14. Elaboration and execution

14.1 General

The process by which a declaration achieves its effect is called the *elaboration* of the declaration. After its elaboration, a declaration is said to be elaborated. Prior to the completion of its elaboration (including before the elaboration), the declaration is not yet elaborated.

Elaboration is also defined for design hierarchies, declarative parts, statement parts (containing concurrent statements), and concurrent statements. Elaboration of such constructs is necessary in order ultimately to elaborate declarative items that are declared within those constructs.

In order to execute a model, the design hierarchy defining the model shall first be elaborated. Initialization of nets (see 14.7.3.4) in the model then occurs. Finally, simulation of the model proceeds. Simulation consists of the repetitive execution of the *simulation cycle*, during which processes are executed and nets updated.

14.2 Elaboration of a design hierarchy

The elaboration of a design hierarchy creates a collection of processes interconnected by nets; this collection of processes and nets can then be executed to simulate the behavior of the design.

At the beginning of the elaboration of a design hierarchy, every registered and enabled `vhpiCbStartOfElaboration` callback is executed. Once the elaboration of a given design hierarchy is complete, every registered and enabled `vhpiCbEndOfElaboration` callback is executed.

A design hierarchy is defined either by a design entity or by a configuration.

An implementation may allow PSL verification units, in addition to any whose binding is specified as part of the design hierarchy, to be bound to design entities within the design hierarchy. The manner in which such PSL verification units are identified and the manner in which binding is specified for such PSL verification units that are not explicitly bound are not defined by this standard.

Elaboration of a design hierarchy defined by a design entity consists of the elaboration of the block statement equivalent to the external block defined by the design entity. The architecture of this design entity is assumed to contain an implicit configuration specification (see 7.3) for each component instance that is unbound in this architecture; each configuration specification has an entity aspect denoting an anonymous configuration declaration identifying the visible entity declaration (see 7.3.3) and supplying an implicit block configuration (see 3.4.2) that binds and configures a design entity identified according to the rules of 7.3.3. The equivalent block statement is defined in 11.7.3. Elaboration of a block statement is defined in 14.5.2.

Elaboration of a configuration consists of the elaboration of the block statement equivalent to the external block defined by the design entity configured by the configuration. The configuration contains an implicit component configuration (see 3.4.3) for each unbound component instance contained within the external block and an implicit block configuration (see 3.4.2) for each internal block contained within the external block.

An implementation may allow, but is not required to allow, a design entity at the root of a design hierarchy to have generics and ports. If an implementation allows these *top-level* interface objects, it may restrict their allowed forms (that is, whether they are allowed to be interface types, subprograms, packages, or objects), and, in the case of interface objects, their allowed types and modes in an implementation-defined manner.

Similarly, the means by which top-level interface objects are associated with the external environment of the hierarchy are also defined by an implementation supporting top-level interface objects.

Elaboration of a block statement involves first elaborating each not-yet-elaborated package primary unit or package instantiation primary unit containing declarations referenced by the block. Similarly, elaboration of a given package primary unit or package instantiation primary unit involves first elaborating each not-yet-elaborated package primary unit or package instantiation primary unit containing declarations referenced by the given package or package instantiation. Elaboration of a package primary unit consists additionally of the following:

a) Elaboration of the package declaration, eventually followed by

b) Elaboration of the corresponding package body, if the package has a corresponding package body.

Elaboration of a package instantiation primary unit consists of elaboration of the equivalent generic-mapped package declaration, eventually followed by elaboration of the corresponding equivalent generic-mapped package body, if such a package body is defined (see 4.9).

Step b), the elaboration of a package body, may be deferred until the declarations of other packages have been elaborated, if necessary, because of the dependencies created between packages by their interpackage references. Similarly, elaboration of an equivalent generic-mapped package body may be deferred if necessary.

Elaboration of a package is defined in 14.4.2.9.

For a block statement implied by a design entity, whether the design entity at the root of the design hierarchy or a design entity bound to a component instance, to which one or more PSL verification units are bound, after elaboration of the implied block statement, each PSL verification unit bound to the design entity is elaborated. Elaboration of a PSL verification unit involves first elaborating each not-yet-elaborated package primary unit or package instantiation primary unit containing declarations referenced by the PSL verification unit. Further interpretation of the PSL verification unit is defined in IEEE Std 1850-2005.

Elaboration of a design hierarchy is completed as follows:

— The drivers identified during elaboration of process statements (see 14.5.5) are created.

— The initial transaction defined by the default value associated with each scalar signal driven by a process statement is inserted into the corresponding driver.

During elaboration of a design hierarchy, if an external name or alias of an external name appears in a declaration or statement being elaborated, then in the following cases, the declaration of the object denoted by the external name or alias shall have been previously elaborated:

— If the external name or alias is a primary or a prefix of a primary in an expression that is evaluated during elaboration of the design hierarchy, when the primary is read during evaluation of the expression.

— If the external name or alias, or a name in which the external name or alias is a prefix, is associated as an actual in an association element in a port map aspect, when the association element is elaborated.

NOTE—Since elaboration of declarations and statements occurs in the order of their appearance in a description, prior elaboration of an object denoted by an external name may be ensured by an appropriate ordering of the declarations and statements in the description.

Examples:

```
--  In the following example, because of the dependencies between
--  the packages, the elaboration of either package body shall
--  follow the elaboration of both package declarations.
```



```
package P1 is
   constant C1: INTEGER := 42;
   constant C2: INTEGER;
end package P1;

package P2 is
   constant C1: INTEGER := 17;
   constant C2: INTEGER;
end package P2;

package body P1 is
   constant C2: INTEGER := Work.P2.C1;
end package body P1;

package body P2 is
   constant C2: INTEGER := Work.P1.C1;
end package body P2;

--  If a design hierarchy is described by the following design entity:

entity E is end;

architecture A of E is
   component comp
      port (...);
   end component;
begin
   C: comp port map (...);
   B: block
      ...
   begin
      ...
   end block B;
end architecture A;

--  then its architecture contains the following implicit configuration
--  specification at the end of its declarative part:

for C: comp use configuration anonymous;

--  and the following configuration declaration is assumed to exist
--  when E(A) is elaborated:

configuration anonymous of L.E is    --  L is the library in which
                                     --  E(A) is found.
   for A                             --  The most recently analyzed
                                     --  architecture of L.E.
   end for;
end configuration anonymous;

--  In the following example, each appearance of an external name is
--  legal or illegal as noted.
```

```
entity TOP is
end entity TOP;

architecture ARCH of TOP is
   signal S1, S2, S3: BIT;
   alias DONE_SIG is <<signal .TOP.DUT.DONE: BIT>>;  -- Legal
   constant DATA_WIDTH: INTEGER
                        := <<signal .TOP.DUT.DATA: BIT_VECTOR>>'LENGTH;
      --  Illegal, because .TOP.DUT.DATA has not yet been elaborated
      --  when the expression is evaluated
begin
   P1: process ( DONE_SIG ) is  -- Legal
   begin
      if DONE_SIG then  -- Legal
         ...;
      end if;
   end process P1;
   MONITOR: entity WORK.MY_MONITOR port map (DONE_SIG);
      --  Illegal, because .TOP.DUT.DONE has not yet been elaborated
      --  when the association element is elaborated
   DUT: entity WORK.MY_DESIGN port map (s1, S2, S3);
   MONITOR2: entity WORK.MY_MONITOR port map (DONE_SIG);
      -- Legal, because .TOP.DUT.DONE has now been elaborated
   B1: block
      constant DATA_WIDTH: INTEGER
         := <<signal .TOP.DUT.DATA: BIT_VECTOR>>'LENGTH
         -- Legal, because .TOP.DUT.DATA has now been elaborated
   begin
   end block B1;
   B2: block
      constant C0: INTEGER := 6;
      constant C1: INTEGER := <<constant .TOP.B3.C2: INTEGER>>;
         -- Illegal, because .TOP.B3.C2 has not yet been elaborated
   begin
   end block B2;
   B3: block
      constant C2: INTEGER
                   := <<constant .TOP.B2.C0: INTEGER>>;  -- Legal
   begin
   end block B3;
   --  Together, B2 and B3 are illegal, because they cannot be ordered
   --  so that the objects are elaborated in the order .TOP.B2.C0,
   --  then .TOP.B3.C2, and finally .TOP.B2.C1.
end architecture ARCH;
```

14.3 Elaboration of a block, package, or subprogram header

14.3.1 General

Elaboration of a block header consists of the elaboration of the generic clause, the generic map aspect, the port clause, and the port map aspect. Similarly, elaboration of a package header consists of the elaboration of the generic clause and the generic map aspect; and elaboration of a subprogram header consists of the

elaboration of the generic clause equivalent to the generic list of the subprogram header and the generic map aspect.

14.3.2 Generic clause

Elaboration of a generic clause consists of the elaboration of each of the equivalent single generic declarations contained in the clause, in the order given. The elaboration of a generic declaration establishes that the generic can subsequently be referenced.

14.3.3 Generic map aspect

14.3.3.1 General

Elaboration of a generic map aspect consists of elaborating the generic association list. The generic association list contains an implicit association element for each generic constant that is not explicitly associated with an actual or that is associated with the reserved word **open**; the actual part of such an implicit association element is the default expression appearing in the declaration of that generic constant. Similarly, the generic association list contains an implicit association element for each generic subprogram that is not explicitly associated with an actual or that is associated with the reserved word **open**; the actual part of such an implicit association element is determined by the interface subprogram default as described in 6.5.6.2. The generic association list also contains implicit association elements for the predefined equality (=) operator and inequality (/=) operators of each generic type; the actual part of such an implicit association element is the name of the predefined equality operator or inequality operator for the base type of the subtype indication in the actual part of the association element corresponding to the generic type.

Elaboration of a generic association list consists of the elaboration of the generic association element or elements in the association list associated with each generic declaration, in the order given by the generic declarations in the generic clause.

14.3.3.2 Association elements for generic constants

Elaboration of the generic association elements associated with a generic constant declaration proceeds as follows:

a) The subtype indication of the corresponding generic declaration is elaborated.

b) The formal part or parts of the generic association elements corresponding to the generic declaration are elaborated.

c) If the type of the generic constant is an array type or contains a subelement that is of an array type, the rules of 5.3.2.2 are applied to determine the index ranges.

d) The generic constant is created.

The generic constant or subelement or slice thereof designated by each formal part is then initialized with the value resulting from the evaluation of the corresponding actual part. It is an error if the value of the actual does not belong to the subtype denoted by the subtype indication of the formal. If the subtype denoted by the subtype indication of the declaration of the formal is a composite subtype, then an implicit subtype conversion is performed prior to this check. It is also an error if the type of the formal is an array type and the value of each element of the actual does not belong to the element subtype of the formal.

14.3.3.3 Association elements for generic types

Elaboration of the generic association element associated with a generic type declaration involves the elaboration of the subtype indication in the actual part followed by creating the generic type and defining it to denote the subtype resulting from elaboration of the actual part.

14.3.3.4 Association elements for generic subprograms

Elaboration of the generic association element associated with a generic subprogram declaration proceeds as follows:

a) The parameter list of the formal generic subprogram declaration is elaborated. This involves the elaboration of the subtype indication of each interface element to determine the subtype of each formal parameter of the formal generic subprogram.

b) The generic subprogram is then defined to denote the subprogram denoted by the subprogram name in the actual part.

14.3.3.5 Association elements for generic packages

For a generic association element associated with a generic package declaration, if the generic package declaration contains an interface package generic map aspect in the form that includes the box (<>) symbol, elaboration of the generic association element involves defining the generic package to denote the instantiated package denoted by the instantiated package name in the actual part. Otherwise, elaboration of the generic association element proceeds as follows:

a) An implicit package header formed from the generic clause of the uninstantiated package named in the formal package declaration and the generic map aspect (whether explicit or implicit, see 6.5.5) of the interface package generic map aspect is elaborated.

b) A check is made that the generic map aspect of the package instantiation declaration that declares the instantiated package denoted by the instantiated package name in the actual part matches the elaborated generic map aspect of the implicit package header.

c) The generic package is defined to denote the instantiated package denoted by the instantiated package name in the actual part.

14.3.4 Port clause

Elaboration of a port clause consists of the elaboration of each of the equivalent single port declarations contained in the clause, in the order given. The elaboration of a port declaration establishes that the port can subsequently be referenced.

14.3.5 Port map aspect

Elaboration of a port map aspect consists of elaborating the port association list.

Elaboration of a port association list consists of the elaboration of the port association element or elements in the association list associated with each port declaration. If the actual in a port association element is an expression that is not globally static, or if the actual part includes the reserved word **inertial**, then elaboration of the port association element first consists of constructing and elaborating the equivalent anonymous signal declaration, concurrent signal assignment statement, and port association element (see 6.5.6.3); the port or subelement or slice thereof designated by the formal part is then associated with the anonymous signal.

Elaboration of the port association elements associated with a port declaration proceeds as follows:

a) The subtype indication of the corresponding port declaration is elaborated.

b) The formal part or parts of the port association elements corresponding to the port declaration are elaborated.

c) If the type of the port is an array type or contains a subelement that is of an array type, the rules of 5.3.2.2 are applied to determine the index ranges.

d) For each port association element associated with the port declaration, if the actual is not the reserved word **open**, the port or subelement or slice thereof designated by the formal part is then associated with the signal or expression designated by the actual part. This association involves a check that the restriction on port associations (see 6.5.6.3) are met. It is an error if this check fails.

If a given port is a port of mode **in** whose declaration includes a default expression, and if no association element associates a signal or expression with that port, then the default expression is evaluated and the effective and driving value of the port is set to the value of the default expression. Similarly, if a given port of mode **in** is associated with an expression that is globally static and the reserved word **inertial** does not appear in the actual part of the association element, that expression is evaluated and the effective and driving value of the port is set to the value of the expression. In the event that the value of a port is derived from an expression in either fashion, references to the predefined attributes 'DELAYED, 'STABLE, 'QUIET, 'EVENT, 'ACTIVE, 'LAST_EVENT, 'LAST_ACTIVE, 'LAST_VALUE, 'DRIVING, and 'DRIVING_VALUE of the port return values indicating that the port has the given driving value with no activity at any time (see 14.7.4).

If an actual signal is associated with a port of mode **in** or **inout**, and if the type of the formal is a scalar type, then it is an error if (after applying any conversion function or type conversion expression present in the actual part) the subtype of the actual is not compatible with the subtype of the formal. If an actual expression is associated with a formal port (of mode **in**), and if the type of the formal is a scalar type, then it is an error if the value of the expression does not belong to the subtype denoted by the subtype indication of the declaration of the formal.

Similarly, if an actual signal is associated with a port of mode **out**, **inout**, or **buffer**, and if the type of the actual is a scalar type, then it is an error if (after applying any conversion function or type conversion expression present in the formal part) the subtype of the formal is not compatible with the subtype of the actual.

If an actual signal or expression is associated with a formal port, and if the formal is of a composite subtype, then it is an error if the actual does not contain a matching element for each element of the formal. This check is made after applying the rules of 5.3.2.2 and, in the case of an actual signal, after applying any conversion function or type conversion that is present in the actual part. It is also an error if the mode of the formal is **in** or **inout** and the value of each element of the actual (after applying any conversion function or type conversion present in the actual part) does not belong to the corresponding element subtype of the formal. If the formal port is of mode **out**, **inout**, or **buffer**, it is also an error if the value of each element of the formal (after applying any conversion function or type conversion present in the formal part) does not belong to the corresponding element subtype of the actual.

14.4 Elaboration of a declarative part

14.4.1 General

The elaboration of a declarative part consists of the elaboration of the declarative items, if any, in the order in which they are given in the declarative part. This rule holds for all declarative parts, with the following three exceptions:

a) The entity declarative part of a design entity whose corresponding architecture is decorated with the 'FOREIGN attribute defined in package STANDARD (see 7.2 and 16.3) and for which the value of the attribute is not of the form described in 20.2.4.

b) The architecture declarative part of a design entity whose architecture is decorated with the 'FOREIGN attribute defined in package STANDARD and for which the value of the attribute is not of the form described in 20.2.4.

c) A subprogram declarative part whose subprogram is decorated with the 'FOREIGN attribute defined in package STANDARD.

For these cases, the declarative items are not elaborated; instead, the design entity or subprogram is subject to implementation-dependent elaboration.

In certain cases, the elaboration of a declarative item involves the evaluation of expressions that appear within the declarative item. The value of any object denoted by a primary in such an expression shall be defined at the time the primary is read (see 6.5.2). In addition, if a primary in such an expression is a function call, then the value of any object denoted by or appearing as a part of an actual designator in the function call shall be defined at the time the expression is evaluated. Additionally, it is an error if a primary that denotes a shared variable, or a method of the protected type of a shared variable, is evaluated during the elaboration of a declarative item. During static elaboration, the function STD.STANDARD.NOW (see 16.3) returns the value 0 ns.

NOTE 1—It is a consequence of this rule that the name of a signal declared within a block cannot be referenced in expressions appearing in declarative items within that block, an inner block, or process statement; nor can it be passed as a parameter to a function called during the elaboration of the block. These restrictions exist because the value of a signal is not defined until after the design hierarchy is elaborated. However, a signal parameter name may be used within expressions in declarative items within a subprogram declarative part, provided that the subprogram is only called after simulation begins, because the value of every signal will be defined by that time.

NOTE 2—A function called in an expression evaluated during elaboration of a declarative item may be a foreign function.

14.4.2 Elaboration of a declaration

14.4.2.1 General

Elaboration of a declaration has the effect of creating the declared item.

For each declaration, the language rules (in particular scope and visibility rules) are such that it is either impossible or illegal to use a given item before the elaboration of its corresponding declaration. For example, it is not possible to use the name of a type for an object declaration before the corresponding type declaration is elaborated. Similarly, it is illegal to call a subprogram before its corresponding body is elaborated.

Rules for creation of PSL declarations are defined in IEEE Std 1850-2005.

14.4.2.2 Subprogram declarations, bodies, and instantiations

Elaboration of a subprogram declaration, other than a subprogram declaration that defines an uninstantiated subprogram, involves the elaboration of the subprogram header, if present, followed by the elaboration of the parameter interface list of the subprogram declaration; the latter in turn involves the elaboration of the subtype indication of each interface element to determine the subtype of each formal parameter of the subprogram. Elaboration of an uninstantiated subprogram declaration simply establishes that the name of the subprogram may be referenced subsequently in subprogram instantiation declarations.

Elaboration of a subprogram body, other than the subprogram body of an uninstantiated subprogram, has no effect other than to establish that the body can, from then on, be used for the execution of calls of the subprogram. Elaboration of a subprogram body of an uninstantiated subprogram has no effect.

Elaboration of a subprogram instantiation declaration consists of elaboration of the equivalent generic-mapped subprogram declaration, followed by elaboration of the corresponding equivalent generic-mapped subprogram body (see 4.4). If the subprogram instantiation declaration occurs immediately within an enclosing package declaration, elaboration of the equivalent generic-mapped subprogram body occurs as

part of elaboration of the body, whether explicit or implicit, of the enclosing package. Similarly, if the subprogram instantiation declaration occurs immediately within an enclosing protected type declaration, elaboration of the equivalent generic-mapped subprogram body occurs as part of elaboration of the protected type body.

14.4.2.3 Type declarations

Elaboration of a type declaration generally consists of the elaboration of the definition of the type and the creation of that type. For a constrained type declaration that declares a partially or fully constrained composite subtype, however, elaboration consists of the elaboration of the equivalent anonymous unconstrained type followed by the elaboration of the named subtype of that unconstrained type.

Elaboration of an enumeration type definition has no effect other than the creation of the corresponding type.

Elaboration of an integer, floating-point, or physical type definition consists of the elaboration of the corresponding range constraint. For a physical type definition, each unit declaration in the definition is also elaborated. Elaboration of a physical unit declaration has no effect other than to create the unit defined by the unit declaration.

Elaboration of an unbounded array type definition that defines an unconstrained array type consists of the elaboration of the element subtype indication of the array type.

Elaboration of a record type definition consists of the elaboration of the equivalent single element declarations in the given order. Elaboration of an element declaration consists of elaboration of the element subtype indication.

Elaboration of an access type definition consists of the elaboration of the corresponding subtype indication.

Elaboration of a protected type definition consists of the elaboration, in the order given, of each of the declarations occurring immediately within the protected type definition.

Elaboration of a protected type body has no effect other than to establish that the body, from then on, can be used during the elaboration of objects of the given protected type.

14.4.2.4 Subtype declarations

Elaboration of a subtype declaration consists of the elaboration of the subtype indication. The elaboration of a subtype indication creates a subtype. If the subtype does not include a constraint, then the subtype is the same as that denoted by the type mark. The elaboration of a subtype indication that includes a constraint proceeds as follows:

a) The constraint is first elaborated.

b) A check is then made that the constraint is compatible with the type or subtype denoted by the type mark (see 5.2.1, 5.3.2.2, and 5.3.3).

Elaboration of a range constraint consists of the evaluation of the range. The evaluation of a range defines the bounds and direction of the range. Elaboration of an index constraint consists of the elaboration of each of the discrete ranges in the index constraint in some order that is not defined by the language. Elaboration of an array constraint consists of the elaboration of the index constraint, if present, and the elaboration of the array element constraint, if present. The order of elaboration of the index constraint and the array element constraint, if both are present, is not defined by the language. Elaboration of a record constraint consists of the elaboration of each of the record element constraints in the record constraint in some order that is not defined by the language.

14.4.2.5 Object declarations

The rules of this subclause apply only to explicitly declared objects (see 6.4.2.1). Generic declarations, port declarations, and other interface declarations are elaborated as described in 14.3.2 through 14.3.5 and 14.6.

Elaboration of an object declaration that declares an object other than a file object or an object of a protected type proceeds as follows:

a) The subtype indication is first elaborated; this establishes the subtype of the object.

b) If the object declaration includes an explicit initialization expression, then the initial value of the object is obtained by evaluating the expression. It is an error if the value of the expression does not belong to the subtype of the object; if the object is a composite object, then an implicit subtype conversion is first performed on the value unless the object is a constant whose subtype indication denotes an unconstrained type. Otherwise, any implicit initial value for the object is determined.

c) The object is created.

d) Any initial value is assigned to the object.

The initialization of such an object (either the declared object or one of its subelements) involves a check that the initial value belongs to the subtype of the object. For a composite object declared by an object declaration, an implicit subtype conversion is first applied as for an assignment statement, unless the object is a constant whose subtype is an unconstrained type.

The elaboration of a file object declaration consists of the elaboration of the subtype indication followed by the creation of the object. If the file object declaration contains file open information, then the implicit call to FILE_OPEN is then executed (see 6.4.2.5).

The elaboration of an object of a protected type consists of the elaboration of the subtype indication, followed by creation of the object. Creation of the object consists of elaborating, in the order given, each of the declarative items in the protected type body.

NOTE 1—The expression initializing a constant object need not be a static expression.

NOTE 2—Each object whose type is a protected type involves creation of separate instances of the objects declared by object declarations within the protected type body.

14.4.2.6 Alias declarations

Elaboration of an alias declaration consists of the elaboration of the subtype indication to establish the subtype associated with the alias, followed by the creation of the alias as an alternative name for the named entity. The creation of an alias for a composite object involves a check that the subtype associated with the alias includes a matching element for each element of the named object. It is an error if this check fails.

14.4.2.7 Attribute declarations

Elaboration of an attribute declaration has no effect other than to create a template for defining attributes of items.

14.4.2.8 Component declarations

Elaboration of a component declaration has no effect other than to create a template for instantiating component instances.

14.4.2.9 Packages

Elaboration of a package declaration, other than a package declaration that defines an uninstantiated package, consists of the elaboration of the package header, if present, followed by the elaboration of the declarative part of the package declaration. Elaboration of a package body, other than a package body of an uninstantiated package, consists of the elaboration of the declarative part of the package body. Elaboration of an uninstantiated package declaration simply establishes that the name of the package may be referenced subsequently in package instantiation declarations. Elaboration of a package body of an uninstantiated package has no effect.

Elaboration of a package instantiation declaration consists of elaboration of the equivalent generic-mapped package declaration, followed by elaboration of the corresponding equivalent generic-mapped package body, if such a package body is defined (see 4.9). If the package instantiation declaration occurs immediately within an enclosing package declaration and the uninstantiated package has a package body, elaboration of the equivalent generic-mapped package body occurs as part of elaboration of the body, whether explicit or implicit, of the enclosing package.

14.4.3 Elaboration of a specification

14.4.3.1 General

Elaboration of a specification has the effect of associating additional information with a previously declared item.

14.4.3.2 Attribute specifications

Elaboration of an attribute specification proceeds as follows:

a) The entity specification is elaborated in order to determine which items are affected by the attribute specification.
b) The expression is evaluated to determine the value of the attribute. It is an error if the value of the expression does not belong to the subtype of the attribute; if the attribute is of a composite type, then an implicit subtype conversion is first performed on the value, unless the subtype indication of the attribute denotes an unconstrained type.
c) A new instance of the designated attribute is created and associated with each of the affected items.
d) Each new attribute instance is assigned the value of the expression.

The assignment of a value to an instance of a given attribute involves a check that the value belongs to the subtype of the designated attribute. For an attribute of a partially or fully constrained composite type, an implicit subtype conversion is first applied as for an assignment statement. No such conversion is necessary for an attribute of an unconstrained type; the constraints on the value determine the constraints on the attribute.

NOTE—The expression in an attribute specification need not be a static expression.

14.4.3.3 Configuration specifications

Elaboration of a configuration specification proceeds as follows:

a) The component specification is elaborated in order to determine which component instances are affected by the configuration specification.
b) The binding indication is elaborated to identify the design entity to which the affected component instances will be bound.
c) The binding information is associated with each affected component instance label for later use in instantiating those component instances.

As part of this elaboration process, a check is made that both the entity declaration and the corresponding architecture body implied by the binding indication exist within the specified library. It is an error if this check fails.

14.4.3.4 Disconnection specifications

Elaboration of a disconnection specification proceeds as follows:

a) The guarded signal specification is elaborated in order to identify the signals affected by the disconnection specification.

b) The time expression is evaluated to determine the disconnection time for drivers of the affected signals.

c) The disconnection time is associated with each affected signal for later use in constructing disconnection statements in the equivalent processes for guarded assignments to the affected signals.

14.5 Elaboration of a statement part

14.5.1 General

Concurrent statements appearing in the statement part of a block shall be elaborated before execution begins. Elaboration of the statement part of a block consists of the elaboration of each concurrent statement in the order given. This rule holds for all block statement parts except for those blocks equivalent to a design entity whose corresponding architecture is decorated with the 'FOREIGN attribute defined in package STANDARD (see 16.3).

For this case, there are two subcases:

— If the value of the attribute is of the form described in 20.2.4, the statements are not elaborated; instead, the elaboration function of the foreign model is invoked, as described in 20.4.1, at the point where elaboration of the statements of the block statement corresponding to the architecture body would otherwise occur.

— Otherwise, the statements are not elaborated; instead, the design entity is subject to implementation-dependent elaboration.

Rules for interpretation of PSL directives are defined in IEEE Std 1850-2005.

14.5.2 Block statements

Elaboration of a block statement consists of the elaboration of the block header, if present, followed by the elaboration of the block declarative part, followed by the elaboration of the block statement part.

Elaboration of a block statement may occur under the control of a configuration declaration. In particular, a block configuration, whether implicit or explicit, within a configuration declaration may supply a sequence of additional implicit configuration specifications to be applied during the elaboration of the corresponding block statement. If a block statement is being elaborated under the control of a configuration declaration, then the sequence of implicit configuration specifications supplied by the block configuration is elaborated as part of the block declarative part, following all other declarative items in that part.

The sequence of implicit configuration specifications supplied by a block configuration, whether implicit or explicit, consists of each of the configuration specifications implied by component configurations (see 3.4.3) occurring immediately within the block configuration, in the order in which the component configurations themselves appear.

14.5.3 Generate statements

Elaboration of a generate statement consists of the replacement of the generate statement with zero or more copies of a block statement whose declarative part consists of declarative items contained within the generate statement and whose statement part consists of concurrent statements contained within the generate statement. These block statements are said to be *represented* by the generate statement. Each block statement is then elaborated.

For a for generate statement, elaboration consists of the elaboration of the discrete range, followed by the generation of one block statement for each value in the range. The block statements all have the following form:

a) The label of the block statement is the same as the label of the for generate statement.

b) The block declarative part has, as its first item, a single constant declaration that declares a constant with the same simple name as that of the applicable generate parameter; the value of the constant is the value of the generate parameter for the generation of this particular block statement. The type of this declaration is determined by the base type of the discrete range of the generate parameter. The remainder of the block declarative part consists of a copy of the declarative items contained within the generate statement.

c) The block statement part consists of a copy of the concurrent statements contained within the generate statement.

For an if generate statement, elaboration consists of the evaluation, in succession, of the condition specified after **if** and any conditions specified after **elsif** (treating a final **else** as **elsif** TRUE **generate**) until one evaluates to TRUE or all conditions are evaluated and yield FALSE. If one condition evaluates to TRUE, then exactly one block statement is generated; otherwise, no block statement is generated. If generated, the block statement has the following form:

— The block label is the same as the label of the if generate statement.

— The block declarative part consists of a copy of the declarative items contained within the generate statement body following the condition that evaluated to TRUE. If the condition is preceded by an alternative label, the label is implicitly declared at the beginning of the block declarative part.

— The block statement part consists of a copy of the concurrent statements contained within the generate statement body following the condition that evaluated to TRUE.

For a case generate statement, elaboration consists of the evaluation of the expression followed by the generation of a block statement for the chosen alternative. A given case generate alternative is the chosen alternative if and only if the expression “E = V” evaluates to TRUE, where “E” is the expression, “V” is the value of one of the choices of the given case generate alternative (if a choice is a discrete range, then this latter condition is fulfilled when V is an element of the discrete range), and the operator “=” in the expression is the predefined “=” operator on the base type of E. The generate block statement has the following form:

— The block label is the same as the label of the case generate statement.

— The block declarative part consists of a copy of the declarative items contained within the generate statement body of the chosen alternative. If the choices of the chosen alternative are preceded by an alternative label, the label is implicitly declared at the beginning of the block declarative part.

— The block statement part consists of a copy of the concurrent statements contained within the generate statement body of the chosen alternative.

Examples:

```
--  The following generate statement:
```

```
LABL: for I in 1 to 2 generate
   signal s1: INTEGER;
begin
   s1 <= p1;
   Inst1: and_gate port map (s1, p2(I), p3);
end generate LABL;

--  is equivalent to the following two block statements:

LABL: block
   constant I: INTEGER := 1;
   signal s1: INTEGER;
begin
   s1 <= p1;
   Inst1: and_gate port map (s1, p2(I), p3);
end block LABL;

LABL: block
   constant I: INTEGER := 2;
   signal s1: INTEGER;
begin
   s1 <= p1;
   Inst1: and_gate port map (s1, p2(I), p3);
end block LABL;

--  The following generate statement:

LABL: if (g1 = g2) generate
   signal s1: INTEGER;
begin
   s1 <= p1;
   Inst1: and_gate port map (s1, p4, p3);
end generate LABL;

--  is equivalent to the following statement if g1 = g2;
--  otherwise, it is equivalent to no statement at all:

LABL: block
   signal s1: INTEGER;
begin
   s1 <= p1;
   Inst1: and_gate port map (s1, p4, p3);
end block LABL;
```

NOTE—The repetition of the block labels in the case of a for generate statement does not produce multiple declarations of the label on the generate statement. The multiple block statements represented by the generate statement constitute multiple references to the same implicitly declared label.

14.5.4 Component instantiation statements

Elaboration of a component instantiation statement that instantiates a component declaration has no effect unless the component instance is either fully bound to a design entity defined by an entity declaration and architecture body or bound to a configuration of such a design entity. If a component instance is so bound, then elaboration of the corresponding component instantiation statement consists of the elaboration of the implied block statement representing the component instance and (within that block) the implied block

statement representing the design entity to which the component instance is bound. The implied block statements are defined in 11.7.2.

Elaboration of a component instantiation statement whose instantiated unit denotes either a design entity or a configuration declaration consists of the elaboration of the implied block statement representing the component instantiation statement and (within that block) the implied block statement representing the design entity to which the component instance is bound. The implied block statements are defined in 11.7.3.

14.5.5 Other concurrent statements

All other concurrent statements are either process statements or are statements for which there is an equivalent process statement.

Elaboration of a process statement proceeds as follows:

a) The process declarative part is elaborated.
b) The drivers required by the process statement are identified.

Elaboration of all concurrent signal assignment statements and concurrent assertion statements consists of the construction of the equivalent process statement followed by the elaboration of the equivalent process statement.

14.6 Dynamic elaboration

The execution of certain constructs that involve sequential statements rather than concurrent statements also involves elaboration. Such elaboration occurs during the execution of the model.

There are three particular instances in which elaboration occurs dynamically during simulation. These are as follows:

a) Execution of a loop statement with a for iteration scheme involves the elaboration of the loop parameter specification prior to the execution of the statements enclosed by the loop (see 10.10). This elaboration creates the loop parameter and evaluates the discrete range.
b) Execution of a subprogram call involves the elaboration of the parameter association list. This involves the elaboration of the parameter association element or elements in the association list associated with each interface declaration. Elaboration of the parameter association elements associated with a formal parameter declaration proceeds as follows:
    1) The subtype indication of the corresponding formal parameter declaration is elaborated.
    2) The formal part or parts of the parameter association elements corresponding to the formal parameter declaration are elaborated.
    3) If the type of the formal parameter is an array type or contains a subelement that is of an array type, the rules of 5.3.2.2 are applied to determine the index ranges.
    4) For each parameter association element associated with the formal parameter declaration, the parameter or subelement or slice thereof designated by the formal part is then associated with the actual part.
    5) If the formal parameter is a variable of mode **out**, then the implicit initial value for the object is determined.

    Next, if the subprogram is a method of a protected type (see 5.6.2) or an implicitly declared file operation (see 5.5.2), the elaboration *blocks* (suspends execution while retaining all state), if necessary, until exclusive access to the object denoted by the prefix of the method or to the file object denoted by the file parameter of the file operation is secured. Finally, if the designator of the subprogram is not decorated with the 'FOREIGN attribute defined in package STANDARD, the declarative part of the corresponding subprogram body is elaborated and the sequence of statements

in the subprogram body is executed. If the designator of the subprogram is decorated with the 'FOREIGN attribute defined in package STANDARD, there are two cases:

— If the value of the attribute is of the form described in 20.2.4, the declarative part of the corresponding subprogram body is not elaborated nor is the sequence of statements in the subprogram body executed; instead, the execution function of the foreign model is invoked, as described in 20.2.4.

— Otherwise, the subprogram body is subject to implementation-dependent elaboration and execution.

c) Evaluation of an allocator that contains a subtype indication involves the elaboration of the subtype indication prior to the allocation of the created object.

NOTE 1—It is a consequence of these rules that declarative items appearing within the declarative part of a subprogram body are elaborated each time the corresponding subprogram is called; thus, successive elaborations of a given declarative item appearing in such a place may create items with different characteristics. For example, successive elaborations of the same subtype declaration appearing in a subprogram body may create subtypes with different constraints.

NOTE 2—If two or more processes access the same set of shared variables, livelock or deadlock may occur. That is, it may not be possible to ever grant exclusive access to the shared variable as outlined in the preceding item b). Implementations are allowed to, but not required to, detect and, if possible, resolve such conditions.

14.7 Execution of a model

14.7.1 General

The elaboration of a design hierarchy produces a *model* that can be executed in order to simulate the design represented by the model. Simulation involves the execution of user-defined processes that interact with each other and with the environment. Simulation also involves interpretation of PSL directives to verify the properties that they specify.

The *kernel process* is a conceptual representation of the agent that coordinates the activity of user-defined processes during a simulation. This agent causes the propagation of signal values to occur and causes the values of implicit signals (such as S'STABLE) to be updated. Furthermore, this process is responsible for detecting events that occur and for causing the appropriate processes to execute in response to those events.

For any given signal that is explicitly declared within a model, the kernel process contains variables representing the driving value and current value of that signal. Any evaluation of a name denoting a given signal retrieves the current value of the corresponding variable in the kernel process. During simulation, the kernel process updates these variables from time to time, based upon the current values of sources of the corresponding signal.

In addition, the kernel process contains a variable representing the current value of any implicitly declared GUARD signal resulting from the appearance of a guard condition on a given block statement. Furthermore, the kernel process contains both a driver for, and a variable representing the current value of, any signal S'STABLE(T), for any prefix S and any time T, that is referenced within the model; likewise, for any signal S'QUIET(T) or S'TRANSACTION.

14.7.2 Drivers

Every signal assignment statement in a process statement defines a set of *drivers* for certain scalar signals. There is a single driver for a given scalar signal S in a process statement, provided that there is at least one signal assignment statement in that process statement and that the longest static prefix of the target signal of that signal assignment statement denotes S or denotes a composite signal of which S is a subelement. Each such signal assignment statement is said to be *associated* with that driver. Execution of a signal assignment statement affects only the associated driver(s).

A driver for a scalar signal is represented by a *projected output waveform*. A projected output waveform consists of a sequence of one or more *transactions*, where each transaction is a pair consisting of a value component and a time component. For a given transaction, the value component represents a value that the driver of the signal is to assume at some point in time, and the time component specifies which point in time. These transactions are ordered with respect to their time components.

A driver always contains at least one transaction. The initial contents of a driver associated with a given signal are defined by the default value associated with the signal (see 6.4.2.3). The kernel process contains a variable representing the *current value* of the driver. The initial value of the variable is the value component of the initial transaction of the driver.

For any driver, if, as the result of the advance of time, the current time becomes equal to the time component of the second transaction of the driver, the first transaction is deleted from the projected output waveform, and what was the second transaction becomes the first transaction. Then, or if a force or deposit is scheduled for the driver, the variable containing the current value of the driver is updated as follows:

— If a force is scheduled for the driver, the driver becomes forced and the variable containing the current value of the driver is updated with the force value for the driver.
— If the driver is forced and no force is scheduled for the driver, the variable containing the current value of the driver is unchanged from its previous value.
— If a deposit is scheduled for the driver and the driver is not forced, the variable containing the current value of the driver is updated with the deposit value for the driver.
— Otherwise, the variable containing the current value of the driver is updated with the value component of the first transaction of the driver.

When this action occurs on a driver, any registered and enabled `vhpiCbTransaction` callbacks associated with the given driver are executed. Moreover, if the current value of the driver changes as a result of this action, any registered and enabled `vhpiCbValueChange` callbacks associated with the given driver are executed.

14.7.3 Propagation of signal values

14.7.3.1 General

As simulation time advances, the transactions in the projected output waveform of a given driver (see 14.7.2) will each, in succession, become the value of the driver. When a driver acquires a new value in this way or as a result of a force or deposit scheduled for the driver, regardless of whether the new value is different from the previous value, that driver is said to be *active* during that simulation cycle. For the purposes of defining driver activity, a driver acquiring a value from a null transaction is assumed to have acquired a new value. A signal is said to be *active* during a given simulation cycle if

— One of its sources is active.
— One of its subelements is active.
— The signal is named in the formal part of an association element in a port association list and the corresponding actual is active.
— The signal is a subelement of a resolved signal and the resolved signal is active.
— A force, a deposit, or a release is scheduled for the signal.
— The signal is a subelement of another signal for which a force or a deposit is scheduled.

If a signal of a given composite type has a source that is of a different type (and therefore a conversion function or type conversion appears in the corresponding association element), then each scalar subelement of that signal is considered to be active if the source itself is active. Similarly, if a port of a given composite type is associated with a signal that is of a different type (and therefore a conversion function or type

conversion appears in the corresponding association element), then each scalar subelement of that port is considered to be active if the actual signal itself is active.

In addition to the preceding information, an implicit signal is said to be active during a given simulation cycle if the kernel process updates that implicit signal within the given cycle.

If a signal is not active during a given simulation cycle, then the signal is said to be *quiet* during that simulation cycle.

The kernel process determines two values for certain signals during certain simulation cycles. The *driving value* of a given signal is the value that signal provides as a source of other signals. The *effective value* of a given signal is the value obtainable by evaluating a reference to the signal within an expression. The driving value and the effective value of a signal are not always the same, especially when resolution functions and conversion functions or type conversions are involved in the propagation of signal values.

NOTE 1—In a given simulation cycle, situations can occur where a subelement of a composite signal is quiet, and the signal itself is active.

NOTE 2—The rules concerning association of actuals with formals (see 6.5.7.1) imply that, if a composite signal is associated with a composite port of mode **out**, **inout**, or **buffer**, and if no conversion function or type conversion appears in either the actual or formal part of the association element, then each scalar subelement of the formal is a source of the matching subelement of the actual. In such a case, a given subelement of the actual will be active if and only if the matching subelement of the formal is active.

NOTE 3—A signal of kind **register** may be active even if its associated resolution function does not execute in the current simulation cycle if the values of all of its drivers are determined by the null transaction and at least one of its drivers is also active.

14.7.3.2 Driving values

A *basic signal* is a signal that has all of the following properties:

- It is either a scalar signal or a resolved signal (see 6.4.2.3).
- It is not a subelement of a resolved signal.
- Is not an implicit signal of the form S'STABLE(T), S'QUIET(T), or S'TRANSACTION (see 16.2).
- It is not an implicit signal GUARD (see 11.2).

Basic signals are those that determine the driving values for all other signals.

The driving value of any signal S is determined by the following steps:

a) If a driving-value release is scheduled for S or for a signal of which S is a subelement, S becomes driving-value released, that is, no longer driving-value forced. Proceed to step b).

b) If a driving-value force is scheduled for S or for a signal of which S is a subelement, S becomes driving-value forced and the driving value of S is the driving force value for S or the element of the driving force value for the signal of which S is a subelement, as appropriate; no further steps are required. Otherwise, proceed to step c).

c) If S is driving-value forced, the driving value of S is unchanged from its previous value; no further steps are required. Otherwise, proceed to step d).

d) If a driving-value deposit is scheduled for S or for a signal of which S is a subelement, the driving value of S is the driving deposit value for S or the element of the driving deposit value for the signal of which S is a subelement, as appropriate; no further steps are required. Otherwise, proceed to step e) or f), as appropriate.

e) If S is a basic signal:

   — If S has no source, then the driving value of S is given by the default value associated with S (see 6.4.2.3).

— If S has one source that is a driver and S is not a resolved signal (see 6.4.2.3), then the driving value of S is the current value of that driver.

— If S has one source that is a port and S is not a resolved signal, then the driving value of S is the driving value of the formal part of the association element that associates S with that port (see 6.5.7.1). The driving value of a formal part is obtained by evaluating the formal part as follows: If no conversion function or type conversion is present in the formal part, then the driving value of the formal part is the driving value of the signal denoted by the formal designator. Otherwise, the driving value of the formal part is the value obtained by applying either the conversion function or type conversion (whichever is contained in the formal part) to the driving value of the signal denoted by the formal designator.

— If S is a resolved signal and has one or more sources, then the driving values of the sources of S are examined. It is an error if any of these driving values is a composite where one or more subelement values are determined by the null transaction (see 10.5.2.2) and one or more subelement values are not determined by the null transaction. If S is of signal kind **register** and all the sources of S have values determined by the null transaction, then the driving value of S is unchanged from its previous value. Otherwise, the driving value of S is obtained by executing the resolution function associated with S, where that function is called with an input parameter consisting of the concatenation of the driving values of the sources of S, with the exception of the value of any source of S whose current value is determined by the null transaction.

f) If S is not a basic signal:

— If S is a subelement of a resolved signal R, the driving value of S is the corresponding subelement value of the driving value of R.

— Otherwise (S is a nonresolved, composite signal), the driving value of S is equal to the aggregate of the driving values of each of the basic signals that are the subelements of S.

NOTE 1—The algorithm for computing the driving value of a scalar signal S is recursive. For example, if S is a local signal appearing as an actual in a port association list whose formal is of mode **out** or **inout**, the driving value of S can only be obtained after the driving value of the corresponding formal part is computed. This computation may involve multiple executions of the preceding algorithm.

NOTE 2—The definition of the driving value of a basic signal exhausts all cases, with the exception of a non-resolved signal with more than one source. This condition is defined as an error in 6.4.2.3.

NOTE 3—The driving value of a port that has no source is the default value of the port (see 6.5.2).

14.7.3.3 Effective values

For a scalar signal S, the *effective value* of S is determined by the following steps:

a) If an effective-value release is scheduled for S or for a signal of which S is a subelement, S becomes effective-value released, that is, no longer effective-value forced. Proceed to step b).

b) If an effective-value force is scheduled for S or for a signal of which S is a subelement, S becomes effective-value forced and the effective value of S is the effective force value for S or the element of the effective force value for the signal of which S is a subelement, as appropriate; no further steps are required. Otherwise, proceed to step c).

c) If S is effective-value forced, the effective value of S is unchanged from its previous value; no further steps are required. Otherwise, proceed to step d).

d) If an effective-value deposit is scheduled for S or for a signal of which S is a subelement, the effective value of S is the effective deposit value for S or the element of the effective deposit value for the signal of which S is a subelement, as appropriate; no further steps are required. Otherwise, proceed to step e).

e) The effective value of S is then determined as follows:

— If S is a signal declared by a signal declaration, a port of mode **out** or **buffer**, or an unconnected port of mode **inout**, then the effective value of S is the same as the driving value of S.

— If S is a connected port of mode **in** or **inout**, then the effective value of S is the same as the effective value of the actual part of the association element that associates an actual with S (see 6.5.7.1). The effective value of an actual part is obtained by evaluating the actual part, using the effective value of the signal denoted by the actual designator in place of the actual designator.

— If S is an unconnected port of mode **in**, the effective value of S is given by the default value associated with S (see 6.4.2.3).

For a composite signal R, the effective value of R is the aggregate of the effective values of each of the subelements of R.

NOTE 1—The algorithm for computing the effective value of a signal S is recursive. For example, if a formal port S of mode **in** corresponds to an actual A, the effective value of A shall be computed before the effective value of S can be computed. The actual A may itself appear as a formal port in a port association list.

NOTE 2—No effective value is specified for **linkage** ports, since these ports cannot be read.

14.7.3.4 Signal update

For a scalar signal S, both the driving and effective values shall belong to the subtype of the signal. For a composite signal R, an implicit subtype conversion is performed to the subtype of R; for each element of R, there shall be a matching element in both the driving and the effective value, and vice versa.

In order to update a signal during a given simulation cycle, the kernel process first determines the driving and effective values of that signal. The kernel process then updates the variable containing the driving value with the newly determined driving value. The kernel also updates the variable containing the current value of the signal with the newly determined effective value, as follows:

a) If S is a scalar signal, the effective value of S is used to update the current value of S. A check is made that the effective value of S belongs to the subtype of S. An error occurs if this subtype check fails. Finally, the effective value of S is assigned to the variable representing the current value of the signal.

b) If S is a composite signal (including a slice of an array), the effective value of S is implicitly converted to the subtype of S. The subtype conversion checks that for each element of S there is a matching element in the effective value and vice versa. An error occurs if this check fails. The result of this subtype conversion is then assigned to the variable representing the current value of S.

The current value of a signal of type T is said to *change* if and only if application of the predefined "=" operator for type T to the current value of the signal and the value of the signal prior to the update evaluates to FALSE. If updating a signal causes the current value of that signal to change, then an *event* is said to have occurred on the signal, unless the update occurs by application of the `vhpi_put_value` function with an update mode of `vhpiDeposit` or `vhpiForce` to an object that represents the signal. This definition applies to any updating of a signal, whether such updating occurs according to the preceding rules or according to the rules for updating implicit signals given in 14.7.4. The occurrence of an event will cause the resumption and subsequent execution of certain processes during the simulation cycle in which the event occurs, if and only if those processes are currently sensitive to the signal on which the event has occurred.

Each time a signal S is updated, any registered and enabled `vhpiCbTransaction` callbacks associated with S are executed. Each time there is an event on a signal S, any registered and enabled `vhpiCbValueChange` callbacks associated with S are executed.

A *net* is a collection of drivers, signals (including ports and implicit signals), conversion functions, and resolution functions that, taken together, determine the effective and driving values of every signal on the net.

For any signal that is part of a given net, the driving and effective values of the signal are determined and the variables containing the driving value and current value of that signal are updated as previously described in those simulation cycles in which any driver or signal on the net is active.

Implicit signals GUARD, S'STABLE(T), S'QUIET(T), and S'TRANSACTION, for any prefix S and any time T, are not updated according to the preceding rules; such signals are updated according to the rules described in 14.7.4.

NOTE 1—Overloading the operator "=" has no effect on the propagation of signal values.

NOTE 2—If a net includes an implicitly declared GUARD signal, the drivers of signals referred to in the corresponding guard condition determine the value of the GUARD signal. Hence, those drivers are part of the net, and when any of the drivers are active, the signals that are part of the net are updated.

14.7.4 Updating implicit signals

The kernel process updates the value of each implicit signal GUARD associated with a block statement that has a guard condition. Similarly, the kernel process updates the values of each implicit signal S'STABLE(T), S'QUIET(T), or S'TRANSACTION for any prefix S and any time T; this also involves updating the drivers of S'STABLE(T) and S'QUIET(T).

For any implicit signal GUARD, the current value of the signal is modified if and only if the corresponding guard condition contains a reference to a signal S and if S is active during the current simulation cycle. In such a case, the implicit signal GUARD is updated by evaluating the corresponding guard condition and assigning the result of that evaluation to the variable representing the current value of the signal. Whenever an implicit signal GUARD is updated, any registered and enabled `vhpiCbTransaction` callbacks associated with the given signal are executed.

For any implicit signal S'STABLE(T), the current value of the signal (and likewise the current state of the corresponding driver) is modified if and only if one of the following statements is true:

— An event has occurred on S in this simulation cycle.

— The driver of S'STABLE(T) is active.

If an event has occurred on signal S, then S'STABLE(T) is updated by assigning the value FALSE to the variable representing the current value of S'STABLE(T), and the driver of S'STABLE(T) is assigned the waveform TRUE **after** T. Otherwise, if the driver of S'STABLE(T) is active, then S'STABLE(T) is updated by assigning the current value of the driver to the variable representing the current value of S'STABLE(T). Otherwise, neither the variable nor the driver is modified. Whenever a signal of the form S'STABLE(T) is updated, any registered and enabled `vhpiCbTransaction` callbacks associated with the given signal are executed.

Similarly, for any implicit signal S'QUIET(T), the current value of the signal (and likewise the current state of the corresponding driver) is modified if and only if one of the following statements is true:

— S is active.

— The driver of S'QUIET(T) is active.

If signal S is active, then S'QUIET(T) is updated by assigning the value FALSE to the variable representing the current value of S'QUIET(T), and the driver of S'QUIET(T) is assigned the waveform TRUE **after** T. Otherwise, if the driver of S'QUIET(T) is active, then S'QUIET(T) is updated by assigning the current value of the driver to the variable representing the current value of S'QUIET(T). Otherwise, neither the variable nor the driver is modified. Whenever a signal of the form S'QUIET(T) is updated, any registered and enabled `vhpiCbTransaction` callbacks associated with the given signal are executed.

Finally, for any implicit signal S'TRANSACTION, the current value of the signal is modified if and only if S is active. If signal S is active, then S'TRANSACTION is updated by assigning the value of the expression (**not** S'TRANSACTION) to the variable representing the current value of S'TRANSACTION. At most one such assignment will occur during any given simulation cycle. Whenever a signal of the form S'TRANSACTION is updated, any registered and enabled `vhpiCbTransaction` callbacks associated with the given signal are executed.

For any implicit signal S'DELAYED(T), the signal is not updated by the kernel process. Instead, it is updated by constructing an equivalent process (see 16.2) and executing that process.

Each time there is an event on a signal S, where S is any one of

— An implicit signal GUARD

— P'STABLE(T), for any prefix P and any time T

— P'QUIET(T), for any prefix P and any time T

— P'TRANSACTION, for any prefix P

any registered and enabled `vhpiCbValueChange` callbacks associated with S are executed.

The current value of a given implicit signal denoted by R is said to *depend* upon the current value of another signal S if one of the following statements is true:

— R denotes an implicit GUARD signal and S is any other implicit signal named within the guard condition that defines the current value of R.

— R denotes an implicit signal S'STABLE(T).

— R denotes an implicit signal S'QUIET(T).

— R denotes an implicit signal S'TRANSACTION.

— R denotes an implicit signal S'DELAYED(T).

Similarly, the current value of a given interface signal denoted by R is said to *depend* upon the current value of an implicit signal S if R denotes a port of mode **in** and S is the actual associated with that port.

These rules define a partial ordering on all signals within a model. The updating of signals by the kernel process is guaranteed to proceed in such a manner that, if a given implicit signal R depends upon the current value of another signal S, or if a given interface signal R depends upon the value of an implicit signal S, then the current value of S will be updated during a particular simulation cycle prior to the updating of the current value of R.

NOTE—These rules imply that, if the driver of S'STABLE(T) is active, then the new current value of that driver is the value TRUE. Furthermore, these rules imply that, if an event occurs on S during a given simulation cycle, and if the driver of S'STABLE(T) becomes active during the same cycle, the variable representing the current value of S'STABLE(T) will be assigned the value FALSE, and the current value of the driver of S'STABLE(T) during the given cycle will never be assigned to that signal.

14.7.5 Model execution

14.7.5.1 General

The execution of a model consists of an initialization phase followed by the repetitive execution of process statements in the description of that model. Each such repetition is said to be a *simulation cycle*. In each cycle, the values of all signals in the description are computed. If as a result of this computation an event occurs on a given signal, process statements that are sensitive to that signal will resume and will be executed as part of the simulation cycle.

At certain stages during the initialization phase and each simulation cycle, the current time, T_c, and the time of the next simulation cycle, T_n, are calculated. T_n is calculated by setting it to the earliest of

a) TIME'HIGH,

b) The next time at which a driver or signal becomes active,

c) The next time at which a process resumes, or

d) The next time at which a registered and enabled `vhpiCbAfterDelay`, `vhpiCbRepAfterDelay`, `vhpiCbTimeOut`, or `vhpiCbRepTimeOut` callback is to occur.

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

14.7.5.2 Initialization

At the beginning of initialization, the current time, T_c, is assumed to be 0 ns.

The initialization phase consists of the following steps:

a) Each registered and enabled `vhpiCbStartofInitialization` callback is executed.

b) Each registered and enabled `vhpiCbStartOfNextCycle` and `vhpiCbRepStartOfNextCycle` callback is executed.

c) The signals in the model are updated as follows in an order such that if a given signal R depends upon the current value of another signal S, then the current value of S is updated prior to the updating of the current value of R:

   — The driving value and the effective value of each explicitly declared signal are computed, and the variables representing the driving value and current value of the signal are set to the driving value and effective value, respectively. The current value is assumed to have been the value of the signal for an infinite length of time prior to the start of simulation. If a force, deposit, or release was scheduled for any driver or signal, the force, deposit or release is no longer scheduled for the driver or signal.

   — The value of each implicit signal of the form S'STABLE(T) or S'QUIET(T) is set to TRUE. The value of each implicit signal of the form S'DELAYED(T) is set to the initial value of its prefix, S.

   — The value of each implicit GUARD signal is set to the result of evaluating the corresponding guard condition.

d) Any action required to give effect to a PSL directive is performed (see IEEE Std 1850-2005).

e) Each registered and enabled `vhpiCbStartOfProcesses` and `vhpiCbRepStartOfProcesses` callback is executed.

f) For each nonpostponed process P in the model, the following actions occur in the indicated order:

   1) The process executes until it suspends.

   2) Each registered and enabled `vhpiCbSuspend` callback associated with P is executed.

g) For each elaborated instance of a registered foreign architecture, the corresponding execution function is invoked.

h) Each registered and enabled `vhpiCbEndOfProcesses` and `vhpiCbRepEndOfProcesses` callback is executed.

i) Each registered and enabled `vhpiCbStartOfPostponed` and `vhpiCbRepStartOfPostponed` callback is executed.

j) For each postponed process P in the model, the following actions occur in the indicated order:

   1) The process executes until it suspends.

   2) Each registered and enabled `vhpiCbSuspend` callback associated with P is executed.

k) The time of the next simulation cycle (which in this case is the first simulation cycle), T_n, is calculated according to the rules of 14.7.5.1.

l) If the VHDL tool executing the initialization phase has requested a model save that has not yet been satisfied, the model is saved as described in 20.7.

m) Each registered and enabled `vhpiCbEndOfInitialization` callback is executed.

NOTE 1—The initial value of any implicit signal of the form S'TRANSACTION is not defined.

NOTE 2—Updating of explicit signals is described in 14.7.3; updating of implicit signals is described in 14.7.4.

NOTE 3—`vhpiCbResume` callbacks are not executed during initialization as processes do not resume during initialization.

14.7.5.3 Simulation cycle

A simulation cycle consists of the following steps:

a) The current time, T_c, is set equal to T_n. Simulation is complete when T_n = TIME'HIGH and there are no active drivers, process resumptions, or registered and enabled `vhpiCbAfterDelay`, `vhpiCbRepAfterDelay`, `vhpiCbTimeOut`, or `vhpiCbRepTimeOut` callbacks to occur at T_n.

b) The following actions occur in the indicated order:

   1) If the current simulation cycle is not a delta cycle, each registered and enabled `vhpiCbNextTimeStep` and `vhpiCbRepNextTimeStep` callback is executed.

   2) Each registered and enabled `vhpiCbStartOfNextCycle` and `vhpiCbRepStartOfNextCycle` callback is executed.

   3) Each registered and enabled `vhpiCbAfterDelay` and `vhpiCbRepAfterDelay` callback is executed.

c) Each active driver in the model is updated. If a force or deposit was scheduled for any driver, the force or deposit is no longer scheduled for the driver.

d) Each signal on each net in the model that includes active drivers is updated in an order that is consistent with the dependency relation between signals (see 14.7.4). (Events may occur on signals as a result.) If a force, deposit, or release was scheduled for any signal, the force, deposit, or release is no longer scheduled for the signal.

e) Any action required to give effect to a PSL directive is performed (see IEEE Std 1850-2005).

f) The following actions occur in the indicated order:

   1) Each registered and enabled `vhpiCbStartOfProcesses` and `vhpiCbRepStartOfProcesses` callback is executed. If an event has occurred on a signal S in this simulation cycle, then each registered and enabled `vhpiCbSensitivity` callback associated with S is executed.

   2) For each process, P, if P is currently sensitive to a signal, S, and if an event has occurred on S in this simulation cycle, then P resumes.

   3) Each registered and enabled `vhpiCbTimeOut` and `vhpiCbRepTimeOut` callback whose triggering condition is met is executed. For each nonpostponed process P that has resumed in the current simulation cycle, the following actions occur in the indicated order:

      — Each registered and enabled `vhpiCbResume` callback associated with P is executed.

      — The process executes until it suspends.

      — Each registered and enabled `vhpiCbSuspend` callback associated with P is executed.

   4) Each registered and enabled `vhpiCbEndOfProcesses` and `vhpiCbRepEndOfProcesses` callback is executed.

g) The time of the next simulation cycle, T_n, is calculated according to the rules of 14.7.5.1.

h) If the next simulation cycle will be a delta cycle, the remainder of step h) is skipped. Otherwise, the following actions occur in the indicated order:

1) Each registered and enabled `vhpiCbLastKnownDeltaCycle` and `vhpiCbRepLastKnownDeltaCycle` callback is executed. T_n is recalculated according to the rules of 14.7.5.1.
2) If the next simulation cycle will be a delta cycle, the remainder of step h) is skipped.
3) Each registered and enabled `vhpiCbStartOfPostponed` and `vhpiCbRepStartOfPostponed` callback is executed.
4) For each postponed process P, if P has resumed but has not been executed since its last resumption, the following actions occur in the indicated order:
   — Each registered and enabled `vhpiCbResume` callback associated with P is executed.
   — The process executes until it suspends.
   — Each registered and enabled `vhpiCbSuspend` callback associated with P is executed.
5) T_n is recalculated according to the rules of 14.7.5.1.
6) Each registered and enabled `vhpiCbEndOfTimeStep` and `vhpiCbRepEndOfTimeStep` callback is executed.
7) If T_n = TIME'HIGH and there are no active drivers, process resumptions, or registered and enabled `vhpiCbAfterDelay`, `vhpiCbRepAfterDelay`, `vhpiCbTimeOut`, or `vhpiCbRepTimeOut` callbacks to occur at T_n, then each registered and enabled `vhpiCbQuiescence` callback is executed. T_n is recalculated according to the rules of 14.7.5.1.

It is an error if the execution of any postponed process or any callback executed in substeps 3) through 7) of step h) causes a delta cycle to occur immediately after the current simulation cycle.

i) If the VHDL tool executing the simulation cycle has requested a model save that has not yet been satisfied, the model is saved as described in 20.7.

Immediately prior to the execution of the first simulation cycle, each registered and enabled `vhpiCbStartOfSimulation` callback is executed. Immediately subsequent to the execution of the final simulation cycle (i.e., when simulation is complete), each registered and enabled `vhpiCbEndOfSimulation` callback is executed.

NOTE 1—Updating of explicit signals is described in 14.7.3; updating of implicit signals is described in 14.7.4.

NOTE 2—When a process resumes, it is added to one of two sets of processes to be executed (the set of postponed processes and the set of nonpostponed processes). However, no process actually begins to execute until all signals have been updated and all executable processes for this simulation cycle have been identified. Nonpostponed processes are always executed during step f) of every simulation cycle, while postponed processes are executed during step h) of every simulation cycle that does not immediately precede a delta cycle.

NOTE 3—The `vhpiCbEndOfTimeStep` and `vhpiCbRepEndOfTimeStep` callbacks cannot cause activity or register callbacks that would result in a change to the time of the next simulation cycle, T_n (see 21.3.6.8).

