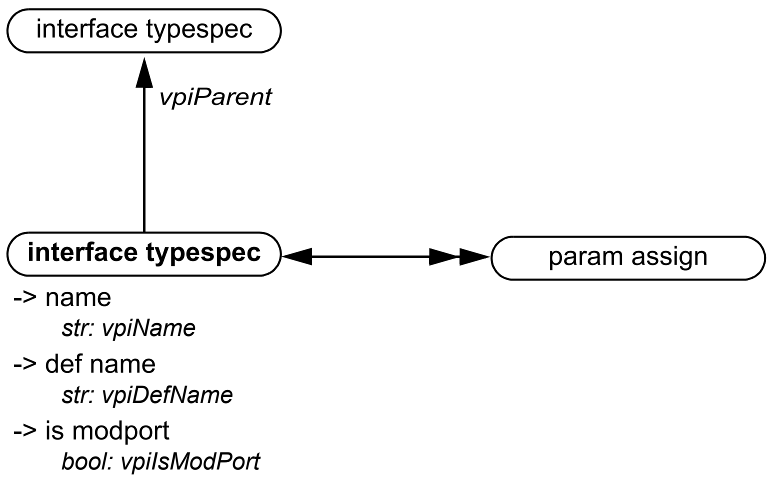
In this example, `V16` is a virtual interface, while `V32_Array` is an array var. The **vpiVariables** iteration from module `top` includes both `V16` and `V32_Array`, while the **vpiVirtualInterfaceVar** iteration returns `V16` together with the individual elements of `V32_Array`, that is, `V32_Array[1]` through `V32_Array[4]`.

Example 2: Virtual interface declaration in a class definition:

```
interface SBus; // A Simple bus interface
   logic req, grant;
   logic [7:0] addr, data;
endinterface

class SBusTransactor;         // SBus transactor class
   virtual SBus bus;          // virtual interface of type SBus
   function new( virtual SBus s );
      bus = s;                // initialize the virtual interface
   endfunction
   task request();            // request the bus
      bus.req <= 1'b1;
   endtask
   task wait_for_bus();       // wait for the bus to be granted
      @(posedge bus.grant);
   endtask
endclass

module devA( SBus s ); ... endmodule  // devices that use SBus

module devB( SBus s ); ... endmodule

module top;
   SBus s[1:4] ();            // instantiate 4 interfaces
   devA a1( s[1] );           // instantiate 4 devices
   devB b1( s[2] );
   devA a2( s[3] );
   devB b2( s[4] );
   initial begin
      SbusTransactor t[1:4];  // create 4 bus-transactors and bind
      t[1] = new( s[1] );
      t[2] = new( s[2] );
      t[3] = new( s[3] );
      t[4] = new( s[4] );
   end
endmodule
```

A *virtual interface var* is returned for the left-hand side expression of the statement "`bus = s`" in the constructor of the class definition `SBusTransactor`. The **vpiName** of the virtual interface var is "`bus`", and it has a **vpiInterfaceTypespec** for which the **vpiDefName** is "`SBus`". The **vpiActual** relationship returns the interface instance associated with that particular call to **`new`** after the assignment has executed. For example, if it was "**`new`**`(s[1])`", **vpiActual** would return the interface `s[1]`. If **vpiActual** is queried before the assignment is executed, the method shall return NULL if the virtual interface is uninitialized. In addition, the right-hand side expression of "`bus = s`" returns a virtual interface var for which **vpiActual** is the interface instance passed to the call to **`new`**.

37.28 Interface typespec



Details:

1) The **vpiDefName** of an interface typespec that represents a modport shall be the modport identifier. The **vpiDefName** of an interface typespec that represents an interface shall be the identifier of the interface declaration.

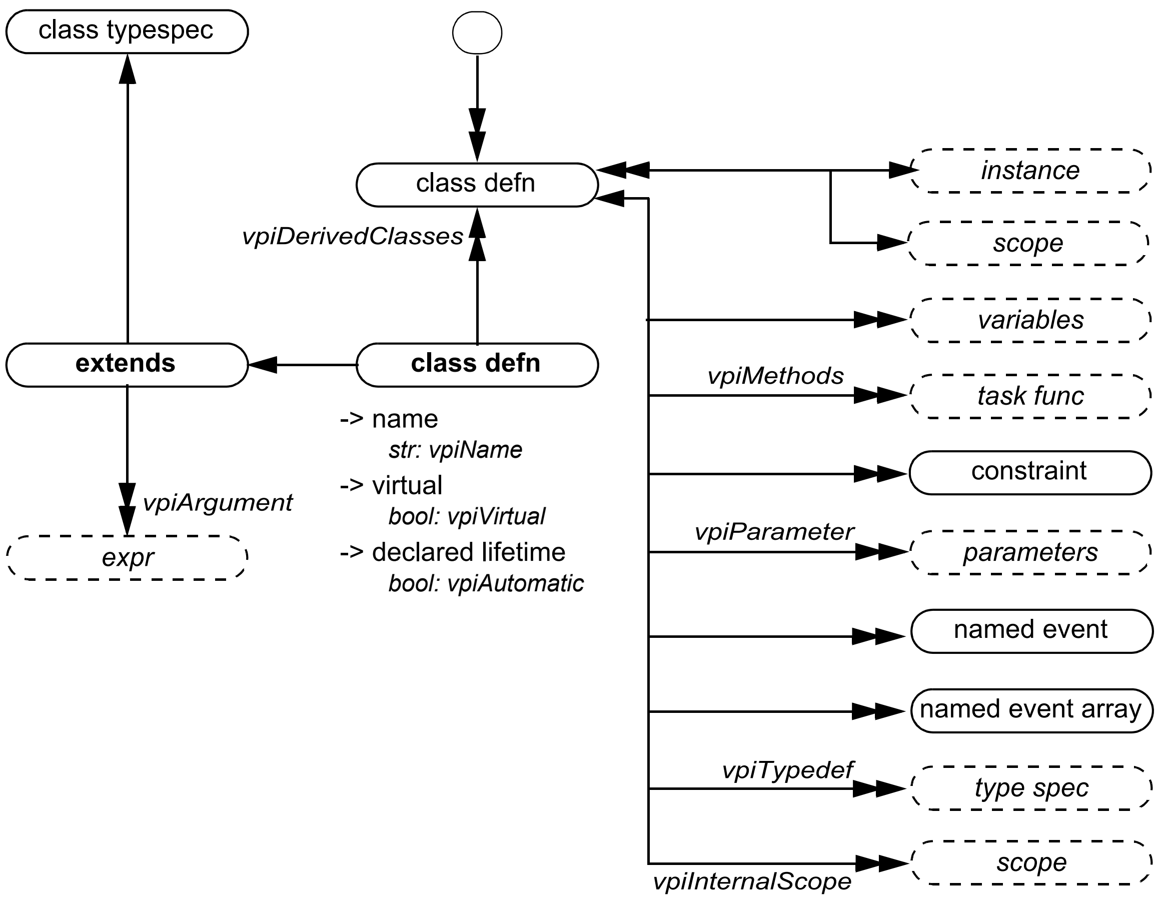
2) For an interface typespec that represents a modport, **vpiParent** shall return an interface typespec of the corresponding interface. For an interface typespec that represents an interface, **vpiParent** shall return NULL.

3) In the following example, the first typedef defines an interface typespec corresponding to "**`virtual`** `SBus#(16)`" whose **vpiName** is `SB16`. The **vpiDefname** of this typespec shall be `SBus`, and the assigned parameter value of 16 shall be derived by iterating on **vpiParamAssign**. The typedef `SBphy`, however, is an array typespec for which the **vpiElemTypespec** returns an interface typespec corresponding to "**`virtual`** `SBus#(32).phy`".

   The **vpiTypedef** iteration from the module `top` returns handles to both `SB16` and `SBphy` interface typespecs.

```
interface SBus #(parameter WIDTH=8);
   logic req, grant;
   logic [WIDTH-1:0] addr, data;
   modport phy(input addr, inout data);
endinterface

module top;

   parameter SIZE = 4;

   typedef virtual SBus#(16) SB16;
   typedef virtual SBus#(32).phy SBphy [1:SIZE];
   ...
endmodule
```

37.29 Class definition



Details:

1) The iterations over **vpiVariables**, **vpiMethods**, **vpiNamedEvent**, and **vpiNamedEventArray** shall return both static and automatic properties or methods. However, the iteration over **vpiMethods** shall not include built-in methods for which there is no explicit declaration.

2) **vpi_get_value**() and **vpi_put_value()** are not allowed for variable and event handles obtained from class defn handles.

3) The iterator to constraints returns only normal constraints and not inline constraints.

4) The **vpiConstraint** iteration shall return the constraints in syntactic declaration order. The position within this order of a constraint declared as **extern** shall be determined by the position of its prototype. To get constraints inherited from base classes, it is necessary to traverse the extends relation to obtain the base class typespec.

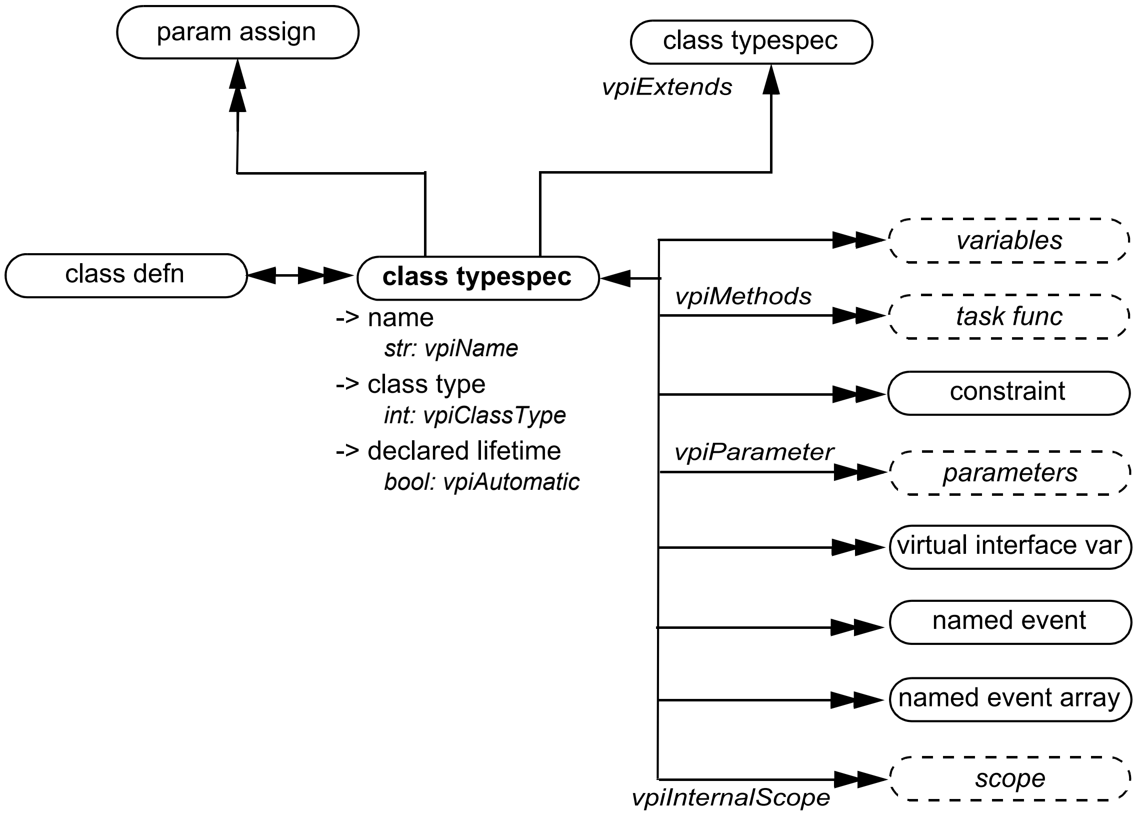
5) The **vpiDerivedClasses** iterator shall return all the class defns derived from the given class defn.

6) The relation to **vpiExtends** exists whenever one class is derived from another class (refer to 8.13). The relation from extends to class typespec provides the base class. The **vpiArgument** iterator from extends shall provide the arguments used in constructor chaining (refer to 8.17).

7) The **vpiParameter** iteration shall return both the parameters declared in the parameter port list of the class declaration and the parameters declared within the body of the class declaration as class items. The property **vpiLocalParam** (see 37.26) shall return TRUE for parameters declared within the body.

8) For details on lifetime and memory allocation properties, see 37.3.7.

37.30 Class typespec



Details:

1) According to how it is obtained, a class typespec may represent either a lexical construct or a class specialization.

   If the class typespec is obtained as part of a class defn, it represents a lexical construct from the SystemVerilog source code. In particular, it shall represent a lexical construct under the following conditions:

   — It is obtained from a class defn via the **vpiTypedef** iteration. In this case it represents a user-defined typedef.

   — It is part of the declaration of a class item (variable or method) obtained from the class defn.

   — It is obtained from the extends object associated with the class defn.

   A class typespec object that has all parameter values resolved shall represent a class specialization. In particular, it shall represent a class specialization under the following conditions:

   — It is obtained from a class defn by iterating over **vpiClassTypespec**.

   — It is the type of a variable or method for which no containing scope is a class defn. If the variable or method is declared using the name of a typedef, the class typespec shall be the corresponding class instantiation rather than the class typespec for the typedef itself.

   A class typespec derived from a class defn for which the parameter port list is empty may represent both a lexical construct and a class specialization.

2) For a class typespec that represents only a lexical construct, the one-to-many relations **vpiVariables**, **vpiMethods**, **vpiConstraint**, **vpiNamedEvent**, **vpiNamedEventArray**, **vpiTypedef**, and **vpiInternalScope** are not supported.

3) In the case of a class typespec that represents a lexical construct, if the class type construct includes an explicit parameter expression or type, the object for that parameter or type shall constitute the **vpiRhs** part of the corresponding param assign (see 37.26); otherwise the **vpiRhs** part shall reference the default expression or type with which the parameter was declared. However, if the class typespec represents a class specialization, the **vpiRhs** of each param assignment may be any object that has the correct value (in the case of a non-type parameter) or type (in the case of a type parameter).

4) A class typespec that represents a class specialization shall have a valid, though tool-dependent, name.

5) From a class typespec that represents a class specialization, the iterations over **vpiVariables**, **vpiMethods**, **vpiNamedEvent**, and **vpiNamedEventArray** shall return both static and automatic properties or methods. However, the iteration over **vpiMethods** shall not include built-in methods for which there is no explicit declaration.

6) **vpi_get_value()** and **vpi_put_value()** are not allowed for non-static variable and event handles obtained from class typespec handles.

7) The iterator to constraints returns only normal constraints and not inline constraints.

8) The **vpiConstraint** iteration shall return the constraints in syntactic declaration order. The position within this order of a constraint declared as `extern` shall be determined by the position of its prototype. To get constraints inherited from a base class typespec, it is necessary to traverse the extends relation to obtain the base class typespec.

9) The **vpiExtends** relation shall return the base class typespec, if any, from which a given class typespec is derived. The base class typespec of a class specialization shall also be a specialization.

10) The **vpiClassTypespec** iteration from a class defn shall return the class specializations derived directly (and not by inheritance) from that class defn.

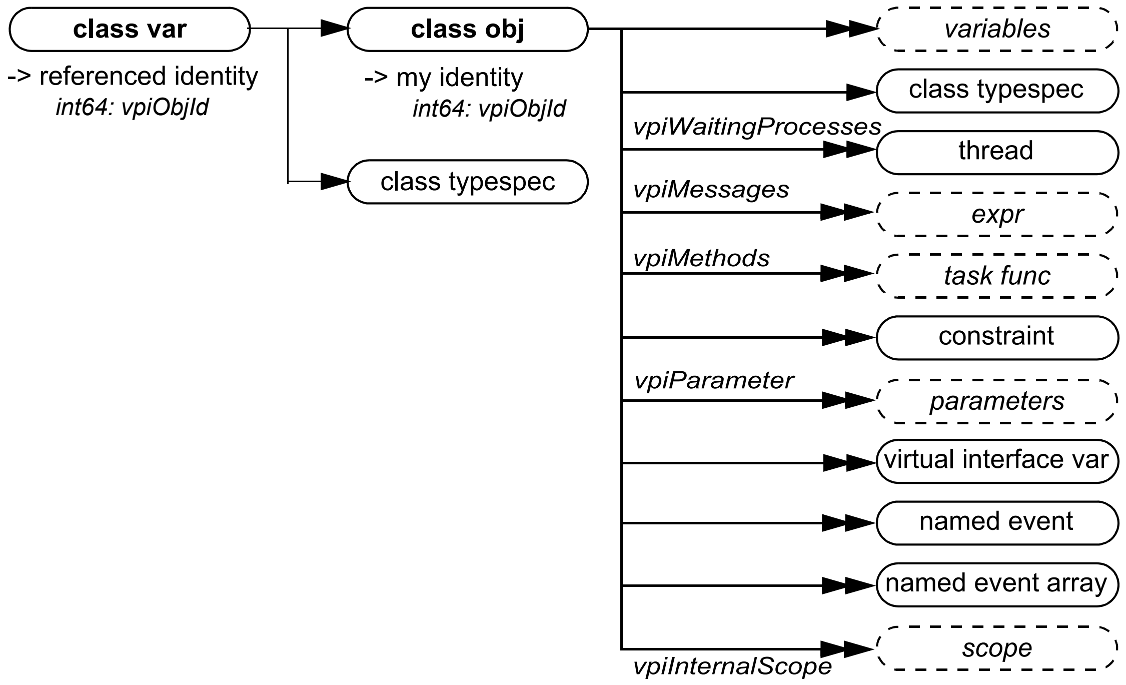
11) The **vpiVirtualInterfaceVar** iteration (formerly **vpiInterfaceDecl**—now deprecated in this standard—see C.4.3, item 5) shall return the virtual interface var declarations in the class specialization (see 37.12 detail 7). If an array of virtual interfaces is declared, the **vpiVirtualInterfaceVar** iteration shall return each element of the array separately. However, the **vpiVariables** iteration shall return the array declaration as a single **vpiArrayVar**.

12) The **vpiParameter** iteration shall return parameters corresponding both to those declared in the parameter port list of the class declaration and to those declared within the body of the class declaration as class items. The property **vpiLocalParam** (see 37.26) shall return TRUE for parameters declared within the body.

13) The **vpiClassDefn** relation shall return NULL for built-in classes.

14) For details on lifetime and memory allocation properties, see 37.3.7.

37.31 Class variables and class objects



Details:

1) The property **vpiObjId** is a class object's identifier. It is a property of a live object and guaranteed to be unique with respect to all other dynamic objects that support this property for as long as the object is alive. After the object is destroyed by garbage collection, its particular **vpiObjId** value may be reused.

2) For a class var, its **vpiObjId** is the identifier of the object it references or 0, indicating it is not referencing any object.

3) The **vpiWaitingProcesses** iterator on a mailbox or semaphore shall return the threads waiting on the class object or object resource. A waiting process is a static or dynamic process represented by its suspended thread. A process may be waiting to retrieve a message from a mailbox or waiting for a semaphore resource key.

4) A **vpiMessages** iteration shall return all the messages in a mailbox.

5) For a class var, **vpiClassTypespec** shall return the class typespec with which the class var was declared in the SystemVerilog source text. If the class var has the value of `NULL`, the **vpiClassObj** relationship applied to the class var shall return a null handle. **vpiClassTypespec** when applied to a class obj handle shall return the class typespec with which the class obj was created. The difference between the two usages of **vpiClassTypespec** can be seen in the following example:

```
class Packet;
   ...
endclass : Packet
class LinkedPacket extends Packet;
   ...
endclass : LinkedPacket
LinkedPacket l = new;
Packet p = l;
```

In this example, the **vpiClassTypespec** of variable p is `Packet`, but the **vpiClassTypespec** of the class obj associated with variable `p` is "`LinkedPacket`".

NOTE—When a class var is obtained as a data member of a class typespec, the application must use **vpiScope** (see 37.12) rather than **vpiClassTypespec** to obtain the enclosing scope.

6) From a class obj, the iterations over **vpiVariables**, **vpiMethods**, **vpiNamedEvent**, and **vpiNamedEventArray** shall return both static and automatic properties or methods. However, the iteration over **vpiMethods** shall not include built-in methods for which there is no explicit declaration.

7) The **vpiVirtualInterfaceVar** iteration (formerly **vpiInterfaceDecl**—now deprecated in this standard—see C.4.3, item 5) shall return the virtual interface var declarations in the class object. If an array of virtual interfaces is declared, the **vpiVirtualInterfaceVar** iteration shall return each element of the array separately. However, the **vpiVariables** iteration shall return the array declaration as a single **vpiArrayVar**.

8) The **vpiParameter** iteration shall return parameters corresponding both to those declared in the parameter port list of the class declaration and to those declared within the body of the class declaration as class items. The property **vpiLocalParam** (see 37.26) shall return TRUE for parameters declared with the body. The value of a parameter derived from a class obj shall be the same as that of the same parameter derived from the corresponding class typespec.

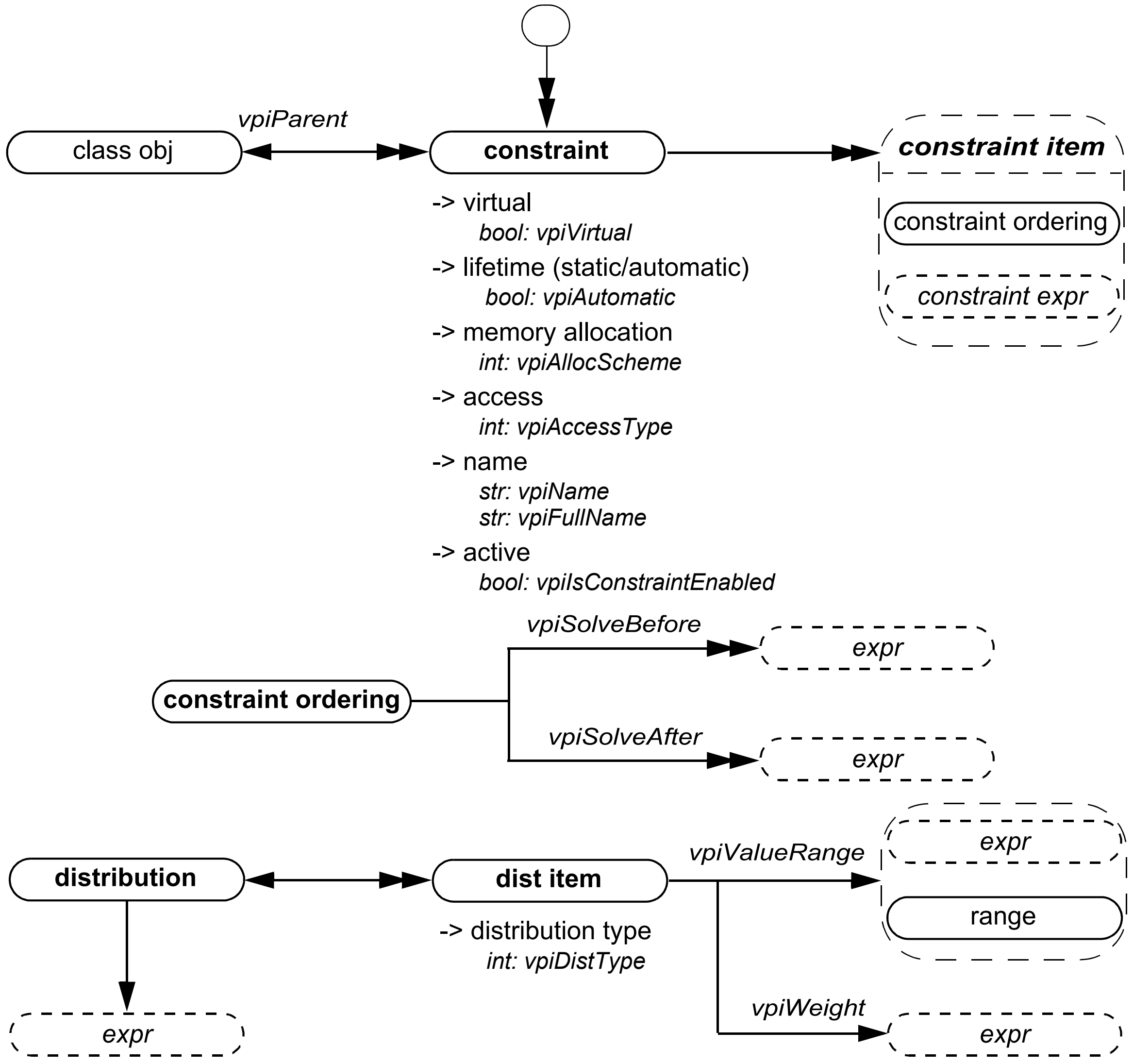
9) **vpi_handle_by_name()** shall accept a full name to a non-static data member, even though it does not have a **vpiFullName** property. For example:

```
module top;
   class Packet ;
      integer Id ;
      ....
   endclass
   Packet p;
   c = p.Id;
   ....
```

vpi_handle_by_name() accepts “top.p.Id”.

10) For details on class object specific callbacks, see 38.36.1.

37.32 Constraint, constraint ordering, distribution



Details:

1) For a constraint, **vpiAutomatic** property does not mean lifetime, but reflects the keyword used in the constraint declaration. **vpiAutomatic** == 0 implies the constraint was declared static. See 18.5.11 for meaning.

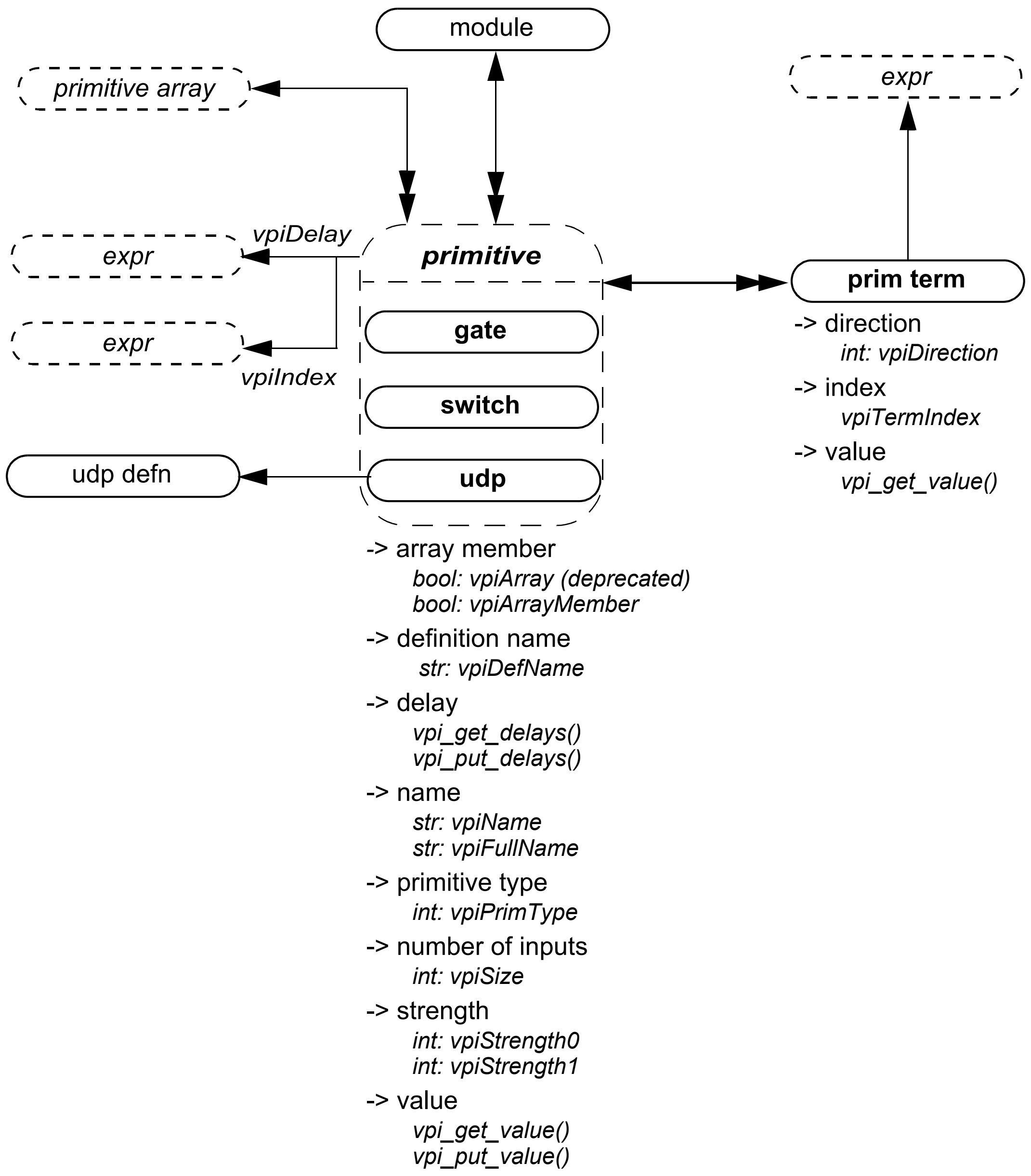
2) For details on memory allocation property, see 37.3.7.

3) Possible return values for the **vpiAccessType** property (see 37.8) for a constraint are **vpiExternAcc** or zero, indicating whether it was declared outside its enclosing class declaration or not (see 18.5.1).

4) The **vpiConstraint** iteration shall return the constraints in syntactic declaration order. The position within this order of a constraint declared as `extern` shall be determined by the position of its prototype.

5) The **vpiConstraintItem** iteration shall return the constraint items in the order in which they occur within the constraint.

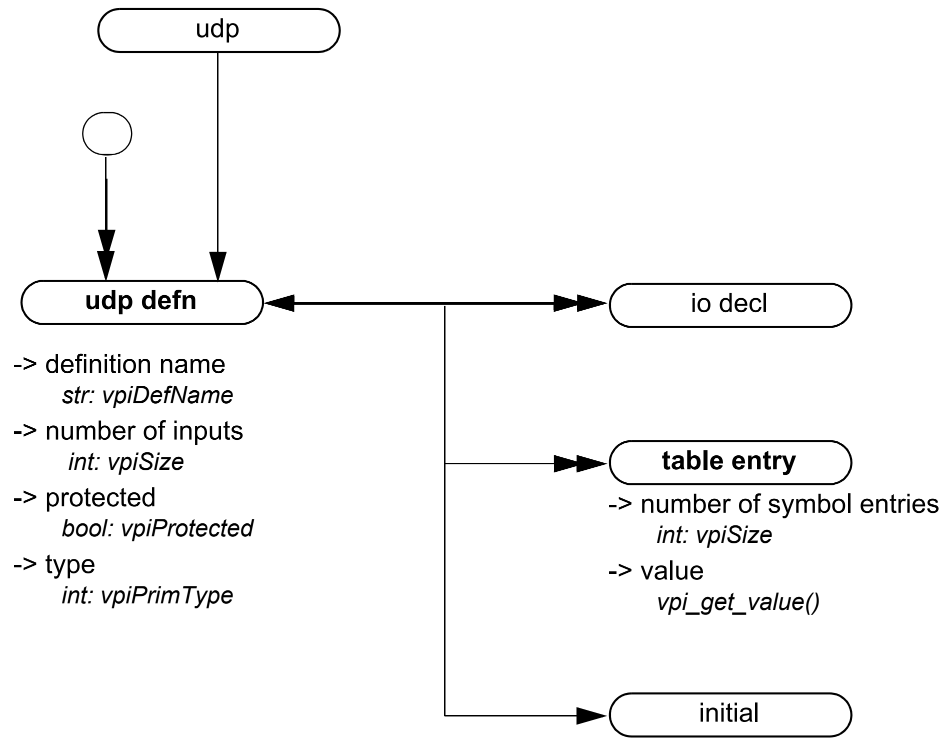
37.33 Primitive, prim term



Details:

1) **vpiSize** shall return the number of inputs.
2) For primitives, **vpi_put_value()** shall only be used with sequential UDP primitives.
3) **vpiTermIndex** can be used to determine the terminal order. The first terminal has a term index of zero.
4) If a primitive is an element within a primitive array, the **vpiIndex** transition is used to access the index within the array. If a primitive is not part of a primitive array, this transition shall return NULL.

37.34 UDP

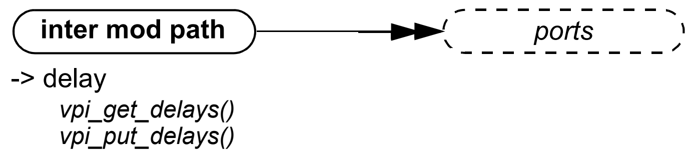


Details:

1) Only string (decompilation) and vector (ASCII values) shall be obtained for table entry objects using **vpi_get_value()**. Refer to the definition of **vpi_get_value()** for additional details.

2) **vpiPrimType** returns **vpiSeqPrim** for sequential UDPs and **vpiCombPrim** for combinational UDPs.

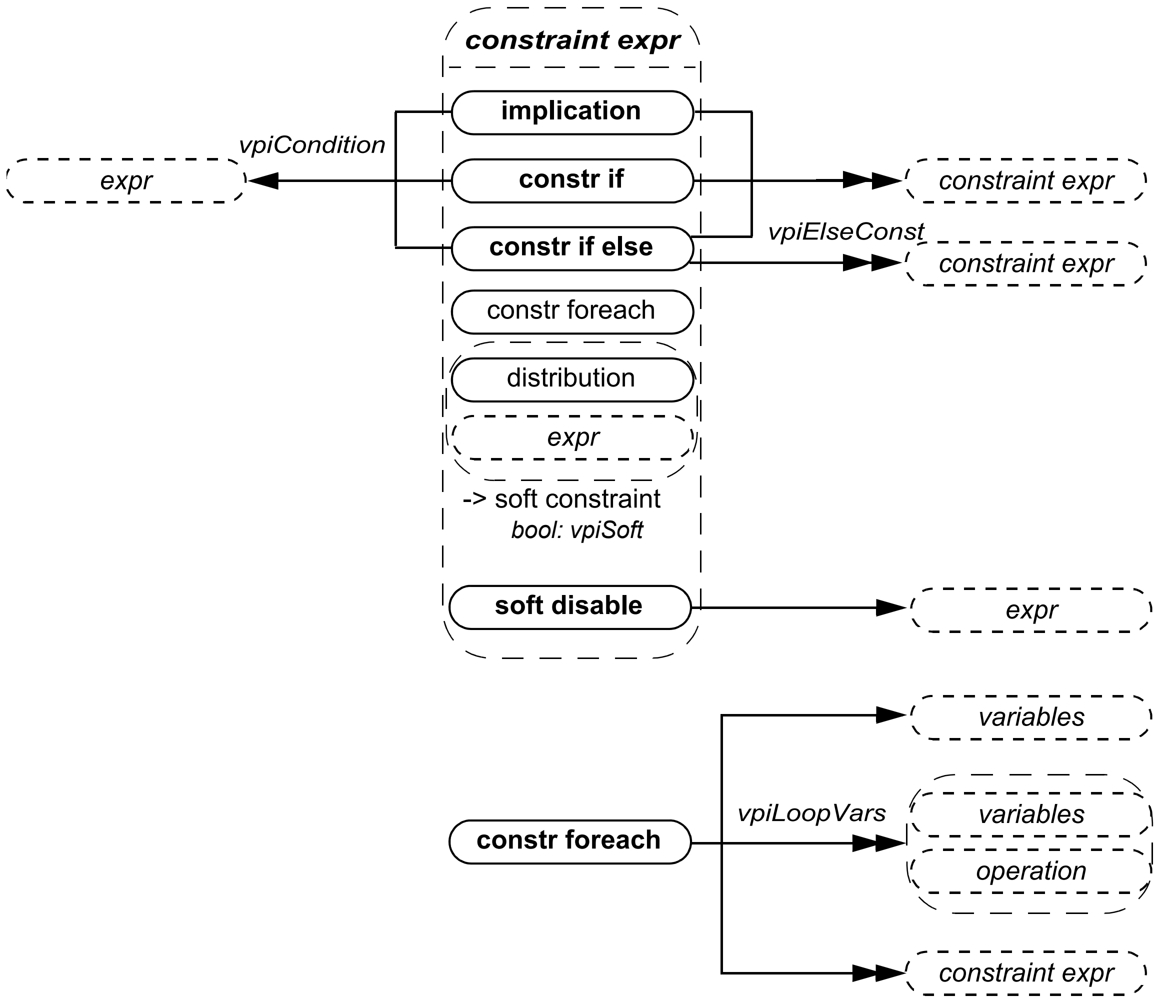
37.35 Intermodule path



Details:

1) To get to an intermodule path, **vpi_handle_multi(vpiInterModPath**, port1, port2**)** can be used.

37.36 Constraint expression



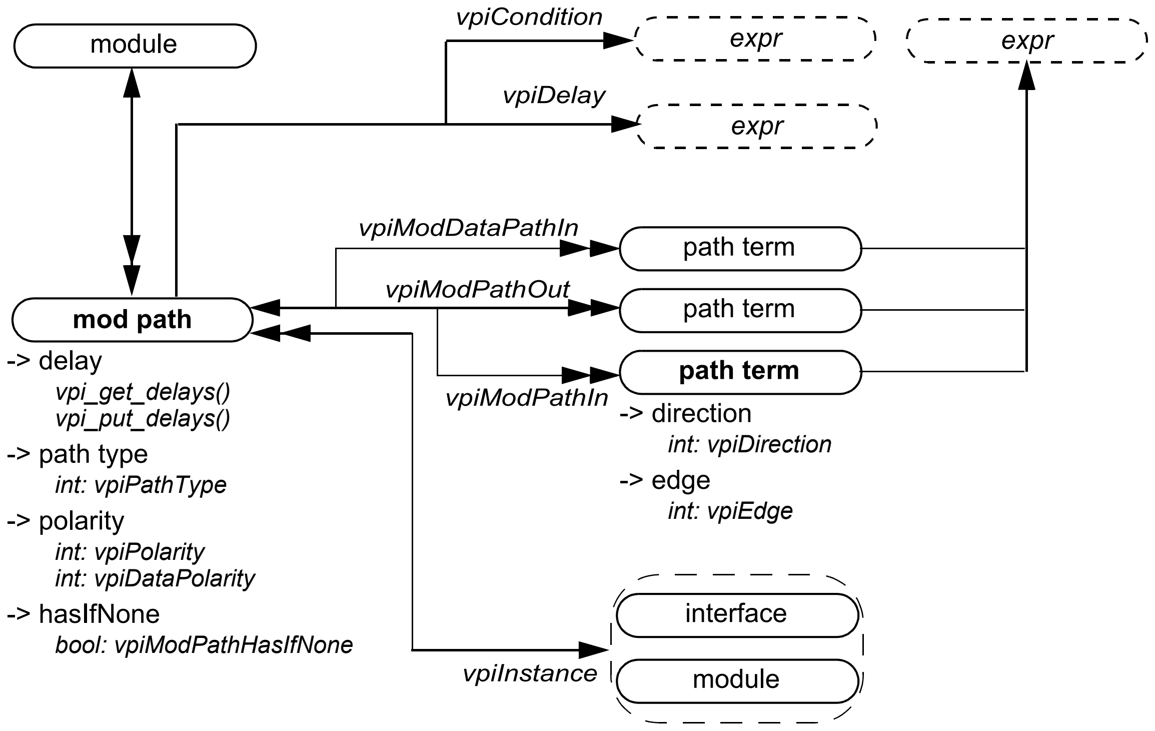
Details:

1) The variable obtained via the **vpiVariables** relation from a **vpiConstrForeach** shall represent the array being indexed.

2) The **vpiLoopVars** iteration shall return the index variables of the foreach constraint in left-to-right order. If an index variable is skipped, its place shall be represented as a **vpiOperation** for which the **vpiOpType** is **vpiNullOp**.

3) Each **vpiConstraintExpr** iteration shall return the expressions in the order in which they occur in the containing implication, **`if`**, **`if-else`**, or **`foreach`** constraint.

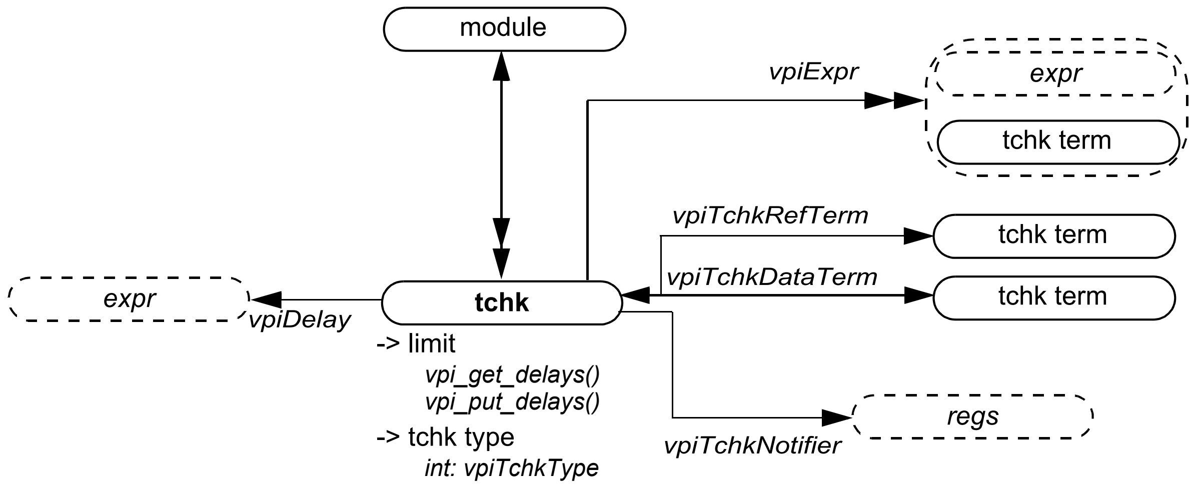
37.37 Module path, path term

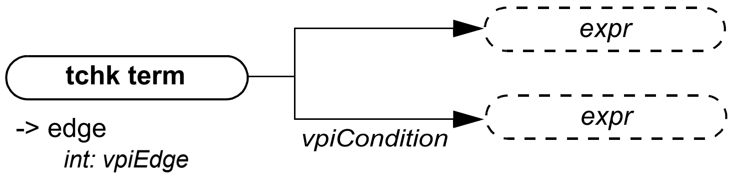


Details:

1) Specify blocks can occur in both modules and interfaces. For backwards compatibility the **vpiModule** relation has been preserved; however this relation shall return `NULL` for **specify** blocks in interfaces. For new code, it is recommended that the **vpiInstance** relation be used instead.

37.38 Timing check



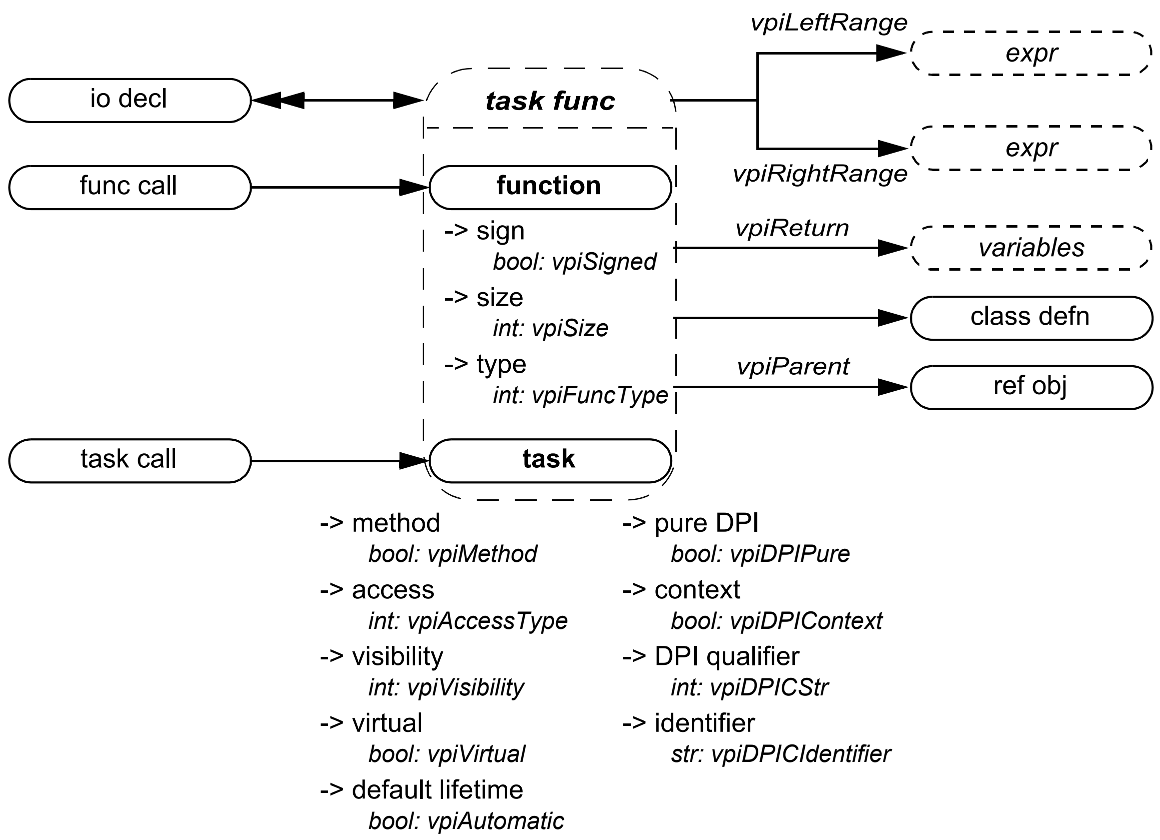


Details:

1) For the timing checks in 31.2 the relationship **vpiTchkRefTerm** shall denote the *reference_event* or *controlled_reference_event*, while **vpiTchkDataTerm** shall denote the *data_event*, if any.

2) When iterating over **vpiExpr** from a tchk, the handles returned for a *reference_event*, a *controlled_reference_event*, or a *data_event* shall have the type **vpiTchkTerm**. All other arguments shall have types matching the expression.

37.39 Task and function declaration



Details:

1) A SystemVerilog function shall contain an object with the same name, size, and type as the function. This object shall be used to capture the return value for this function.

2) For a function where the return type is a user-defined type, **vpi_handle(vpiReturn,** function_handle**)** shall return the implicit variable handle representing the return of the function from which the user can get the details of that user-defined type.

3) **vpiReturn** shall always return a `var` object, even for simple returns.

4) **vpiVisibility** denotes the visibility (**local**, **protected**, or default) of a task or function that is a class member (a method). **vpiVisibility** shall return **vpiPublicVis** for a class member that is not local or protected, or for a task or function that is not a class member.

5) **vpiFullName** of a task or function declared inside a package or class defn shall begin with the full name of the package or class defn followed by "**::**" and immediately followed with the name of the task or function.

6) **vpiAccessType** shall return **vpiDPIExportAcc** for "DPI" and "DPI-C" export functions/tasks, and shall return **vpiDPIImportAcc** for "DPI" and "DPI-C" import functions/tasks.

7) **vpiDPIPure** shall return TRUE for pure "DPI" and "DPI-C" import functions.

8) **vpiDPIContext** shall return TRUE for context import "DPI" and "DPI-C", functions/tasks.

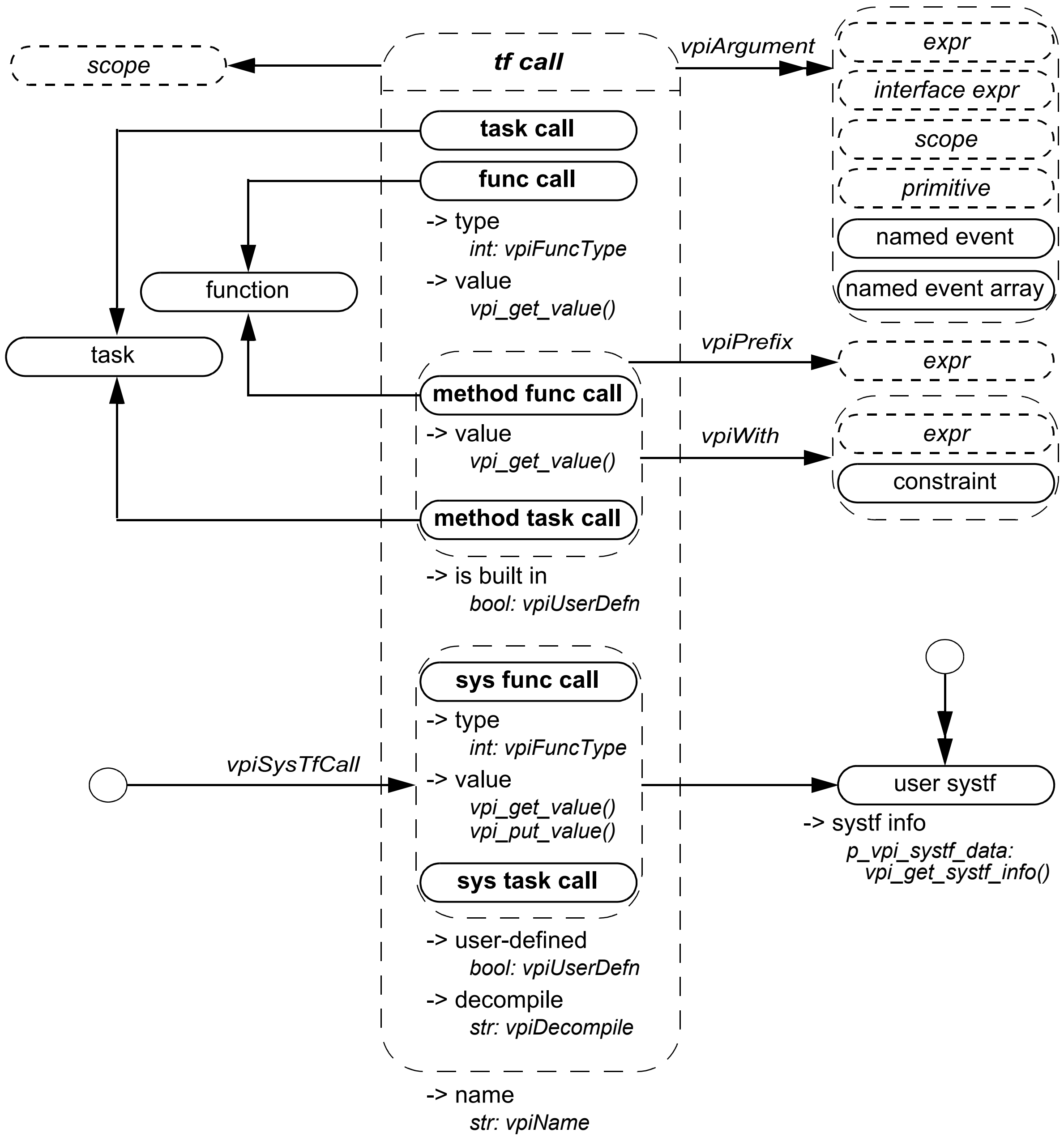
9) **vpiDPICStr** shall return **vpiDPI** for a "DPI" function/task, and **vpiDPIC** for a "DPI-C" function/task.

10) **vpiDPICIdentifier** shall return a string corresponding to the C linkage name for the "DPI"/"DPI-C" function/task.

11) For details on lifetime and memory allocation properties, see 37.3.7.

12) If the **vpiSize** of the **vpiReturn** variable is defined (see 37.17, detail 9) and can be determined without evaluating the function, **vpiSize** for the function shall return the same value as **vpiSize** for the **vpiReturn** variable. For a void function, **vpiSize** shall return 0. For all other cases the behavior of **vpiSize** is undefined.

37.40 Task and function call



Details:

1) The **vpiWith** relation is only available for randomize methods (see 18.7) and for array locator methods (see 7.12.1).

2) For methods (method func call, method task call), the **vpiPrefix** relation shall return the object to which the method is being applied. For example, for the class method invocation

```
packet.send();
```

the prefix for the “`send`” method is the class var “`packet`”.

3) The system task or function that invoked an application shall be accessed with **vpi_handle**(**vpiSysTfCall**, `NULL`).

4) **vpi_get_value()** shall return the current value of the system function.

5) If the **vpiUserDefn** property of a system task or function call is true, then the properties of the corresponding systf object shall be obtained via **vpi_get_systf_info()**.

6) All user-defined system tasks or functions shall be retrieved using **vpi_iterate()**, with **vpiUserSystf** as the type argument, and a NULL reference argument.

7) The simulator shall not evaluate arguments to system tasks or functions when calling those tasks or functions (36.4). Effectively, the value of any argument expression, or of any operand or argument of the expression, is not known until an application asks for it using **vpi_get_value()** (38.15), a **cbValueChange** callback (38.36.1), or other equivalent operation. If no application asks for the value of the argument, it is never evaluated.

8) An empty (omitted) argument (see 21.2.1) shall be represented as an expression with a **vpiType** of **vpiOperation** and a **vpiOpType** of **vpiNullOp**. An argument consisting of the special value **`null`** shall be represented as an expression with a **vpiType** of **vpiConstant** and a **vpiConstType** of **vpiNullConst**.

   Example:

```
logic my_var;
$my_task(my_var, "", , null, );
```

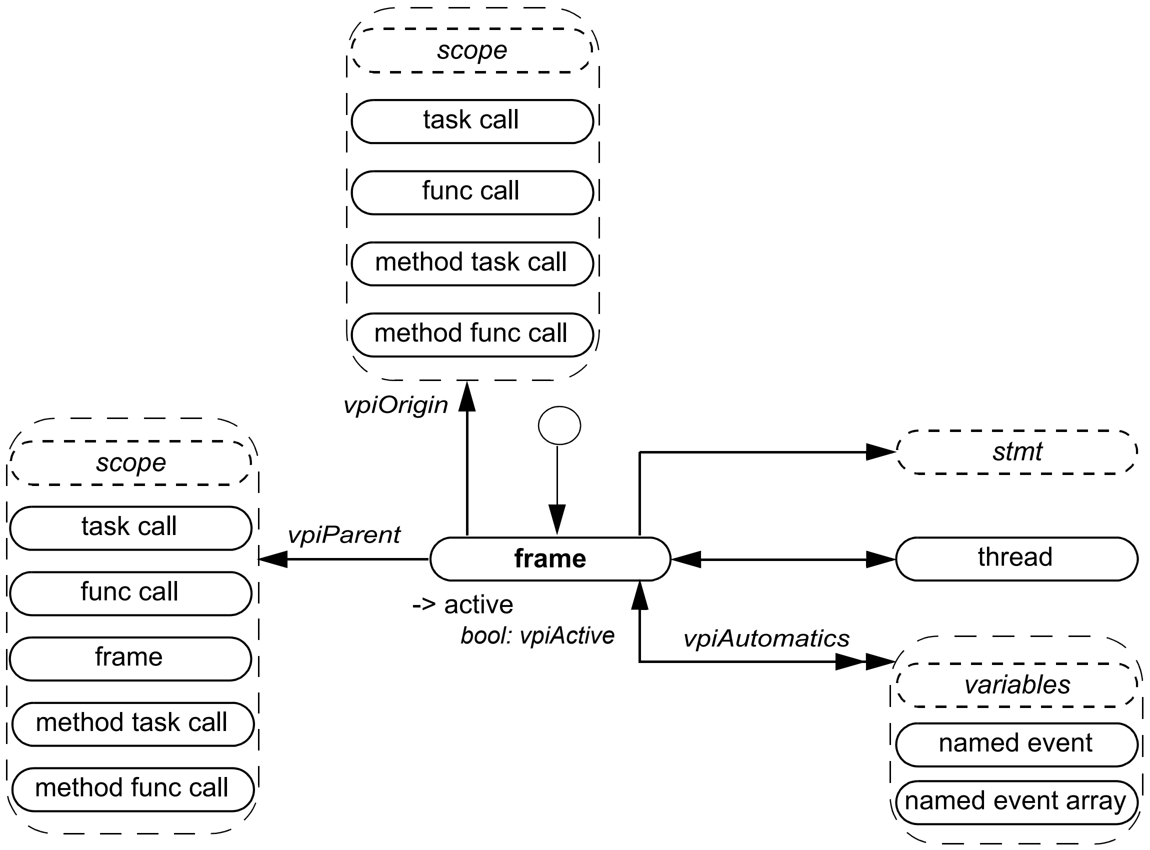
   In the call to the user-defined system task `$my_task()`, `my_var` is an ordinary argument of type **vpiLogicVar**. The second argument, an empty string (but not an empty argument), is a **vpiConstant** for which the **vpiConstType** is **vpiStringConst**. The third and fifth arguments are empty arguments, while the fourth argument is a **vpiConstant** with a **vpiConstType** of **vpiNullConst**. VPI shall represent the third and fifth arguments as **vpiOperations** with a **vpiOpType** of **vpiNullOp**.

9) The property **vpiDecompile** shall return a string with a functionally equivalent system task or function call to what was in the original source code. The arguments shall be decompiled using the same manner as any expression is decompiled. See 37.57 for a description of expression decompilation.

10) System task and function calls that are protected shall allow iteration over the **vpiArgument** relationship.

11) For a built-in method func call, **vpiFunction** shall return `NULL`, while **vpiTask** shall return `NULL` for a built-in method task call.

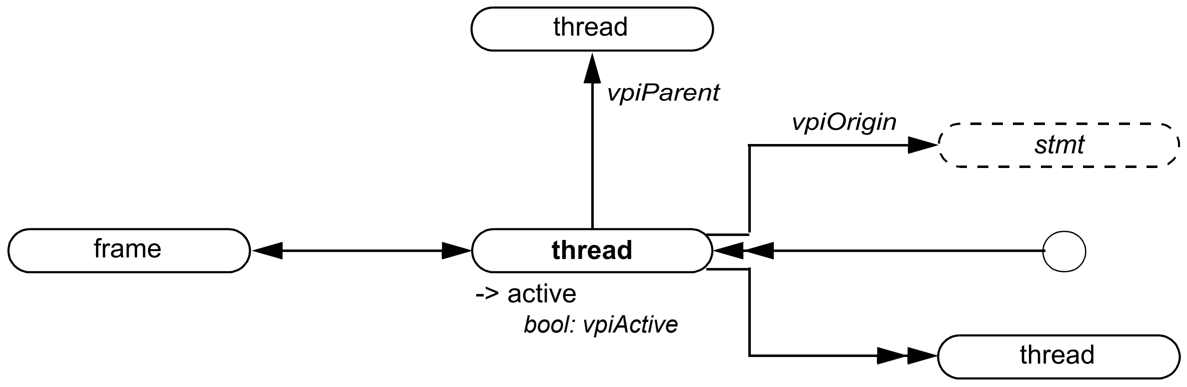
37.41 Frames



Details:

1) Frames correspond to the set of automatic variables declared in a given task or function.
2) It shall be illegal to place value change callbacks on automatic variables.
3) It shall be illegal to put a value with a delay on automatic variables.
4) There is at most only one active frame at any time in a given thread. To get a handle to the currently active frame, use **vpi_handle(vpiFrame**, `NULL`**)**. The frame to stmt transition shall return the currently active statement within the frame.
5) The frame object model is not backwards compatible with IEEE Std 1364-2005.
6) For details on frame specific callbacks, see 38.36.1.

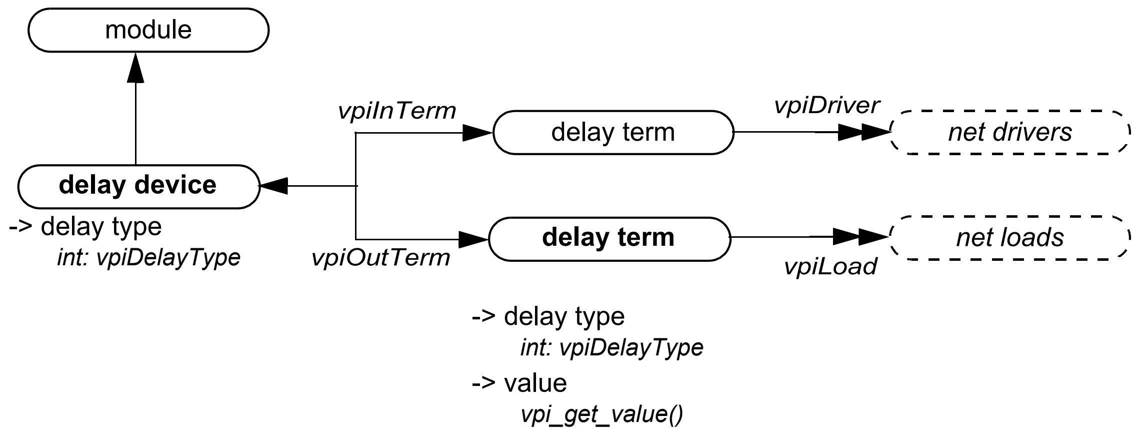
37.42 Threads



Details:

1) A thread is a SystemVerilog process such as an **`always`** procedure or a branch of a **`fork`** construct. As a thread works its way down a call chain of tasks and/or functions, a new frame is activated as each new task or function is entered.
2) For details on thread specific callbacks, see 38.36.1.

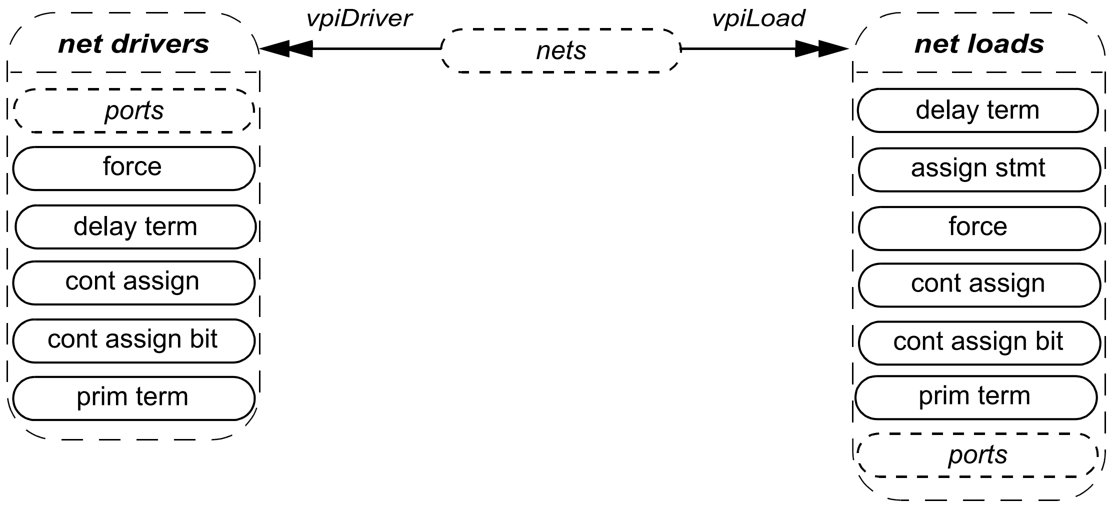
37.43 Delay terminals



Details:

1) The value of the input delay term shall change before the delay associated with the delay device.
2) The value of the output delay term shall not change until after the delay has occurred.

37.44 Net drivers and loads



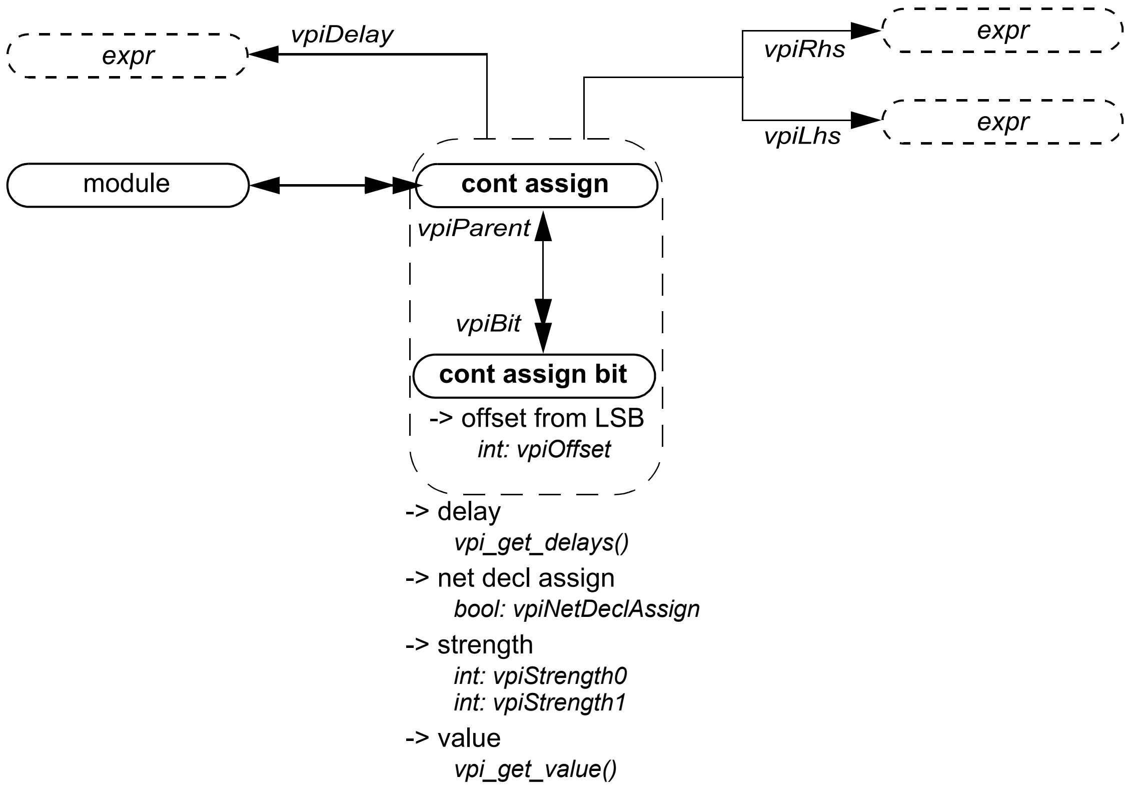
Details:

1) Complex expressions on input ports that are not concatenations shall be considered a load for a net. Iterating on loads for *trinet* in the following example will cause the fourth port of *ram* to be a load:

```
module my_module;
   tri trinet;
   ram r0 (a, write, read, !trinet);
endmodule
```

Access to the complex expression shall be available using **vpi_handle(vpiHighConn**, portH**)** where portH is the handle to the port returned when iterating on loads.

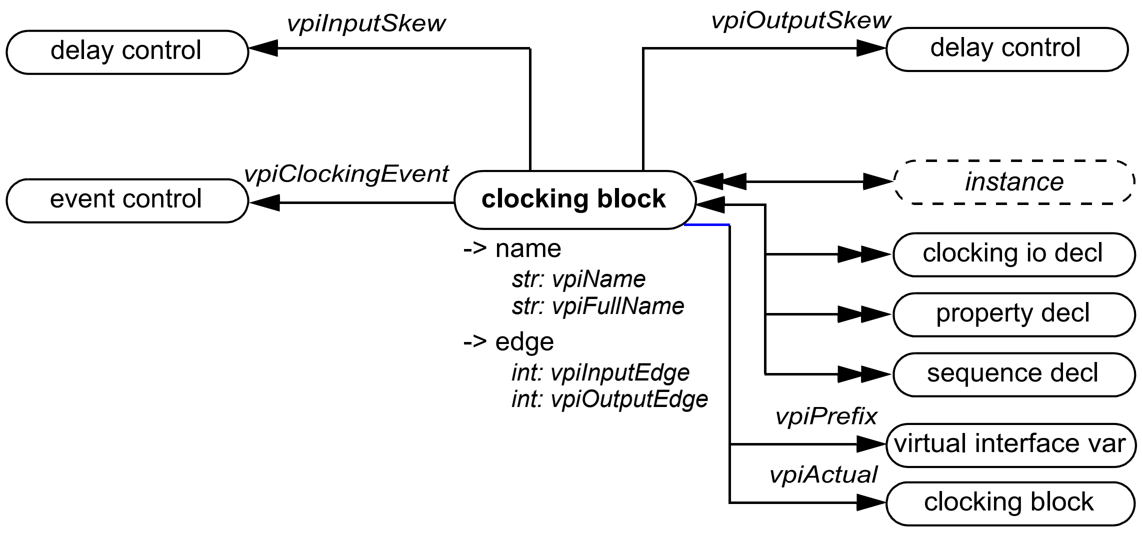
37.45 Continuous assignment

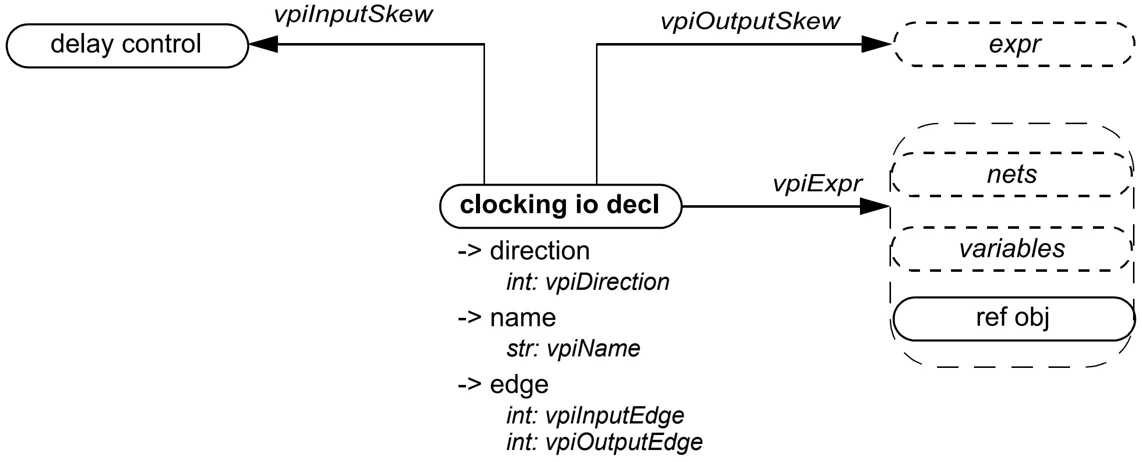


Details:

1) The size of a cont assign bit is always scalar.
2) Callbacks for value changes can be placed onto cont assign or a cont assign bit.
3) **vpiOffset** shall return zero for the LSB.

37.46 Clocking block

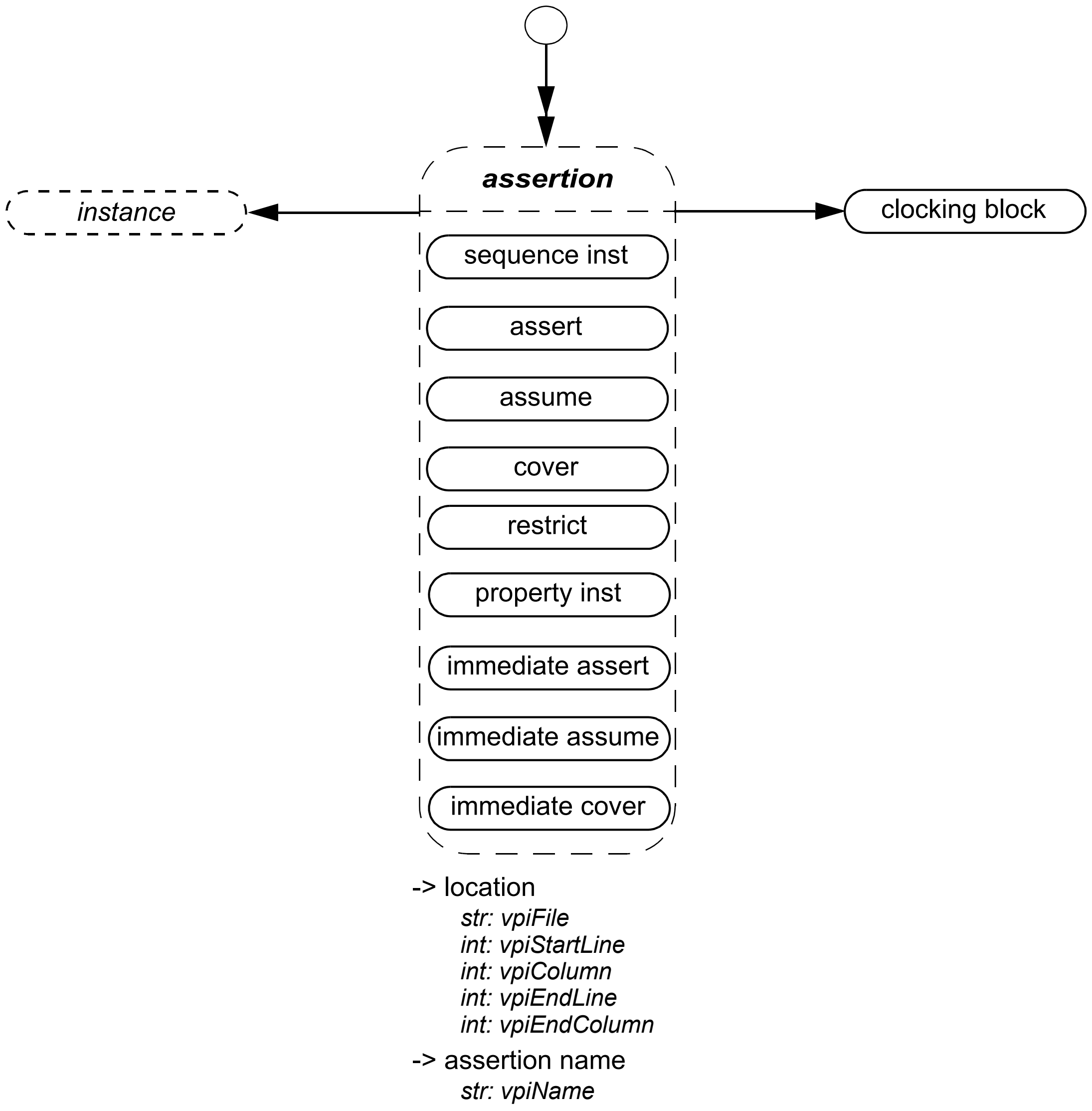




Details:

1) The methods, **vpiInputSkew** and **vpiOutputSkew**, and properties **vpiInputEdge** and **vpiOutputEdge**, on the `clocking` block apply to the default constructs. The same methods and properties on the clocking io decl apply to the clocking io decl itself.
2) The **vpiPrefix** relation shall be non-NULL when the clocking block represents an expression in the SystemVerilog source code immediately prefixed by a virtual interface.
3) If a prefix of a clocking block is a virtual interface that has no value at the current simulation time, the **vpiActual** relation shall return NULL.
4) **vpiExpr** shall return the expression or ref obj referenced by the clocking io decl. Consider input `enable = top.mem1.enable`. Here, "`enable`" is represented by a clocking io decl, and the **vpiExpr** relation returns a handle to "`top.mem1.enable`".

37.47 Assertion

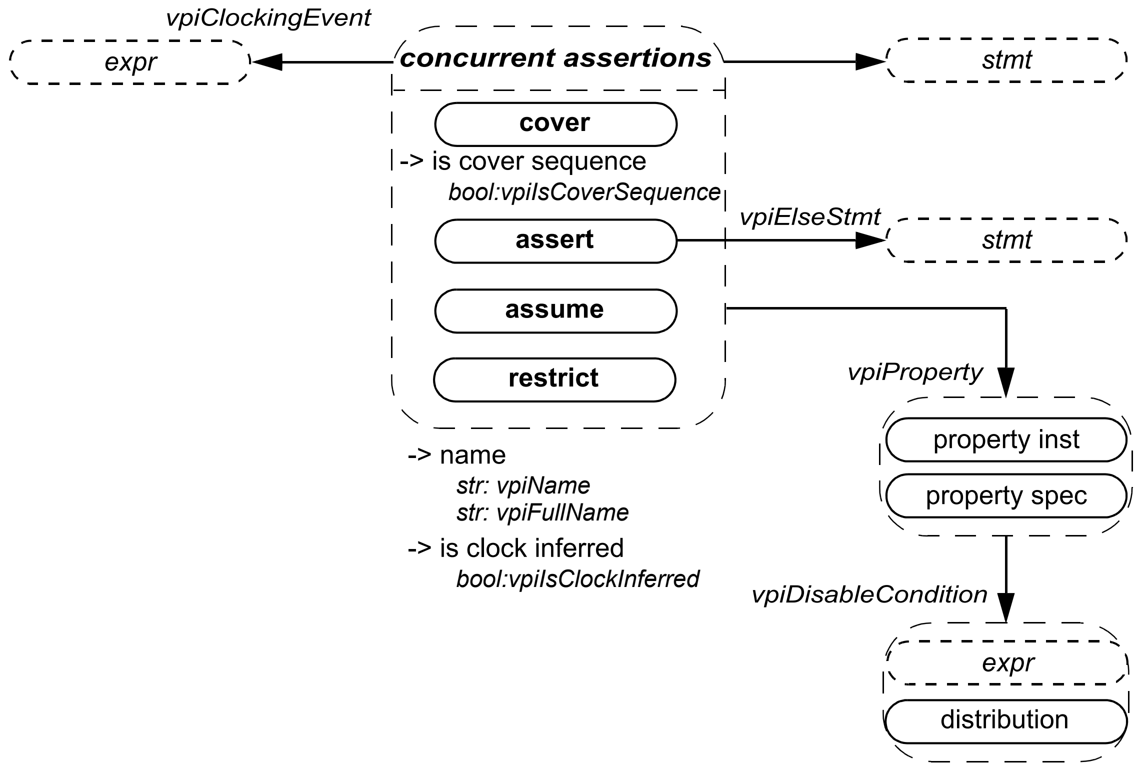


Details:

1) For details on using VPI to obtain static and dynamic assertion information as well as assertion callbacks and control, see Clause 39.

2) For details on using VPI to obtain assertion coverage, see 40.5.3.

37.48 Concurrent assertions

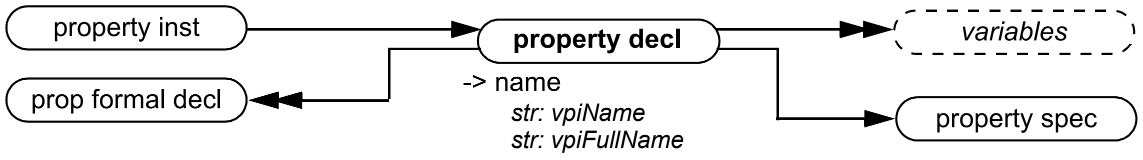


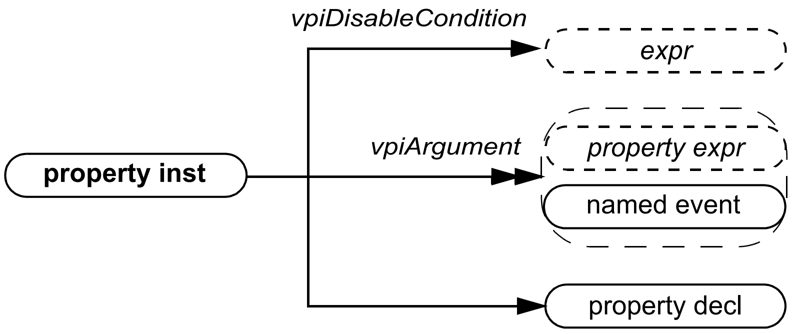
Details:

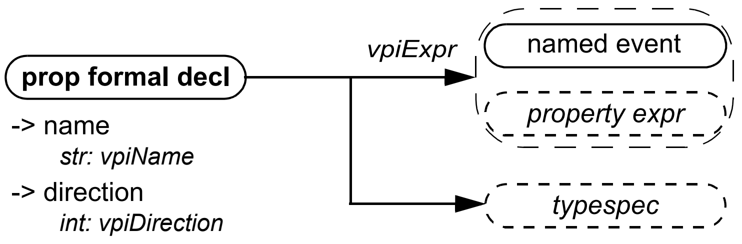
1) Clocking event is always the actual clocking event on which the assertion is being evaluated, regardless of whether this is explicit or implicit (inferred).

2) The **`restrict property`** statement has no pass and no fail action statement. Also, it is not simulated and hence generates no run-time information.

37.49 Property declaration



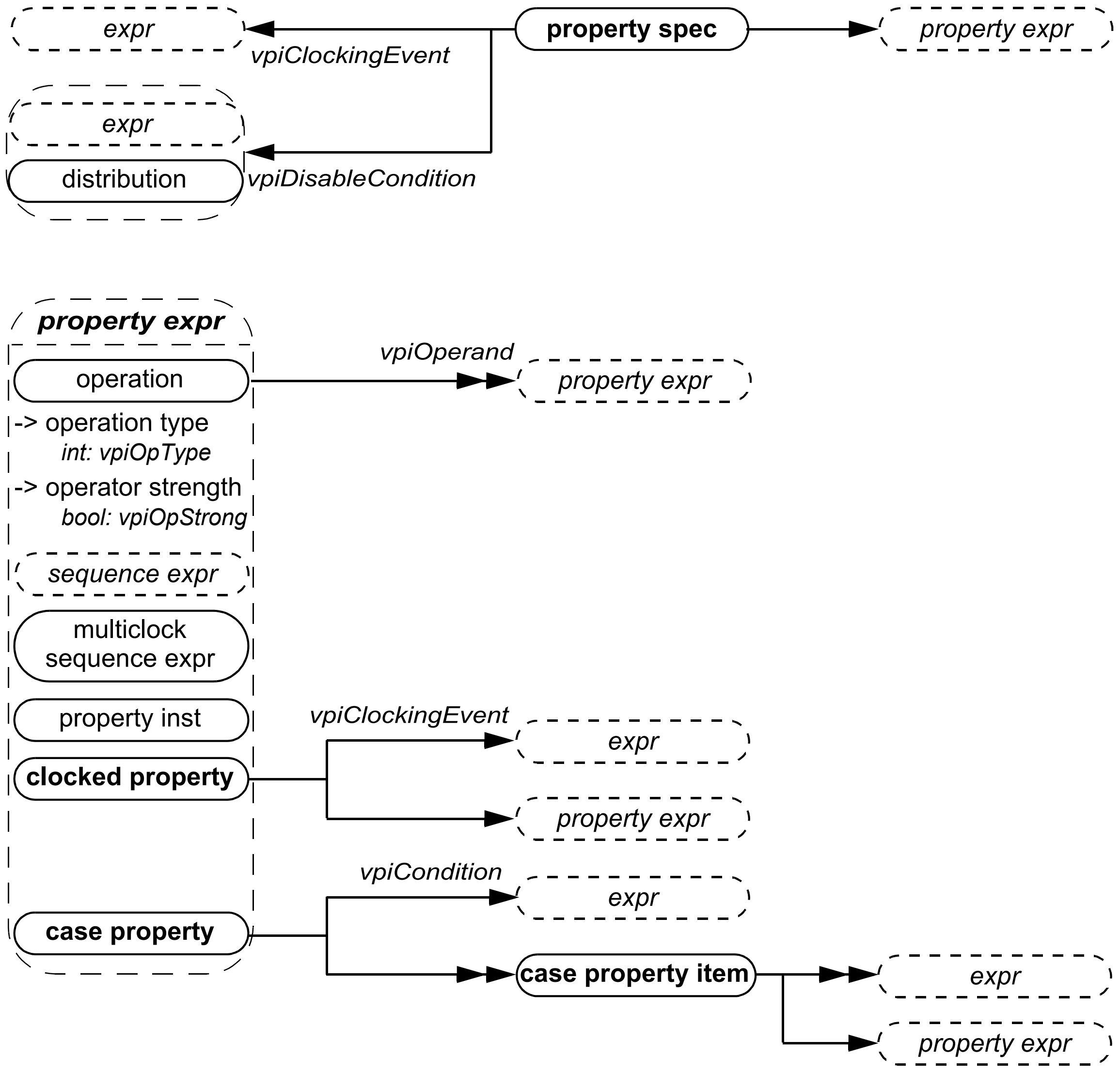




Details:

1) The **vpiPropFormalDecl** iterator shall return the property declaration arguments in the order that the formals for the property are declared.
2) The **vpiArgument** iterator shall return the property instance arguments in the order that the formals for the property are declared, so that the correspondence between each argument and its respective formal can be made. If a formal has a default value, that value shall appear as the argument should the instantiation not provide a value for that argument.
3) The **vpiTypespec** relation shall return NULL if the formal is untyped.
4) If the formal has an initialization expression, the expression can be obtained using the **vpiExpr** relation.
5) **vpiDirection** returns **vpiNoDirection** if the formal argument is not a local variable argument. Otherwise, **vpiDirection** returns **vpiInput**.

37.50 Property specification



Details:

1) Variables are declarations of property variables. The value of these variables cannot be accessed.

2) Within the context of a property expr, **vpiOpType** can be any one of **vpiAcceptOnOp**, **vpiAlwaysOp**, **vpiCompAndOp**, **vpiCompOrOp**, **vpiEventuallyOp**, **vpiIfElseOp**, **vpiIfOp**, **vpiIffOp**, **vpiImpliesOp**, **vpiNexttimeOp**, **vpiNonOverlapFollowedByOp**, **vpiNonOverlapImplyOp**, **vpiNotOp**, **vpiOverlapFollowedByOp**, **vpiOverlapImplyOp**, **vpiRejectOnOp**, **vpiSyncAcceptOnOp**, **vpiSyncRejectOnOp**, **vpiUntilOp**, or **vpiUntilWithOp**.

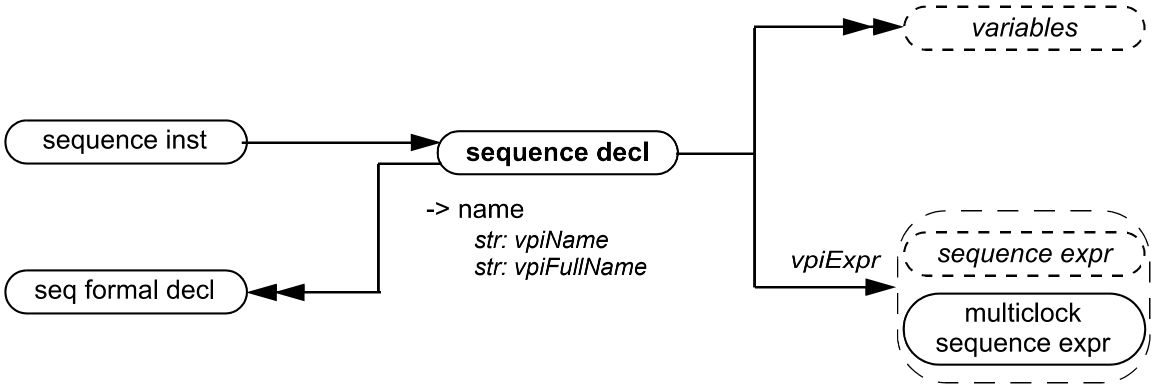
   Operands to these operations shall be provided in the same order as shown in the BNF, with the following exceptions:
   - **vpiNexttimeOp**: Arguments shall be: property, constant. constant shall only be given if different from 1.
   - **vpiAlwaysOp** and **vpiEventuallyOp**: Arguments shall be: property, left range, right range.

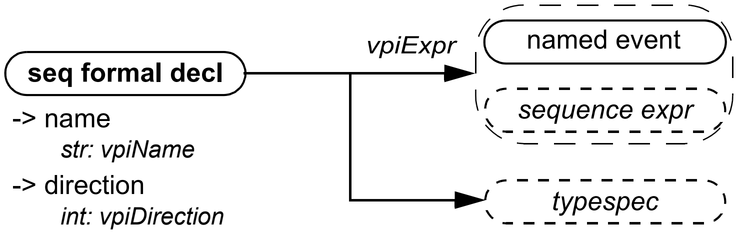
3) **vpiOpStrong** is valid only for operations **vpiNexttimeOp**, **vpiAlwaysOp**, **vpiEventuallyOp**, **vpiUntilOp**, **vpiUntilWithOp**, and for sequence expression. **vpiOpStrong** shall return TRUE to indicate the strong version of the corresponding operator.

4) The case property item shall group all case conditions that branch to the same property statement.

5) vpi_iterate() shall return NULL for the default case item because there is no expression with the default case.

37.51 Sequence declaration





Details:

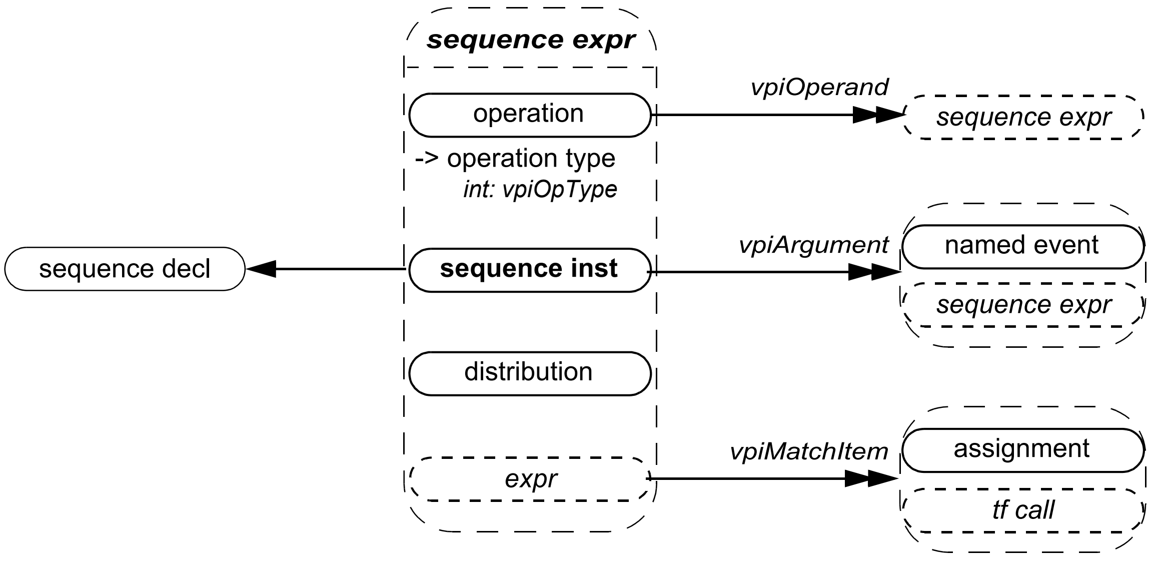
1) The **vpiSeqFormalDecl** iterator shall return the sequence declaration arguments in the order that the formals for the sequence are declared.

2) The **vpiTypespec** relation shall return NULL if the formal is untyped.

3) If the formal has an initialization expression, the expression can be obtained using the **vpiExpr** relation.

4) **vpiDirection** returns **vpiNoDirection** if the formal argument is not a local variable argument. Otherwise, **vpiDirection** returns either **vpiInput**, **vpiOutput**, or **vpiInout**.

37.52 Sequence expression



Details:

1) The **vpiArgument** iterator shall return the sequence instance arguments in the order that the formals for the sequence are declared, so that the correspondence between each argument and its respective formal can be made. If a formal has a default value, that value shall appear as the argument should the instantiation not provide a value for that argument.

2) Within a sequence expression, **vpiOpType** can be any one of **vpiCompAndOp**, **vpiIntersectOp**, **vpiCompOrOp**, **vpiFirstMatchOp**, **vpiThroughoutOp**, **vpiWithinOp**, **vpiUnaryCycleDelayOp**, **vpiCycleDelayOp**, **vpiRepeatOp**, **vpiConsecutiveRepeatOp**, or **vpiGotoRepeatOp**.

3) For operations, the operands shall be provided in the same order as the operands appear in BNF, with the following exceptions:

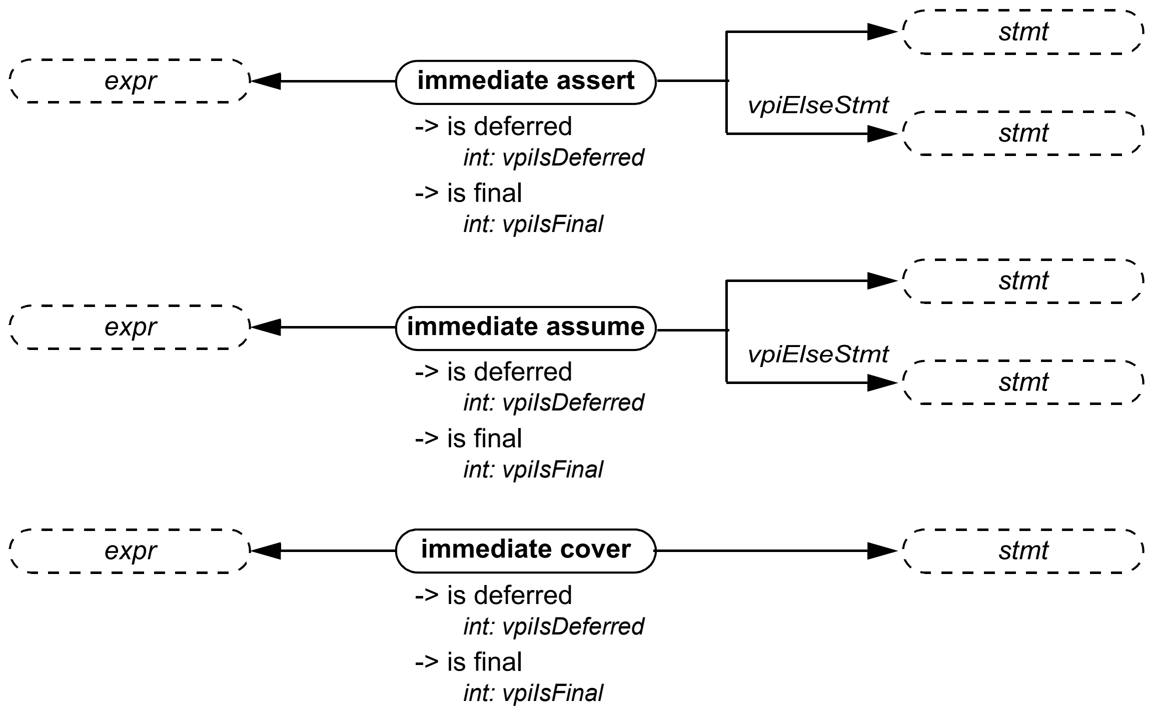
   — **vpiUnaryCycleDelayOp**: Arguments shall be: sequence, left range, right range. Right range shall only be given if different from left range.

   — **vpiCycleDelayOp**: Arguments shall be: left-hand side sequence, right-hand side sequence, left range, right range. Right range shall only be provided if different than left range.

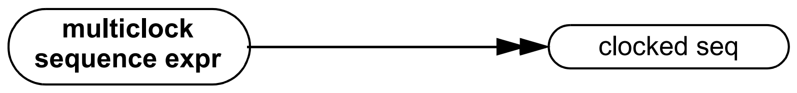
   — All the repeat operators: The first argument shall be the sequence being repeated, and the next argument shall be the left repeat bound, followed by the right repeat bound. The right repeat bound shall only be provided if different from left repeat bound.

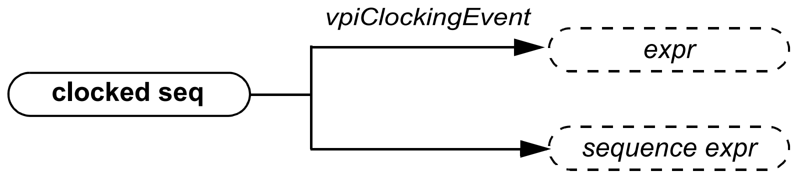
```
and, intersect, or,
first_match,
throughout, within,
##,
[*], [=], [->]
```

37.53 Immediate assertions

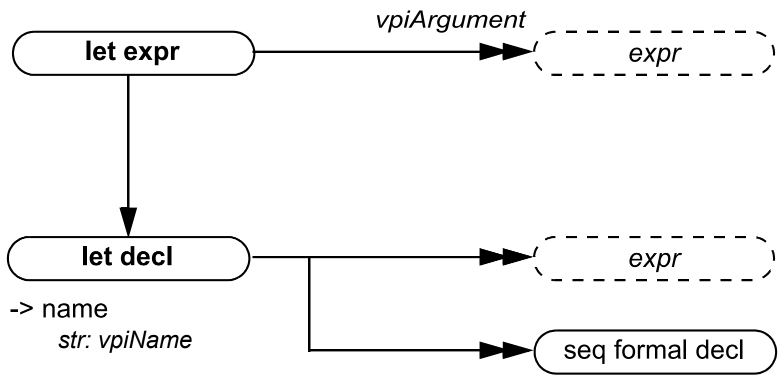


37.54 Multiclock sequence expression





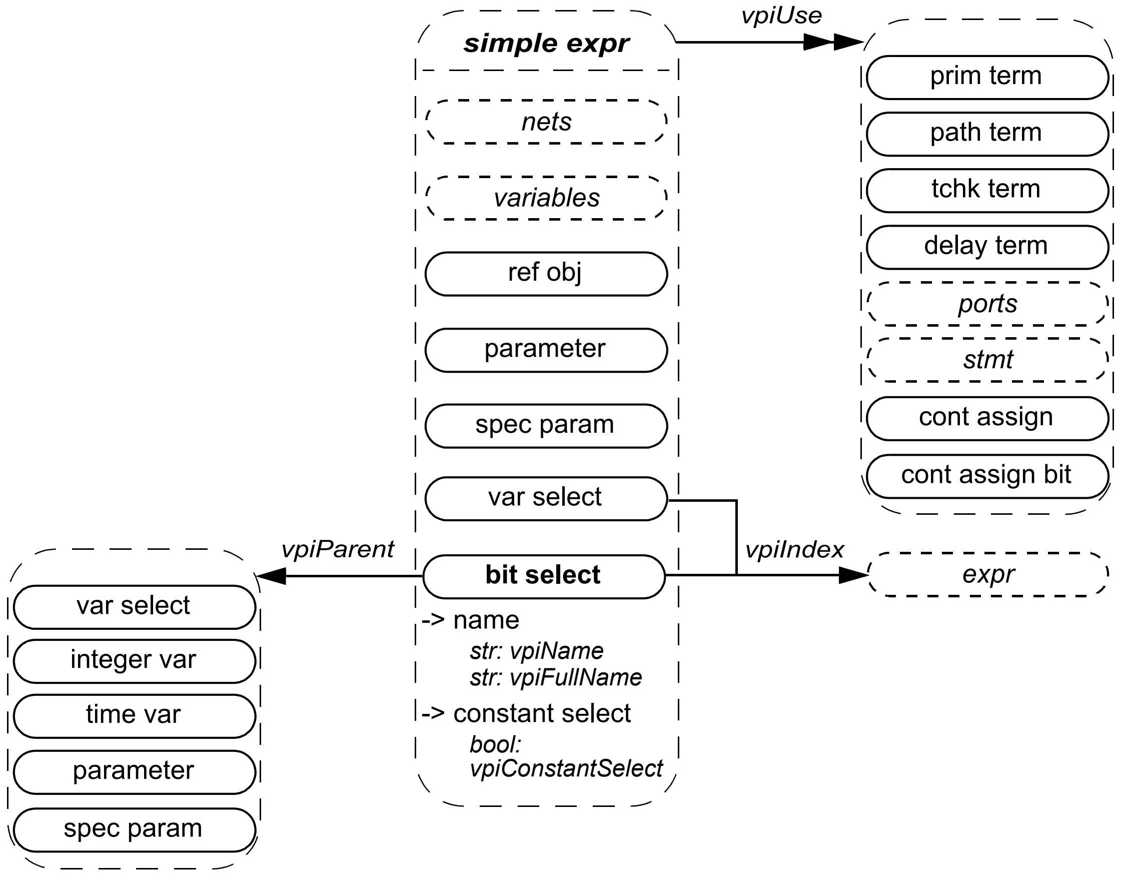
37.55 Let



Details:

1) The **vpiArgument** iterator shall return the let expression arguments in the order that the formals for the let are declared, so that the correspondence between each argument and its respective formal can be made. If a formal has a default value, that value shall appear as the argument should the instantiation not provide a value for that argument.

37.56 Simple expressions



Details:

1) For vectors, the **vpiUse** relationship shall access any use of the vector or of the part-selects or bit-selects of the vector.

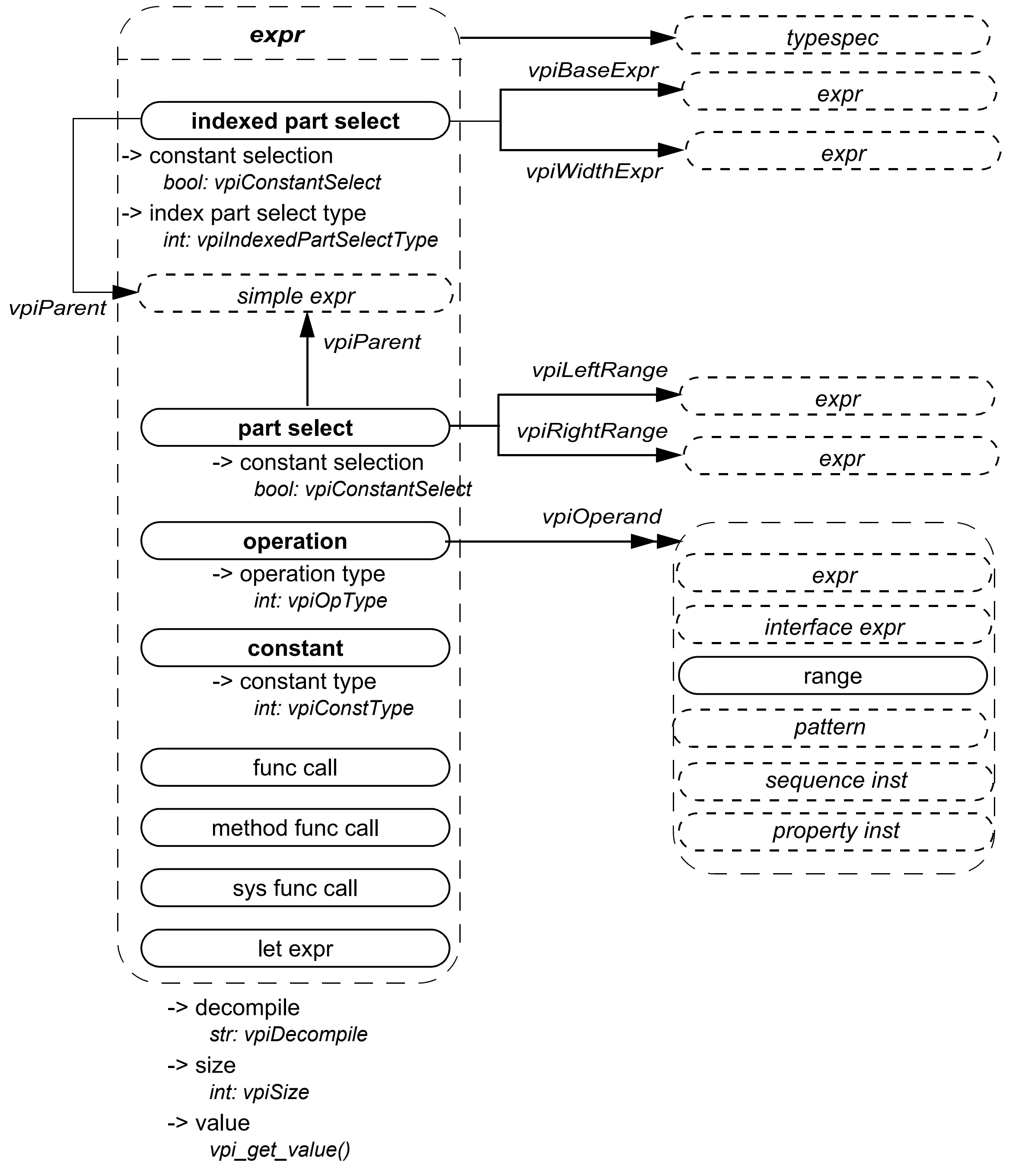
2) For bit-selects, the **vpiUse** relationship shall access any specific use of that bit, any use of the parent vector, and any part-select that contains that bit.

3) The property **vpiConstantSelect** shall return TRUE for a bit-select if
   - every associated index expression is an elaboration time constant expression, and
   - **vpiConstantSelect** returns TRUE for the parent of the bit-select.

   Otherwise, **vpiConstantSelect** shall return FALSE.

   NOTE—If **vpiConstantSelect** is TRUE, then if the handle refers to a valid underlying simulation object at the beginning of simulation (or at any point in the simulation), it refers to the same object at all points in the simulation. Moreover, if an index expression of the bit-select or of any of its parents is in or out of bounds at the beginning of simulation, it is in or out of bounds at all subsequent simulation times as well.

37.57 Expressions



Details:

1) For an operator whose type is **vpiMultiConcatOp**, the first operand shall be the multiplier expression. The remaining operands shall be the expressions within the concatenation.

2) The property **vpiDecompile** shall return a string with a functionally equivalent expression to the original expression within the source code. Parentheses shall be added only to preserve precedence. Each operand and operator shall be separated by a single space character. No additional white space shall be added due to parentheses.

3) The cast operation, for which **vpiOpType** returns **vpiCastOp**, is represented as a unary operation, with its sole argument being the expression being cast, and the typespec of the cast expression being the type to which the argument is being cast.

4) The constant type **vpiUnboundedConst** represents the **$** value used in assertion ranges.

5) The one-to-one relation to typespec shall always be available for **vpiCastOp** operations, for simple expressions, and for **vpiAssignmentPatternOp** and **vpiMultiAssignmentPatternOp** expressions when the curly braces of the assignment pattern are prefixed by a data type name to form an assignment pattern expression. For other expressions, it is implementation dependent as to whether or not there is any associated typespec.

6) For an operation of type **vpiAssignmentPatternOp**, the operand iteration shall return the expressions as if the assignment pattern were written with the positional notation. Nesting of assignment patterns shall be preserved.

Example 1:

```
struct {
   int A;
   struct {
     logic B;
     real C;
   } BC1, BC2;
} ABC = '{BC1: '{1'b1, 1.0}, int: 0, BC2: '{default: 0}};
```

The assignment pattern that initializes the struct variable `ABC` uses member, type, and default keys. The **vpiOperand** traversal would represent this assignment pattern expression as:

```
'{0, '{1'b1, 1.0}, '{0, 0}}
```

or some other equivalent positional assignment pattern.

Example 2:

```
logic [2:0] varr [0:3] = '{3: 3'b1, default: 3'b0};
```

The assignment pattern that initializes the array variable `varr` uses index and default keys. The **vpiOperand** traversal would represent this assignment pattern as:

```
'{3'b0, 3'b0, 3'b0, 3'b1}
```

7) For an operator whose type is **vpiMultiAssignmentPatternOp**, the first operand shall be the multiplier expression. The remaining operands shall be the expressions within the assignment pattern.

Example:

```
bit unpackedbits [1:0];
initial unpackedbits = '{2 {y}} ; // same as '{y, y}
```

For the assignment pattern `'{2{y}}`, the **vpiOpType** property shall return **vpiMultiAssignmentPatternOp**, and the first operand shall be the constant 2. The next operand shall represent the expression `y`.

8) Expressions that are protected shall permit access to the **vpiSize** property.

9) The property **vpiConstantSelect** shall return TRUE for a part-select or indexed part-select if

— **vpiConstantSelect** returns TRUE for its parent, and

— the parent is a packed or unpacked array with static bounds, and

— each range expression in the part-select or indexed part-select is an elaboration time constant expression.

Otherwise, **vpiConstantSelect** shall return FALSE.

NOTE—If **vpiConstantSelect** is TRUE, then if the handle refers to a valid underlying simulation object at the beginning of simulation (or at any point in the simulation), it refers to the same object at all points in the simulation. Moreover, if any index expression of the part-select or indexed part-select or of any of its parents is in or out of bounds at the beginning of simulation, it is in or out of bounds at all subsequent simulation times as well.

10) For a part-select or indexed part-select, the **vpiParent** object shall correspond to the expression formed by removing the part-select range from the expression represented by the part-select or indexed part-select itself. For example, given the declaration

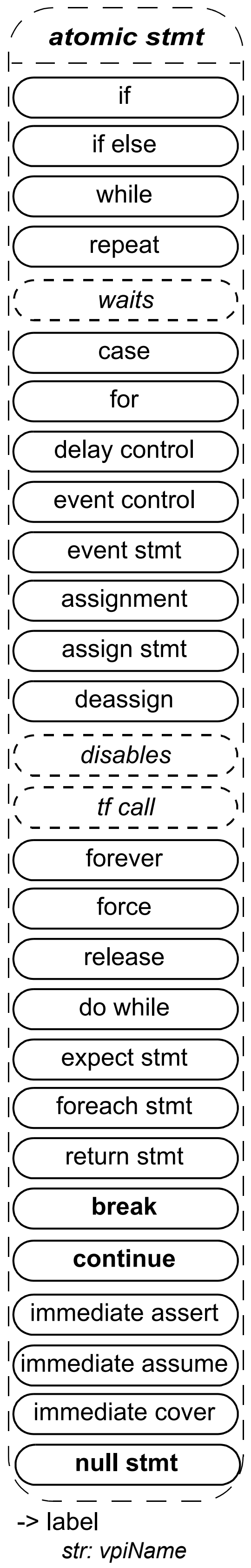
```
logic [0:3][7:0] r [1:4];
```

then the parents of various part-selects or indexed part-selects shall be as shown in Table 37-1:

Table 37-1—Part-select parent expressions

Part-select or indexed part-select expression	Parent expression
`r[4][3][1:0]`	`r[4][3]`
`r[i+1][3][j+:2]`	`r[i+1][3]`
`r[0][j-:4]`	`r[0]`
`r[0:2]`	`r`

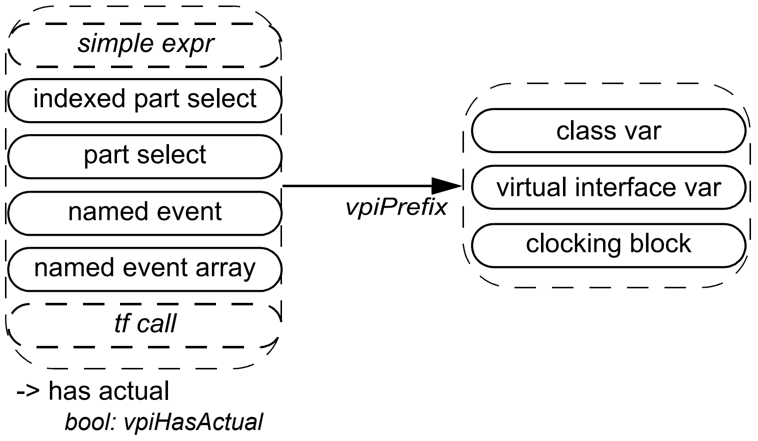
37.58 Atomic statement



Details:

1) The **vpiName** property shall provide the statement label if one was given; otherwise, the name is `NULL`.

37.59 Dynamic prefixing



Details:

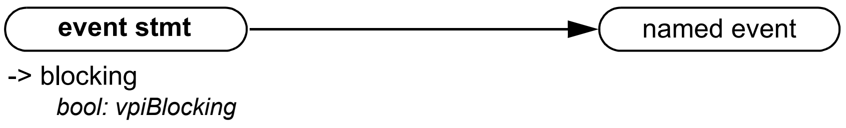
1) The **vpiPrefix** relation shall be non-NULL when the object represents an expression or task call in the SystemVerilog source code prefixed by a virtual interface or a clocking block, or when the object is all or part of a non-static class property prefix ed by a class var.

2) The memory allocation scheme value for an object for which a class var or virtual interface var **vpiPrefix** is non-NULL shall be the same as for the prefix.

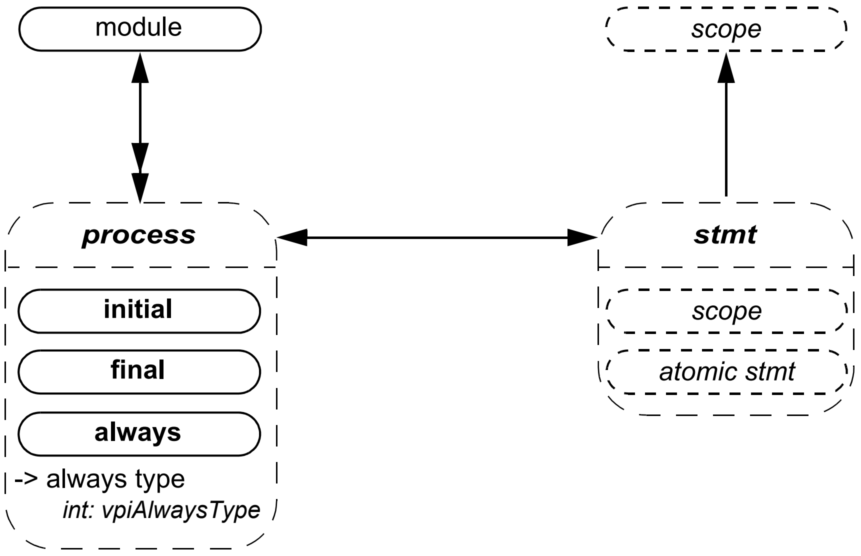
3) The property **vpiHasActual** shall return TRUE:
   - whenever the prefix object has a corresponding actual at the current simulation time.
   - if the object is all or part of a statically declared object in an elaborated context.
   - if the object is part or all of an automatically allocated variable obtained from a frame (see 37.41).

   The property **vpiHasActual** shall return FALSE:
   - whenever the prefix object has no corresponding actual at the current simulation time.
   - if the object is obtained from a lexical context, such as from a class defn (see 37.29).
   - if the object is part or all of a non-static class property variable referenced relative to its class typespec (see 37.30).
   - if the object is part or all of an automatically allocated variable obtained from a task or function declaration (see 37.39).

37.60 Event statement



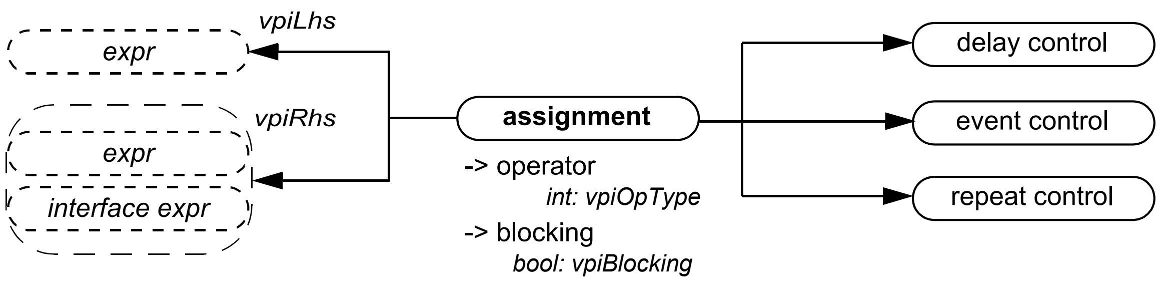
37.61 Process



Details:

1) **vpiAlwaysType** can be one of **vpiAlways**, **vpiAlwaysComb**, **vpiAlwaysFF**, or **vpiAlwaysLatch**.

37.62 Assignment



Details:

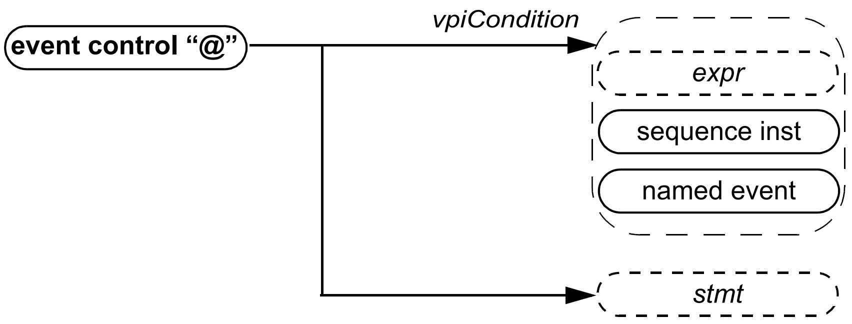
1) **vpiOpType** shall return **vpiAssignmentOp** for normal assignments (both blocking "=" and nonblocking "<="). For assignment operators, **vpiOpType** shall return a value that corresponds to the operator that is combined with the assignment as described in 11.4.1.

   For example, the assignment

```
a += 2;
```

   shall return **vpiAddOp** for the **vpiOpType** property.

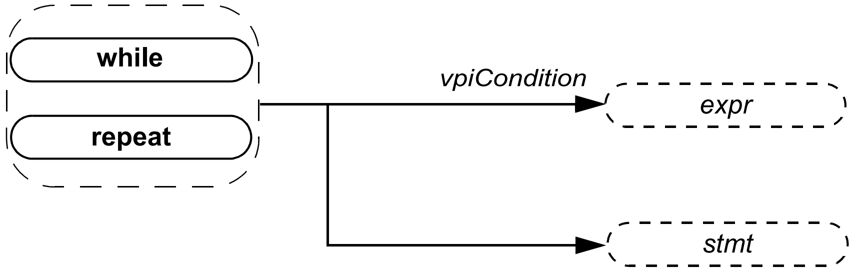
37.63 Event control



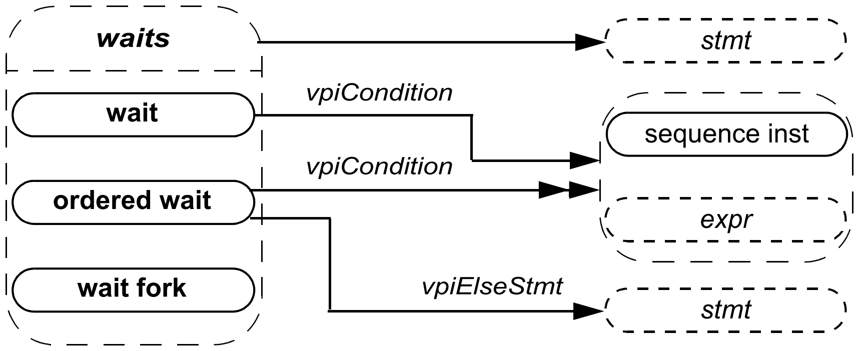
Details:

1) For event control associated with assignment, the statement shall always be `NULL`.

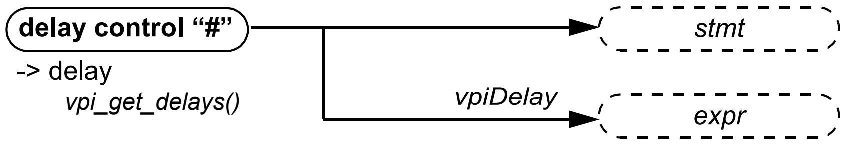
37.64 While, repeat



37.65 Waits



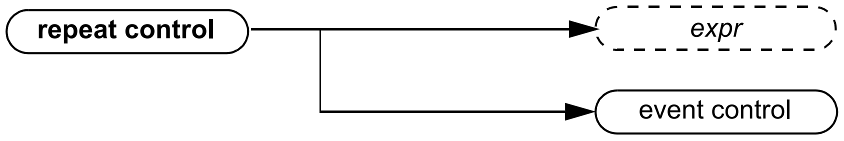
37.66 Delay control



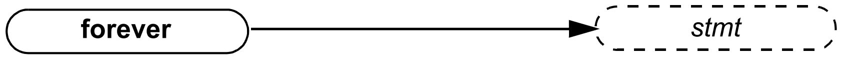
Details:

1) For delay control associated with assignment, the statement shall always be `NULL`.

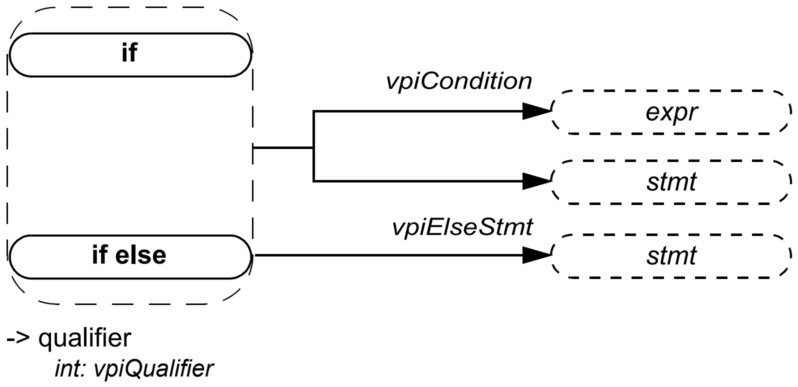
37.67 Repeat control



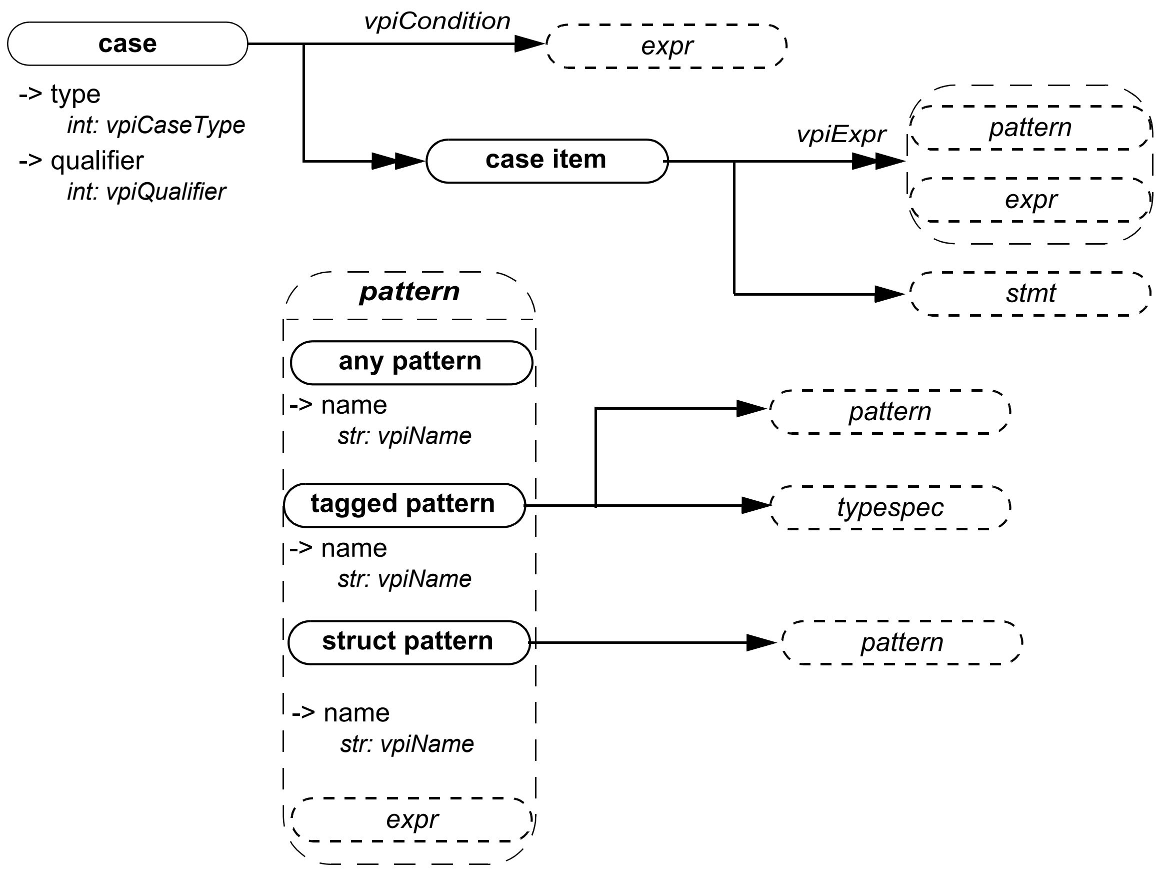
37.68 Forever



37.69 If, if–else



37.70 Case, pattern

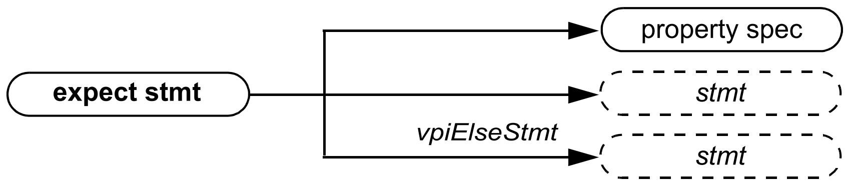


Details:

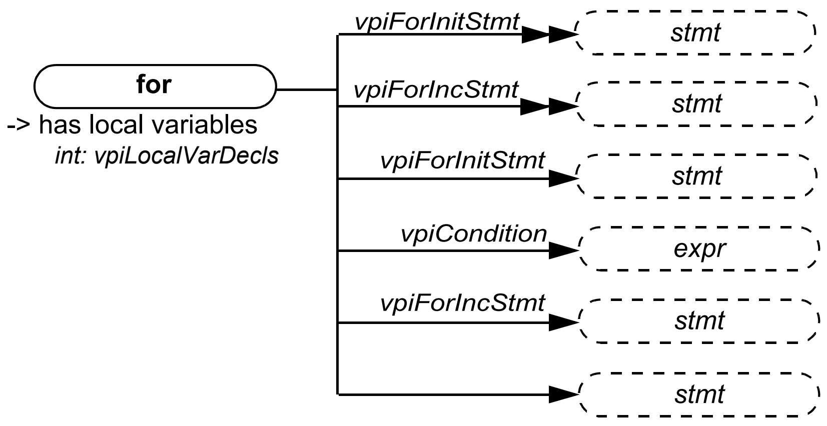
1) The case item shall group all case conditions that branch to the same statement.

2) **vpi_iterate()** shall return NULL for the default case item because there is no expression with the default case.

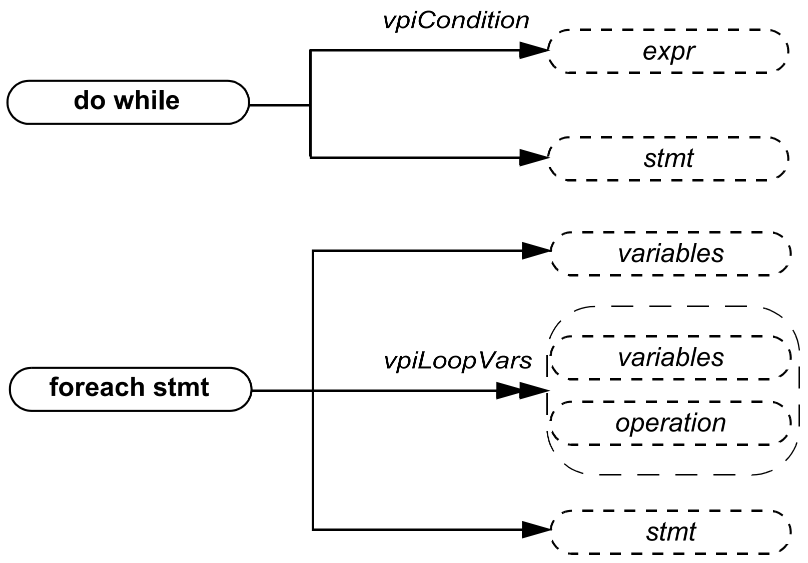
37.71 Expect



37.72 For



37.73 Do-while, foreach

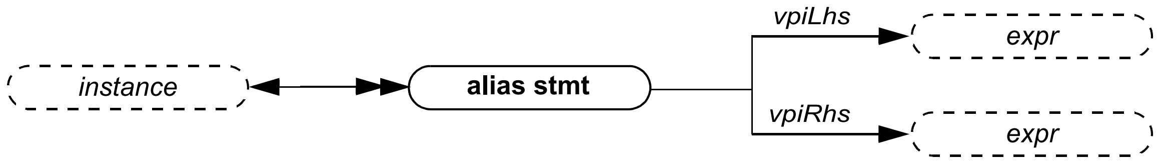


Details:

1) The variable obtained via the **vpiVariables** relation from a foreach stmt shall represent the packed array, unpacked array, or string var being indexed.

2) The **vpiLoopVars** iteration shall return the index variables of the foreach statement in left-to-right order. If an index variable is skipped, its place shall be represented as a **vpiOperation** for which the **vpiOpType** is **vpiNullOp**.

37.74 Alias statement



Example:

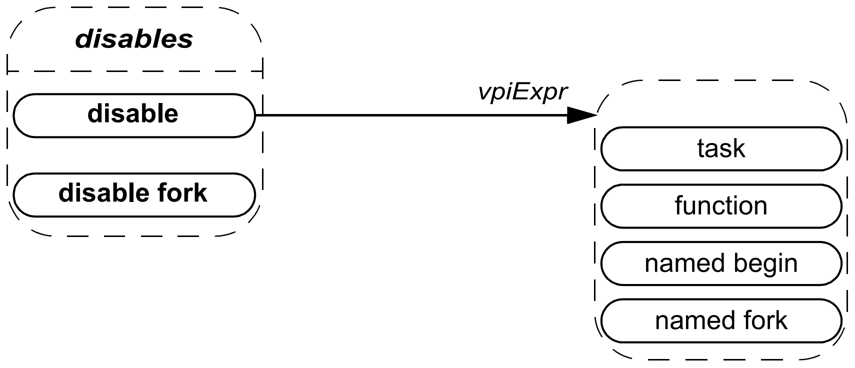
```
    alias a=b=c=d;
```

results in 3 aliases:

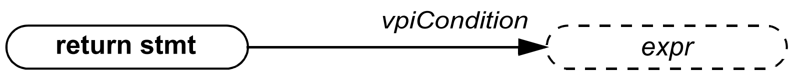
```
        alias a=d;
        alias b=d;
        alias c=d;
```

`d` is the right-hand side for all.

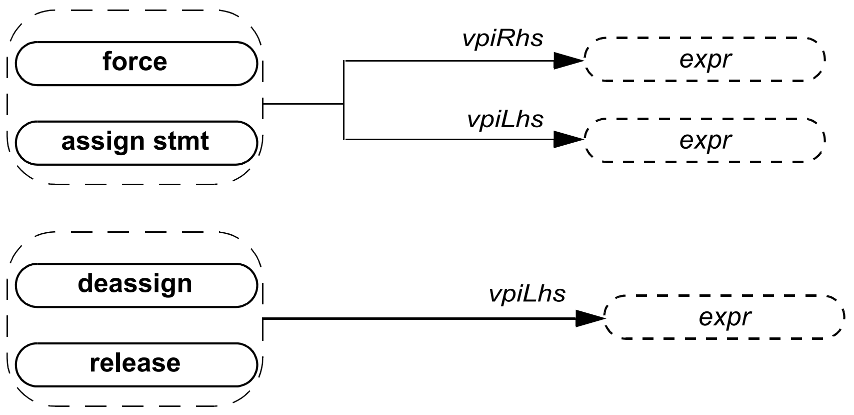
37.75 Disables



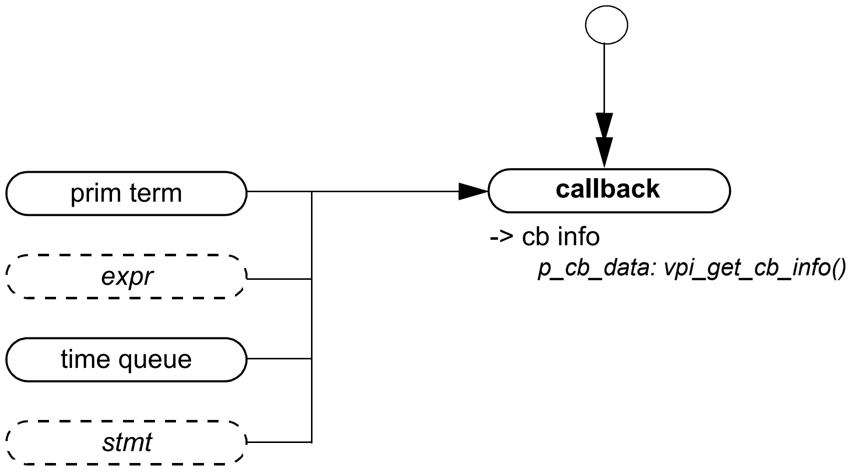
37.76 Return statement



37.77 Assign statement, deassign, force, release



37.78 Callback

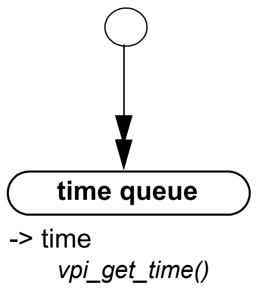


Details:

1) To get information about the callback object, the routine **vpi_get_cb_info()** can be used..

2) To get callback objects not related to the above objects, the second argument to **vpi_iterate()** shall be NULL.

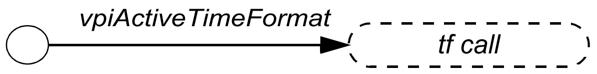
37.79 Time queue



Details:

1) The time queue objects shall be returned in increasing order of simulation time.
2) **vpi_iterate()** shall return NULL if there is nothing left in the simulation time queue.
3) The current time queue shall only be returned as part of the iteration if there are events that precede read only sync.

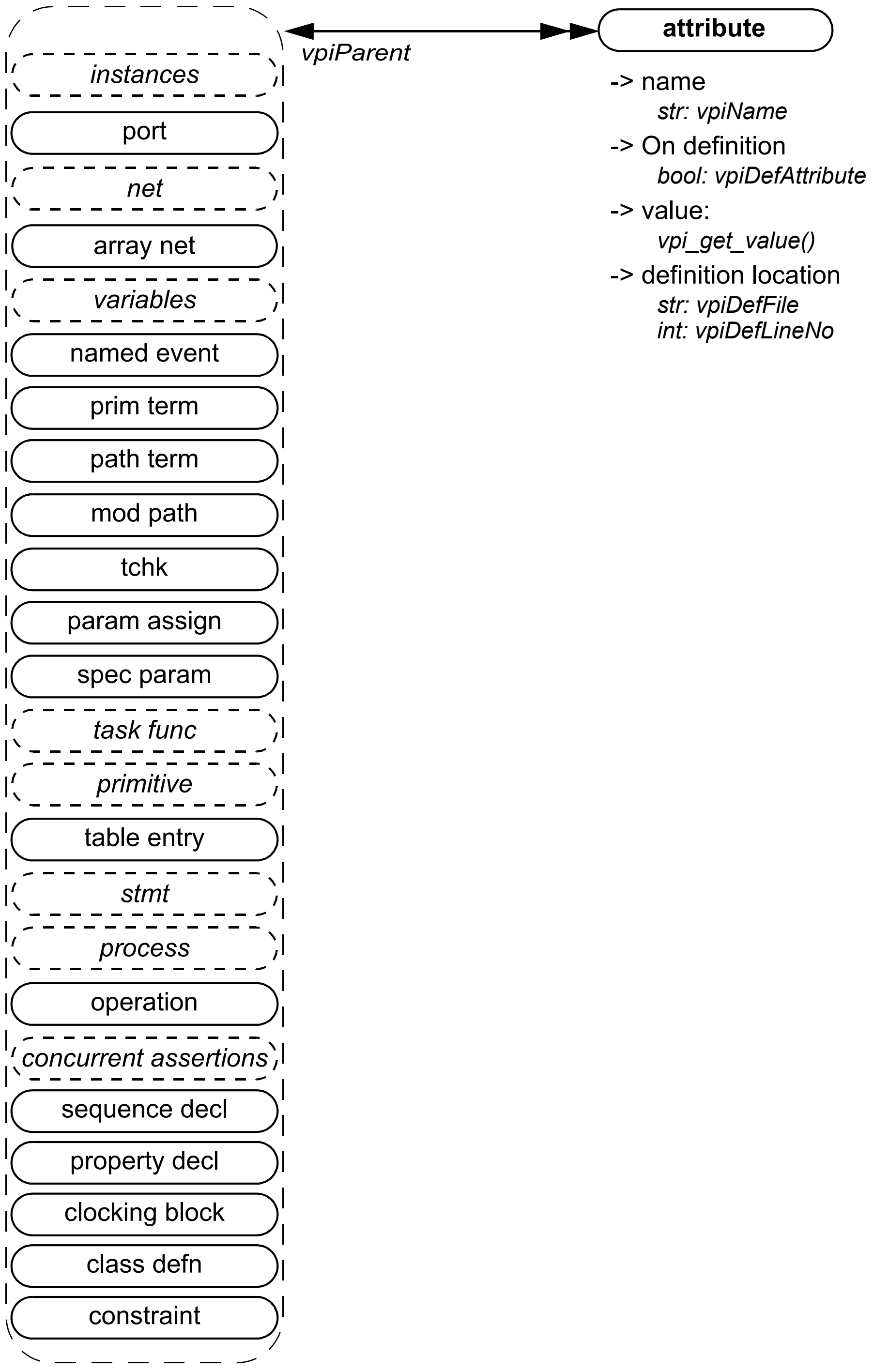
37.80 Active time format



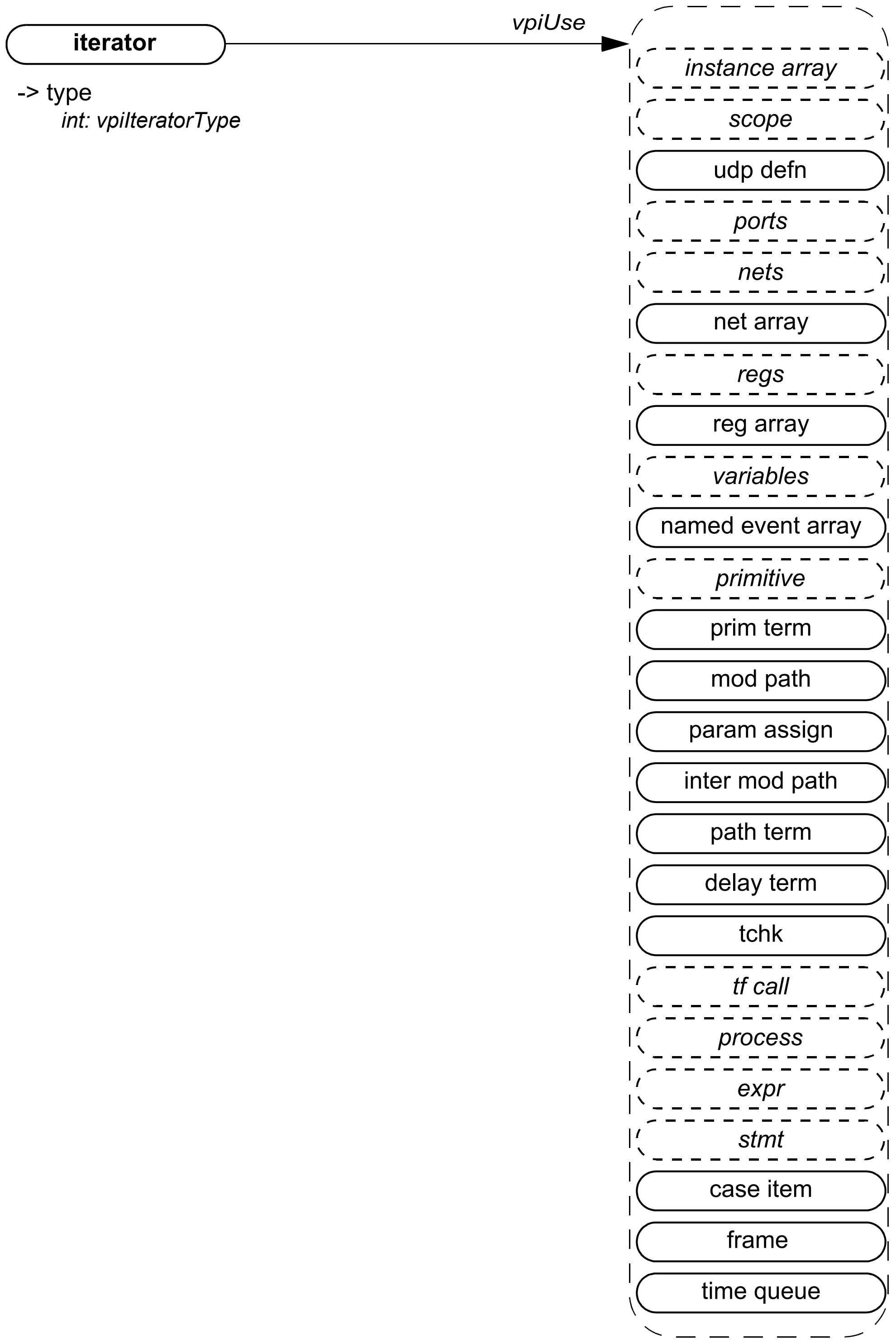
Details:

1) If `$timeformat()` has not been called, **vpi_handle(vpiActiveFormat,** NULL**)** shall return NULL.

37.81 Attribute



37.82 Iterator

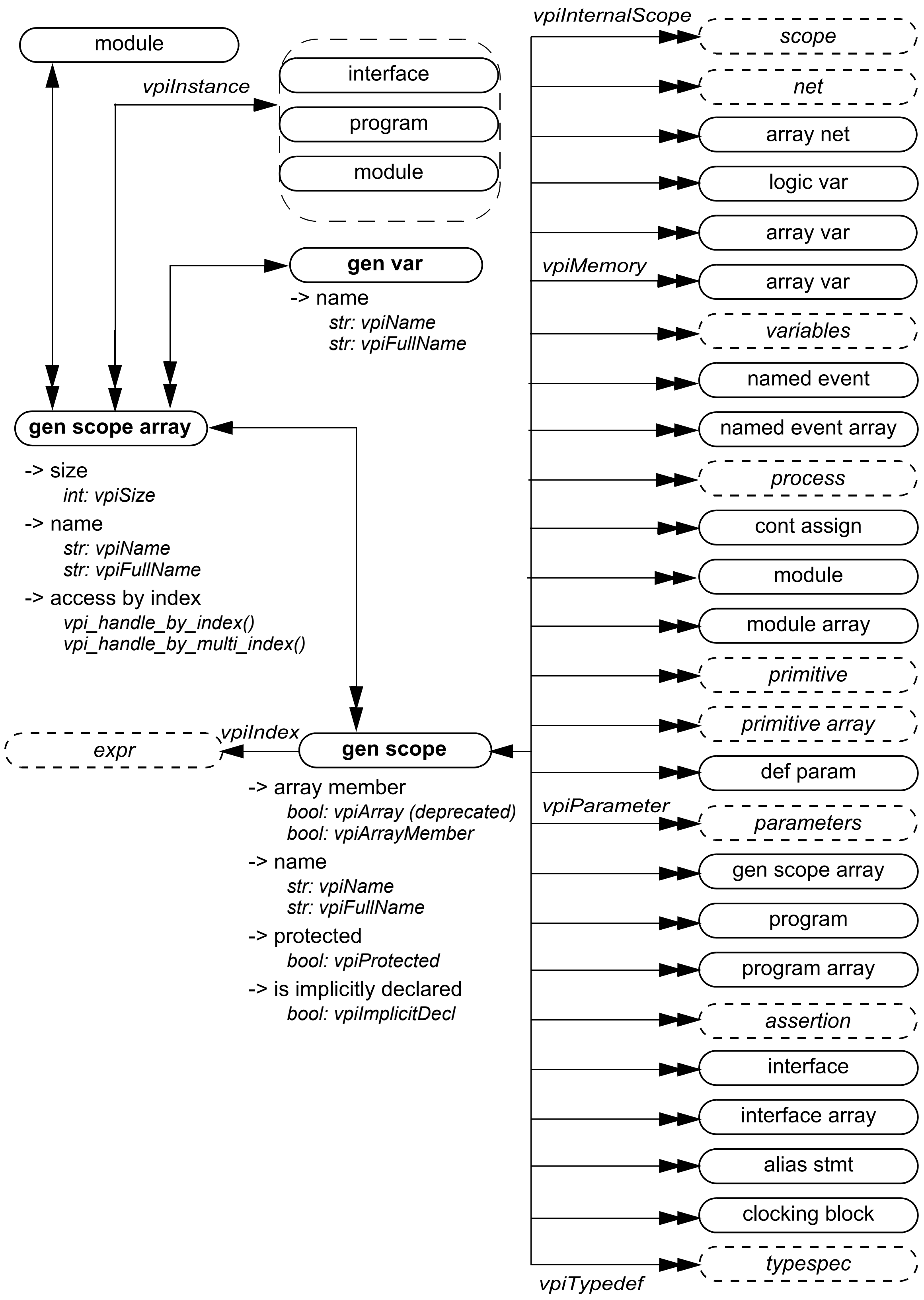


Details:

1) **vpi_handle(vpiUse,** iterator_handle**)** shall return the reference handle used to create the iterator.

2) It is possible to have a NULL reference handle, in which case **vpi_handle(vpiUse**, iterator_handle**)** shall return NULL.

37.83 Generates


module
vpiInstance
interface
program
module
gen var
-> name
str: vpiName
str: vpiFullName
gen scope array
-> size
int: vpiSize
-> name
str: vpiName
str: vpiFullName
-> access by index
vpi_handle_by_index()
vpi_handle_by_multi_index()
expr
vpiIndex
gen scope
-> array member
bool: vpiArray (deprecated)
bool: vpiArrayMember
-> name
str: vpiName
str: vpiFullName
-> protected
bool: vpiProtected
-> is implicitly declared
bool: vpiImplicitDecl
vpiInternalScope
scope
net
array net
logic var
array var
vpiMemory
array var
variables
named event
named event array
process
cont assign
module
module array
primitive
primitive array
def param
vpiParameter
parameters
gen scope array
program
program array
assertion
interface
interface array
alias stmt
clocking block
typespec
vpiTypedef