

## Clause 12

### Elaboration and execution

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 must first be elaborated. Initialization of nets (see 12.6.2) 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.

#### 12.1 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.

A design hierarchy is defined either by a design entity or by a configuration.<sup>1</sup>

~~A design hierarchy may be defined by a design entity.~~<sup>2</sup> Elaboration of a design hierarchy defined in this manner ~~by a design entity~~<sup>3</sup> 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 5.2) 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 5.2) and supplying an implicit block configuration (see 1.3.1) that binds and configures a design entity identified according to the rules of 5.2.2. The equivalent block statement is defined in 9.6.2. Elaboration of a block statement is defined in 12.4.1.

~~A design hierarchy may also be defined by a configuration.~~<sup>4</sup> 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 1.3.2) for each unbound component instance contained within the external block and an implicit block configuration (see 1.3.1) 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 types and modes in an implementation-defined manner. Similarly, the means by which top-level interface objects

- 
1. IR1000.4.7.
  2. IR1000.4.7.
  3. IR1000.4.7.
  4. IR1000.4.7.

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 containing declarations referenced by the block. Similarly, elaboration of a given package involves first elaborating each not-yet-elaborated package containing declarations referenced by the given package. Elaboration of a package additionally consists of the

- a) Elaboration of the declarative part of the package declaration, eventually followed by
- b) Elaboration of the declarative part of the corresponding package body, if the package has a corresponding package body

Step b above, the elaboration of a package body, may be deferred until the declarative parts of other packages have been elaborated, if necessary, because of the dependencies created between packages by their interpackage references.

Elaboration of a declarative part is defined in 12.3.

*Examples:*

```
-- In the following example, because of the dependencies between the packages, the
-- elaboration of either package body must 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
    for A
        ...
    end for;
end configuration anonymous;

```

-- L is the library in which E(A) is found.  
-- The most recently analyzed architecture  
-- of L.E.

## 12.2 Elaboration of a block header

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, in that order.

### 12.2.1 The 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 consists of elaborating the subtype indication and then creating a generic constant of that subtype.

The value of a generic constant is not defined until a subsequent generic map aspect is evaluated or, in the absence of a generic map aspect, until the default expression associated with the generic constant is evaluated to determine the value of the constant.

### 12.2.2 The generic map aspect

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.

Elaboration of a generic association list consists of the elaboration of each generic association element in the association list. Elaboration of a generic association element consists of the elaboration of the formal part and the evaluation of the actual part. The generic constant or subelement or slice thereof designated by the 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 constrained array 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.

### 12.2.3 The 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 consists of elaborating the subtype indication and then creating a port of that subtype.

### 12.2.4 The 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 each port association element in the association list whose actual is not the reserved word **open**. Elaboration of a port association element consists of the elaboration of the formal part; 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 restrictions on port associations (see 1.1.1.2) 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 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 12.6.3).

If an actual signal is associated with a port of any mode, 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 bounds and direction of the subtype denoted by the subtype indication of the formal are not identical to the bounds and direction of the subtype denoted by the subtype indication of the actual. 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.

If an actual signal or expression is associated with a formal port, and if the formal is of a constrained array subtype, then it is an error if the actual does not contain a matching element for each element of the formal. In the case of an actual signal, this check is made after applying any conversion function or type conversion that is present in the actual part. If an actual signal or expression is associated with a formal port, and if the subtype denoted by the subtype indication of the declaration of the formal is an unconstrained array type, then the subtype of the formal is taken from the actual associated with that formal. It is also an error if the mode of the formal is **in** or **inout** and the value of each element of the actual array (after applying any conversion function or type conversion present in the actual part) does not belong to the 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 element subtype of the actual.

If an actual signal or expression is associated with a formal port, and if the formal is of a record subtype, then it is an error if the rules of the preceding three paragraphs do not apply to each element of the record subtype. In the case of an actual signal, these checks are made after applying any conversion function or type conversion that is present in the actual part.

## 12.3 Elaboration of a declarative part

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 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 5.1 and 14.2)

- b) The architecture declarative part of a design entity whose architecture is decorated with the 'FOREIGN' attribute defined in package STANDARD
- 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 must be defined at the time the primary is read (see 4.3.2). 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 must 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 14.2) returns the value 0 ns.<sup>5</sup>

#### NOTE

—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.

### 12.3.1 Elaboration of a declaration

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.

#### 12.3.1.1 Subprogram declarations and bodies

Elaboration of a subprogram declaration involves the elaboration of the parameter interface list of the subprogram declaration; this 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 a subprogram body has no effect other than to establish that the body can, from then on, be used for the execution of calls of the subprogram.

#### 12.3.1.2 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 array type declaration, however, elaboration consists of the elaboration of the equivalent anonymous unconstrained array 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.

---

5. LCS 9.

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 unconstrained array type definition 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 can, from then on, be used during the elaboration of objects of the given protected type.

### 12.3.1.3 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 3.1 and 3.2.1.1).

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.

### 12.3.1.4 Object declarations

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 an array object, then an implicit subtype conversion is first performed on the value unless the object is a constant whose subtype indication denotes an unconstrained array 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 an array 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 array 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 4.3.1.4).

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.

#### NOTES

- 1—These rules apply to all object declarations other than port and generic declarations, which are elaborated as outlined in 12.2.1 through 12.2.4.
- 2—The expression initializing a constant object need not be a static expression.
- 3—Each object whose type is a protected type creates an instance of the shared objects.

### 12.3.1.5 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 an array 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.

### 12.3.1.6 Attribute declarations

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

### 12.3.1.7 Component declarations

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

## 12.3.2 Elaboration of a specification

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

### 12.3.2.1 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 an array type, then an implicit subtype conversion is first performed on the value, unless the subtype indication of the attribute denotes an unconstrained array 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 constrained array type, an implicit subtype conversion is first applied as for an assignment statement. No such conversion is necessary for an attribute of an unconstrained array 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.

### 12.3.2.2 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.

### 12.3.2.3 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.

## 12.4 Elaboration of a statement part

Concurrent statements appearing in the statement part of a block must 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 14.2).

For this case, the statements are not elaborated; instead, the design entity is subject to implementation-dependent elaboration.

### 12.4.1 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 1.3.2) occur-



ring immediately within the block configuration, in the order in which the component configurations themselves appear.

### 12.4.2 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 the declarative items contained within the generate statement and whose statement part consists of the 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 generate statement with a for generation scheme, 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 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 a generate statement with an if generation scheme, elaboration consists of the evaluation of the Boolean expression, followed by the generation of exactly one block statement if the expression evaluates to TRUE, and no block statement otherwise. If generated, the block statement has the following form:

- The block label is the same as the label of the generate statement.
- The block declarative part consists of a copy of the declarative items contained within the generate statement.
- The block statement part consists of a copy of the concurrent statements contained within the generate statement.

*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 generation scheme 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.

### 12.4.3 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 statements<sup>6</sup> representing the design entity to which the component instance is bound. The implied block statements are defined in 9.6.1.

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 in-

---

6. LCS 3.

stantiation statement and (within that block) the implied block statements<sup>7</sup> representing the design entity to which the component instance is bound. The implied block statements are defined in 9.6.2.

#### 12.4.4 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 created.
- c) The initial transaction defined by the default value associated with each scalar signal driven by the process statement is inserted into the corresponding driver.

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.

#### 12.5 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 8.9). This elaboration creates the loop parameter and evaluates the discrete range.
- b) Execution of a subprogram call involves the elaboration of the parameter interface list of the corresponding subprogram declaration; this involves the elaboration of each interface declaration to create the corresponding formal parameters. Actual parameters are then associated with formal parameters. Next, if the subprogram is a method of a given<sup>8</sup> protected type (see 3.5.1) or an implicitly declared file operation (see 3.4.1)<sup>9</sup>, the elaboration *blocks* (suspends execution while retaining all state)<sup>10</sup>, if necessary, until exclusive access to the object denoted by the prefix of the method or to the file objected denoted by the file parameter of the file operation<sup>11</sup> 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, then 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.

---

7. LCS 3.

8. Usage correction noted by Ashenden.

9. Ashenden, P1076-2000/D1 review.

10. Added definition as part of D1 review.

11. Ashenden, P1076-2000/D1 review.

## NOTES

- 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.
- 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 b), above. Implementations are allowed to, but not required to, detect and, if possible, resolve such conditions.

## 12.6 Execution of a model

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.

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(T)] 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 a variable representing the 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 that variable 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 expression 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.

### 12.6.1 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 4.3.1.2).

For any driver, there is exactly one transaction whose time component is not greater than the current simulation time. The *current value* of the driver is the value component of this transaction. If, as the result of the advance of time, the current time becomes equal to the time component of the next transaction, then the first transaction is deleted from the projected output waveform and the next becomes the current value of the driver.

## 12.6.2 Propagation of signal values

As simulation time advances, the transactions in the projected output waveform of a given driver (see 12.6.1) will each, in succession, become the value of the driver. When a driver acquires a new value in this way, 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
- If one of its subelements is active
- If the signal is named in the formal part of an association element in a port association list and the corresponding actual is active
- If the signal is a subelement of a resolved signal and the resolved signal is active

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 any given simulation cycle. 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.

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

- It is either a scalar signal or a resolved signal (see 4.3.1.2);
- 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 14.1); and
- It is not an implicit signal GUARD (see 9.1).

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

The driving value of any basic signal S is determined as follows:

- If S has no source, then the driving value of S is given by the default value associated with S (see 4.3.1.2).
- If S has one source that is a driver and S is not a resolved signal (see 4.3.1.2), then the driving value of S is the current<sup>12</sup> value of that driver.

---

12. Boyer.

- 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 4.3.2.2). 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 8.4.1) 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.

The driving value of any signal S that is not a basic signal is determined as follows:

- 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.

For a scalar signal S, the *effective value* of S is determined in the following manner:

- If S is a signal declared by a signal declaration, a port of mode **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 4.3.2.2). 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 4.3.1.2).

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

For a scalar signal S, both the driving and effective values must 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 must be a matching element in both the driving and the *resolved effective*<sup>13</sup> 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 current value of the signal with the newly determined effective value, as follows:

- a) If S is a signal of some type that is not an array type, 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.

---

13. Boyer.

- b) If S is an array 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.

Updating a signal S of type T is said to *change* the current value of S if and only if the expression “S = S'Delayed” evaluates to False, where the “=” operator in the expression is the predefined “=” on type T.<sup>14</sup> If updating a signal causes the current value of that signal to change, then an *event* is said to have occurred on the signal. This definition applies to any updating of a signal, whether such updating occurs according to the above rules or according to the rules for updating implicit signals given in 12.6.3. The occurrence of an event *may will*<sup>15</sup> 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<sup>16</sup>.

For any signal other than one declared with the signal kind **register**, the driving and effective values of the signal are determined and the current value of that signal is updated as described above in every simulation cycle. A signal declared with the signal kind **register** is updated in the same fashion during every simulation cycle except those in which all of its sources have current values that are determined by null transactions.

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.

Implicit signals  $\text{GUARD}_i$ <sup>17</sup>,  $\text{S'Stable}(T)$ ,  $\text{S'Quiet}(T)$ , and  $\text{S'Transaction}$ , for any prefix S and any time T, are not updated according to the above rules; such signals are updated according to the rules described in 12.6.3.

#### NOTES

- 1—In a given<sup>18</sup> simulation cycle, situations can occur where<sup>19</sup> a subelement of a composite signal may be is<sup>20</sup> quiet, but and<sup>21</sup> the signal itself may be is<sup>22</sup> active.
- 2—The rules concerning association of actuals with formals (see 4.3.2.2) 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.
- 3—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 above algorithm.
- 4—Similarly, 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 must 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.
- 5—No effective value is specified for **out** and **linkage** ports, since these ports ~~may not~~ cannot<sup>23</sup> be read.
- 6—Overloading the operator “=” has no effect on the propagation of signal values.

- 
14. LCS 14.
  15. IR1000.4.7.
  16. IR1000.4.7.
  17. Boyer.
  18. IR1000.4.7.
  19. IR1000.4.7.
  20. IR1000.4.7.
  21. IR1000.4.7.
  22. IR1000.4.7.
  23. IR1000.4.7.

7—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.

8—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 4.3.1.2.

### 12.6.3 Updating implicit signals

The kernel process updates the value of each implicit signal **GUARD** associated with a block statement that has a guard expression. 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 expression 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 expression and assigning the result of that evaluation to the variable representing the current value of the signal.

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.

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.

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.

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 14.1) and executing that process.

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 expression 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).

These rules define a partial ordering on all signals within a model. The updating of implicit 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, 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.

### 12.6.4 The simulation cycle

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

The initialization phase consists of the following steps:

- The driving value and the effective value of each explicitly declared signal are computed, and the current value of the signal is set to the effective value. This value is assumed to have been the value of the signal for an infinite length of time prior to the start of simulation.
- 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 expression.
- Each nonpostponed process in the model is executed until it suspends.
- Each postponed process in the model is executed until it suspends.
- 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 step f of the simulation cycle, below.

A simulation cycle consists of the following steps:

- a) The current time,  $T_c$  is set equal to  $T_n$ . Simulation is complete when  $T_n = \text{TIME'HIGH}$  and there are no active drivers or process resummptions at  $T_n$ .
- b) Each active explicit signal in the model is updated. (Events may occur on signals as a result.)
- c) Each implicit signal in the model is updated. (Events may occur on signals as a result.)

- d) 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.
- e) Each nonpostponed process that has resumed in the current simulation cycle is executed until it suspends.
- f) The time of the next simulation cycle,  $T_n$ , is determined by setting it to the earliest of
  - 1) TIME'HIGH,
  - 2) The next time at which a driver becomes active, or
  - 3) The next time at which a process resumes.If  $T_n = T_c$ , then the next simulation cycle (if any) will be a *delta cycle*.
- g) If the next simulation cycle will be a delta cycle, the remainder of this step is skipped. Otherwise, each postponed process that has resumed but has not been executed since its last resumption is executed until it suspends. Then  $T_n$  is recalculated according to the rules of step f. It is an error if the execution of any postponed process causes a delta cycle to occur immediately after the current simulation cycle.

#### NOTES

- 1—The initial value of any implicit signal of the form S'Transaction is not defined.
- 2—Updating of explicit signals is described in 12.6.2; updating of implicit signals is described in 12.6.3.
- 3—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 e of every simulation cycle, while postponed processes are executed during step g of every simulation cycle that does not immediately precede a delta cycle.
- 4—The second and third steps of the initialization phase and steps b and c of the simulation cycle may occur in interleaved fashion. This interleaving may occur because the implicit signal GUARD may be used as the prefix of another implicit signal; moreover, implicit signals may be associated as actuals with explicit signals, making the value of an explicit signal a function of an implicit signal.