# A standard for an Advanced Library Format (ALF) describing Integrated Circuit (IC) technology, cells, and blocks

**This is an unapproved draft for an IEEE standard and subject to change**

**IEEE P1603 Draft 6**

**August 15, 2002**

put in IEEE verbiage

The following individuals contributed to the creation, editing, and review of this document

| | | |
|---|---|---|
| Wolfgang Roethig, Ph.D. | wroethig@eda.org | Official Reporter and WG Chair |
| Joe Daniels | chippewea@aol.com | Technical Editor |

Revision history:

| | |
|---|---|
| IEEE P1596 Draft 0 | August 19, 2001 |
| IEEE P1603 Draft 1 | September 17, 2001 |
| IEEE P1603 Draft 2 | November 12, 2001 |
| IEEE P1596 Draft 3 | April 17, 2002 |
| IEEE P1603 Draft 4 | May 15, 2002 |
| IEEE P1603 Draft 5 | June 21, 2002 |

# Table of Contents

# List of Figures

# List of Tables

# IEEE Standard for an Advanced Library Format (ALF) describing Integrated Circuit (IC) technology, cells, and blocks

## 1. Introduction

**Add a lead-in OR change this to parallel an IEEE intro section**

### 1.1 Motivation

Designing digital integrated circuits has become an increasingly complex process. More functions get integrated into a single chip, yet the cycle time of electronic products and technologies has become considerably shorter. It would be impossible to successfully design a chip of today's complexity within the time-to-market constraints without extensive use of EDA tools, which have become an integral part of the complex design flow. The efficiency of the tools and the reliability of the results for simulation, synthesis, timing and power analysis, layout and extraction rely significantly on the quality of available information about the cells in the technology library.

New challenges in the design flow, especially signal integrity, arise as the traditional tools and design flows hit their limits of capability in processing complex designs. As a result, new tools emerge, and libraries are needed in order to make them work properly. Library creation (generation) itself has become a very complex process and the choice or rejection of a particular application (tool) is often constrained or dictated by the availability of a library for that application. The library constraint can prevent designers from choosing an application program that is best suited for meeting specific design challenges. Similar considerations can inhibit the development and productization of such an application program altogether. As a result, competitiveness and innovation of the whole electronic industry can stagnate.

In order to remove these constraints, an industry-wide standard for library formats, the Advanced Library Format (ALF), is proposed. It enables the EDA industry to develop innovative products and ASIC designers to choose the best product without library format constraints. Since ASIC vendors have to support a multitude of libraries according to the preferences of their customers, a common standard library is expected to significantly reduce the library development cycle and facilitate the deployment of new technologies sooner.

## 1.2 Goals

The basic goals of the proposed library standard are

- *simplicity* - library creation process needs to be easy to understand and not become a cumbersome process only known by a few experts.
- *generality* - tools of any level of sophistication need to be able to retrieve necessary information from the library.
- *expandability* - this needs to be done for early adoption and future enhancement possibilities.
- *flexibility* - the choice of keeping information in one library or in separate libraries needs to be in the hand of the user not the standard.
- *efficiency* - the complexity of the design information requires the process of retrieving information from the library does not become a bottleneck. The right trade-off between compactness and verbosity needs to be established.
- *ease of implementation* - backward compatibility with existing libraries shall be provided and translation to the new library needs to be an easy task.
- *conciseness* - unambiguous description and accuracy of contents shall be detailed.
- *acceptance* - there needs to be a preference for the new standard library over existing libraries.

## 1.3 Target applications

The fundamental purpose of ALF is to serve as the primary database for all third-party applications of ASIC cells. In other words, it is an elaborate and formalized version of the *databook*.

In the early days, databooks provided all the information a designer needed for choosing a cell in a particular application: Logic symbols, schematics, and a truth table provided the functional specification for simple cells. For more complex blocks, the name of the cell (e.g., asynchronous ROM, synchronous 2-port RAM, or 4-bit synchronous up-down counters) and timing diagrams conveyed the functional information. The performance characteristics of each cell were provided by the loading characteristics, delay and timing constraints, and some information about DC and AC power consumption. The designers chose the cell type according to the functionality, estimated the performance of the design, and eventually re-implemented it in an optimized way as necessary to meet performance constraints.

Design automation enabled tremendous progress in efficiency, productivity, and the ability to deal with complexity, yet it did not change the fundamental requirements for ASIC design. Therefore, ALF needs to provide models with *functional* information and *performance* information, primarily including timing and power. Signal integrity characteristics, such as noise margin can also be included under performance category. Such informa-

tion is typically found in any databook for analog cells. At deep sub-micron levels, digital cells behave similar to analog cells as electronic devices bound by physical laws and therefore are not infinitely robust against noise.

Table 1 shows a list of applications used in ASIC design flow and their relationship to ALF.

NOTE — ALF covers *library* data, whereas *design* data needs to be provided in other formats.

**Table 1—Target applications and models supported by ALF**

| Application | Functional model | Performance model | Physical model |
|---|---|---|---|
| *Simulation* | Derived from ALF | N/A | N/A |
| *Synthesis* | Supported by ALF | Supported by ALF | Supported by ALF |
| *Design for test* | Supported by ALF | N/A | N/A |
| *Design planning* | Supported by ALF | Supported by ALF | Supported by ALF |
| *Timing analysis* | N/A | Supported by ALF | N/A |
| *Power analysis* | N/A | Supported by ALF | N/A |
| *Signal integrity* | N/A | Supported by ALF | N/A |
| *Layout* | N/A | N/A | Supported by ALF |

Historically, a functional model was virtually identical to a simulation model. A functional gate-level model was used by the proprietary simulator of the ASIC company and it was easy to lump it together with a rudimentary timing model. Timing analysis was done through dynamic functional simulation. However, with the advanced level of sophistication of both functional simulation and timing analysis, this is no longer the case. The capabilities of the functional simulators have evolved far beyond the gate-level and timing analysis has been decoupled from simulation.

RTL design planning is an emerging application type aiming to produce “virtual prototypes” of complex for system-on-chip (SOC) designs. RTL design planning is thought of as a combination of some or all of RTL floorplanning and global routing, timing budgeting, power estimation, and functional verification, as well as analysis of signal integrity, EMI, and thermal effects. The library components for RTL design planning range from simple logic gates to parameterizeable macro-functions, such as memories, logic building blocks, and cores.

From the point of view of library requirements, applications involved in RTL design planning need functional, performance, and physical data. The functional aspect of design planning includes RTL simulation and formal verification. The performance aspect covers timing and power as primary issues, while signal integrity, EMI, and thermal effects are emerging issues. The physical aspect is floorplanning. As stated previously, the functional and performance models of components can be described in ALF.

ALF also covers the requirements for physical data, including layout. This is important for the new generation of tools, where logical design merges with physical design. Also, all design steps involve optimization for timing, power, signal integrity, i.e. electrical correctness and physical correctness. EDA tools need to be knowledgeable about an increasing number of design aspects. For example, a place and route tool needs to consider congestion as well as timing, crosstalk, electromigration, antenna rules etc. Therefore it is a logical step to combine the functional, electrical and physical models needed by such a tool in a unified library.

Figure 1 shows how ALF provides information to various design tools.

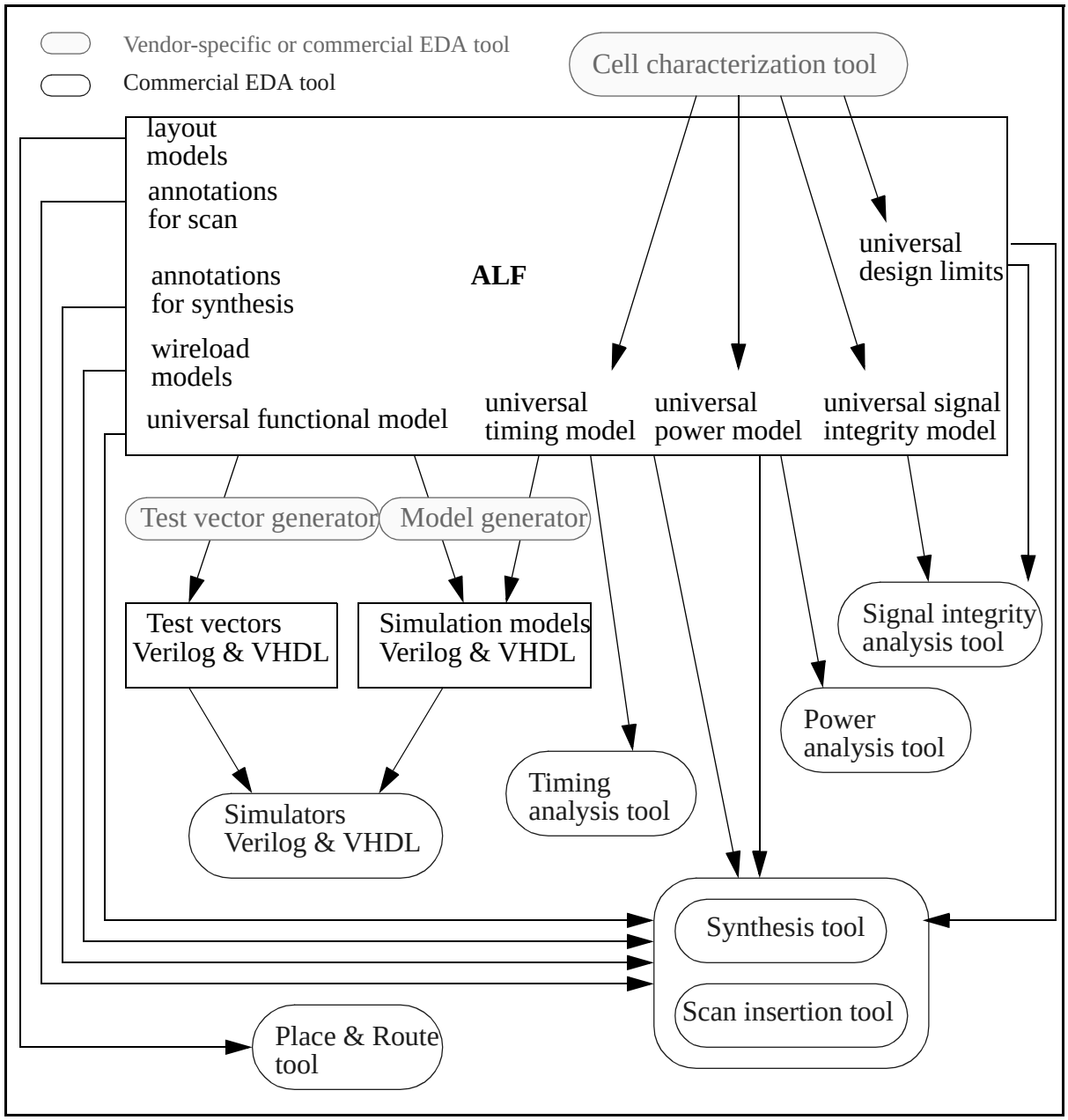



**Figure 1—ALF and its target applications**

The worldwide accepted standards for hardware description and simulation are VHDL and Verilog. Both languages have a wide scope of describing the design at various levels of abstraction: behavioral, functional, synthesizable RTL, and gate level. There are many ways to describe gate-level functions. The existing simulators are implemented in such a way that some constructs are more efficient for simulation run time than others. Also, how the simulation model handles timing constraints is a trade-off between efficiency and accuracy. Developing efficient simulation models which are functionally reliable (i.e., pessimistic for detecting timing constraint violation) is a major development effort for ASIC companies.

Hence, the use of a particular VHDL or Verilog simulation model as primary source of functional description of a cell is not very practical. Moreover, the existence of two simulation standards makes it difficult to pick one as a

reference with respect to the other. The purpose of a generic functional model is to serve as an absolute reference for all applications that require functional information. Applications such as synthesis, which need functional information merely for recognizing and choosing cell types, can use the generic functional model directly. For other applications, such as simulation and test, the generic functional model enables automated simulation model and test vector generation and verification, which has a tremendous benefit for the ASIC industry.

With progress of technology, the set of physical constraints under which the design functions have increased dramatically, along with the cost constraints. Therefore, the requirements for detailed characterization and analysis of those constraints, especially timing and power in deep submicron design, are now much more sophisticated. Only a subset of the increasing amount of characterization data appears in today's databooks.

ALF provides a generic format for all type of characterization data, without restriction to state-of-the art timing models. Power models are the most immediate extension and they have been the starter and primary driver for ALF.

Detailed timing and power characterization needs to take into account the *mode of operation* of the ASIC cell, which is related to the functionality. ALF introduces the concept of *vector-based modeling*, which is a generalization and a superset of today's timing and power modeling approaches. All existing timing and power analysis applications can retrieve the necessary model information from ALF.

## 1.4 Conventions

The syntax for description of lexical and syntax rules uses the following conventions.
**Consider using the BNF nomenclature from IEEE 1481-1999**

```
::= definition of a syntax rule
|   alternative definition
[item]an optional item
[item1 | item2 | ... ]  optional item with alternatives
{item}optional item that can be repeated
{item1 | item2 | ... }  optional items with alternatives
                        which can be repeated
item item in boldface font is taken verbatim
item item in italic is for explanation purpose only
```

The syntax for explanation of semantics of expressions uses the following conventions.

```
=== left side and right side expressions are equivalent
<item>a placeholder for an item in regular syntax
```

## 1.5 Contents of this standard

The organization of the remainder of this standard is

- Clause 2 (References) provides references to other applicable standards that are assumed or required for ALF.
- Clause 3 (Definitions) defines terms used throughout the different specifications contained in this standard.
- Clause 4 (Acronyms and abbreviations) defines the acronyms used in this standard.
- Clause 5 (ALF language construction principles and overview) defines the language construction principles.
- Clause 6 (Lexical rules) specifies the lexical rules.
- Clause 7 (Auxiliary syntax rules) defines syntax and semantics of auxiliary items used in this standard.

— Clause 8 (Generic objects and related statements) defines syntax and semantics of generic objects used in this standard.
— Clause 9 (Library-specific objects and related statements) defines syntax and semantics of library-specific objects used in this standard.
— Clause 10 (Constructs for modeling of functional behavior) defines syntax and semantics of the control expression language used in this standard
— Clause 11 (Constructs for electrical and physical modeling) defines syntax and semantics of arithmetic models used in this standard.
— Annexes. Following Clause 11are a series of normative and informative annexes.

## 2. References

**Fill in applicable references, i.e. standards on which the herein proposed standard depends.

This standard shall be used in conjunction with the following publication. When the following standard is superseded by an approved revision, the revision shall apply.

**The following is only an example. ALF does not depend on C.

ISO/IEC 9899:1990, Programming Languages—C.[1]

[ISO 8859-1 : 1987(E)] ASCII character set

[1]ISO publications are available from the ISO Central Secretariat, Case Postale 56, 1 rue de Varembé, CH-1211, Genève 20, Switzerland/Suisse (http://www.iso.ch/). IEC publications are available from the Sales Department of the International Electrotechnical Commission, Case Postale 131, 3, rue de Varembé, CH-1211, Genève 20, Switzerland/Suisse (http://www.iec.ch/). ISO/IEC publications are also available in the United States from the Sales Department, American National Standards Institute, 11 West 42nd Street, 13th Floor, New York, NY 10036, USA.

## 3. Definitions

For the purposes of this standard, the following terms and definitions apply. The *IEEE Standard Dictionary of Electrical and Electronics Terms* [B4] should be consulted for terms not defined in this standard.

**Fill in definitions of terms which are used in the herein proposed standard.

3.1 **advanced library format:** The format of any file that can be parsed according to the syntax and semantics defined within this standard.

3.2 **application**, **electric design automation (EDA) application:** Any software program that uses data represented in the Advanced Library Format (ALF). Examples include RTL (Register Transfer Level) synthesis tools, static timing analyzers, etc. *See also:* **advanced library format**; **register transfer level**.

3.3 **arc**: *See:* **timing arc**.

3.4 **argument:** A data item required for the mathematical evaluation of an arithmetic model. *See also:* **arithmetic model**.

3.5 **arithmetic model:** A representation of a library quantity that can be mathematically evaluated.

3.6 ...

3.7 **register transfer level:** A behavioral representation of a digital electronic design allowing inference of sequential and combinational logic components.

3.8 ...

3.9 **timing arc:** An abstract representation of a measurement between two points in time during operation of a library component.

3.10 ...

## 4. Acronyms and abbreviations

This clause lists the acronyms and abbreviations used in this standard.

| | |
|---|---|
| ALF | advanced library format, title of the herein proposed standard |
| ASIC | application specific integrated circuit |
| AWE | asymptotic waveform evaluation |
| BIST | built-in self test |
| BNF | Backus-Naur Form |
| CAE | computer-aided engineering [the term electronic design automation (EDA) is preferred] |
| CAM | content-addressable memory |
| CLF | Common Library Format from Avant! Corporation |
| CPU | central processing unit |
| DCL | Delay Calculation Language from IEEE 1481-1999 std |
| DEF | Design Exchange Format from Cadence Design Systems Inc. |
| DLL | delay-locked loop |
| DPCM | Delay and Power Calculation Module from IEEE 1481-1999 std |
| DPCS | Delay and Power Calculation System from IEEE 1481-1999 std |
| DSP | digital signal processor |
| DSPF | Detailed Standard Parasitic Format |
| EDA | electronic design automation |
| EDIF | Electronic Design Interchange Format |
| HDL | hardware description language |
| IC | integrated circuit |
| IP | intellectual property |
| ILM | Interface Logic Model from Synopsys Inc. |
| LEF | Library Exchange Format from Cadence Design Systems Inc. |
| LIB | Library Format from Synopsys Inc. |
| LSSD | level-sensitive scan design |
| MPU | micro processor unit |
| OLA | Open Library Architecture from Silicon Integration Initiative Inc. |
| PDEF | Physical Design Exchange Format from IEEE 1481-1999 std |
| PLL | Phase-locked loop |
| PVT | process/voltage/temperature (denoting a set of environmental conditions) |
| QTM | Quick Timing Model |
| RAM | random access memory |
| RC | resistance times capacitance |
| RICE | rapid interconnect circuit evaluator |
| ROM | read-only memory |
| RSPF | Reduced Standard Parasitic Format |
| RTL | Register Transfer Level |
| SDF | Standard Delay Format from IEEE 1497 std |
| SDC | Synopsys Design Constraint format from Synopsys Inc. |
| SPEF | Standard Parasitic Exchange Format from IEEE 1481-1999 std |
| SPF | Standard Parasitic Format |
| SPICE | Simulation Program with Integrated Circuit Emphasis |
| STA | Static Timing Analysis |

| | |
|---|---|
| STAMP | (STA Model Parameter ?) format from Synopsys Inc. |
| TCL | Tool Command Language (supported by multiple EDA vendors) |
| TLF | Timing Library Format from Cadence Design Systems Inc. |
| VCD | Value Change Dump format (from IEEE 1364 std ?) |
| VHDL | VHSIC Hardware Description Language |
| VHSIC | very-high-speed integrated circuit |
| VITAL | VHDL Initiative Towards ASIC Libraries from IEEE ??? std |
| VLSI | very-large-scale integration |

# 5. ALF language construction principles and overview

**Add lead-in text**

This section presents the ALF language construction principles and gives an overview of the language features. The types of ALF statements and rules for parent/child relationships between types are presented summarily. Most of the types are associated with predefined keywords. The keywords in ALF shall be case-insensitive. However, uppercase is used for keywords throughout this section for clarity.

## 5.1 ALF meta-language

Syntax 1 establishes an *ALF meta-language*.

```
ALF_statement ::=
    ALF_type [ALF_name ] [ = ALF_value ] ALF_statement_termination
ALF_type ::=
      non_escaped_identifier [ index ]
    | @
    | :
ALF_name ::=
      identifier [ index ]
    | control_expression
ALF_value ::=
      identifier
    | number
    | arithmetic_expression
    | boolean_expression
    | control_expression
ALF_statement_termination ::=
      ;
    | { { ALF_value | : | ; } }
    | { { ALF_statement } }
```

*Syntax 1—Syntax construction for ALF meta-language*

An *ALF statement* uses the delimiters "**;**", "**{**" and "**}**" to indicate its termination.

The *ALF type* is defined by a *keyword* (see 6.12) eventually in conjunction with an *index* (see 7.8) or by the *operator* "**@**" (6.4) or by the *delimiter* "**:**" (see 6.3). The usage of keyword, index, operator, or delimiter as ALF type is defined by ALF language rules concerning the particular ALF type.

The *ALF name* is defined by an *identifier* (see 6.11) eventually in conjunction with an index or by a *control expression* (see 10.16). Depending on the ALF type, the ALF name is mandatory or optional or not applicable. The usage of identifier, index, or control expression as ALF name is defined by ALF language rules concerning the particular ALF type.

The *ALF value* is defined by an identifier, a *number* (see 6.5), an *arithmetic expression* (see 11.1), a *boolean expression* (see 10.9), or a control expression. Depending on the type of the ALF statement, the ALF value is mandatory or optional or not applicable. The usage of identifier, number, arithmetic expression, boolean expression or control expression as ALF value is defined by ALF language rules concerning the particular ALF type.

An ALF statement can contain one or more other ALF statements. The former is called *parent* of the latter. Conversely, the latter is called *child* of the former. An ALF statement with child is called a *compound* ALF statement.

An ALF statement containing one or more ALF values, eventually interspersed with the delimiters "**;**" or "**:**", is called a *semi-compound* ALF statement. The items between the delimiters "**{**" and "**}**" are called *contents* of the ALF statement. The usage of the delimiters "**;**" or "**:**" within the contents of an ALF statement is defined by ALF language rules concerning the particular ALF statement.

An ALF statement without child is called an *atomic* ALF statement. An ALF statement which is either compound or semi-compound is called a *non-atomic* ALF statement.

*Examples*

a) ALF statement describing an unnamed object without value:

```
ARBITRARY_ALF_TYPE {
    // put children here
}
```

b) ALF statement describing an unnamed object with value:

```
ARBITRARY_ALF_TYPE = arbitrary_ALF_value;
```

or

```
ARBITRARY_ALF_TYPE = arbitrary_ALF_value {
    // put children here
}
```

c) ALF statement describing a named object without value:

```
ARBITRARY_ALF_TYPE arbitrary_ALF_name;
```

or

```
ARBITRARY_ALF_TYPE arbitrary_ALF_name {
    // put children here
}
```

d) ALF statement describing a named object with value:

```
ARBITRARY_ALF_TYPE arbitrary_ALF_name = arbitrary_ALF_value;
```

or

```
ARBITRARY_ALF_TYPE arbitrary_ALF_name = arbitrary_ALF_value {
    // put children here
}
```

## 5.2 Categories of ALF statements

In this section, the terms *statement*, *type*, *name*, *value* are used for shortness in lieu of *ALF statement*, *ALF name*, *ALF value*, respectively.

Statements are divided into the following categories: *generic object*, *library-specific object*, *arithmetic model*, *arithmetic submodel*, *arithmetic model container*, *geometric model*, *annotation*, *annotation container*, and *auxiliary statement*, as shown in Table 2.

**Table 2—Categories of ALF statements**

| Category | Purpose | Syntax particularity |
|---|---|---|
| Generic object | Provide a definition for use within other ALF statements. | Statement is atomic, semi-compound or compound.<br>Name is mandatory.<br>Value is either mandatory or not applicable. |

**Table 2—Categories of ALF statements (Continued)**

| Category | Purpose | Syntax particularity |
|---|---|---|
| Library-specific object | Describe the contents of a IC technology library. | Statement is atomic or compound.<br>Name is mandatory.<br>Value does not apply.<br>Category of parent is exclusively *library-specific object.* |
| Arithmetic model | Describe an abstract mathematical quantity that can be calculated and eventually measured within the design of an IC. | Statement is atomic or compound.<br>Name is optional.<br>Value is mandatory, if atomic. |
| Arithmetic submodel | Describe an arithmetic model under a specific measurement condition. | Statement is atomic or compound.<br>Name does not apply.<br>Value is mandatory, if atomic.<br>Category of parent is exclusively *arithmetic model.* |
| Arithmetic model container | Provide a context for an arithmetic model. | Statement is compound.<br>Name and value do not apply.<br>Category of child is exclusively *arithmetic model.* |
| Geometric model | Describe an abstract geometrical form used in physical design of an IC. | Statement is semi-compound or compound.<br>Name is optional.<br>Value does not apply. |
| Annotation | Provide a qualifier or a set of qualifiers for an ALF statement. | Statement is atomic, semi-compound or compound.<br>Name does not apply.<br>Value is mandatory, if atomic or compound.<br>Value does not apply, if semi-compound.<br>Category of child is exclusively *annotation.* |
| Annotation container | Provide a context for an annotation. | Statement is compound.<br>Name and value do not apply.<br>Category of child is exclusively *annotation.* |
| Auxiliary statement | Provide an additional description within the context of a library-specific object, an arithmetic model, an arithmetic submodel, geometric model or another auxiliary statement. | Dependent on subcategory. |

Figure 2 illustrates the parent/child relationship between categories of statements.

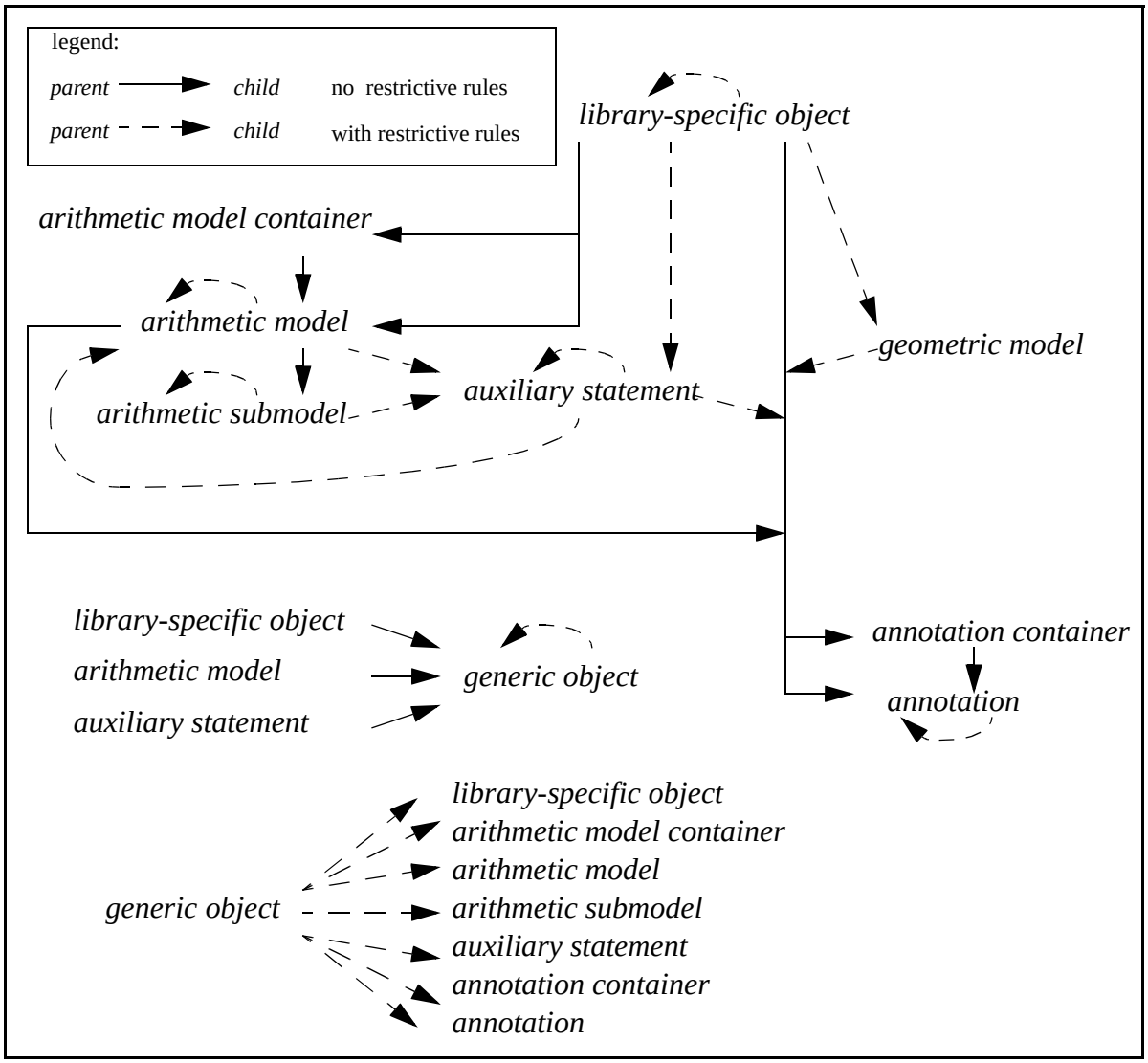



**Figure 2—Parent/child relationship between ALF statements**

More detailed rules for parent/child relationships for particular types of statements apply.

## 5.3 Generic objects and library-specific objects

Statements with mandatory name are called *objects*, i.e., *generic object* and *library-specific object*.

Table 3 lists the keywords and items in the category *generic object*. The keywords used in this category are called *generic keywords.*

**Table 3—Generic objects**

| Keyword | Item | Section |
|---|---|---|
| ALIAS | Alias declaration | See 8.1. |

**Table 3—Generic objects (Continued)**

| Keyword | Item | Section |
|---|---|---|
| CONSTANT | Constant declaration | See 8.2. |
| CLASS | Class declaration | See 8.3. |
| GROUP | Group declaration | See 8.7. |
| KEYWORD | Keyword declaration | See 8.4. |
| SEMANTICS | Semantics declaration | See 8.6. |
| TEMPLATE | Template declaration | See 8.8. |

Table 4 lists the keywords and items in the category *library-specific object.* The keywords used in this category are called *library-specific keywords.*

**Table 4—Library-specific objects**

| Keyword | Item | Section |
|---|---|---|
| LIBRARY | Library declaration | See 9.1. |
| SUBLIBRARY | Sublibrary declaration | See 9.1. |
| CELL | Cell declaration | See 9.3. |
| PRIMITIVE | Primitive declaration | See 9.11. |
| WIRE | Wire declaration | See 9.12. |
| PIN | Pin declaration | See 9.7. |
| PINGROUP | Pin group declaration | See 9.8. |
| VECTOR | Vector declaration | See 9.16. |
| NODE | Node declaration | See 9.15. |
| LAYER | Layer declaration | See 9.18. |
| VIA | Via declaration | See 9.20. |
| RULE | Rule declaration | See 9.23. |
| ANTENNA | Antenna declaration | See 9.24. |
| SITE | Site declaration | See 9.28. |
| ARRAY | Array declaration | See 9.30. |
| BLOCKAGE | Blockage declaration | See 9.25. |
| PORT | Port declaration | See 9.26. |
| PATTERN | Pattern declaration | See 9.32. |
| REGION | Region declaration | See 9.34. |

Figure 3 illustrates the parent/child relationship between statements within the category *library-specific object.*

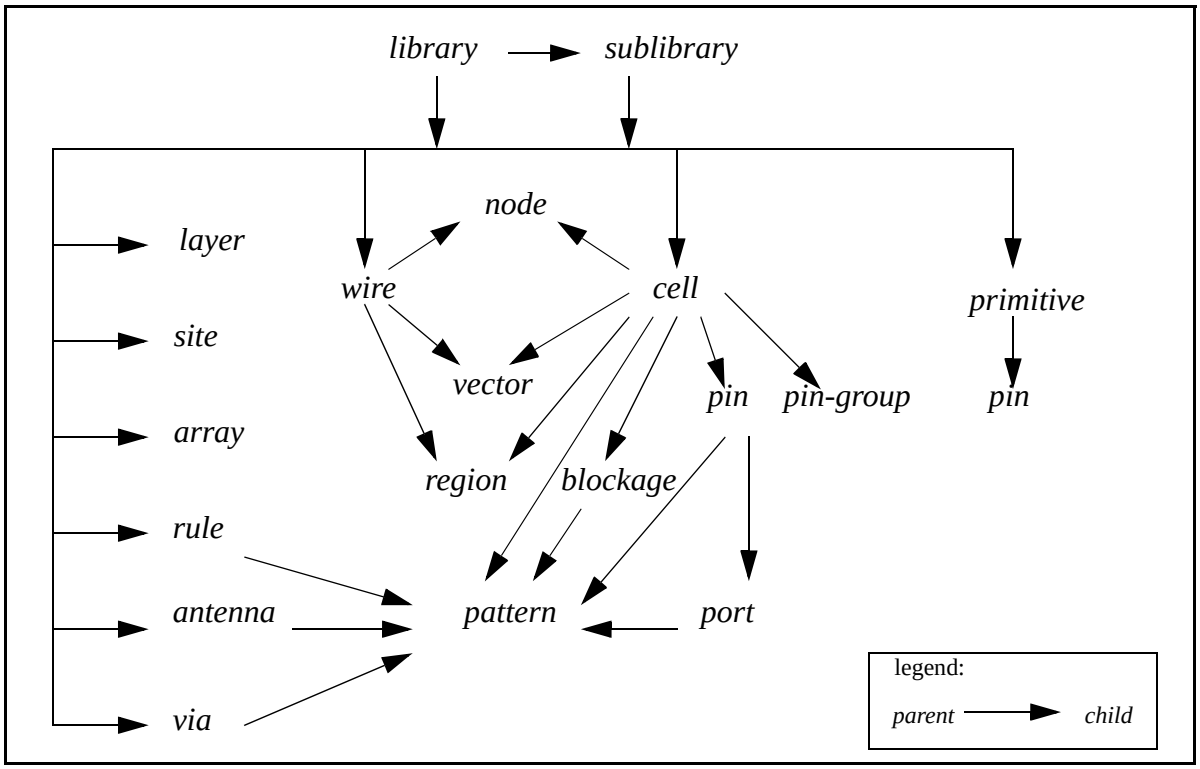



**Figure 3—Parent/child relationship amongst library-specific objects**

A parent can have multiple library-specific objects of the same type as children. Each child is distinguished by name.

## 5.4 Singular statements and plural statements

Auxiliary statements with predefined keywords are divided in the following subcategories: *singular statement* and *plural statement*.

Auxiliary statements with predefined keywords and without name are called *singular statements*. Auxiliary statements with predefined keywords and with name, yet without value, are called *plural statements*.

Table 5 lists the singular statements.

**Table 5—Singular statements**

| Keyword | Item | Value | Complexity | Section |
|---|---|---|---|---|
| FUNCTION | Function statement | N/A | Compound | See 10.1. |
| TEST | Test statement | N/A | Compound | See 10.2. |
| RANGE | Range statement | N/A | Semi-compound | See 10.8. |
| FROM | From statement | N/A | Compound | See 11.4.10. |
| TO | To statement | N/A | Compound | See 11.4.10. |

**Table 5—Singular statements (Continued)**

| Keyword | Item | Value | Complexity | Section |
|---|---|---|---|---|
| VIOLATION | Violation statement | N/A | Compound | See 11.4.12. |
| HEADER | Header statement | N/A | Compound (or semi-compound?) | See 11.3.1. |
| TABLE | Table statement | N/A | Semi-compound | See 11.3.2. |
| EQUATION | Equation statement | N/A | Semi-compound | See 11.3.3. |
| BEHAVIOR | Behavior statement | N/A | Compound | See 10.4. |
| STRUCTURE | Structure statement | N/A | Compound | See 10.5. |
| NON_SCAN_CELL | Non-scan cell statement | Optional | Compound or semi-compound | See 10.7. |
| ARTWORK | Artwork statement | Mandatory | Compound or atomic | See 9.38. |

Table 6 lists the plural statements.

**Table 6—Plural statements**

| Keyword | Item | Name | Complexity | Section |
|---|---|---|---|---|
| STATETABLE | State table statement | Optional | Semi-compound | See 10.6. |
| @ | Control statement | Mandatory | Compound | See 10.4. |
| : | Alternative control statement | Mandatory | Compound | See 10.4. |

Figure 4 illustrates the parent/child relationship for singular statements and plural statements.

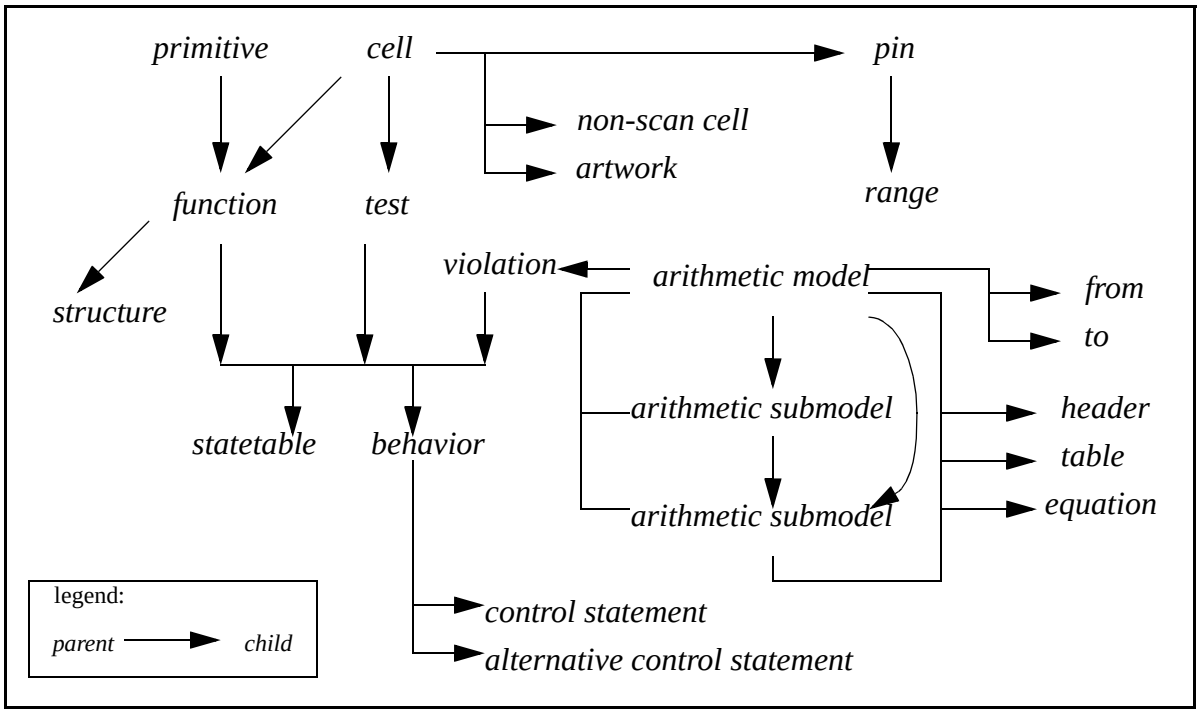



**Figure 4—Parent/child relationship involving singular statements and plural statements**

A parent can have at most one child of a particular type in the category singular statements, but multiple children of a particular type in the category plural statements.

## 5.5 Instantiation statement and assignment statement

Auxiliary statements without predefined keywords use the name of an object as keyword. Such statements are divided in the following subcategories: *instantiation statement* and *assignment statement*.

Compound or semi-compound statements using the name of an object as keyword are called *instantiation statements.* Their purpose is to specify an instance of the object.

Table 7 lists the instantiation statements.

**Table 7—Instantiation statements**

| Item | Name | Value | Section |
|---|---|---|---|
| Cell instantiation | Optional | N/A | See 9.4. |
| Primitive instantiation | Optional | N/A | See 10.4. |
| Template instantiation | N/A | Optional | See 8.9. |
| Via instantiation | Mandatory | N/A | See 9.21. |
| Wire instantiation | Mandatory | N/A | *Proposed* for IEEE. |

Atomic statements without name using an identifier as keyword which has been defined within the context of another object are called assignment statements. A value is mandatory for assignment statements, as their purpose is to assign a value to the identifier. Such an identifier is called a *variable*.

Table 8 lists the assignment statements.

**Table 8—Assignment statements**

| Item | Section |
|---|---|
| Pin assignment | See 7.10. |
| Arithmetic assignment | See 8.9. |
| Boolean assignment | See 10.4. |

Figure 5 illustrates the parent/child relationship involving instantiation and assignment statements.

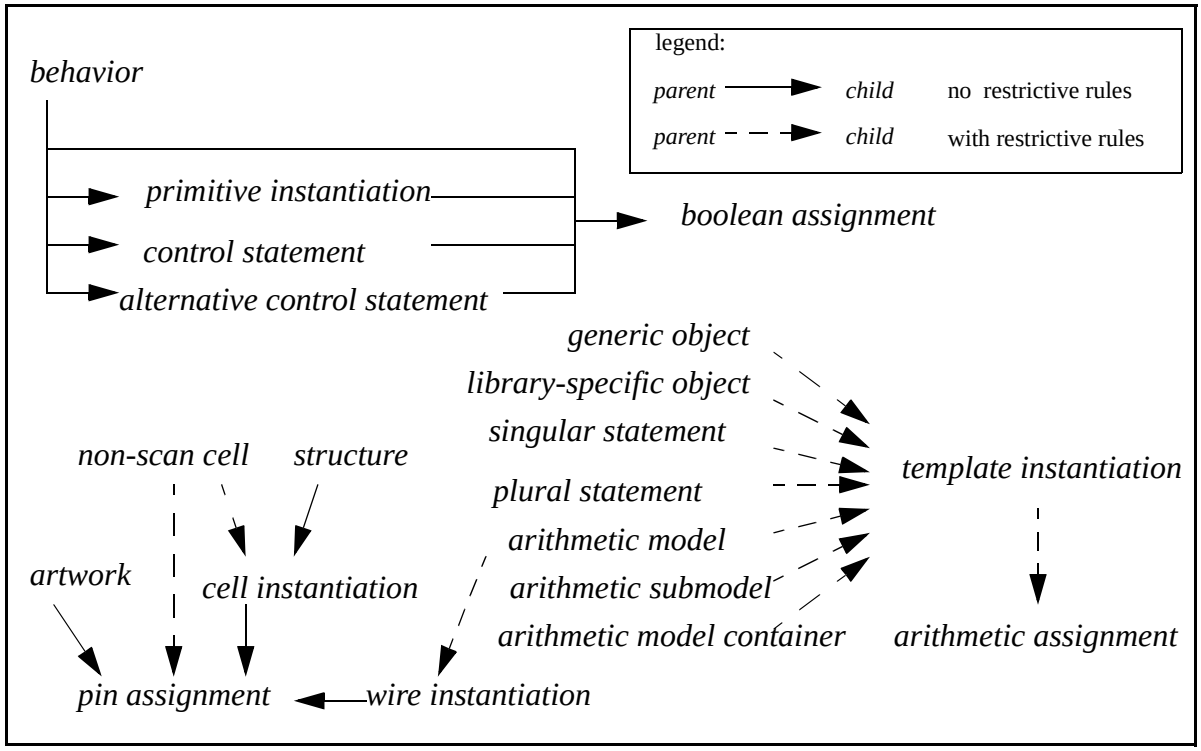



**Figure 5—Parent/child relationship involving instantiation and assignment statements**

A parent can have multiple children using the same keyword in the category instantiation statement, but at most one child using the same variable in the category assignment statement.

## 5.6 Annotation, arithmetic model, and related statements

Multiple keywords are predefined in the categories *arithmetic model*, *arithmetic model container*, *arithmetic submodel*, *annotation*, *annotation container*, and *geometric model*. Their semantics are established within the

context of their parent. Therefore they are called *context-sensitive keywords*. In addition, the ALF language allows additional definition of keywords in these categories.

Table 9 provides a reference to sections where more definitions about these categories can be found.

**Table 9—Other categories of ALF statements**

| Item | Section |
|---|---|
| Arithmetic model | See 11.2. |
| Arithmetic submodel | See 11.4.3. |
| Arithmetic model container | See 11.4.7. |
| Annotation | See 7.11. |
| Annotation container | See 7.12. |
| Geometric model | See 9.35. |

There exist predefined keywords with generic semantics in the category *annotation* and *annotation container.* They are called *generic keywords*, like the keywords for *generic objects*.

Table 10 lists the generic keywords in the category annotation and annotation container.

**Table 10—Annotations and annotation containers with generic keyword**

| Keyword | Item / subcategory | Section |
|---|---|---|
| `PROPERTY` | Annotation container. | See 7.14. |
| `ATTRIBUTE` | Multi-value annotation. | See 7.13. |
| `INFORMATION` | Annotation container. | See 9.2.1. |

Table 11 lists predefined keywords in categories related to arithmetic model.

**Table 11—Keywords related to arithmetic model**

| Keyword | Item / category | Section |
|---|---|---|
| `LIMIT` | Arithmetic model container. | See 11.4.8. |
| `MIN` | Arithmetic submodel, also operator within *arithmetic expression.* | See 11.4.3, 11.1.3. |
| `MAX` | Arithmetic submodel, also operator within *arithmetic expression.* | See 11.4.4, 11.1.3. |
| `TYP` | Arithmetic submodel. | See 11.4.5. |
| `DEFAULT` | Annotation. | See 11.5.4. |
| `ABS` | Operator within *arithmetic expression*. | See 11.1.3. |
| `EXP` | Operator within *arithmetic expression*. | See 11.1.3. |

**Table 11—Keywords related to arithmetic model (Continued)**

| Keyword | Item / category | Section |
|---|---|---|
| LOG | Operator within *arithmetic expression*. | See 11.1.3. |

The definitions of other predefined keywords, especially in the category arithmetic model, can be self-described in ALF using the *keyword declaration* statement (see 8.4).

## 5.7 Statements for parser control

Table 12 provides a reference to statements used for ALF parser control.

**Table 12—Statements for ALF parser control**

| Keyword | Statement | Section |
|---|---|---|
| INCLUDE | Include statement | See 7.15. |
| ASSOCIATE | Associate statement | See 7.16. |
| ALF_REVISION | Revision statement | See 7.17. |

The statements for parser control do not necessarily follow the ALF meta-language shown in Syntax 1.

## 5.8 Name space and visibility of statements

The following rules for name space and visibility shall apply:

a) A statement shall be visible within its parent statement, but not outside its parent statement.
b) A statement visible within another statement shall also be visible within a child of that other statement.
c) All objects (i.e., generic objects and library-specific objects) shall share a common name space within their scope of visibility. No object shall use the same name as any other visible object. Conversely, an object can use the same name as any other object outside the scope of its visibility.
d) The following exception of rule c) is allowed for specific objects and with specific semantic implications. An object of the same type and the same name can be redeclared, if semantic support for this redeclaration is provided. The purpose of such a redeclaration is to supplement the original declaration with new children statements which augment the original declaration without contradicting it.
e) All statements with optional names (i.e., property, arithmetic model, geometric model) shall share a common name space within their scope of visibility. No statement with optional name shall use the same name as any other visible statement with optional name. Conversely, a statement can use the same optional name as any other statement with optional name outside the scope of its visibility.

# 6. Lexical rules

This section discusses the lexical rules.

The ALF source text files shall be a stream of *lexical tokens* and *whitespace*. Lexical tokens shall be divided into the categories *delimiter, operator, comment, number, bit literal, based literal, edge, quoted string,* and *identifier.*

Each lexical token shall be composed of one or more characters. Whitespace shall be used to separate lexical tokens from each other. Whitespace shall not be allowed within a lexical token with the exception of *comment* and *quoted string*.

The specific rules for construction of lexical tokens and for usage of whitespace are defined in this section.

## 6.1 Character set

This standard shall use the ASCII character set [ISO 8859-1 : 1987(E)].

The *ASCII character set* shall be divided into the following categories: *whitespace, letter, digit,* and *special,* as shown in Syntax 2.

```
character ::=
      whitespace
    | letter
    | digit
    | special
whitespace ::=
    space | vertical_tab | horizontal_tab | new_line | carriage_return | form_feed
letter ::=
    uppercase | lowercase
uppercase ::=
    A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W
    | X | Y | Z
lowercase ::=
    a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z
digit ::=
    0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
special ::=
    & | | | ^ | ~ | + | - | * | / | % | ? | ! | : | ; | , | " | ' | @ | = | \ | . | $ | _ | #
    | ( | ) | < | > | [ | ] | { | }
```

*Syntax 2—ASCII character set*

Table 13 shows the list of *whitespace* characters and their ASCII code.

**Table 13—List of whitespace characters**

| Name | ASCII code (octal) |
|---|---|
| Space | 200 |
| Horizontal tab | 011 |
| New line | 012 |
| Vertical tab | 013 |

**Table 13—List of whitespace characters (Continued)**

| Name | ASCII code (octal) |
|---|---|
| Form feed | 014 |
| Carriage return | 015 |

Table 14 shows the list of *special* characters and their names used in this standard

**Table 14—List of special characters**

| Symbol | ASCII code (octal) | Name |
|---|---|---|
| & | | Amperesand |
| \| | | Vertical bar |
| ^ | | Caret |
| ~ | | Tilde |
| + | | Plus |
| - | | Minus |
| * | | Asterix |
| / | | Slash |
| % | | Percent |
| ? | | Question mark |
| ! | | Exclamation mark |
| : | | Colon |
| ; | | Semicolon |
| , | | Comma |
| ” | | Double quote |
| ’ | | Single quote |
| @ | | At sign |
| = | | Equal sign |
| \ | | Backslash |
| . | | Dot |
| $ | | Dollar |
| _ | | Underscore |

**Table 14—List of special characters (Continued)**

| Symbol | ASCII code (octal) | Name |
|---|---|---|
| # | | Pound |
| ( ) | | Parenthesis (open, close) |
| < > | | Angular bracket (open, close) |
| [ ] | | Square bracket (open, close) |
| { } | | Curly bracket (open, close) |

## 6.2 Comment

A *comment* shall be divided into the subcategories *in-line comment* and *block comment*, as shown in Syntax 3.

```
comment ::=
      in_line_comment
    | block_comment
in_line_comment ::=
      //{character}new_line
    | //{character}carriage_return
block_comment ::=
    /*{character}*/
```

*Syntax 3—Comment*

The start of an in-line comment shall be determined by the occurence of two subsequent *slash* characters without whitespace in-between. The end of an in-line comment shall be determined by the occurence of a *new line* or of a *carriage return* character.

The start of a block comment shall be determined by the occurence of a *slash* character followed by an *asterix* without whitespace in-between. The end of a block comment shall be determined by the occurence of an *asterix* character followed by a *slash* character.

A comment shall have the same semantic meaning as a whitespace. Therefore, no syntax rule shall involve a comment.

## 6.3 Delimiter

The special characters shown in Syntax 4 shall be considered *delimiters*.

```
delimiter ::=
    ( | ) | [ | ] | { | } | : | ; | ,
```

*Syntax 4—Delimiter*

When appearing in a syntax rule, a delimiter shall be used to indicate the end of a statement or of a partial statement, the begin and end of an expression or of a partial expression.

## 6.4 Operator

*Operators* shall be divided into the following subcategories: *arithmetic operator*, *boolean operator*, *relational operator*, *shift operator*, *event sequence operator*, and *meta operator*, as shown in Syntax 5

```
operator ::=
      arithmetic_operator
    | boolean_operator
    | relational_operator
    | shift_operator
    | event_sequence_operator
    | meta_operator
arithmetic_operator ::=
    + | - | * | / | % | **
boolean_operator ::=
    && | || | ~& | ~| | ^ | ~^ | ~ | ! | & | |
relational_operator ::=
    == | != | >= | <= | > | <
shift_operator ::=
    << | >>
event_sequence_operator ::=
    -> | ~> | <-> | <~> | &> | <&>
meta_operator ::=
    = | ? | @
```

*Syntax 5—Operator*

When appearing in a syntax rule, an operator shall be used within a statement or within an expression. An operator with one operand shall be called *unary operator*. An unary operator shall precede the operand. An operator with two operands shall be called *binary operator*. A binary operator shall succeed the first operand and precede the second operand.

### 6.4.1 Arithmetic operator

Table 15 shows the list of arithmetic operators and their names used in this standard.

**Table 15—List arithmetic operators**

| Symbol | Operator name | Unary / binary | Section |
|---|---|---|---|
| + | Plus | Binary | See 10.10.2. |
| - | Minus | Both | See 10.10.2. |
| * | Multiply | Binary | See 10.10.2. |
| / | Divide | Binary | See 10.10.2. |
| % | Modulo | Binary | See 10.10.2. |
| ** | Power | Binary | See 11.1. |

Arithmetic operators shall be used to specify arithmetic operations.

### 6.4.2 Boolean operator

Table 16 shows the list of boolean operators and their names used in this standard.

**Table 16—List of boolean operators**

| Symbol | Operator name | Unary / binary | Section |
|---|---|---|---|
| ! | Logical inversion | Unary | See 10.10.1. |
| && | Logical and | Binary | See 10.10.1. |
| \|\| | Logical or | Binary | See 10.10.1. |
| ~ | bit-wise inversion | Unary | **See 10.12.?? |
| & | bit-wise and | Both | **See 10.12.?? |
| ~& | bit-wise nand | Both | **See 10.12.?? |
| \| | bit-wise or | Both | **See 10.12.?? |
| ~\| | bit-wise nor | Both | **See 10.12.?? |
| ^ | Exclusive or | Both | **See 10.12.?? |
| ~^ | Exclusive nor | Both | **See 10.12.?? |

Boolean operators shall be used to specify boolean operations.

### 6.4.3 Relational operator

Table 17 shows the list of relational operators and their names used in this standard.

**Table 17—List of relational operators**

| Symbol | Operator name | Unary / binary | Section |
|---|---|---|---|
| == | Equal | Binary | See 10.10.2. |
| != | Not equal | Binary | See 10.10.2. |
| > | Greater | Binary | See 10.10.2. |
| < | Lesser | Binary | See 10.10.2. |
| >= | Greater or equal | Binary | See 10.10.2. |
| <= | Lesser or equal | Binary | See 10.10.2. |

Relational operators shall be used to specify mathematical relationships between numerical quantities.

### 6.4.4 Shift operator

Table 18 shows the list of shift operators and their names used in this standard.

**Table 18—List of shift operators**

| Symbol | Operator name | Unary / binary | Section |
|---|---|---|---|
| << | Shift left | Binary | See 10.10.2. |
| >> | Shift right | Binary | See 10.10.2. |

Shift operators shall be used to specify manipulations of discrete mathematical values.

### 6.4.5 Event sequence operator

Table 19 shows the list of event sequence operators and their names used in this standard.

**Table 19—List of event sequence operators**

| Symbol | Operator name | Unary / binary | Section |
|---|---|---|---|
| -> | Immediately followed by | Binary | See 10.12.2. |
| ~> | Eventually followed by | Binary | See 10.12.2. |
| <-> | Immediately following each other | Binary | See 10.12.3. |
| <~> | Eventually following each other | Binary | **where?? |
| &> | Simultaneous or immediately followed by | Binary | See 10.12.3. |
| <&> | Simultaneous or immediately following each other | Binary | See 10.12.3. |

Event sequence operators shall be used to express temporal relationships between discrete events.

### 6.4.6 Meta operator

Table 20 shows the list of meta operators and their names used in this standard.

**Table 20—List of meta operators**

| Symbol | Operator name | Unary / binary | Section |
|---|---|---|---|
| = | Assignment | Binary | See 7.10, 8.9, 10.4. |
| ? | Condition | Binary | See 10.10.1. |
| @ | Control | Unary | See 10.12.6. |

Meta operators shall be used to specify transactions between variables.

## 6.5 Number

*Numbers* shall be divided into subcategories *signed integer*, *signed real*, *unsigned integer*, and *unsigned real*. Furthermore, the categories *signed number*, *unsigned number*, *integer* and *real* shall be defined as shown in Syntax 6.

```
number ::=
    signed_integer | signed_real | unsigned_integer | unsigned_real
signed_number ::=
    signed_integer | signed_real
unsigned_number ::=
    unsigned_integer | unsigned_real
integer ::=
    signed_integer | unsigned_integer
signed_integer ::=
    sign unsigned_integer
unsigned_integer ::=
    digit { [ _ ] digit }
real ::=
    signed_real | unsigned_real
signed_real ::=
    sign unsigned_real
unsigned_real ::=
      mantisse [ exponent ]
    | unsigned_integer exponent
sign ::=
    + | -
mantisse ::=
      . unsigned_integer
    | unsigned_integer . [ unsigned_integer ]
exponent ::=
      E [ sign ] unsigned_integer
    | e [ sign ] unsigned_integer
```

*Syntax 6—Numbers*

Numbers shall be used to represent numerical quantities.

## 6.6 Quantity symbol

A *quantity symbol* shall be defined as shown in Syntax 7.

```
quantity_symbol ::=
      unity { letter } | K { letter } | M E G { letter } | G { letter }
    | M { letter } | U { letter } | N { letter } | P { letter } | F { letter }
unity ::=
    1
K ::=
    K | k
M ::=
    M | m
E ::=
    E | e
G ::=
    G | g
U ::=
    U | u
N ::=
    N | n
P ::=
    P | p
F ::=
    F | f
```

*Syntax 7—Quantity symbol*

The meaning of the quantity symbol is shown in Table 21.

**Table 21—Quantity symbol and corresponding SI-prefix**

| Leading character | SI-prefix (symbol) | SI-prefix (word) | Numerical value |
|---|---|---|---|
| F | f | femto | 1e-15 |
| P | p | pico | 1e-12 |
| N | n | nano | 1e-9 |
| U | µ | micro | 1e-6 |
| M | m | milli | 1e-3 |
| unity | 1 | one | 1 |
| K | k | kilo | 1e+3 |
| MEG | M | mega | 1e+6 |
| G | G | giga | 1e+9 |

A quantity symbol can be used to define a *quantity value* (see Section 7.2).

## 6.7 Bit literal

*Bit literals* shall be divided into the subcategories *alphanumeric bit literal* and *symbolic bit literal*, as shown in Syntax 8.

Bit literals shall be used to specify scalar values within a boolean value system (see Section 10.10).

```
bit_literal ::=
      alphanumeric_bit_literal
    | symbolic_bit_literal
alphanumeric_bit_literal
      numeric_bit_literal
    | alphabetic_bit_literal
numeric_bit_literal ::=
    0 | 1
alphabetic_bit_literal ::=
      X | Z | L | H | U | W
    | x | z | l | h | u | w
symbolic_bit_literal ::=
      ? | *
```

*Syntax 8—Bit literal*

## 6.8 Based literal

*Based literals* shall be divided into subcategories *binary based literal*, *octal based literal*, *decimal based literal*, and *hexadecimal based literal*, as shown in Syntax 9.

```
based_literal ::=
    binary_based_literal | octal_based_literal | decimal_based_literal | hexadecimal_based_literal
binary_based_literal ::=
    binary_base bit_literal { [ _ ] bit_literal }
binary_base ::=
    'B | 'b
octal_based_literal ::=
     octal_base octal_digit { [ _ ] octal_digit }
octal_base ::=
    'O | 'o
octal_digit ::=
    bit_literal | 2 | 3 | 4 | 5 | 6 | 7
decimal_based_literal ::=
     decimal_base digit { [ _ ] digit }
decimal_base ::=
    'D | 'd
hexadecimal_based_literal ::=
     hexadecimal_base hexadecimal_digit { [ _ ] hexadecimal_digit }
hexadecimal_base ::=
    'H | 'h
hexadecimal_digit ::=
      octal_digit | 8 | 9
    | A | B | C | D | E | F
    | a | b | c | d | e | f
```

*Syntax 9—Based literal*

Based literals shall be used to specify vectorized values within a boolean value system.

## 6.9 Edge literal

*Edge literals* shall be divided into subcategories *bit edge literal*, *based edge literal*, and *symbolic edge literal*, as shown in Syntax 10.

```
edge_literal ::=
      bit_edge_literal
    | based_edge_literal
    | symbolic_edge_literal
bit_edge_literal ::=
    bit_literal  bit_literal
based_edge_literal ::=
    based_literal  based_literal
symbolic_edge_literal ::=
    ?~ | ?! | ?-
```

*Syntax 10—Edge literal*

Edge literals shall be used to specify a change of value within a boolean system. In general, bit edge literals shall specify a change of a scalar value, based edge literals shall specify a change of a vectorized value, and symbolic edge literals shall specify a change of a scalar or of a vectorized value.

## 6.10 Quoted string

A *quoted string* shall be a sequence of zero or more characters enclosed between two double quote characters, as shown in Syntax 11.

```
quoted_string ::=
      " { character } "
```

*Syntax 11—Quoted string*

Within a quoted string, a sequence of characters starting with an *escape character* shall represent a symbol for another character, as shown in Table 22.

**Table 22—Character symbols within a quoted string**

| Symbol | Character | ASCII Code (octal) |
|---|---|---|
| \g | Alert or bell. | 007 |
| \h | Backspace. | 010 |
| \t | Horizontal tab. | 011 |
| \n | New line. | 012 |
| \v | Vertical tab. | 013 |
| \f | Form feed. | 014 |
| \r | Carriage return. | 015 |
| \" | Double quote. | 042 |
| \\ | Backslash. | 134 |
| \ digit digit digit | ASCII character represented by three digit octal ASCII code. | digit digit digit |

The start of a quoted string shall be determined by a double quote character. The end of a quoted string shall be determined by a double quote character preceded by an even number of escape characters or by any other character than escape character.

## 6.11 Identifier

*Identifiers* shall be divided into the subcategories *non-escaped identifier*, *escaped identifier*, *placeholder identifier*, and *hierarchical identifier*, as shown in Syntax 12.

```
identifier ::=
      non_escaped_identifier
    | escaped_identifier
    | placeholder_identifier
    | hierarchical_identifier
```

*Syntax 12—Identifier*

Identifiers shall be used to specify a name of an ALF statement or a value of an ALF statement. Identifiers can also appear in an arithmetic expression, in a boolean expression, or in a vector expression, referencing an already defined statement by name.

A lowercase character used within a keyword or within an identifier shall be considered equivalent to the corresponding uppercase character. This makes ALF case-insensitive. However, wherever an identifier is used to specify the name of a statement, the usage of the exact letters shall be preserved by the parser to enable usage of the same name by a case-sensitive application.

### 6.11.1 Non-escaped identifier

A *non-escaped identifier* shall be defined as shown in Syntax 13.

```
non_escaped_identifier ::=
    letter { letter | digit | _ | $ | # }
```

*Syntax 13—Non-escaped identifier*

A non-escaped identifier shall be used, when there is no lexical conflict, i.e., no appearance of a character with special meaning, and no semantic conflict, i.e., the identifier is not used elsewhere as a keyword.

### 6.11.2 Escaped identifier

An *escaped identifier* shall be defined as shown in Syntax 14.

```
escaped_identifier ::=
    \ escapable_character { escapable_character }
escapable_character ::=
    letter | digit | special
```

*Syntax 14—Escaped identifier*

An escaped identifier shall be used, when there is a lexical conflict, i.e., an appearance of a character with special meaning, or a semantic conflict, i.e., the identifier is used elsewhere as a keyword.

### 6.11.3 Placeholder identifier

A *placeholder identifier* shall be defined as a non-escaped identifier enclosed by angular brackets without whitespace, as shown in Syntax 15.

```
placeholder_identifier ::=
    < non_escaped_identifier >
```

*Syntax 15—Placeholder identifier*

A placeholder identifier shall be used to represent a formal parameter in a *template* statement (see 8.8), which is to be replaced by an actual parameter in a *template instantiation* statement (see 8.9).

### 6.11.4 Hierarchical identifier

A *hierarchical identifier* shall be defined as shown in Syntax 16.

```
hierarchical_identifier ::=
    identifier [ \ ] . identifier
```

*Syntax 16—Hierarchical identifier*

A hierarchical identifier shall be used to specify a hierarchical name of a statement, i.e., the name of a child preceded by the name of its parent. A dot within a hierarchical identifier shall be used to separate a parent from a child, unless the dot is directly preceded by an escape character.

*Example*

**\id1.id2.\id3** is a hierarchical identifier, where **id2** is a child of **\id1**, and **\id3** is a child of **id2**.

**id1\.id2.\id3** is a hierarchical identifier, where **\id3** is a child of "**id1.id2**".

**id1\.id2\.id3** specifies the pseudo-hierarchical name "**id1.id2.id3**".

## 6.12 Keyword

*Keywords* shall be lexically equivalent to non-escaped identifiers. Predefined keywords are listed in Table 3 — Table 6 and Table 10 — Table 12. Additional keywords are predefined in 8.4.

The predefined keywords in this standard shall follow a more restrictive lexical rule than general non-escaped identifiers, as shown in Syntax 17.

```
keyword_identifier ::=
    letter { [ _ ] letter }
```

*Syntax 17—Keyword*

The reason for the more restrictive lexical rule is to encourage the use of words taken from a natural language as keywords. Words in a natural language are constructed from lexical characters only, not from numbers. The underscore can be used to indicate that there would be a whitespace or a dash in the word from the natural language.

NOTE—This document presents keywords in all-uppercase letters for clarity.

## 6.13 Rules for whitespace usage

Whitespace shall be used to separate lexical tokens from each other, according to the following rules:

a) Whitespace before and after a *delimiter* shall be optional.
b) Whitespace before and after an *operator* shall be optional.
c) Whitespace before and after a *quoted string* shall be optional.
d) Whitespace before and after a *comment* shall be mandatory. This rule shall override a), b), and c).
e) Whitespace between subsequent quoted strings shall be mandatory. This rule shall override c).
f) Whitespace between subsequent lexical tokens amongst the categories *number*, *bit literal*, *based literal*, and *identifier* shall be mandatory.
g) Whitespace before and after a *placeholder identifier* shall be mandatory. This rule shall override a), b), and c).
h) Whitespace after an *escaped identifier* shall be mandatory. This rule shall override a), b), and c).
i) Either whitespace or delimiter before a *signed number* shall be mandatory. This rule shall override a), b), and c).
j) Either whitespace or delimiter before a *symbolic edge literal* shall be mandatory. This rule shall override a), b), and c).

Whitespace before the first lexical token or after the last lexical token in a file shall be optional. Hence in all rules prescribing mandatory whitespace, “before” shall not apply for the first lexical token in a file, and “after” shall not apply for the last lexical token in a file.

## 6.14 Rules against parser ambiguity

In a syntax rule where multiple legal interpretations of a lexical token are possible, the resulting ambiguity shall be resolved according to the following rules:

a) In a context where both *bit literal* and *identifier* are legal, a *non-escaped identifier* shall take priority over a *symbolic bit literal*.
b) In a context where both *bit literal* and *number* are legal, an *unsigned integer* shall take priority over a *numeric bit literal*.
c) In a context where both *edge literal* and *identifier* are legal, a *non-escaped identifier* shall take priority over a *bit edge literal*.
d) In a context where both *edge literal* and *number* are legal, an *unsigned integer* shall take priority over a *bit edge literal*.

If the interpretation as *bit literal* is desired in case a) or b), a *based literal* can be substituted for a *bit literal*.

If the interpretation as *edge literal* is desired in case c) or d), a *based edge literal* can be substituted for a *bit edge literal*.

# 7. Auxiliary syntax rules

This section specifies auxiliary syntax rules which are used to build other syntax rules.

## 7.1 All-purpose value

An *all-purpose value* shall be defined as shown in Syntax 18.

```
all_purpose_value ::=
      number
    | identifier
    | quoted_string
    | bit_literal
    | based_literal
    | edge_value
    | pin_variable
    | control_expression
```

*Syntax 18—All purpose value*

## 7.2 Quantity value

A *quantity value* shall be defined as shown in Syntax 19.

```
quantity_value ::=
    unsigned_number | quantity_symbol
```

*Syntax 19—Quantity value*

Only the leading characters of the quantity symbol shall be used for identification of a quantity value, as specified in Table 21.

Optional subsequent letters can be used to make the quantity symbol more readable. For example, “pF” can be used to denote “picofarad” etc.

## 7.3 String value

A *string value* shall be defined as shown in Syntax 20.

```
string ::=
    quoted_string | identifier
```

*Syntax 20—String value*

A string value shall represent textual data in general and the name of a referenced object in particular.

## 7.4 Arithmetic value

An *arithmetic value* shall be defined as shown in Syntax 21.

```
arithmetic_value ::=
    number | identifier | bit_literal | based_literal
```

*Syntax 21—Arithmetic value*

An arithmetic value shall represent data for an arithmetic model or for an arithmetic assignment. Semantic restrictions apply, depending on the particular type of arithmetic model.

## 7.5 Boolean value

A *boolean value* shall be defined as shown in Syntax 22.

```
boolean_value ::=
    alphanumeric_bit_literal | based_literal | integer
```

*Syntax 22—Boolean value*

A boolean value shall represent the contents of a pin variable (see 7.9).

## 7.6 Edge value

An *edge value* shall be defined as shown in Syntax 23.

```
edge_value ::=
    ( edge_literal )
```

*Syntax 23—Edge value*

An edge value shall represent a standalone edge literal that is not embedded in a vector expression.

## 7.7 Index value

An *index value* shall be defined as shown in Syntax 24.

```
index_value ::=
    unsigned_integer | identifier
```

*Syntax 24—Index value*

An index value shall represent a particular position within a *vector pin* (see 9.7). The usage of identifier shall only be allowed, if that identifier represents a *constant* (see 8.2) with a value of the category unsigned integer.

## 7.8 Index

An *index* shall be defined as shown in Syntax 25.

An index shall be used in conjunction with the name of a pin or a pingroup. A *single index* shall represent a particular scalar within a one-dimensional vector or a particular one-dimensional vector within a two-dimensional matrix. A *multi index* shall represent a range of scalars or a range of vectors, wherein the most significant bit (MSB) is specified by the left index value and the least significant bit (LSB) is specified by the right index value.

```
index ::=
    single_index | multi_index
single_index ::=
    [ index_value ]
multi_index ::=
    [ index_value : index_value ]
```

*Syntax 25—Index*

## 7.9 Pin variable and pin value

A *pin variable* and a *pin value* shall be defined as shown in Syntax 26.

```
pin_variable ::=
    pin_variable_identifier [ index ]
pin_value ::=
    pin_variable | boolean_value
```

*Syntax 26—Pin variable*

A pin variable shall represent the name of a pin or the name of a pin group, in conjunction with an optional index.

A pin value shall represent the actual value or a pointer to the actual value associated with a pin variable. The actual value is a boolean value (see Section 7.5). A pin variable represents a pointer to the actual value.

## 7.10 Pin assignment

A *pin assignment* shall be defined as shown in Syntax 27.

```
pin_assignment ::=
    pin_variable = pin_value ;
```

*Syntax 27—Pin assignment*

A pin assignment represents an association between a pin variable and a pin value.

The datatype of the left hand side (LHS) and the right hand side (RHS) of the assignment shall be compatible with each other. The following rules shall apply:

a) The bitwidth of the RHS must be equal to the bitwidth of the LHS.
b) A scalar pin at the LHS can be assigned a bit literal or a based literal representing a single bit.
c) A pin group, a vector pin, or a one-dimensional slice of a matrix pin at the LHS can be assigned a based literal or an unsigned integer, representing a binary number.

## 7.11 Annotation

An *annotation* shall be divided into the subcategories *single value annotation* and *multi value annotation*, as shown in Syntax 28

An annotation shall represent an association between an identifier and a set of *annotation values* (*values* for shortness). In case of a single value annotation, only one value shall be legal. In case of a multi value annotation,

```
annotation ::=
      single_value_annotation
    | multi_value_annotation
single_value_annotation ::=
    annotation_identifier = annotation_value ;
annotation_value ::=
      number
    | identifier
    | quoted_string
    | bit_literal
    | based_literal
    | edge_value
    | pin_variable
    | control_expression
    | boolean_expression
    | arithmetic_expression
multi_value_annotation ::=
    annotation_identifier { annotation_value { annotation_value } }
```

*Syntax 28—Annotation*

one or more values shall be legal. The annotation shall serve as a semantic qualifier of its parent statement. The value shall be subject to semantic restrictions, depending on the identifier.

The annotation identifier can be a keyword used for the declaration of an object (i.e., a generic object or a library-specific object). An annotation using such an annotation identifier shall be called a *reference annotation*. The annotation value of a reference annotation shall be the name of an object of matching type. A reference annotation can be a single-value annotation or a multi-value annotation. The semantic meaning of a reference annotation shall be defined in the context of its parent statement.

## 7.12 Annotation container

An *annotation container* shall be defined as shown in Syntax 29

```
annotation_container ::=
    annotation_container_identifier { annotation { annotation } }
```

*Syntax 29—Annotation container*

An annotation container shall represent a collection of annotations. The annotation container shall serve as a semantic qualifier of its parent statement. The annotation container identifier shall be a keyword. An annotation within an annotation container shall be subject to semantic restrictions, depending on the annotation container identifier.

## 7.13 ATTRIBUTE statement

An *attribute* statement shall be defined as shown in Syntax 30.

```
attribute ::=
    ATTRIBUTE { identifier { identifier } }
```

*Syntax 30—ATTRIBUTE statement*

The attribute statement shall be used to associate arbitrary identifiers with the parent of the attribute statement. Semantics of such identifiers can be defined depending on the parent of the attribute statement. The attribute statement has a similar syntax definition as a multi-value annotation (see 7.11). While a multi-value annotation can have restricted semantics and a restricted set of applicable values, identifiers with and without predefined semantics can co-exist within the same attribute statement.

*Example*

```
CELL myRAM8x128 {
    ATTRIBUTE { rom asynchronous static }
}
```

## 7.14 PROPERTY statement

A *property* statement shall be defined as shown in Syntax 31.

property ::=
**PROPERTY** [ identifier ] **{** annotation { annotation } **}**

*Syntax 31—PROPERTY statement*

The property statement shall be used to associate arbitrary annotations with the parent of the property statement. The property statement has a similar syntax definition as an annotation container (see 7.12). While the keyword of an annotation container usually restricts the semantics and the set of applicable annotations, the keyword "property" does not. Annotations shall have no predefined semantics, when they appear within the property statement, even if annotation identifiers with otherwise defined semantics are used.

*Example*

```
PROPERTY myProperties {
    parameter1 = value1 ;
    parameter2 = value2 ;
    parameter3 { value3 value4 value5 }
}
```

## 7.15 INCLUDE statement

An *include* statement shall be defined as shown in Syntax 32.

include ::=
**INCLUDE** quoted_string **;**

*Syntax 32—INCLUDE statement*

The quoted string shall specify the name of a file. When the include statement is encountered during parsing of a file, the application shall parse the specified file and then continue parsing the former file. The format of the file containing the include statement and the format of the file specified by the include statement shall be the same.

*Example*

```
LIBRARY myLib {
    INCLUDE "templates.alf";
```

```
    INCLUDE "technology.alf";
    INCLUDE "primitives.alf";
    INCLUDE "wires.alf";
    INCLUDE "cells.alf";
}
```

Note: The filename specified by the quoted string shall be interpreted according to the rules of the application and/or the operating system. The ALF parser itself shall make no semantic interpretation of the filename.

## 7.16 ASSOCIATE statement

An *associate* statement shall be defined as shown in .

associate ::=
**ASSOCIATE** quoted_string **;**
| **ASSOCIATE** quoted_string **{** *FORMAT*_single_value_annotation **}**

*Syntax 33—ASSOCIATE statement*

## 7.17 REVISION statement

A *revision statement* shall be defined as shown in Syntax 34

revision ::=
**ALF_REVISION** string_value

*Syntax 34—Revision statement*

A revision statement shall be used to identify the revision or version of the file to be parsed. One, and only one, revision statement can appear at the beginning of an ALF file.

The set of legal string values within the revision statement shall be defined as shown in Table 23

**Table 23—Legal string values within the REVISION statement**

| String value | Revision or version |
|---|---|
| "1.1" | Version 1.1 by Open Verilog International (OVI), released on April 6, 1999. |
| "2.0" | Version 2.0 by Accellera, released on December 14, 2000. |
| "P1603.2002-06-21" | IEEE draft version as described in this document. |
| TBD | IEEE 1603 release version. |

The revision statement shall be optional, as the application program parsing the ALF file can provide other means of specifying the revision or version of the file to be parsed. If a revision statement is encountered while a revision has already been specified to the parser (e.g. if an included file is parsed), the parser shall be responsible

to decide whether the newly encountered revision is compatible with the originally specified revision and then either proceed assuming the original revision or abandon.

This document suggests, but does not certify, that the IEEE version of the ALF standard proposed herein be backward compatible with the Accellera version 2.0 and the OVI version 1.1.

## 7.18 Generic object

A *generic object* shall be defined as shown in Syntax 35.

```
generic_object ::=
      alias_declaration
    | constant_declaration
    | class_declaration
    | keyword_declaration
    | semantics_declaration
    | group_declaration
    | template_declaration
```

*Syntax 35—Generic object*

The syntax items introduced in Syntax 35 are defined in Section 8.

## 7.19 Library-specific object

A *library-specific object* shall be defined as shown in Syntax 36.

```
library_specific_object ::=
      library
    | sublibrary
    | cell
    | primitive
    | wire
    | pin
    | pingroup
    | vector
    | node
    | layer
    | via
    | rule
    | antenna
    | site
    | array
    | blockage
    | port
    | pattern
    | region
```

*Syntax 36—Library-specific object*

The syntax items introduced in Syntax 36 are defined in Section 9.

## 7.20 All purpose item

An *all purpose item* shall be defined as shown in Syntax 37.

```
all_purpose_item ::=
      generic_object
    | include_statement
    | associate_statement
    | annotation
    | annotation_container
    | arithmetic_model
    | arithmetic_model_container
    | all_purpose_item_template_instantiation
```

*Syntax 37—All purpose item*

The syntax items introduced in Syntax 37 are defined in this Section 7 , in Section 8 and in Section 11.

# 8. Generic objects and related statements

**Add lead-in text**

## 8.1 ALIAS declaration

An *alias* shall be declared as shown in Syntax 38.

```
alias_declaration ::=
    ALIAS alias_identifier = original_identifier ;
```

*Syntax 38—ALIAS declaration*

The alias declaration shall specify an identifier which can be used instead of an original identifier to specify a name or a value of an ALF statement. The identifier shall be semantically interpreted in the same way as the original identifier.

*Example*

```
ALIAS reset = clear;
```

## 8.2 CONSTANT declaration

A *constant* shall be declared as shown in Syntax 39.

```
constant_declaration ::=
    CONSTANT constant_identifier = constant_value ;
constant_value ::=
    number | based_literal
```

*Syntax 39—CONSTANT declaration*

The constant declaration shall specify an identifier which can be used instead of a *constant value*, i.e., a number or a based literal. The identifier shall be semantically interpreted in the same way as the constant value.

*Example*

```
CONSTANT vdd = 3.3;
CONSTANT opcode = 'h0f3a;
```

## 8.3 CLASS declaration

A *class* shall be declared as shown in Syntax 40.

```
class_declaration ::=
    CLASS class_identifier ;
  | CLASS class_identifier { { all_purpose_item } }
```

*Syntax 40—CLASS declaration*

A class declaration shall be used to establish a semantic association between ALF statements, including, but not restricted to, other class declarations. ALF statements shall be associated with each other, if they contain a reference to the same class. The semantics specified by an all purpose item within a class declaration shall be inherited by the statement containing the reference.

*Example*

```
CLASS \1stclass { ATTRIBUTE { everything } }
CLASS \2ndclass { ATTRIBUTE { nothing } }
CELL cell1 { CLASS = \1stclass; }
CELL cell2 { CLASS = \2ndclass; }
CELL cell3 { CLASS { \1stclass \2ndclass } }
// cell1 inherits “everything”
// cell2 inherits “nothing”
// cell3 inherits “everything” and “nothing”
```

## 8.4 KEYWORD declaration

A *keyword* shall be declared as shown in Syntax 41.

keyword_declaration ::=
**KEYWORD** *keyword*_identifier **=** *syntax_item*_identifier **;**
| **KEYWORD** *keyword*_identifier **=** *syntax_item*_identifier **{** { keyword_item } **}**
keyword_item ::=
*VALUETYPE*_single_value_annotation
| *VALUES*_multi_value_annotation
| *DEFAULT*_single_value_annotation
| *CONTEXT*_annotation

*Syntax 41—KEYWORD declaration*

A keyword declaration shall be used to define a new keyword in a category or in a subcategory of ALF statements specified by a *syntax item* identifier. One or more annotations (see 8.5) can be used to qualify the contents of the keyword declaration.

A legal syntax item identifier shall be defined as shown in Table 24.

**Table 24—Syntax item identifier**

| Identifier | Semantic meaning |
|---|---|
| `annotation` | The keyword shall specify an *annotation* (see 7.11). |
| `single_value_annotation` | The keyword shall specify a *single value annotation* (see 7.11). |
| `multi_value_annotation` | The keyword shall specify a *multi-value annotation* (see 7.11). |
| `annotation_container` | The keyword shall specify an *annotation container* (see 7.12). |
| `arithmetic_model` | The keyword shall specify an *arithmetic model* (see 11.2). |
| `arithmetic_submodel` | The keyword shall specify an *arithmetic submodel* (see 11.4.3). |

**Table 24—Syntax item identifier (Continued)**

| Identifier | Semantic meaning |
|---|---|
| `arithmetic_model_container` | The keyword shall specify an *arithmetic model container* (see 11.4.7). |

## 8.5 Annotations for a KEYWORD

This subsection defines annotations which can be used as legal children of a keyword declaration statement.

### 8.5.1 VALUETYPE annotation

The *valuetype* annotation shall be a *single value annotation*. The set of legal values shall depend on the syntax item identifier associated with the keyword declaration, as shown in Table 25.

**Table 25—VALUETYPE annotation**

| Syntax item identifier | Set of legal values for VALUETYPE | Default value for VALUETYPE | Comment |
|---|---|---|---|
| `annotation` or `single_value_annotation` or `multi_value_annotation` | `number`, `identifier`, `quoted_string`, `edge_value`, `pin_variable`, `control_expression`, `boolean_expression`, `arithmetic_expression`. | `identifier` | See Syntax 28, definition of *annotation value.* |
| `annotation_container` | N/A | N/A | An *annotation container* (see Syntax 29) has no value. |
| `arithmetic_model` | `number`, `identifier`, `bit_literal`, `based_literal`. | `number` | See Syntax 21, definition of *arithmetic value.* |
| `arithmetic_submodel` | N/A | N/A | An *arithmetic submodel* (see 11.4.3) shall always have the same `valuetype` as its parent arithmetic model. |
| `arithmetic_model_container` | N/A | N/A | An *arithmetic model container* (see 11.4.7) has no value. |

The valuetype annotation shall specify the category of legal ALF values applicable for an ALF statement whose ALF type is given by the declared keyword.

*Example:*

This example shows a correct and an incorrect usage of a declared keyword with specified valuetype.

```
KEYWORD Greeting = annotation { VALUETYPE = identifier ; }
CELL cell1 { Greeting = HiThere ; } // correct
CELL cell2 { Greeting = “Hi There” ; } // incorrect
```

The first usage is correct, since `HiThere` is an identifier. The second usage is incorrect, since `“Hi There”` is a quoted string and not an identifier.

### 8.5.2 VALUES annotation

The *values* annotation shall be a *multi value annotation* applicable in the case where the *valuetype* annotation is also applicable.

The *values* annotation shall specify a discrete set of legal values applicable for an ALF statement using the declared keyword. Compatibility between the *values* annotation and the *valuetype* annotation shall be mandatory.

*Example:*

This example shows a correct and an incorrect usage of a declared keyword with specified valuetype and values.

```
KEYWORD Greeting = annotation {
    VALUETYPE = identifier ;
    VALUES { HiThere Hello HowDoYouDo }
}
CELL cell3 { Greeting = Hello ; } // correct
CELL cell4 { Greeting = GoodBye ; } // incorrect
```

The first usage is correct, since `Hello` is contained within the set of values. The second usage is incorrect, since `GoodBye` is not contained within the set of values.

### 8.5.3 DEFAULT annotation

The *default* annotation shall be a *single value annotation* applicable in the case where the valuetype annotation is also applicable. Compatibility between the *default* annotation, the *valuetype* annotation, and the *values* annotation shall be mandatory.

The default annotation shall specify a presumed value in absence of an ALF statement specifying a value.

*Example:*

```
KEYWORD Greeting = annotation {
    VALUETYPE = identifier ;
    VALUES { HiThere Hello HowDoYouDo }
    DEFAULT = Hello ;
}
CELL cell5 { /* no Greeting */ }
```

In this example, the absence of a `Greeting` statement is equivalent to the following:

```
CELL cell5 { Greeting = Hello ; }
```

### 8.5.4 CONTEXT annotation

The *context* annotation shall specify the ALF type of a legal parent of the statement using the declared keyword. The ALF type of a legal parent can be a predefined keyword or a declared keyword.

*Example:*

```
KEYWORD LibraryQualifier = annotation { CONTEXT { LIBRARY SUBLIBRARY } }
KEYWORD CellQualifier = annotation { CONTEXT = CELL ; }
KEYWORD PinQualifier = annotation { CONTEXT = PIN ; }
LIBRARY library1 {
    LibraryQualifier = foo ; // correct
    CELL cell1 {
    CellQualifier = bar ; // correct
    PinQualifier = foobar ; // incorrect
    }
}
```

The following change would legalize the example above:

```
KEYWORD PinQualifier = annotation { CONTEXT { PIN CELL } }
```

### 8.5.5 SI_MODEL annotation

** see IEEE proposal, June 2002, chapter 27**

## 8.6 SEMANTICS declaration

*Semantics* shall be declared as shown in Syntax 42—.

semanics_declaration ::=
**SEMANTICS** *semantics*_identifier **=** *syntax_item*_identifier **;**
| **SEMANTICS** *semantics*_identifier [ **=** *syntax_item*_identifier ] **{** { semantics_item } **}**
semantics_item ::=
*VALUES*_multi_value_annotation
| *DEFAULT*_single_value_annotation
| *CONTEXT*_annotation

*Syntax 42—SEMANTICS declaration*

A semantics declaration shall be used to define context-specific rules in a category or in a subcategory of ALF statements. The *semantics item identifier* shall make reference to a legal ALF statement or to a category or subcategory of legal ALF statements.

The semantics identifier shall be a keyword identifier or a syntax item identifier or a hierarchical identifier. In the latter case, the hierarchical identifier shall involve one or more keyword identifiers and/or syntax item identifiers.

If the ALF type of the referenced ALF statement is `annotation`, the optional syntax item identifier `single_value_annotation` or `multi_value_annotation` can be used.

A *semantic item* can be used to qualify the contents of the semantics declaration. Legal semantic items include *values* annotation (see 8.5.2), *default* annotation (see 8.5.3) and *context* annotation (see 8.5.4).

A rule specified by a *semantic item* shall be compatible with the set of rules specified for the referenced ALF statement. A rule specified within a semantics declaration can not invalidate a rule specified within the referenced ALF statement.

*Example:*

```
KEYWORD myAnnotation = annotation {
    VALUETYPE = identifier ;
    VALUES { value1 value2 value3 value4 value5 }
    CONTEXT { CELL PIN }
}
SEMANTICS CELL.myAnnotation = multi_value_annotation {
    VALUES { value1 value2 value3 }
}
SEMANTICS PIN.myAnnotation = single_value_annotation {
    VALUES { value4 value5 }
    DEFAULT = value4;
}
CELL myCell {
    myAnnotation { value1 value2 }
    PIN myPin {
    myAnnotation = value5;
    }
}
```

## 8.7 GROUP declaration

A *group* shall be declared as shown in Syntax 43.

group_declaration ::=
**GROUP** *group*_identifier **{** all_purpose_value { all_purpose_value } **}**
| **GROUP** *group*_identifier **{** *left*_index_value **:** *right*_index_value **}**

*Syntax 43—GROUP declaration*

A group declaration shall be used to specify the semantic equivalent of multiple similar ALF statements within a single ALF statement. An ALF statement containing a group identifier shall be semantically replicated by substituting each *group value* for the *group identifier*, or, by substituting subsequent index values bound by the left index value and by the right index value for the group identifier. The ALF parser shall verify whether each substitution results in a legal statement.

The ALF statement which has the same parent as the group declaration shall be semantically replicated, if the group identifier is found within the statement itself or within a child of the statement or within a child of a child of the statement etc. If the group identifier is found more than once within the statement or within its children, the same group value or index value per replication shall be substituted for the group identifier, but no additional replication shall occur.

The group identifier (i.e., the name associated with the group declaration) can be re-used as name of another statement. As a consequence, the other statement shall be interpreted as multiple statements wherein the group identifier within each replication shall be replaced by the all-purpose value. On the other hand, no name of any visible statement shall be allowed to be re-used as group identifier.

*Examples*

The following example shows substitution involving group values.

```
// statement using GROUP:
CELL myCell {
    GROUP data { data1 data2 data3 }
    PIN data { DIRECTION = input ; }
}
// semantically equivalent statement:
CELL myCell {
    PIN data1 { DIRECTION = input ; }
    PIN data2 { DIRECTION = input ; }
    PIN data3 { DIRECTION = input ; }
}
```

The following example shows substitution involving index values.

```
// statement using GROUP:
CELL myCell {
    GROUP dataIndex { 1 : 3 }
    PIN [1:3] data { DIRECTION = input ; }
    PIN clock { DIRECTION = input ; }
    SETUP = 0.5 { FROM { PIN = data[dataIndex]; } TO { PIN = clock ; } }
}
// semantically equivalent statement:
CELL myCell {
    GROUP dataIndex { 1 : 3 }
    PIN [1:3] data { DIRECTION = input ; }
    PIN clock { DIRECTION = input ; }
    SETUP = 0.5 { FROM { PIN = data[1]; } TO { PIN = clock ; } }
    SETUP = 0.5 { FROM { PIN = data[2]; } TO { PIN = clock ; } }
    SETUP = 0.5 { FROM { PIN = data[3]; } TO { PIN = clock ; } }
}
```

The following example shows multiple occurrences of the same group identifier within a statement.

```
// statement using GROUP:
CELL myCell {
    GROUP dataIndex { 1 : 3 }
    PIN [1:3] Din { DIRECTION = input ; }
    PIN [1:3] Dout { DIRECTION = input ; }
    DELAY = 1.0 { FROM {PIN=Din[dataIndex];} TO {PIN=Dout[dataIndex];} }
}
// semantically equivalent statement:
CELL myCell {
    GROUP dataIndex { 1 : 3 }
    PIN [1:3] Din { DIRECTION = input ; }
    PIN [1:3] Dout { DIRECTION = input ; }
    DELAY = 1.0 { FROM {PIN=Din[1];} TO {PIN=Dout[1];} }
    DELAY = 1.0 { FROM {PIN=Din[2];} TO {PIN=Dout[2];} }
    DELAY = 1.0 { FROM {PIN=Din[3];} TO {PIN=Dout[3];} }
}
```

## 8.8 TEMPLATE declaration

A *template* shall be declared as shown in Syntax 44.

```
template_declaration ::=
    TEMPLATE template_identifier { ALF_statement { ALF_statement } }
```

*Syntax 44—TEMPLATE declaration*

A template declaration shall be used to specify one or more ALF statements with variable contents that can be used many times. A template instantiation (see 8.9) shall specify the usage of such an ALF statement. Within the template declaration, the variable contents shall be specified by a placeholder identifier (see 6.11.3).

## 8.9 TEMPLATE instantiation

A *template* shall be instantiated in form of a *static template instantiation* or a *dynamic template instantiation*, as shown in Syntax 45

```
template_instantiation ::=
      static_template_instantiation
    | dynamic_template_instantiation
static_template_instantiation ::=
      template_identifier [ = STATIC ] ;
    | template_identifier [ = STATIC ] { { all_purpose_value } }
    | template_identifier [ = STATIC ] { { annotation } }
dynamic_template_instantiation ::=
    template_identifier = DYNAMIC { { dynamic_template_instantiation_item } }
dynamic_template_instantiation_item ::=
      annotation
    | arithmetic_model
    | arithmetic_assignment
arithmetic_assignment ::=
      identifier = arithmetic_expression ;
```

*Syntax 45—TEMPLATE instantiation*

A template instantiation shall be semantically equivalent to the ALF statement or the ALF statements found within the template declaration, after replacing the placeholder identifiers with replacement values. A static template instantiation shall support replacement by order, using one or more all-purpose values, or alternatively, replacement by reference, using one or more annotations (see 7.11). A dynamic template instantiation shall support replacement by reference only, using one or more annotations and/or one or more arithmetic models (see 7.11 and 11.2).

In the case of replacement by reference, the reference shall be established by a non-escaped identifier matching the placeholder identifier when the angular brackets are removed. The matching shall be case-insensitive.

The following rules shall apply:

a) A static template instantiation shall be used when the replacement value of any placeholder identifier can be determined during compilation of the library. Only a matching identifier shall be considered a legal annotation identifier. Each occurrence of the placeholder identifier shall be replaced by the annotation value associated with the annotation identifier.
b) A dynamic template instantiation shall be used when the replacement value of at least one placeholder identifier can only determined during runtime of the application. Only a matching identifier shall be considered a legal annotation identifier, or alternatively, a arithmetic model identifier, or alternatively, a legal arithmetic value.

c) Multiple replacement values within a multi-value annotation shall be legal if and only if the syntax rules for the ALF statement within the template declaration allow substitution of multiple values for one placeholder identifier.

d) In the case replacement by order, subsequently occurring placeholder identifiers in the template declaration shall be replaced by subsequently occurring all-purpose values in the template instantiation. If a placeholder identifier occurs more than once within the template declaration, all occurrences of that placeholder identifier shall be immediately replaced by the same all-purpose value. The first amongst the remaining placeholder identifiers shall then be considered the next placeholder to be replaced by the next all-purpose value.

e) A static template instantiation for which a placeholder identifier is not replaced shall be legal if and only if the semantic rules for the ALF statement support a placeholder identifier outside a template declaration. However, the semantics of a placeholder identifier as an item to be substituted shall only apply within the template declaration statement.

*Examples*

The following example illustrates rule a).

```
// statement using TEMPLATE declaration and instantiation:
TEMPLATE someAnnotations {
    KEYWORD <oneAnnotation> = single_value_annotation ;
    KEYWORD annotation2 = single_value_annotation ;
    <oneAnnotation> = value1 ;
    annotation2 = <anotherValue> ;
}
someAnnotations {
    oneAnnotation = annotation1 ;
    anotherValue = value2 ;
}
// semantically equivalent statement:
KEYWORD annotation1 = single_value_annotation ;
KEYWORD annotation2 = single_value_annotation ;
annotation1 = value1 ;
annotation2 = value2 ;
```

The following example illustrates rule b).

```
// statement using TEMPLATE declaration and instantiation:
TEMPLATE someNumbers {
    KEYWORD N1 = single_value_annotation { VALUETYPE=number ; }
    KEYWORD N2 = single_value_annotation { VALUETYPE=number ; }
    N1 = <number1> ;
    N2 = <number2> ;
}
someNumbers = DYNAMIC {
    number2 = number1 + 1;
}
// semantically equivalent statement, assuming number1=3 at runtime:
N1 = 3 ;
N2 = 4 ;
```

The following example illustrates rule c).

```
TEMPLATE moreAnnotations {
    KEYWORD annotation3 = annotation ;
    KEYWORD annotation4 = annotation ;
    annotation3 { <someValue> }
    annotation4 = <yetAnotherValue> ;
}
moreAnnotations {
    someValue { value1 value2 }
    yetAnotherValue = value3 ;
}
// semantically equivalent statement:
KEYWORD annotation3 = annotation ;
KEYWORD annotation4 = annotation ;
annotation3 { value1 value2 }
annotation4 = value3 ;
```

The following example illustrates rule e).

```
TEMPLATE evenMoreAnnotations {
    KEYWORD <thisAnnotation> = single_value_annotation ;
    KEYWORD <thatAnnotation> = single_value_annotation ;
    <thatAnnotation> = <thisValue> ;
    <thisAnnotation> = <thatValue> ;
}
// template instantiation by reference:
evenMoreAnnotations = STATIC {
    thatAnnotation = day ;
    thisAnnotation = month;
    thatValue = April;
    thisValue = Monday;
}
// semantically equivalent template instantiation by order:
evenMoreAnnotations = STATIC { day month Monday April }

// semantically equivalent statement:
KEYWORD day = single_value_annotation ;
KEYWORD month = single_value_annotation ;
month = April;
day = Monday;
```

The following example illustrates rule d).

```
// statement using TEMPLATE declaration and instantiation:
TEMPLATE encoreAnnotation {
    KEYWORD context1 = annotation_container;
    KEYWORD context2 = annotation_container;
    KEYWORD annotation5 = single_value_annotation {
    CONTEXT { context1 context2 }
    VALUES { <something> <nothing> }
    }
    context1 { annotation5 = <nothing> ; }
    context2 { annotation5 = <something> ; }
}
encoreAnnotation {
```

```
    something = everything ;
}
// semantically equivalent statement:
KEYWORD context1 = annotation_container;
KEYWORD context2 = annotation_container;
KEYWORD annotation5 = single_value_annotation {
    CONTEXT { context1 context2 }
    VALUES { everything <nothing> }
}
context1 { annotation5 = <nothing> ; }
context2 { annotation5 = all ; }
// Both everything (without brackets) and <nothing> (with brackets)
// are legal values for annotation5.
```

# 9. Library-specific objects and related statements

**Add lead-in text**

## 9.1 LIBRARY and SUBLIBRARY declaration

A *library* and a *sublibrary* shall be declared as shown in Syntax 46.

```
library ::=
      LIBRARY library_identifier ;
    | LIBRARY library_identifier { { library_item } }
    | library_template_instantiation
library_item ::=
      sublibrary
    | sublibrary_item
sublibrary ::=
      SUBLIBRARY sublibrary_identifier ;
    | SUBLIBRARY sublibrary_identifier { { sublibrary_item } }
    | sublibrary_template_instantiation
sublibrary_item ::=
      all_purpose_item
    | cell
    | primitive
    | wire
    | layer
    | via
    | rule
    | antenna
    | array
    | site
    | region
```

*Syntax 46—LIBRARY and SUBLIBRARY declaration*

A library shall serve as a repository of technology data for creation of an electronic integrated circuit. A sublibrary can optionally be used to create different scopes of visibility for particular statements describing technology data.

If any two objects of the same ALF type and the same ALF name appear in two libraries, or in two sublibraries with the same library as parents, their usage for creation of an electronic circuit shall be mutually exclusive. For example, two cells with the same name shall not be instantiated in the same integrated circuit. It shall be the responsibility of the application tool to detect and properly handle such cases, as the selection of a library or a sublibrary is controlled by the user of the application tool.

## 9.2 Annotations for LIBRARY and SUBLIBRARY

**Add lead-in text**

### 9.2.1 INFORMATION annotation container

**Single subheader**

An *information* annotation container shall be defined as shown in Semantics 1.

```
KEYWORD INFORMATION = annotation_container {
  CONTEXT { LIBRARY SUBLIBRARY CELL WIRE PRIMITIVE }
}
KEYWORD PRODUCT = single_value_annotation {
  VALUETYPE = string_value; DEFAULT = “”;
  CONTEXT = INFORMATION;
}
KEYWORD TITLE = single_value_annotation {
  VALUETYPE = string_value; DEFAULT = “”;
  CONTEXT = INFORMATION;
}
KEYWORD VERSION = single_value_annotation {
  VALUETYPE = string_value; DEFAULT = “”;
  CONTEXT = INFORMATION;
}
KEYWORD AUTHOR = single_value_annotation {
  VALUETYPE = string_value; DEFAULT = “”;
  CONTEXT = INFORMATION;
}
KEYWORD DATETIME = single_value_annotation {
  VALUETYPE = string_value; DEFAULT = “”;
  CONTEXT = INFORMATION;
}
```

*Semantics 1—INFORMATION statement*

The information annotation container shall be used to associate its parent statement with a product specification. The following semantic restrictions shall apply:

a) A library, a sublibrary, or a cell can be a legal parent of the information statement.
b) A wire, or a primitive can be a legal parent of the information statement, provided the parent of the wire or the primitive is a library or a sublibrary.

The semantics of the *information* contents are specified in Table 26.

**Table 26—Annotations within an INFORMATION statement**

| Annotation identifier | Semantics of annotation value |
|---|---|
| PRODUCT | A code name of a product described herein. |
| TITLE | A descriptive title of the product described herein. |
| VERSION | A version number of the product description. |
| AUTHOR | The name of a person or company generating this product description. |
| DATETIME | Date and time of day when this product description was created. |

The product developer shall be responsible for any rules concerning the format and detailed contents of the string value itself.

*Example*

```
LIBRARY myProduct {
    INFORMATION {
     PRODUCT = p10sc;
     TITLE = “0.10 standard cell”;
     VERSION = “v2.1.0”;
     AUTHOR = “Major Asic Vendor, Inc.”;
     DATETIME = “Mon Apr 8 18:33:12 PST 2002”;
    }
}
```

## 9.3 CELL declaration

A *cell* shall be declared as shown in Syntax 47.

```
cell ::=
      CELL cell_identifier ;
    | CELL cell_identifier { { cell_item } }
    | cell_template_instantiation
cell_item ::=
      all_purpose_item
    | pin
    | pingroup
    | primitive
    | function
    | non_scan_cell
    | test
    | vector
    | wire
    | blockage
    | artwork
    | pattern
    | region
```

*Syntax 47—CELL declaration*

A cell shall represent an electronic circuit which can be used as a building block for a larger electronic circuit.

## 9.4 CELL instantiation

A *cell* shall be instantiated as shown in Syntax 48.

```
named_cell_instantiation ::=
      cell_identifier instance_identifier ;
    | cell_identifier instance_identifier { pin_value { pin_value } }
    | cell_identifier instance_identifier { pin_assignment { pin_assignment } }
unnamed_cell_instantiation ::=
      cell_identifier { pin_value { pin_value } }
    | cell_identifier { pin_assignment { pin_assignment } }
```

*Syntax 48—CELL instantiation*

The purpose of a *named cell instantiation* is to describe a structural circuit or netlist in the context of a *structure* statement, where multiple instances of the same cell can appear (see Section 10.5).

The purpose of an *unnamed cell instantiation* is to establish a correspondence between a cell and another cell in the context of a non-scan cell statement (see Section 10.7).

The mapping between the reference cell and the cell instance can be established by order, using *pin value* (see Section 7.9), or by name, using *pin assignment* (see Section 7.10). The left-hand side of a pin assignment shall represent the name of a pin within reference cell, and the right-hand side of the pin assignment shall represent the name of the corrrepsonding pin within the cell instance.

## 9.5 Annotations for a CELL

This section defines annotations and attribute values in the context of a cell declaration.

### 9.5.1 CELLTYPE annotation

A *celltype* annotation shall be defined as shown in Semantics 2.

```
KEYWORD CELLTYPE = single_value_annotation {
  CONTEXT = CELL;
  VALUETYPE = identifier;
  VALUES {
    buffer combinational multiplexor flipflop latch
    memory block core special
  }
}
```

*Semantics 2—CELLTYPE annotation*

The *celltype* shall divide cells into categories, as specified in Table 27.

**Table 27—CELLTYPE annotation values**

| Annotation value | Description |
|---|---|
| `buffer` | `CELL` is a *buffer*, i.e., an element for transmission of a digital signal without performing a logic operation, except for possible logic inversion. |
| `combinational` | `CELL` is a combinatorial logic element, i.e., an element performing a logic operation on two or more digital input signals. |
| `multiplexor` | `CELL` is a *multiplexor*, i.e., an element for selective transmission of digital signals. |
| `flipflop` | `CELL` is a *flip-flop*, i.e., a one-bit storage element with edge-sensitive clock |
| `latch` | `CELL` is a *latch*, i.e., a one-bit storage element without edge-sensitive clock |
| `memory` | `CELL` is a *memory*, i.e., a multi-bit storage element with selectable addresses. |
| `block` | `CELL` is a hierarchical *block*, i.e., a complex element which has an associated netlist for implementation purpose. All instances of the netlist are library elements, i.e., there is a `CELL` model for each of them in the library. |
| `core` | `CELL` is a *core*, i.e., a complex element which has no associated netlist for implementation purpose. However, a netlist representation can exist for modeling purpose. |

**Table 27—CELLTYPE annotation values (Continued)**

| Annotation value | Description |
|---|---|
| `special` | `CELL` is a special element, which does not fall into any other category of cells. Examples: bus holder, protection diode, filler cell. |

### 9.5.2 SWAP_CLASS annotation

A *swap_class* annotation shall be defined as shown in Semantics 3.

```
KEYWORD SWAP_CLASS = annotation {
  CONTEXT = CELL;
  VALUETYPE = identifier;
}
```

*Semantics 3—SWAP_CLASS annotation*

The *value* is the name of a declared `CLASS`. Multi-value annotation can be used. Cells referring to the same `CLASS` can be swapped for certain applications.

Cell-swapping is only allowed, if the `RESTRICT_CLASS` annotation (see 9.5.3) authorizes usage of the cell and the cells to be swapped are compatible from an application standpoint.

### 9.5.3 RESTRICT_CLASS annotation

A *restrict-class* annotation shall be defined as shown in Semantics 4.

```
KEYWORD RESTRICT_CLASS = annotation {
  CONTEXT { CELL CLASS }
  VALUETYPE = identifier;
}
CLASS synthesis { USAGE = RESTRICT_CLASS ; }
CLASS scan { USAGE = RESTRICT_CLASS ; }
CLASS datapath { USAGE = RESTRICT_CLASS ; }
CLASS clock { USAGE = RESTRICT_CLASS ; }
CLASS layout { USAGE = RESTRICT_CLASS ; }
```

*Semantics 4—RESTRICT_CLASS annotation*

The *value* shall be the name of a declared `CLASS`.

The restrict-class annotation shall establish a necessary condition for the usage of a cell by an application performing a design transformation involving instantiations of cells. An application other than a design transformation (e.g. analysis, file format translation) can disregard the restrict-class annotation or use it for informational purpose only..

The meaning of the predefined restrict-class values in Semantics 4 is specified in Table 28.

**Table 28—Predefined values for RESTRICT_CLASS**

| Annotation value | Description |
|---|---|
| `synthesis` | Cell is suitable for creation or modification of a structual design description (i.e., a netlist) while providing functional equivalence. |
| `scan` | Cell is suitable for creation or modification of a scan chain within a netlist. |
| `datapath` | Cell is suitable for structural implementation of a data flow graph. |
| `clock` | Cell is suitable for distribution of a global synchronization signal. |
| `layout` | Cell is suitable for usage within a physical artwork. |

Additional restrict-class values can be defined within the context of a LIBRARY or a SUBLIBRARY, using the CLASS declaration and the SEMANTICS declaration in a similar way as shown in Semantics 4.

From the application standpoint, the following usage model for restrict-class shall apply:

a) A set of restrict-class values shall be associated with the application. These values are considered “known” by the application. Usage of a cell shall only be authorized, if the set of restrict-class values associated with the cell is a subset of the “known” restrict-class values.
b) Optionally, a boolean condition involving the set of “known” restrict-class values or a subset thereof can be associated with the application. In addition to a), usage of a cell shall only be authorized, if the set of restrict-class values associated with the cell satisfies the boolean condition.

*Example:*

Specification within the library:

```
CELL X { RESTRICT_CLASS { A B } }
CELL Y { RESTRICT_CLASS { C } }
CELL Z { RESTRICT_CLASS { A C F } }
```

Specification for the application:

Set of “known” restrict-class values = ( A, B, C, D, E)
Boolean condition = ( A and not B ) or C

Result:

Usage of CELL X is not authorized, because boolean condition is not true.
Usage of CELL Y is authorized, because all values are “known”, and boolean condition is true.
Usage of CELL Z is not authorized, because value F is not “known”.

### 9.5.4 SCAN_TYPE annotation

A *scan_type* annotation shall be defined as shown in Semantics 5.

```
KEYWORD SCAN_TYPE = single_value_annotation {
  CONTEXT = CELL;
  VALUETYPE = identifier;
  VALUES { muxscan clocked lssd control_0 control_1 }
}
```

*Semantics 5—SCAN_TYPE annotation*

It can take the *values* shown in Table 29.

**Table 29—SCAN_TYPE annotations for a CELL object**

| Annotation value | Description |
|---|---|
| `muxscan` | Cell contains a multiplexor for selection between non-scan-mode and scan-mode data. |
| `clocked` | Cell supports a dedicated scan clock. |
| `lssd` | Cell is suitable for level sensitive scan design. |
| `control_0` | Combinatorial cell, controlling pin shall be `0` in scan mode. |
| `control_1` | Combinatorial cell, controlling pin shall be `1` in scan mode. |

### 9.5.5 SCAN_USAGE annotation

A *scan_usage* annotation shall be defined as shown in Semantics 6.

```
KEYWORD SCAN_USAGE = single_value_annotation {
  CONTEXT = CELL;
  VALUETYPE = identifier;
  VALUES { input output hold }
}
```

*Semantics 6—SCAN_USAGE annotation*

It can take the *values* shown in Table 30.

**Table 30—SCAN_USAGE annotations for a CELL object**

| Annotation value | Description |
|---|---|
| `input` | Primary input cell in a scan chain. |
| `output` | Primary output cell in a scan chain. |
| `hold` | Intermediate cell in a scan chain. |

The `SCAN_USAGE` annotation applies for a cell which is designed to be the primary input, output or intermediate stage of a scan chain. It also applies for a block in case there is a particular scan-ordering requirement.

### 9.5.6 BUFFERTYPE annotation

A *buffertype* annotation shall be defined as shown in Semantics 7.

```
KEYWORD BUFFERTYPE = single_value_annotation {
  CONTEXT = CELL;
  VALUETYPE = identifier;
  VALUES { input output inout internal }
  DEFAULT = internal;
}
```

*Semantics 7—BUFFERTYPE annotation*

It can take the *values* shown in Table 31.

**Table 31—BUFFERTYPE annotations for a CELL object**

| Annotation value | Description |
|---|---|
| `input` | `CELL` has an external (i.e., off-chip) input pin. |
| `output` | `CELL` has an external output pin. |
| `inout` | `CELL` has an external bidirectional pin or an external input pin and an external output pin. |
| `internal` | `CELL` has no external pin. |

### 9.5.7 DRIVERTYPE annotation

A *drivertype* annotation shall be defined as shown in Semantics 8.

```
KEYWORD DRIVERTYPE = single_value_annotation {
  CONTEXT = CELL;
  VALUETYPE = identifier;
  VALUES { predriver slotdriver both }
}
```

*Semantics 8—DRIVERTYPE annotation*

It can take the *values* shown in Table 32.

**Table 32—DRIVERTYPE annotations for a CELL object**

| Annotation value | Description |
|---|---|
| `predriver` | `CELL` is a predriver, i.e., the core part of an I/O buffer. |
| `slotdriver` | `CELL` is a slotdriver, i.e., the pad of an I/O buffer with off-chip connection. |
| `both` | `CELL` is both a predriver and a slot driver, i.e., a complete I/O buffer. |

DRIVERTYPE applies only for a cell with BUFFERTYPE value input or output or inout.

### 9.5.8 PARALLEL_DRIVE annotation

A *parallel_drive* annotation shall be defined as shown in Semantics 9.

```
KEYWORD PARALLEL_DRIVE = single_value_annotation {
  CONTEXT = CELL;
  VALUETYPE = unsigned;
  DEFAULT = 1;
}
```

*Semantics 9—PARALLEL_DRIVE annotation*

The annotation value shall specify the number of cells connected in parallel. This number shall be greater than zero (0) ; the default shall be 1.

### 9.5.9 PLACEMENT_TYPE annotation

A *placement_type* annotation shall be defined as shown in Semantics 10.

```
KEYWORD PLACEMENT_TYPE = single_value_annotation {
  CONTEXT = CELL;
  VALUETYPE = identifier;
  VALUES { pad core ring block connector }
  DEFAULT = core;
}
```

*Semantics 10—PLACEMENT_TYPE annotation*

The purpose of the placement-type annotation is to establish categories of cells in terms of placement and power routing requirements.

It can take the *values* shown in Table 33.

**Table 33—PLACEMENT_TYPE annotations for a CELL object**

| Annotation value | Description |
|---|---|
| pad | The cell is an element to be placed in the I/O area of a die. |
| core | The cell is a regular element to be placed in the core area of a die, using a regular power structure. |
| ring | The cell is a macro element with built-in power structure. |
| block | The cell is an abstraction of a collection of regular elements, each of which uses a regular power structure. |
| connector | The cell is to be placed at the border of the core area of a die in order to establish a connection between a regular power structure and a power ring in the I/O area. |

### 9.5.10 SITE reference annotation

A *site* reference annotation shall be defined as shown in Semantics 11.

```
SEMANTICS SITE = annotation {
  CONTEXT { CELL CLASS }
}
```

*Semantics 11—SITE reference annotation*

The purpose of a site reference annotation is to indicate one or more legal placement locations for a cell. The annotation value shall be the name of a declared *site* (see Section 9.28).

## 9.6 ATTRIBUTE values for a CELL

An attribute in the context of a cell declaration shall specify more specific information within the category given by the celltype annotation.

The attribute values shown in Table 34 can be used within a `CELL` with `CELLTYPE=memory`.

**Table 34—Attribute values for a CELL with CELLTYPE=memory**

| Attribute item | Description |
|---|---|
| `RAM` | Random Access Memory |
| `ROM` | Read Only Memory |
| `CAM` | Content Addressable Memory |
| `static` | Static memory, needs no refreshment |
| `dynamic` | Dynamic memory, needs refreshment |
| `asynchronous` | operation self-timed |
| `synchronous` | operation synchronized with a clock signal |

The attributes shown in Table 35 can be used within a `CELL` with `CELLTYPE=block`.

**Table 35—Attributes within a CELL with CELLTYPE=block**

| Attribute item | Description |
|---|---|
| `counter` | `CELL` is a *counter*, i.e., a complex sequential circuit going through a predefined sequence of states in its normal operation mode where each state represents an encoded control value. |
| `shift_register` | `CELL` is a *shift register*, i.e., a complex sequential circuit going through a predefined sequence of states in its normal operation mode, where each subsequent state can be obtained from the previous one by a shift operation. Each bit represents a data value. |

**Table 35—Attributes within a CELL with CELLTYPE=block (Continued)**

| Attribute item | Description |
|---|---|
| adder | CELL is an *adder*, i.e., a combinatorial circuit performing an addition of two operands. |
| subtractor | CELL is a *subtractor*, i.e., a combinatorial circuit performing a subtraction of two operands. |
| multiplier | CELL is a *multiplier*, i.e., a combinatorial circuit performing a multiplication of two operands. |
| comparator | CELL is a *comparator*, i.e., a combinatorial circuit comparing the magnitude of two operands. |
| ALU | CELL is an *arithmetic logic unit*, i.e., a combinatorial circuit combining the functionality of adder, subtractor, and comparator. |

The attributes shown in Table 36 can be used within a CELL with CELLTYPE=core.

**Table 36—Attributes within a CELL with CELLTYPE=core**

| Attribute item | Description |
|---|---|
| PLL | CELL is a *phase-locked loop*. |
| DSP | CELL is a *digital signal processor.* |
| CPU | CELL is a *central processing unit*. |
| GPU | CELL is a *graphical processing unit*. |

The attributes shown in Table 37 can be used within a CELL with CELLTYPE=special.

**Table 37—Attributes within a CELL with CELLTYPE=special**

| Attribute item | Description |
|---|---|
| busholder | CELL enables a tristate bus to hold its last value before all drivers went into high-impedance state (see 10.1). |
| clamp | CELL connects a net to a constant value (logic value and drive strength; see 10.1). |
| diode | CELL is a *diode* (no FUNCTION statement). |
| capacitor | CELL is a *capacitor* (no FUNCTION statement). |
| resistor | CELL is a *resistor* (no FUNCTION statement). |
| inductor | CELL is an *inductor* (no FUNCTION statement). |
| fillcell | CELL is used to fill unused space in layout (no PIN, no FUNCTION statement). |

## 9.7 PIN declaration

A *pin* shall be declared as a *scalar pin* or as a *vector pin* or a *matrix pin*, as shown in Syntax 49.

```
pin ::=
    scalar_pin | vector_pin | matrix_pin
scalar_pin ::=
      PIN pin_identifier ;
    | PIN pin_identifier { { scalar_pin_item } }
    | scalar_pin_template_instantiation
scalar_pin_item ::=
      all_purpose_item
    | port
vector_pin ::=
      PIN multi_index pin_identifier ;
    | PIN multi_index pin_identifier { { vector_pin_item } }
    | vector_pin_template_instantiation
vector_pin_item ::=
      all_purpose_item
    | range
matrix_pin ::=
      PIN first_multi_index pin_identifier second_multi_index ;
    | PIN first_multi_index pin_identifier second_multi_index { { matrix_pin_item } }
    | matrix_pin_template_instantiation
matrix_pin_item ::=
    vector_pin_item
```

*Syntax 49—PIN declaration*

A pin shall represent a terminal of an electronic circuit. The purpose of a pin is exchange of information or energy between the circuit and its environment. A constant value of information shall be called *state*. A time-dependent value of information shall be called *signal*.

A reference to a pin in general shall be established by the pin identifier.

The order of pin declarations within a cell declaration shall reflect the order of appearance of pins, when the cell is instantiated in a netlist and the pins are refered to by order. The *view* annotation (see Section 9.9.1) shall further specify which pins are visible in a netlist.

A scalar pin can be associated with a general electrical signal. However, a vector pin or a matrix pin can only be associated with digital signals. One element of a vector pin or of a matrix pin shall be associated with one bit of information, i.e., a binary digital signal.

A vector-pin can be considered as a *bus*, i.e., a combination of scalar pins. The declaration of a vector-pin shall involve a *multi index* (see Section 7.8). A reference to a scalar within the vector-pin shall be established by the pin identifier followed by a *single index* (see Section 7.8). A reference to a subvector within the vector-pin shall be established by the pin identifier followed by a *multi index*.

A matrix-pin can be considered as a combination of vector-pins. A reference to a vector or to a submatrix, respectively, within the matrix-pin shall be established by the pin identifier followed by a single index or by a multi index, respectively.

Within a matrix-pin declaration, the first multi index shall specify the range of scalars or bits, and the second multi index shall specify the range of vectors. Support for direct reference of a scalar within a matrix is not provided.

*Example*

```
PIN [5:8] myVectorPin ;
PIN [3:0] myMatrixPin [1:1000] ;
```

The pin variable `myVectorPin[5]` refers to the scalar associated with the MSB of `myVectorPin`.
The pin variable `myVectorPin[8]` refers to the scalar associated with the LSB of `myVectorPin`.
The pin variable `myVectorPin[6:7]` refers to a subvector within `myVectorPin`.
The pin variable `myMatrixPin[500]` refers to a vector within `myMatrixPin`.
The pin variable `myMatrixPin[500:502]` refers to 3 subsequent vectors within `myMatrixPin`.

Consider the following pin assignment:

```
myVectorPin=myMatrixPin[500];
```

This establishes the following exchange of information:

`myVectorPin[5]` receives information from element `[3]` of `myMatrixPin[500]`.
`myVectorPin[6]` receives information from element `[2]` of `myMatrixPin[500]`.
`myVectorPin[7]` receives information from element `[1]` of `myMatrixPin[500]`.
`myVectorPin[8]` receives information from element `[0]` of `myMatrixPin[500]`.

## 9.8 PINGROUP declaration

A *pingroup* shall be declared as a *simple pingroup* or as a *vector pingroup*, as shown in Syntax 50.

```
pingroup ::=
    simple_pingroup | vector_pingroup
simple_pingroup ::=
     PINGROUP pingroup_identifier { members { all_purpose_item } }
    | simple_pingroup_template_instantiation
members ::=
    MEMBERS { pin_identifier pin_identifier { pin_identifier } }
vector_pingroup ::=
    | PINGROUP [ index_value : index_value ] pingroup_identifier
        { members { vector_pingroup_item } }
    | vector_pingroup_template_instantiation
vector_pingroup_item ::=
      all_purpose_item
    | range
```

*Syntax 50—PINGROUP declaration*

A pingroup in general shall serve the purpose to specify items applicable to a combination of pins. The combination of pins shall be specified by the *members* statement.

A *vector pingroup* can only combine scalar pins. A vector pingroup can be used as a pin variable, in the same capacity as a vector pin.

A *simple pingroup* can combine pins of any format, i.e., scalar pins, vector pins, and matrix pins. A simple pingroup can not be used as a pin variable.

## 9.9 Annotations for a PIN and a PINGROUP

This section defines annotations and attribute values in the context of a pin declaration or a pingroup declaration.

### 9.9.1 VIEW annotation

A *view* annotation shall be defined as shown in Semantics 12.

```
KEYWORD VIEW = single_value_annotation {
  CONTEXT { PIN PINGROUP }
  VALUETYPE = identifier;
  VALUES { functional physical both none }
  DEFAULT = both
}
```

*Semantics 12—VIEW annotation*

The purpose of the view annotation is to specify the visibility of a pin in a netlist.

It can take the values shown in Table 38.

**Table 38—VIEW annotations for a PIN object**

| Annotation value | Description |
|---|---|
| `functional` | `pin` appears in functional netlist. |
| `physical` | `pin` appears in physical netlist. |
| `both` (default) | `pin` appears in both functional and physical netlist. |
| `none` | `pin` does not appear in netlist. |

### 9.9.2 PINTYPE annotation

A *pintype* annotation shall be defined as shown in Semantics 13.

```
KEYWORD PINTYPE = single_value_annotation {
  CONTEXT = PIN;
  VALUETYPE = identifier;
  VALUES { digital analog supply }
  DEFAULT = digital;
}
```

*Semantics 13—PINTYPE annotation*

The purpose of the pintype annotation is to establish broad categories of pins.

It can take the values shown in Table 39.

**Table 39—PINTYPE annotations for a PIN object**

| Annotation value | Description |
|---|---|
| `digital` (default) | Digital signal pin. |
| `analog` | Analog signal pin. |
| `supply` | Power supply or ground pin. |

### 9.9.3 DIRECTION annotation

A *direction* annotation shall be defined as shown in Semantics 14.

```
KEYWORD DIRECTION = single_value_annotation {
  CONTEXT = PIN;
  VALUETYPE = identifier;
  VALUES { input output both none }
}
```

*Semantics 14—DIRECTION annotation*

The purpose of the direction annotation is to establish the flow of information and/or electrical energy through a pin. Information/energy can flow into a cell or out of a cell through a pin. The information/energy flow is not to be mistaken as the flow of electrical current through a pin.

The direction annotation can take the values shown in Table 40.

**Table 40—DIRECTION annotations for a PIN object**

| Annotation value | Description |
|---|---|
| `input` | Information/energy flows through the pin into the cell. The pin is a receiver or a sink. |
| `output` | Information/energy flows through the pin out of the cell. The pin is a driver or a source. |
| `both` | Information/energy flows through the pin in and out of the cell. The pin is both a receiver/sink and driver/source, dependent on the mode of operation. |
| `none` | No information/energy flows through the pin in or out of the cell. The pin can be an internal pin without connection to its environment or a feedthrough where both ends are represented by the same pin. |

The *direction* annotation shall be orthogonal to the *pintype* annotation, i.e., all combinations of annotation values are possible.

*Examples*

— The power and ground pins of a regular cell have DIRECTION=input.
— A level converter cell has a power supply pin with DIRECTION=input and another power supply pin with DIRECTION=output.
— A level converter can have separate ground pins related to its power supply pins or a common ground pin with DIRECTION=both.
— The power and ground pins of a feed through cell have the DIRECTION=none.

### 9.9.4 SIGNALTYPE annotation

A *signaltype* annotation shall be defined as shown in Semantics 15.

```
KEYWORD SIGNALTYPE = single_value_annotation {
  CONTEXT = PIN;
  VALUETYPE = identifier;
  VALUES {
    data scan_data address control select tie clear set
    enable out_enable scan_enable scan_out_enable
    clock master_clock slave_clock
    scan_master_clock scan_slave_clock
  }
  DEFAULT = data;
}
```

*Semantics 15—SIGNALTYPE annotation*

SIGNALTYPE classifies the functionality of a pin. The currently defined values apply for pins with PIN-TYPE=DIGITAL.

Conceptually, a pin with PINTYPE = ANALOG can also have a SIGNALTYPE annotation. However, no values are currently defined.

The fundamental SIGNALTYPE values are defined in Table 41

**Table 41—Fundamental SIGNALTYPE annotations for a PIN object**

| Annotation value | Description |
|---|---|
| data (default) | General *data* signal, i.e., a signal that carries information to be transmitted, received, or subjected to logic operations within the CELL. |
| address | *Address* signal of a memory, i.e., an encoded signal, usually a bus or part of a bus, driving an address decoder within the CELL. |
| control | General *control* signal, i.e., an encoded signal that controls at least two modes of operation of the CELL, eventually in conjunction with other signals. The signal value is allowed to change during real-time circuit operation. |
| select | *Select* signal, i.e., a signal that selects the data path of a multiplexor or de-multiplexor within the CELL. Each selected signal has the same SIGNALTYPE. |
| enable | The signal enables storage of general input data in a latch or a flip-flop or a memory |

**Table 41—Fundamental SIGNALTYPE annotations for a PIN object (Continued)**

| Annotation value | Description |
|---|---|
| `tie` | The signal needs to be tied to a fixed value statically in order to define a fixed or programmable mode of operation of the `CELL`, eventually in conjunction with other signals. The signal value is not allowed to change during real-time circuit operation. |
| `clear` | *Clear* or *reset* signal of a flip-flop or latch, i.e., a signal that controls the storage of the value `0` within the `CELL`. |
| `set` | *Preset* or *set* signal of a flip-flop or latch, i.e., a signal that controls the storage of the value `1` within the `CELL`. |
| `clock` | *Clock* signal of a flip-flop or latch, i.e., a timing-critical signal that triggers data storage within the `CELL`. |

Figure 6 shows how to construct composite signaltypes.

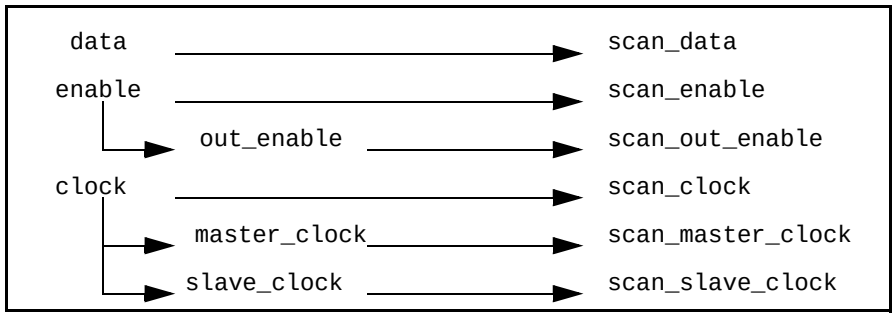



**Figure 6—Scheme for construction of composite signaltype values**

The composite `SIGNALTYPE` values are defined in Table 42

**Table 42—Composite SIGNALTYPE annotations for a PIN object**

| Annotation value | Description |
|---|---|
| `scan_data` | *Scan* data signal, i.e., signal is relevant in scan mode only. |
| `out_enable` | Enables visibility of general data at an output pin of a cell. |
| `scan_enable` | Enables storage of scan input data in a latch or a flipflop. |
| `scan_out_enable` | Enables visibility of scan data at an output pin of a cell. |
| `master_clock` | Triggers storage of input data in the first stage of a flipflop in a two-phase clocking scheme. |
| `slave_clock` | Triggers data transfer from first the stage to the second stage of a flipflop in a two-phase clocking scheme. |
| `scan_clock` | Triggers storage of scan input data within a cell. |
| `scan_master_clock` | Triggers storage of input scan data in the first stage of a flipflop in a two-phase clocking scheme. |

**Table 42—Composite SIGNALTYPE annotations for a PIN object (Continued)**

| Annotation value | Description |
|---|---|
| scan_slave_clock | Triggers scan data transfer from the first stage to the second stage of a flipflop in a two-phase clocking scheme. |

Within the definitions of Table 41 and Table 42, the elements *flipflop*, *latch*, *multiplexor*, or *memory* can be standalone cells or embedded in larger cells. In the former case, the celltype is `flipflop`, `latch`, `multiplexor`, or `memory`, respectively. In the latter case, the celltype can be `block` or `core`.

### 9.9.5 ACTION annotation

An *action* annotation shall be defined as shown in Semantics 16.

```
KEYWORD ACTION = single_value_annotation {
  CONTEXT = PIN;
  VALUETYPE = identifier;
  VALUES { asynchronous synchronous }
}
```

*Semantics 16—ACTION annotation*

The purpose of the action annotation is to define, whether a signal is self-timed or synchronized with a clock signal.

The ACTION annotation can take the values shown in Table 43.

**Table 43—ACTION annotations for a PIN object**

| Annotation value | Description |
|---|---|
| asynchronous | Signal acts in an asynchronous way, i.e., self-timed. |
| synchronous | Signal acts in a synchronous way, i.e., triggered by a clock signal. |

The `ACTION` annotation applies only to pins with certain `SIGNALTYPE` values, as shown in Table 44. The rule applies also to any composite `SIGNALTYPE` values based on the fundamental values.

**Table 44—ACTION applicable in conjunction with SIGNALTYPE values**

| SIGNALTYPE value | ACTION applicable |
|---|---|
| data, scan_data | No |
| address | No |
| control | Yes |
| select | No |

**Table 44—ACTION applicable in conjunction with SIGNALTYPE values (Continued)**

| SIGNALTYPE value | ACTION applicable |
|---|---|
| `enable, scan_enable, out_enable, scan_out_enable` | Yes |
| `tie` | No |
| `clear` | Yes |
| `set` | Yes |
| `clock, scan_clock, master_clock, slave_clock, scan_master_clock, scan_slave_clock` | No |

### 9.9.6 POLARITY annotation

A *polarity* annotation shall be defined as shown in Semantics 17.

```
KEYWORD POLARITY = single_value_annotation {
  CONTEXT = PIN;
  VALUETYPE = identifier;
  VALUES { high low rising_edge falling_edge double_edge }
}
```

*Semantics 17—POLARITY annotation*

The purpose of the polarity annotation is to define the active state or the active edge of an input signal.

The POLARITY annotation can take the values shown in Table 45.

**Table 45—POLARITY annotations for a PIN**

| Annotation value | Description |
|---|---|
| `high` | Signal is active high or to be driven high. |
| `low` | Signal is active low or to be driven low. |
| `rising_edge` | Signal is activated by rising edge. |
| `falling_edge` | Signal is activated by falling edge. |
| `double_edge` | Signal is activated by both rising and falling edge. |

The `POLARITY` annotation applies only to pins with certain `SIGNALTYPE` values, as shown in Table 46..

**Table 46—POLARITY applicable in conjunction with SIGNALTYPE values**

| SIGNALTYPE value | Applicable POLARITY |
|---|---|
| `data, scan_data` | N/A |

**Table 46—POLARITY applicable in conjunction with SIGNALTYPE values (Continued)**

| SIGNALTYPE value | Applicable POLARITY |
|---|---|
| address | N/A |
| control | N/A |
| select | N/A |
| enable, scan_enable, out_enable, scan_out_enable | high, low. |
| tie | high, low. |
| clear | high, low. |
| set | high, low. |
| clock, scan_clock, master_clock, slave_clock, scan_master_clock, scan_slave_clock | high, low, rising_edge, falling_edge, double_edge, |

### 9.9.7 DATATYPE annotation

A *datatype* annotation shall be defined as shown in Semantics 18.

```
KEYWORD DATATYPE = single_value_annotation {
  CONTEXT { PIN PINGROUP }
  VALUETYPE = identifier;
  VALUES { signed unsigned }
 }
```

*Semantics 18—DATATYPE annotation*

The purpose of the datatype annotation is to define the arithmetic representation of a digital signal.

The DATATYPE annotation can take the values shown in Table 47.

**Table 47—DATATYPE annotations for a PIN object**

| Annotation value | Description |
|---|---|
| signed | Result of arithmetic operation is signed 2's complement. |
| unsigned | Result of arithmetic operation is unsigned. |

DATATYPE is only relevant for a vector pin.

### 9.9.8 INITIAL_VALUE annotation

An *initial value* annotation shall be defined as shown in Semantics 19.

```
KEYWORD INITIAL_VALUE = single_value_annotation {
  CONTEXT = CELL;
  VALUETYPE = boolean_value;
  DEFAULT = U;
}
```

*Semantics 19—INITIAL_VALUE annotation*

The purpose of the initial value annotation is to provide an initial value of a signal within a simulation model derived from ALF. A signal shall have the initial value before a simulation event affects the signal. The default value “U” means “uninitialized” (see Table 69).

### 9.9.9 SCAN_POSITION annotation

A *scan position* annotation shall be defined as shown in Semantics 20.

```
KEYWORD SCAN_POSITION = single_value_annotation {
  CONTEXT = PIN;
  VALUETYPE = unsigned;
  DEFAULT = 0;
}
```

*Semantics 20—SCAN_POSITION annotation*

The purpose of the scan position annotation is to specify the position of the pin in scan chain, starting with `1` for the primary input. The value 0 (which is the default) indicates that the pin is not on the scan chain.

### 9.9.10 STUCK annotation

A *stuck* annotation shall be defined as shown in Semantics 21.

```
KEYWORD STUCK = single_value_annotation {
  CONTEXT = PIN;
  VALUETYPE = identifier;
  VALUES { stuck_at_0 stuck_at_1 both none }
  DEFAULT = both;
}
```

*Semantics 21—STUCK annotation*

The purpose of the stuck annotation is to specify a static fault model applicable for the pin.

The STUCK annotation can take the values shown in Table 48.

**Table 48—STUCK annotations for a PIN object**

| Annotation value | Description |
|---|---|
| `stuck_at_0` | Pin can exhibit a faulty static low state. |
| `stuck_at_1` | Pin can exhibit a faulty static high state. |

**Table 48—STUCK annotations for a PIN object (Continued)**

| Annotation value | Description |
|---|---|
| `both` (default) | Pin can exhibit a faulty static high or low state. |
| `none` | Pin can not exhibit a faulty static state. |

### 9.9.11 SUPPLYTYPE annotation

A *supplytype* annotation shall be defined as shown in Semantics 22.

```
KEYWORD SUPPLYTYPE = annotation {
  CONTEXT { PIN CLASS }
  VALUETYPE = identifier;
  VALUES { power ground reference }
}
```

*Semantics 22—SUPPLYTYPE annotation*

The supplytype annotation can take the values shown in Table 49.

**Table 49—SUPPLYTYPE annotations for a PIN object**

| Annotation value | Description |
|---|---|
| `power` | Pin is electrically connected to a power supply, i.e., a constant non-zero voltage source providing energy for operation of a circuit. |
| `ground` | Pin is electrically connected to ground, i.e., a zero voltage source providing the return path for electrical current through a power supply. |
| `reference` | Pin exhibits a constant voltage level without providing significant energy for operation of a circuit. |

The purpose of the supplytype annotation is to define a subcategory of pins with *pintype* value *supply* (see Table 39).

### 9.9.12 SIGNAL_CLASS annotation

A *signal-class* annotation shall be defined as shown in Semantics 23.

```
KEYWORD SIGNAL_CLASS = annotation {
  CONTEXT { PIN PINGROUP }
  VALUETYPE = identifier;
}
```

*Semantics 23—SIGNAL_CLASS annotation*

The value shall be the name of a declared CLASS.

The purpose of the signal-class annotation is to specify which terminals of a cell with are functionally related to each other. The signal-class annotation applies for a pin with any *signaltype* value (see Section 9.9.4).

*Example:*

A multiport memory can have a data bus related to an address bus and another data bus related to another address bus. Note that the term “port” in “multiport” does not relate to the ALF *port* declaration (see Section 9.26).

```
CELL my2PortMemory {
    CLASS ReadPort { USAGE = SIGNAL_CLASS; }
    CLASS WritePort { USAGE = SIGNAL_CLASS; }
    PIN [3:0] addr_A { SIGNALTYPE = address; SIGNAL_CLASS = ReadPort; }
    PIN [7:0] data_A { SIGNALTYPE = data;    SIGNAL_CLASS = ReadPort; }
    PIN [3:0] addr_B { SIGNALTYPE = address; SIGNAL_CLASS = WritePort; }
    PIN [7:0] data_B { SIGNALTYPE = data;    SIGNAL_CLASS = WritePort; }
    PIN write_enable { SIGNALTYPE = enable;   SIGNAL_CLASS = WritePort; }
}
```

### 9.9.13 SUPPLY_CLASS annotation

A *supply-class* annotation shall be defined as shown in Semantics 24.

```
KEYWORD SUPPLY_CLASS = annotation {
  CONTEXT { PIN CLASS }
  VALUETYPE = identifier;
}
```

*Semantics 24—SUPPLY_CLASS annotation*

The value shall be the name of a declared CLASS.

The purpose of the supply-class annotation is to specify which terminals of a cell with are electrically related to each other. The supply-class annotation applies for a pin with any *signaltype* (see Section 9.9.4) or *supplytype* value (see Section 9.9.11). The supply-class annotation also applies for a class with *usage* value *connect-class* (see Section 9.9.16). In this case, the refered class represents a set of global nets which are electrically related to each other.

*Example 1:*

A cell can provide two local power supplies. Each pin is related to at least one power supply.

```
CELL myLevelShifter {
    CLASS supply1 { USAGE = SUPPLY_CLASS; }
    CLASS supply2 { USAGE = SUPPLY_CLASS; }
    PIN Vdd1 { SUPPLYTYPE = power; SUPPLY_CLASS = supply1; }
    PIN Din  { SIGNALTYPE = data;  SUPPLY_CLASS = supply1; }
    PIN Vdd2 { SUPPLYTYPE = power; SUPPLY_CLASS = supply2; }
    PIN Dout { SIGNALTYPE = data;  SUPPLY_CLASS = supply2; }
    PIN Gnd  { SUPPLYTYPE = ground; SUPPLY_CLASS { supply1 supply2 } }
}
```

*Example 2:*

A library can provide two environmental power supplies. A supply pin of a cell has to be connected to a global net related to an environmental power supply.

```
CLASS core { USAGE = SUPPLY_CLASS; }
CLASS io   { USAGE = SUPPLY_CLASS; }
CLASS Vdd1 { USAGE=CONNECT_CLASS; SUPPLYTYPE=power; SUPPLY_CLASS=core; }
CLASS Vss1 { USAGE=CONNECT_CLASS; SUPPLYTYPE=ground; SUPPLY_CLASS=core; }
CLASS Vdd2 { USAGE=CONNECT_CLASS; SUPPLYTYPE=power; SUPPLY_CLASS=io; }
CLASS Vss2 { USAGE=CONNECT_CLASS; SUPPLYTYPE=ground; SUPPLY_CLASS=io; }
CELL myInternalCell {
    PIN vdd { CONNECT_CLASS=Vdd1; }
    PIN vss { CONNECT_CLASS=Vss1; }
}
CELL myPadCell {
    PIN vdd { CONNECT_CLASS=Vdd2; }
    PIN vss { CONNECT_CLASS=Vss2; }
}
```

### 9.9.14 DRIVETYPE annotation

A *drivetype* annotation shall be defined as shown in Semantics 25.

```
KEYWORD DRIVETYPE = single_value_annotation {
  CONTEXT { PIN CLASS }
  VALUETYPE = identifier;
  VALUES {
    cmos nmos pmos cmos_pass nmos_pass pmos_pass
    ttl open_drain open_source
  }
  DEFAULT = cmos;
}
```

*Semantics 25—DRIVETYPE annotation*

The purpose of the drivetype annotation is to specify a category of electrical characteristics for a pin, which relate to the system of logic values and drive strengths specified in Table 69.

The drivetype annotation can take the values shown in Table 50.

**Table 50—DRIVETYPE annotations for a PIN object**

| Annotation value | Description |
|---|---|
| `cmos` (default) | Standard cmos signal. The logic high level is equal to the power supply, the logic low level is equal to ground. The drive strength is strong. No static current flows. Signal is amplified by cmos stage. |
| `nmos` | Nmos or pseudo nmos signal. The logic high level is equal to the power supply and its drive strength is resistive. The logic low level voltage depends on the ratio of pull-up and pull-down transistor. Static current flows in logic low state. |

**Table 50—DRIVETYPE annotations for a PIN object (Continued)**

| Annotation value | Description |
|---|---|
| pmos | Pmos or pseudo pmos signal. The logic low level is equal to ground and its drive strength is resistive. The logic high level voltage depends on the ratio of pull-up and pull-down transistor. Static current flows in logic high state. |
| nmos_pass | Nmos passgate signal. Signal is not amplified by passgate stage. Logic low voltage level is preserved, logic high voltage level is limited by power supply minus nmos threshold voltage. |
| pmos_pass | Pmos passgate signal. Signal is not amplified by passgate stage. Logic high voltage level is preserved, logic high voltage level is limited by pmos threshold voltage. |
| cmos_pass | Cmos passgate signal, i.e., a full transmission gate. Signal is not amplified by passgate stage. Voltage levels are preserved. |
| ttl | TTL signal. Both logic high and logic low voltage levels are load-dependent, as static current can flow. |
| open_drain | Open drain signal. Logic low level is equal to ground. Logic high level corresponds to high impedance state. |
| open_source | Open source signal. Logic high level is equal to the power supply. Logic low level corresponds to high impedance state. |

### 9.9.15 SCOPE annotation

A *scope* annotation shall be defined as shown in Semantics 26.

```
KEYWORD SCOPE = single_value_annotation {
  CONTEXT = PIN;
  VALUETYPE = identifier;
  VALUES { behavior measure both none }
  DEFAULT = both;
}
```

*Semantics 26—SCOPE annotation*

The purpose of the scope annotation is to specify a category of modeling usage for a pin. The scope annotation specifies whether a pin can be involved in a control expression within a vector declaration (see Section 9.16) or within a behavior statement (see Section 10.4).

The scope annotation can take the values shown in Table 51.

**Table 51—SCOPE annotations for a PIN object**

| Annotation value | Description |
|---|---|
| behavior | The pin is used for modeling functional behavior. Pin can be involved in a control expression within a BEHAVIOR statement. |

**Table 51—SCOPE annotations for a PIN object (Continued)**

| Annotation value | Description |
|---|---|
| `measure` | Measurements related to the pin can be described. Pin can be involved in a control expression within a `VECTOR` declaration. |
| `both` (default) | Pin can be involved in a control expression within a BEHAVIOR statement or within a VECTOR declaration. |
| `none` | Pin can not be involved in a control expression. |

### 9.9.16 CONNECT_CLASS annotation

A *connect_class* annotation shall be defined as shown in Semantics 27.

```
KEYWORD CONNECT_CLASS = single_value_annotation {
  CONTEXT = PIN;
  VALUETYPE = identifier;
}
```

*Semantics 27—CONNECT_CLASS annotation*

The value shall be the name of a declared CLASS.

The purpose of the connect-class annotation is to specify a relationship between a pin and an environmental rule for connectivity. For application in conjunction with *supply-class* see Section 9.9.13. For application in conjunction with *connect-rule* see Section 11.42.1.

### 9.9.17 SIDE annotation

A *side* annotation shall be defined as shown in Semantics 28.

```
KEYWORD SIDE = single_value_annotation {
  CONTEXT { PIN PINGROUP }
  VALUETYPE = identifier;
  VALUES { left right top bottom inside }
}
```

*Semantics 28—SIDE annotation*

The purpose of the side annotation is to define an abstract location of a pin relative to the bounding box of a cell.

The side annotation can take the values shown in Table 52.

**Table 52—SIDE annotations for a PIN object**

| Annotation value | Description |
|---|---|
| `left` | `pin` is on the left side of the bounding box. |
| `right` | `pin` is on the right side of the bounding box. |

**Table 52—SIDE annotations for a PIN object (Continued)**

| Annotation value | Description |
|---|---|
| `top` | `pin` is at the top of the bounding box. |
| `bottom` | `pin` is at the bottom of the bounding box. |
| `inside` | `pin` is inside the bounding box. |

### 9.9.18 ROW and COLUMN annotation

A *row* annotation and a *column* annotation shall be defined as shown in Semantics 29.

```
KEYWORD ROW = annotation {
  CONTEXT { PIN PINGROUP }
  VALUETYPE = unsigned;
}
KEYWORD COLUMN = annotation {
  CONTEXT { PIN PINGROUP }
  VALUETYPE = unsigned;
}
```

*Semantics 29—ROW and COLUMN annotations*

The purpose of a row and a column annotation is to indicate a location of a pin when a cell is placed within a placement grid. The count of rows and columns shall start at the lower left corner of the bounding box of the cell, as shown in figure 7.

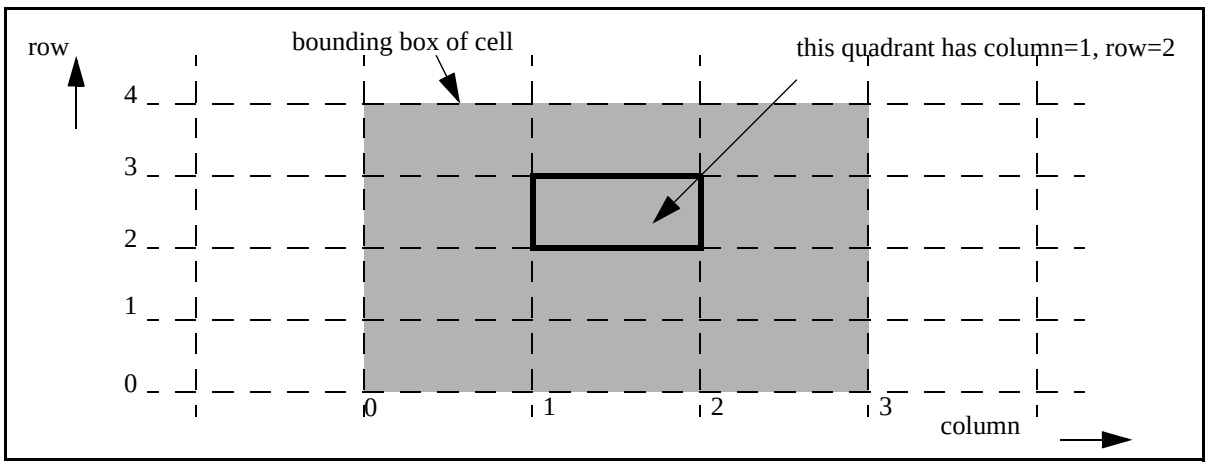



**Figure 7—ROW and COLUMN relative to a bounding box of a CELL**

The row annotion is applicable for a pin with *side* value *left* or *right*. The column annotion is applicable for a pin with *side* value *top* or *bottom*. Both row and column annotation are applicable for a pin with *side* value *inside*.

A single-value annotation is applicable for a scalar pin. A multi-value annotation is applicable for a vector pin or for a vector pingroup. The number of values shall match the number of scalar pins within the vector pin or pingroup. The order of values shall correspond to the order of scalar pins within the vector pin or pingroup.

### 9.9.19 ROUTING_TYPE annotation

A *routing-type* annotation shall be defined as shown in Semantics 30.

```
KEYWORD ROUTING_TYPE = single_value_annotation {
  CONTEXT { PIN PORT }
  VALUETYPE = identifier;
  VALUES { regular abutment ring feedthrough }
  DEFAULT = regular;
}
```

*Semantics 30—ROUTING_TYPE annotation*

The purpose of the routing-type annotation is to specify the physical connection between a pin and a routed wire.

The routing-type annotation can take the values shown in Table 53.

**Table 53—ROUTING-TYPE annotations for a PIN object**

| Annotation value | Description |
|---|---|
| `regular` | Pin has a via, connection by regular routing to the via |
| `abutment` | Pin is the end of a wire segment, connection by abutment |
| `ring` | Pin forms a ring around the cell, connection by abutment to any point of the ring. |
| `feedthrough` | Pin has two aligned ends of a wire segment, connection by abutment on both ends |

### 9.9.20 PULL annotation

A *pull* annotation shall be defined as shown in Semantics 31.

```
KEYWORD PULL = single_value_annotation {
  CONTEXT = PIN;
  VALUETYPE = identifier;
  VALUES { up down both none }
  DEFAULT = none;
}
```

*Semantics 31—PULL annotation*

The purpose of the pull annotation is to specify whether a *pullup* or a *pulldown* device is connected to the pin.

The pull annotation can take the values shown in Table 54.

**Table 54—PULL annotations for a PIN object**

| Annotation value | Description |
|---|---|
| `up` | Pullup device connected to the pin. |
| `down` | Pulldown device connected to the pin. |
| `both` | Both pullup and pulldown device connected to pin. |
| `none` (default) | No pullup or pulldown device connected to the pin. |

A pullup device ties the pin to a logic high level when no other signal is driving the pin. A pulldown device ties the pin to a logic low level when no other signal is driving the pin. If both devices are connected, the pin is tied to an intermediate voltage level, i.e. in-between logic high and logic low, when no other signal is driving the pin.

## 9.10 ATTRIBUTE values for a PIN and a PINGROUP

The attribute values shown in Table 55 can be used within a `PIN` object.

**Table 55—Attributes within a PIN object**

| Attribute item | Description |
|---|---|
| `SCHMITT` | Schmitt trigger signal, i.e., the DC transfer characteristics exhibit a hysteresis. Applicable for output pin. |
| `TRISTATE` | Tristate signal, i.e., the signal can be in high impedance mode. Applicable for output pin. |
| `XTAL` | Crystal/oscillator signal. Applicable for output pin of an oscillator circuit. |
| `PAD` | Pin has external,i.e., off-chip connection. |

The attributes shown in Table 56 are applicable for a pin of a cell with *celltype* value *memory* in conjunction with a specific *signaltype* value.

**Table 56—Attributes for pins of a memory**

| Attribute item | SIGNALTYPE | Description |
|---|---|---|
| `ROW_ADDRESS_STROBE` | `clock` | Samples the row address of the memory. Applicable for scalar pin. |
| `COLUMN_ADDRESS_STROBE` | `clock` | Samples the column address of the memory. Applicable for scalar pin. |
| `ROW` | `address` | Selects an addressable row of the memory. Applicable for pin and pingroup. |

**Table 56—Attributes for pins of a memory (Continued)**

| Attribute item | SIGNALTYPE | Description |
|---|---|---|
| COLUMN | address | Selects an addressable column of the memory. Applicable for pin and pingroup. |
| BANK | address | Selects an addressable bank of the memory. Applicable for pin and pingroup. |

The attributes shown in Table 57 are applicable for a pair of signals.

**Table 57—Attributes for pins representing pairs of signals**

| Attribute item | Description |
|---|---|
| INVERTED | Represents the inverted value within a pair of signals carrying complementary values. |
| NON_INVERTED | Represents the non-inverted value within a pair of signals carrying complementary values. |
| DIFFERENTIAL | Signal is part of a differential pair, i.e., both the inverted and non-inverted values are always required for physical implementation. |

In case there is more than one pair of signals related to each other by the attribute values *inverted*, *non-inverted*, or *differential*, each pair shall be member of a dedicated pingroup.

The following restrictions apply for pairs of signals:

— The PINTYPE, SIGNALTYPE, and DIRECTION of both pins shall be the same.
— One PIN shall have the attribute INVERTED, the other NON_INVERTED.
— Either both pins or none of the pins shall have the attribute DIFFERENTIAL.
— POLARITY, if applicable, shall be complementary as follows:
  HIGH is paired with LOW
  RISING_EDGE is paired with FALLING_EDGE
  DOUBLE_EDGE is paired with DOUBLE_EDGE

The attribute *inverted*, *non-inverted* also applies to pins of a cell for which the implementation of a pair of signals is optional, i.e., one of the signals can be missing. The output pin of a *flipflop* or a *latch* is an example. The *flip-flop* or the *latch* can have an output pin with attribute *non-inverted* and/or another output pin with attribute *inverted*.

The pin ATTRIBUTE values shown in Table 58 shall be defined for memory BIST.

**Table 58—PIN or PINGROUP attributes for memory BIST**

| Attribute item | Description |
|---|---|
| ROW_INDEX | vector pin or pingroup with a contiguous range of values, indicating a physical row of a memory. |

**Table 58—PIN or PINGROUP attributes for memory BIST (Continued)**

| Attribute item | Description |
|---|---|
| COLUMN_INDEX | vector pin or pingroup with a contiguous range of values, indicating a physical column of a memory. |
| BANK_INDEX | vector pin or pingroup with a contiguous range of values, indicating a physical bank of a memory. |
| DATA_INDEX | vector pin or pingroup with a contiguous range of values, indicating the bit position within a data bus of a memory. |
| DATA_VALUE | scalar pin, representing a value stored in a physical memory location. |

These attributes apply to the virtual pins associated with a BIST wrapper around the memory rather than to the physical pins of the memory itself. The BIST wrapper can be represented as a test statement (see Section 10.2).

## 9.11 PRIMITIVE declaration

A *primitive* shall be declared as shown in Syntax 51.

```
primitive ::=
      PRIMITIVE primitive_identifier { { primitive_item } }
    | PRIMITIVE primitive_identifier ;
    | primitive_template_instantiation
primitive_item ::=
      all_purpose_item
    | pin
    | pingroup
    | function
    | test
```

*Syntax 51—PRIMITIVE statement*

The purpose of a primitive is to describe a virtual circuit. The virtual circuit can be functionally equivalent to a physical electronic circuit represented as a cell (see Section 9.3). A primitive can be instantiated within a behavior statement (see Section 10.4).

## 9.12 WIRE declaration

A *wire* shall be declared as shown in Syntax 52.

```
wire ::=
      WIRE wire_identifier { wire_item { wire_item } }
    | WIRE wire_identifier ;
    | wire_template_instantiation
wire_item ::=
      all_purpose_item
    | node
```

*Syntax 52—WIRE declaration*

The purpose of a wire declaration is to describe an interconnect model. The interconnect model can be a statistical wireload model, a description of boundary parasitics within a complex cell, a model for interconnect analysis, or a specification of a load seen by a driver.

## 9.13 WIRE instantiation

A *wire* shall be instantiated as shown in .

```
wire_instantiation ::=
      wire_identifier instance_identifier ;
    | wire_identifier instance_identifier { pin_value { pin_value } }
    | wire_identifier instance_identifier { pin_assignment { pin_assignment } }
```

*Syntax 53—WIRE instantiation*

## 9.14 Annotations for a WIRE

**Add lead-in text**

### 9.14.1 SELECT_CLASS annotation

A *select_class* annotation shall be defined as shown in Semantics 32.

```
KEYWORD SELECT_CLASS = annotation {
  CONTEXT = WIRE;
  VALUETYPE = identifier;
}
```

*Semantics 32—SELECT_CLASS annotation*

The *identifier* shall refer to the name of a declared class.

The purpose of the select class annotation is to provide a mechanism for selection of an interconnect model by an application. The user of the application can select a set of related interconnect models by specifying the name of the class rather than specifying the name of each interconnect model.

## 9.15 NODE declaration

A *node* shall be declared as shown in Syntax 54.

```
node ::=
      NODE node_identifier ;
    | NODE node_identifier { { node_item } }
    | node_template_instantiation
node_item ::=
    all_purpose_item
```

*Syntax 54—NODE statement*

The purpose of a node declaration is to specify an electrical node in the context of a *wire* declaration (see Section 9.12) or in the context of a *cell* declaration (see Section 9.3).

### 9.15.1 NODETYPE annotation

A *nodetype* annotation shall be defined as shown in Semantics 33.

```
KEYWORD NODETYPE = single_value_annotation {
  CONTEXT = NODE;
  VALUETYPE = identifier;
  VALUES { power ground source sink
    driver receiver interconnect }
}
```

*Semantics 33—NODETYPE annotation*

The *values* shall have the semantic meaning shown in Table 59.

**Table 59—NODETYPE annotation values**

| Annotation value | Description |
|---|---|
| `driver` | The node is the interface between an output pin of a cell and an interconnect wire. |
| `receiver` | The node is the interface between an interconnect wire and an input pin of a cell. |
| `source` | The node is a virtual start point of signal propagation; it can be collapsed with a driver node in case of an ideal driver. |
| `sink` | The node is a virtual end point of signal propagation; it can be collapsed with a receiver node in case of an ideal receiver. |
| `power` | The node supports electrical current for a rising signal at a source or a driver node and a reference for a logic high signal at a sink or receiver side. |
| `ground` | The node supports electrical current for a falling signas at a source or a driver node and a reference for logic a low signal at a sink or a receiver node |
| `interconnect` (default) | The node serves for connecting purpose only. |

### 9.15.2 NODE_CLASS annotation

A *node_class* annotation shall be defined as shown in Semantics 34.

```
KEYWORD NODE_CLASS = annotation {
  CONTEXT = NODE;
  VALUETYPE = identifier;
}
```

*Semantics 34—NODE_CLASS annotation*

The *identifier* shall refer to the name of a declared class.

The purpose of the node class annotation is to associate a node with a virtual cell. The virtual cell is represented by the declared class.

## 9.16 VECTOR declaration

A *vector* shall be declared as shown in Syntax 55.

```
vector ::=
    VECTOR control_expression ;
  | VECTOR control_expression { { vector_item } }
  | vector_template_instantiation
vector_item ::=
    all_purpose_item
```

*Syntax 55—VECTOR statement*

The purpose of a vector is to provide a context for electrical characterization data or for functional test data. The *control expression* (see Section 10.16) specifies a stimulus related to the data.

## 9.17 Annotations for VECTOR

**Add lead-in text**

### 9.17.1 PURPOSE annotation

A *purpose* annotation shall be defined as shown in Semantics 35.

```
KEYWORD PURPOSE = annotation {
  CONTEXT { VECTOR CLASS }
  VALUETYPE = identifier ;
  VALUES { bist test timing power noise reliability }
}
```

*Semantics 35—PURPOSE annotation*

The purpose of the *purpose* annotation is to specify a category for the data found in the context of the vector. The purpose annotation can also be inherited from a class referenced within the context of the vector.

The *values* shall have the semantic meaning shown in Table 61.

**Table 60—PURPOSE annotation values**

| Annotation value | Description |
|---|---|
| `bist` | The vector contains data related to *built-in self test* |
| `test` | The vector contains data related to test requiring external circuitry. |
| `timing` | The vector contains an arithmetic model related to timing calculation (see from Section 11.6 to Section 11.17) |

**Table 60—PURPOSE annotation values (Continued)**

| Annotation value | Description |
|---|---|
| power | The vector contains an arithmetic model related to power calculation (see Section 11.24 ) |
| noise | The vector contains an arithmetic model related to noise calculation (see Section 11.28) |
| reliability | The vector contains an arithmetic model related to reliability calculation (see Section 11.25, also Section 11.6 and Section 11.7) |

### 9.17.2 OPERATION annotation

An *operation* annotation shall be defined as shown in Semantics 36.

```
KEYWORD OPERATION = single_value_annotation {
  CONTEXT = VECTOR;
  VALUETYPE = identifier;
  VALUES {
    read write read_modify_write refresh load
    start end iddq
  }
}
```

*Semantics 36—OPERATION annotation*

The purpose of the operation annotation is to associate a mode of operation of the electronic circuit with the stimulus specified within the vector declaration. This assocation can be used by an application for test vector generation or test vector verification.

The *values* shall have the semantic meaning shown in Table 61.

**Table 61—OPERATION annotation values**

| Annotation value | Description |
|---|---|
| read | Read operation at one address of a memory. |
| write | Write operation at one address of a memory |
| read_modify_write | Read followed by write of different value at same address of a memory |
| start | First operation within a sequence of operations required in a particular mode. |
| end | Last operation within a sequence of operations required in a particular mode. |
| refresh | Operation required to maintain the contents of the memory without modifying it. |

**Table 61—OPERATION annotation values (Continued)**

| Annotation value | Description |
|---|---|
| load | Operation for supplying data to a control register. |
| iddq | Operation for supply current measurements in quiescent state. |

### 9.17.3 LABEL annotation

A *label* annotation shall be defined as shown in Semantics 37.

```
KEYWORD LABEL = single_value_annotation {
  CONTEXT = VECTOR;
  VALUETYPE = string_value;
}
```

*Semantics 37—LABEL annotation*

The purpose of the label annotation is to enable a cross-reference between a statement within the context of a vector and a corresponding statement outside the ALF library. For example, a cross-reference between a delay model in context of a vector (see Section 11.8.1) and an annotated delay within an SDF file [**put reference to IEEE1497 here**] can be established, since the SDF standard also supports a LABEL statement.

### 9.17.4 EXISTENCE_CONDITION annotation

An *existence-condition* annotation shall be defined as shown in Semantics 38.

```
KEYWORD EXISTENCE_CONDITION = single_value_annotation {
  CONTEXT { VECTOR CLASS }
  VALUETYPE = boolean_expression;
  DEFAULT = 1;
}
```

*Semantics 38—EXISTENCE_CONDITION annotation*

The purpose of the existence-condition is to define a necessary and sufficient condition for a vector to be relevant for an application. This condition can also be inherited by the vector from a referenced class. A vector shall be relevant unless the existence-condition evaluates *False*.

The set of pin variables involved in the vector declaration and the set of pin variables involved in the existence condition shall be mutually exclusive.

For dynamic evaluation of the control expression within the vector declaration, the boolean expression within the existence-condition can be treated as if it were a co-factor of the control expression.

### 9.17.5 EXISTENCE_CLASS annotation

An *existence-class* annotation shall be defined as shown in Semantics 39.

The identifier shall be the name of a declared class.

```
KEYWORD EXISTENCE_CLASS = annotation {
  CONTEXT { VECTOR CLASS }
  VALUETYPE = identifier;
}
```

*Semantics 39—EXISTENCE_CLASS annotation*

The purpose of the existence-class annotation is to provide a mechanism for selection of a relevant vector by an application. The user of the application can select a set of relevant vectors by specifying the name of the class. Another purpose is to share a common existence-condition amongst multiple vectors.

### 9.17.6 CHARACTERIZATION_CONDITION annotation

A *characterization-condition* annotation shall be defined as shown in Semantics 40.

```
KEYWORD
CHARACTERIZATION_CONDITION = single_value_annotation {
  CONTEXT { VECTOR CLASS }
  VALUETYPE = boolean_expression;
}
```

*Semantics 40—CHARACTERIZATION_CONDITION annotation*

The purpose of the characterization-condition annotation is to specify a unique condition under which the data in the context of the vector were characterized. The characterization condition is only applicable if the vector declaration eventually in conjunction with an existence-condition allows more than one condition.

The set of pin variables involved in the characterization-condition can overlap with the set of pin variables involved in the vector declaration and/or the existence-condition, as long as the characterization condition is compatible with the vector declaration and eventually with the existence-condition.

The characterization condition shall not be relevant for evaluation of either the vector declaration or the existence condition.

### 9.17.7 CHARACTERIZATION_VECTOR annotation

A *characterization-vector* annotation shall be defined as shown in Semantics 41.

```
KEYWORD CHARACTERIZATION_VECTOR =
single_value_annotation {
  CONTEXT { VECTOR CLASS }
  VALUETYPE = control_expression;
}
```

*Semantics 41—CHARACTERIZATION_VECTOR annotation*

The purpose of a characterization-vector annotation is to specify a complete stimulus for characterization in the case where the vector declaration specifies only a partial stimulus.

The characterization-vector annotation and the characterization-condition annotation shall be mutually exclusive within the context of the same vector.

### 9.17.8 CHARACTERIZATION_CLASS annotation

A *characterization-class* annotation shall be defined as shown in Semantics 42.

```
KEYWORD CHARACTERIZATION_CLASS = annotation {
  CONTEXT { VECTOR CLASS }
  VALUETYPE = identifier;
}
```

*Semantics 42—CHARACTERIZATION_CLASS annotation*

The identifier shall be the name of a declared class.

The purpose of the characterization-class annotation is to provide a mechanism for classification of characterization data. Another purpose is to share a common characterization-condition or a common characterization-vector amongst multiple vectors.

### 9.17.9 MONITOR annotation

A *monitor* annotation shall be defined as shown inSemantics 43 .

```
KEYWORD MONITOR = annotation {
  CONTEXT { VECTOR CLASS }
  VALUETYPE = identifier;
}
```

*Semantics 43—MONITOR annotation*

## 9.18 LAYER declaration

A *layer* shall be declared as shown in Syntax 56.

```
layer ::=
    LAYER layer_identifier ;
  | LAYER layer_identifier { { layer_item } }
  | layer_template_instantiation
layer_item ::=
  all_purpose_item
```

*Syntax 56—LAYER declaration*

A layer shall describe process technology for fabrication of an integrated electronic circuit and a set of related physical data and constraints relevant for a design application.

The order of layer declarations within a library or a sublibrary shall reflect the order of physical creation of layers by a manufacturing process.

## 9.19 Annotations for LAYER

**Add lead-in text**

### 9.19.1 LAYERTYPE annotation

A *layertype* annotation shall be defined as shown in Semantics 44.

```
KEYWORD LAYERTYPE = single_value_annotation {
  CONTEXT = LAYER;
  VALUETYPE = identifier;
  VALUES {
    routing cut substrate dielectric reserved abstract
  }
}
```

*Semantics 44—LAYERTYPE annotation*

The values shall have the semantic meaning shown in Table 62.

**Table 62—LAYERTYPE annotation values**

| Annotation value | Description |
|---|---|
| routing | Layer provides electrical connections within a plane. |
| cut | Layer provides electrical connections between planes. |
| substrate | Layer at the bottom. |
| dielectric | Layer provides electrical isolation between planes. |
| reserved | Layer is for proprietary use only. |
| abstract | Layer is virtual, not manufacturable. |

### 9.19.2 PITCH annotation

A *pitch* annotation shall be defined as shown in Semantics 45.

```
KEYWORD PITCH = single_value_annotation {
  CONTEXT = LAYER;
  VALUETYPE = unsigned_number;
}
```

*Semantics 45—PITCH annotation*

The purpose of the pitch annotation is specification of the normative distance between parallel wire segments within a layer with layertype value *routing*. This distance is measured between the center of two adjacent parallel wires.

### 9.19.3 PREFERENCE annotation

A *preference* annotation shall be defined as shown in Semantics 46.

```
KEYWORD PREFERENCE = single_value_annotation {
  CONTEXT = LAYER;
  VALUETYPE = identifier;
  VALUES { horizontal vertical acute obtuse }
}
```

*Semantics 46—PREFERENCE annotation*

The purpose is to indicate the prefered routing direction for wires within a layer with *layertype* value *routing*.

The values shall have the semantic meaning shown in Table 62.

**Table 63—PREFERENCE annotation values**

| Annotation value | Description |
|---|---|
| `horizontal` | Prefered routing direction is horizontal, i.e., 0 degrees. |
| `vertical` | Prefered routing direction is vertical, i.e., 90 degrees. |
| `acute` | Prefered routing direction is 45 degrees. |
| `obtuse` | Prefered routing direction is 135 degrees. |

## 9.20 VIA declaration

A *via* shall be declared as shown in Syntax 57.

```
via ::=
    VIA via_identifier ;
  | VIA via_identifier { { via_item } }
  | via_template_instantiation
via_item ::=
    all_purpose_item
  | pattern
  | artwork
```

*Syntax 57—VIA statement*

A via shall describe a stack of physical artwork for electrical connection between wire segments on different layers.

## 9.21 VIA instantiation

A *via* shall be instantiated as shown in Syntax 58.

The purpose of a via instantiation is to define a design *rule* involving a via (see Section 9.23 ), to describe details of a physical *blockage* (see Section 9.25) or details of a physical *port* (see Section 9.26).

```
via_instantiation ::=
    via_identifier instance_identifier ;
  | via_identifier instance_identifier { { geometric_transformation } }
```

*Syntax 58—VIA instantiation*

## 9.22 Annotations for a VIA

**Add lead-in text**

### 9.22.1 VIATYPE annotation

**Single subheader**

A *viatype* annotation shall be defined as shown in Semantics 47.

```
KEYWORD VIATYPE = single_value_annotation {
  CONTEXT = VIA;
  VALUETYPE = identifier;
  VALUES { default non_default partial_stack full_stack }
  DEFAULT = default;
}
```

*Semantics 47—VIATYPE annotation*

The *values* shall have the semantic meaning shown in Table 64.

**Table 64—VIATYPE annotation values**

| Annotation value | Description |
|---|---|
| `default` | `via` can be used per default. |
| `non_default` | `via` can only be used if authorized by a `RULE`. |
| `partial_stack` | `via` contains three patterns: the lower and upper routing layer and the cut layer in-between. This can only be used to build stacked vias. The bottom of a stack can be a `default` or a `non_default via`. |
| `full_stack` | `via` contains `2N+1` patterns (`N>1`). It describes the full stack from bottom to top. |

## 9.23 RULE declaration

A *rule* shall be declared as shown in Syntax 59.

A rule declaration shall be used to define electrical or physical constraints involving physical objects. A physical object shall be described as a *pattern* (see Section 9.32), a *region* (see Section 9.34), or a *via instantiation* (see Section 9.21). The contraints shall be described as arithmetic models.

```
rule ::=
      RULE rule_identifier ;
    | RULE rule_identifier { { rule_item } }
    | rule_template_instantiation
rule_item ::=
      all_purpose_item
    | pattern
    | region
    | via_instantiation
```

*Syntax 59—RULE statement*

## 9.24 ANTENNA declaration

An *antenna* shall be declared as shown in Syntax 60.

```
antenna ::=
      ANTENNA antenna_identifier ;
    | ANTENNA antenna_identifier { { antenna_item } }
    | antenna_template_instantiation
antenna_item ::=
      all_purpose_item
    | region
```

*Syntax 60—ANTENNA declaration*

An antenna declaration shall be used to define manufacturability constraints involving physical objects or *regions* (see Section 9.34) created by physical objects. The physical objects shall be associated with a *layer* (see Section 9.18). Within the context of an antenna declaration, arithmetic models for *size* (see Section 11.31), *area* (see Section 11.32), *perimeter* (see Section 11.38) associated with a layer or with a region can be described. The arithmetic models can be combined, based on electrical *connectivity* (see Section 11.30) between the layers.

To evaluate connectivity in the context of an antenna declaration, the order of manufacturing given by the order of layer declarations shall be relevant. An object on a layer shall only be considered electrically connected to an object on another layer, if the connection already exists when the uppermost layer of both layers is manufactured. This is illustrated in the following figure 8.

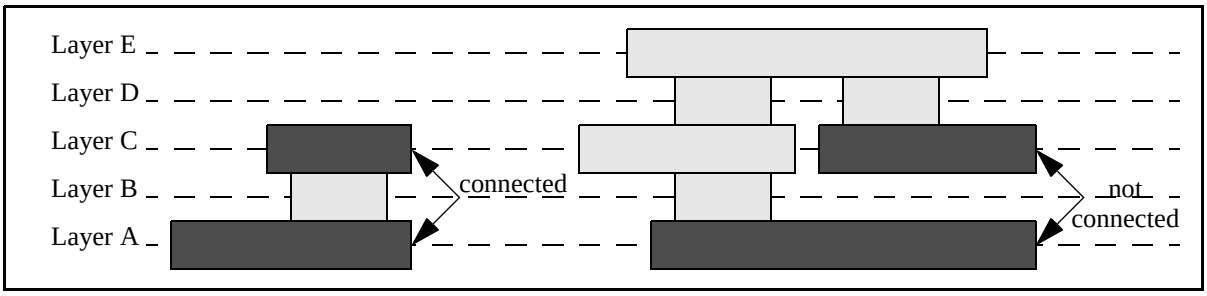



**Figure 8—Connection between layers during manufacturing**

The dark objects on layer A and layer C on the left side of figure 8 are considered connected, because the connection is established through layer B which exists already when layer C is manufactured.

The dark objects on layer A and layer C on the right hand side of figure 8 are not considered connected, because the connection involves layer D and E which do not yet exist when layer C is manufactured.

## 9.25 BLOCKAGE declaration

A *blockage* shall be declared as shown in Syntax 61.

```
blockage ::=
      BLOCKAGE blockage_identifier ;
    | BLOCKAGE blockage_identifier { { blockage_item } }
    | blockage_template_instantiation
blockage_item ::=
      all_purpose_item
    | pattern
    | region
    | rule
    | via_instantiation
```

*Syntax 61—BLOCKAGE statement*

A blockage declaration shall be used in context of a *cell* (see Section 9.3) to describe a part of the physical artwork of the cell. No short circuit shall be created between the physical artwork described by the blockage and a physical artwork created by an application. Physical or electrical constraints involving a blockage can be described by a *rule* (see Section 9.23). A rule within the context of a blockage shall only be applicable for physical objects within the blockage in relation to their environment. The physical objects within the blockage can also be subjected to a more general rule.

## 9.26 PORT declaration

A *port* shall be declared as shown in Syntax 62.

```
port ::=
      PORT port_identifier ;{ { port_item } }
    | PORT port_identifier ;
    | port_template_instantiation
port_item ::=
      all_purpose_item
    | pattern
    | region
    | rule
    | via_instantiation
```

*Syntax 62—PORT declaration*

A port declaration shall be used in context of a *scalar pin* (see Section 9.7) to describe a part of the physical artwork of a cell (see Section 9.3) provided to establish electrical connection between a pin and its environment. Physical or electrical constraints involving a port can be described by a *rule* (see Section 9.23). A rule within the context of a port shall only be applicable for physical objects within the blockage in relation to their environment. The physical objects within the port can also be subjected to a more general rule.

## 9.27 Annotations for PORT

**Add lead-in text**

### 9.27.1 PORT_VIEW annotation

**Single subheader**

A *port_view* annotation shall be defined as shown in Semantics 48.

```
KEYWORD PORT_VIEW = single_value_annotation {
  CONTEXT = PORT;
  VALUETYPE = identifier;
  VALUES { physical electrical both none }
  DEFAULT = both;
}
```

*Semantics 48—PORT_VIEW annotation*

The *values* shall have the semantic meaning shown in Table 65.

**Table 65—PORT_VIEW annotation values**

| Annotation value | Description |
|---|---|
| `physical` | A port for layout with the possibility to connect a routing wire. |
| `electrical` | A port in an electrical netlist (SPEF, SPICE). |
| `both` | Both of the above. |
| `none` | A virtual port for modeling purpose only. |

## 9.28 SITE declaration

A *site* shall be declared as shown in Syntax 63.

```
site ::=
      SITE site_identifier ;
    | SITE site_identifier { { site_item } }
    | site_template_instantiation
site_item ::=
      all_purpose_item
    | WIDTH_arithmetic_model
    | HEIGHT_arithmetic_model
```

*Syntax 63—SITE declaration*

A site declaration shall be used to specify a legal placement location for a cell.

## 9.29 Annotations for SITE

**Add lead-in text**

### 9.29.1 ORIENTATION_CLASS annotation

An *orientation_class* annotation shall be defined as shown in Semantics 49.

```
KEYWORD ORIENTATION_CLASS = annotation {
  CONTEXT { SITE CELL }
  VALUETYPE = IDENTIFIER;
}
```

*Semantics 49—ORIENTATION_CLASS annotation*

### 9.29.2 SYMMETRY_CLASS annotation

A *symmetry_class* annotation shall be defined as shown in Semantics 50.

```
KEYWORD SYMMETRY_CLASS = annotation {
  CONTEXT { SITE CELL }
  VALUETYPE = identifier;
}
```

*Semantics 50—SYMMETRY_CLASS annotation*

The SYMMETRY_CLASS statement shall be used for a SITE to indicate symmetry between legal orientations. Multiple SYMMETRY statements shall be legal to enumerate all possible combinations in case they cannot be described within a single SYMMETRY statement.

Legal orientation of a cell within a site shall be defined as the intersection of legal cell orientation and legal site orientation. If there is a set of common legal orientations for both cell and site without symmetry, the orientation of cell instance and site instance shall match.

If there is a set of common legal orientations for both cell and site with symmetry, the cell can be placed on the side using any orientation within that set.

*Example*

*Case 1*: no symmetry

The site has legal orientations A and B. The cell has legal orientations A and B. When the site appears in orientation A, the cell shall be placed in orientation A. When the site appears in orientation B, the cell shall be placed in orientation B.

*Case 2*: symmetry

The site has legal orientations A and B and symmetry between A and B. The cell has legal orientations A and B. When the site appears in orientation A, the cell can be placed in orientation A or B. When the site appears in orientation B, the cell can also be placed in orientation A or B.

## 9.30 ARRAY declaration

An array shall be declared as shown in Syntax 64.

An array declaration shall be used for the purpose to describe a grid for creating physical objects within design. The geometric transformations *shift* and *repeat* (see Section 9.37) shall be used to define the construction rule for the array. The shift statement shall define the offset between the origin of the basic element within the array and the origin of its context. The repeat statement shall define, how the basic element is replicated.

```
array ::=
      ARRAY array_identifier ;
    | ARRAY array_identifier { { array_item } }
    | array_template_instantiation
array_item ::=
      all_purpose_item
    | geometric_transformation
```

*Syntax 64—ARRAY statement*

## 9.31 Annotations for ARRAY

**Add lead-in text**

### 9.31.1 ARRAYTYPE annotation

An *arraytype* annotation shall be defined as shown in Semantics 51.

```
KEYWORD ARRAYTYPE = single_value_annotation {
  CONTEXT = ARRAY;
  VALUETYPE = identifier;
  VALUES { floorplan placement
    global_routing detailed_routing }
}
```

*Semantics 51—ARRAYTYPE annotation*

The *values* shall have the semantic meaning shown in Table 66.

**Table 66—ARRAYTYPE annotation values**

| Annotation value | Description |
|---|---|
| `floorplan` | The array provides a grid for placing macrocells, i.e., cells with *celltype* value can be *block* or *core* or *memory.*<br>The *placement_type* value shall be *core*. |
| `placement` | The array provides a grid for placing regular cells, i.e., cells with *celltype* value *buffer, combinational, multiplexor, latch, flipflop* or *special*.<br>The *placement_type* value shall be *core*. |
| `global_routing` | The array provides a grid for global routing. |
| `detailed_routing` | The array provides a grid for global routing. |

### 9.31.2 SITE reference annotation

A *site* reference annotation shall be defined as shown in Semantics 52.

The purpose of a site reference annotation is to establish a relation between a *cell* (see Section 9.3) and a *site* (see Section 9.28) or between a site and an array. The site reference annotation in context of a cell shall indicate

```
SEMANTICS SITE = single_value_annotation {
  CONTEXT { ARRAY CELL }
  VALUETYPE = identifier;
}
```

*Semantics 52—SITE reference annotation*

whether the site represents a legal placement location for the cell. The site reference annotation in context of an array shall indicate that the site is the basic element from which the array is constructed.

The site reference annotation is applicable for an array with *arraytype* value *floorplan* or *placement*.

### 9.31.3 LAYER reference annotation

A *layer* reference annotation in the context of an *array* shall be defined as shown in Semantics 53.

```
SEMANTICS ARRAY.LAYER = annotation {
  VALUETYPE = identifier;
}
```

*Semantics 53—LAYER reference annotation for ARRAY*

The layer reference annotation is applicable for an array with *arraytype* value *detailed routing*. It shall specify the applicable *layer* (see Section 9.18) with *layertype* value *routing*.

## 9.32 PATTERN declaration

A *pattern* shall be declared as shown in Syntax 65.

pattern ::=
  **PATTERN** *pattern*_identifier **;**
| **PATTERN** *pattern*_identifier **{** { pattern_item } **}**
| *pattern*_template_instantiation
pattern_item ::=
  all_purpose_item
| geometric_model
| geometric_transformation

*Syntax 65—PATTERN declaration*

The pattern declaration shall be used to describe a physical object associated with a *layer* (see Section 9.18).

## 9.33 Annotations for PATTERN

**Add lead-in text**

### 9.33.1 LAYER reference annotation

A *layer* reference annotation in the context of a *pattern* shall be defined as shown in.

```
SEMANTICS PATTERN.LAYER = single_value_annotation {
  VALUETYPE = identifier;
}
```

*Semantics 54—LAYER reference annotation for PATTERN*

The layer reference annotation shall establish an association between a pattern and a *layer* (see Section 9.18). The physical object represented by the pattern shall reside on a layer. A pattern declaration without layer reference annotation shall be considered incomplete.

### 9.33.2 SHAPE annotation

A *shape* annotation shall be defined as shown in Semantics 55.

```
KEYWORD SHAPE = single_value_annotation {
  CONTEXT = PATTERN;
  VALUETYPE = identifier;
  VALUES { line tee cross jog corner end }
  DEFAULT = line;
}
```

*Semantics 55—SHAPE annotation*

The shape annotation applies for a pattern associated with a layer with *layertype* value *routing*. The meaning of the shape annotation values is illustrated in Figure 9.

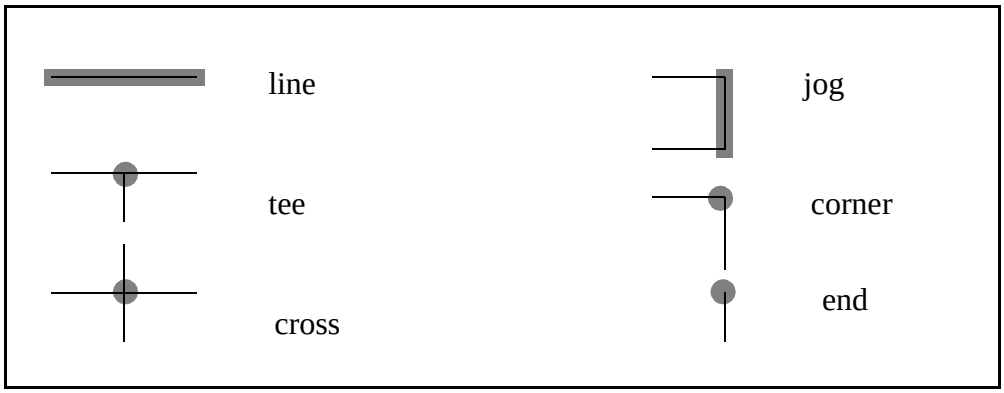



**Figure 9—Shapes of routing patterns**

The annotation values *line* and *jog* shall represent a *routing segment*. The annotation values tee, cross, and corner shall represent an intersection between routing segments. The annotation value *end* shall represent the open end point of an unterminated routing segment.

### 9.33.3 VERTEX annotation

A *vertex* annotation shall be defined as shown in Semantics 56.

The vertex annotation applies for a pattern in conjunction with the *shape* annotation. The meaning of the vertex annotation values is illustrated Figure 10.

```
KEYWORD VERTEX = single_value_annotation {
  CONTEXT = PATTERN;
  VALUETYPE = identifier;
  VALUES { round angular }
  DEFAULT = angular;
}
```

*Semantics 56—VERTEX annotation*

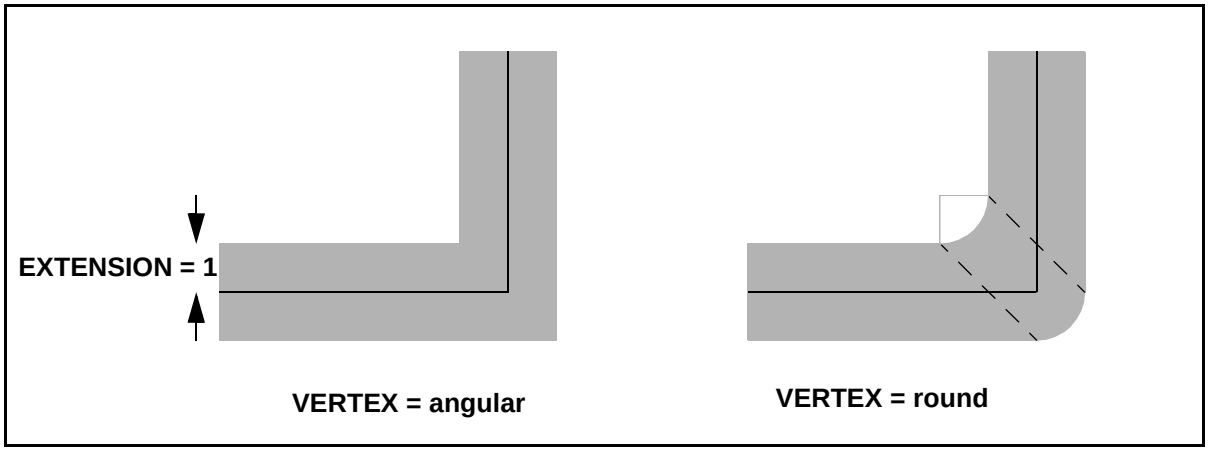



**Figure 10—Illustration of VERTEX annotation**

### 9.33.4 ROUTE annotation

A *route* annotation shall be defined as shown in .

```
KEYWORD ROUTE = single_value_annotation {
  CONTEXT = PATTERN;
  VALUETYPE = identifier;
  VALUES { horizontal acute vertical obtuse }
}
```

*Semantics 57—ROUTE annotation*

## 9.34 REGION declaration

A *region* object shall be declared as shown in .

region ::=
    **REGION** *region_name*_identifier **;**
  | **REGION** *region_name*_identifier **{** { region_item } **}**
region_item ::=
    all_purpose_item
  | geometric_model
  | geometric_transformation
  | *BOOLEAN*_single_value_annotation

*Syntax 66—ROUTE declaration*

### 9.34.1 BOOLEAN annotation

A *boolean* annotation shall be defined as shown in .

```
KEYWORD BOOLEAN = single_value_annotation {
  CONTEXT = REGION ;
  VALUETYPE = boolean_expression ;
}
```

*Semantics 58—BOOLEAN annotation*

## 9.35 Geometric model

A *geometric model* shall be defined as shown in Syntax 67.

```
geometric_model ::=
    nonescaped_identifier [ geometric_model_identifier ]
      { geometric_model_item { geometric_model_item } }
  | geometric_model_template_instantiation
geometric_model_item ::=
    POINT_TO_POINT_single_value_annotation
  | coordinates
coordinates ::=
  COORDINATES { point { point } }
point ::=
  x_number y_number
```

*Syntax 67—Geometric model*

A geometric model shall describe the form of a physical object. A geometric model can appear in the context of a *pattern* (see Section 9.32) or a *region* (see Section 9.34).

The numbers in the *point* statement shall be measured in units of *distance* (see Section 11.36).

The parent object of the geometric model can contain a *geometric transformation* (see Section 9.37) applicable to the geometric model.

Table 67 specifiies the meaning of predefined geometric model identifiers.

**Table 67—Geometric model identifiers**

| Identifier | Description |
|---|---|
| DOT | Describes one point. |
| POLYLINE | Defined by N>1 directly connected points, forming an open object. |

**Table 67—Geometric model identifiers (Continued)**

| Identifier | Description |
|---|---|
| RING | Defined by N>1 directly connected points, forming a closed object, i.e., the last point is connected with first point. The object occupies the boundary of the enclosed space. |
| POLYGON | Defined by N>1 connected points, forming a closed object, i.e., the last point is connected with first point. The object occupies the entire enclosed space. |

The meaning of predefined geometric model identifiers is further illustrated in Figure 11.

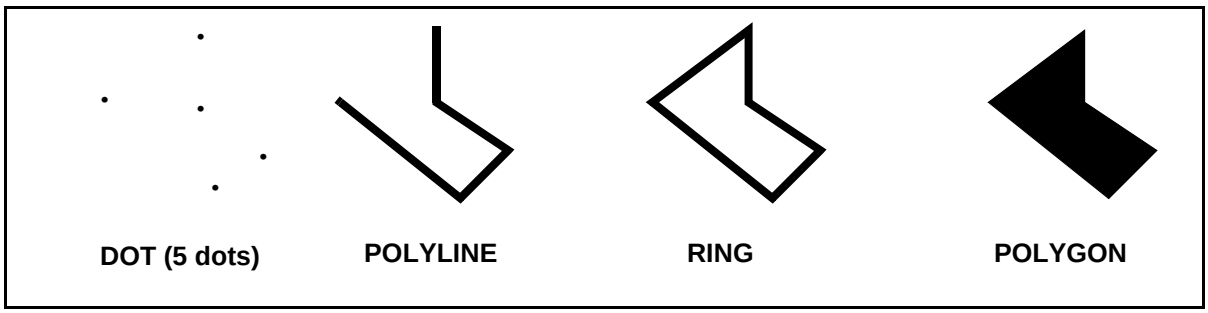



**Figure 11—Illustration of geometric models**

A *point_to_point* annotation shall be defined as shown in Semantics 59.

```
KEYWORD POINT_TO_POINT = single_value_annotation {
  CONTEXT { POLYLINE RING POLYGON }
  VALUETYPE = identifier;
  VALUES { direct manhattan }
  DEFAULT = direct;
}
```

*Semantics 59—POINT_TO_POINT annotation*

The point-to-point annotation applies for a *polyline*, a *ring* or a *polygon*. The annotation value specifies, how subsequent points in the *coordinates* statement are to be connected.

The meaning of the annotation value *direct* is illustrated in Figure 12. It specifies the shortest possible connection between points.

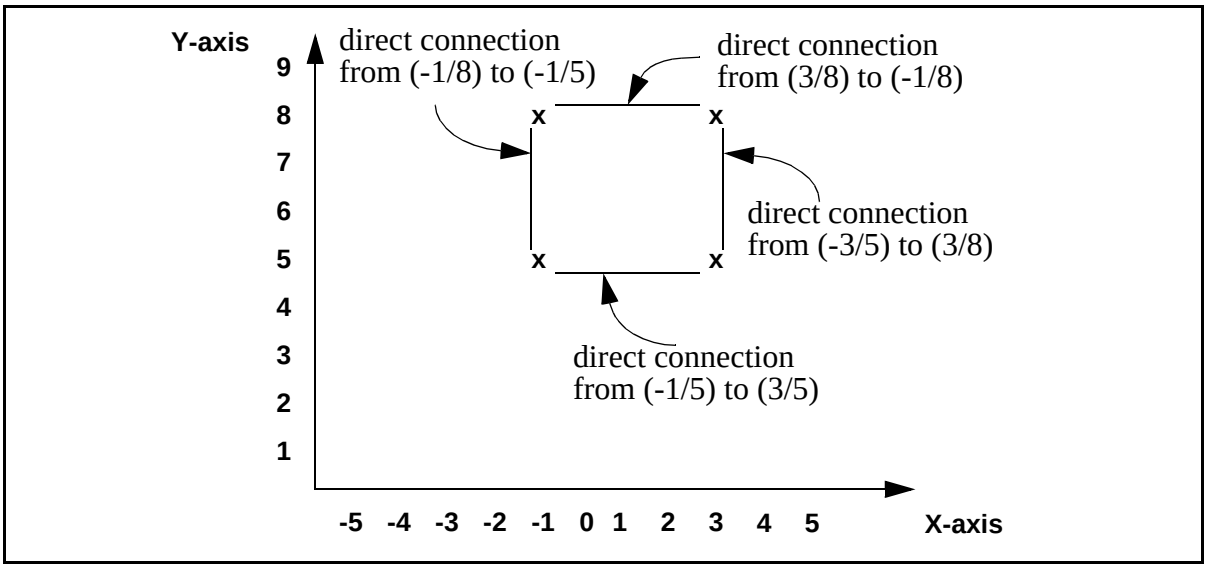



**Figure 12—Illustration of direct point-to-point connection**

The meaning of the annotation value *manhattan* is illustrated in Figure 13. It specifies a connection between points by moving in the x-direction first and then moving in the y-direction. This enables a non-redundant specification of a rectilinear object using `N/2` points instead of `N` points.

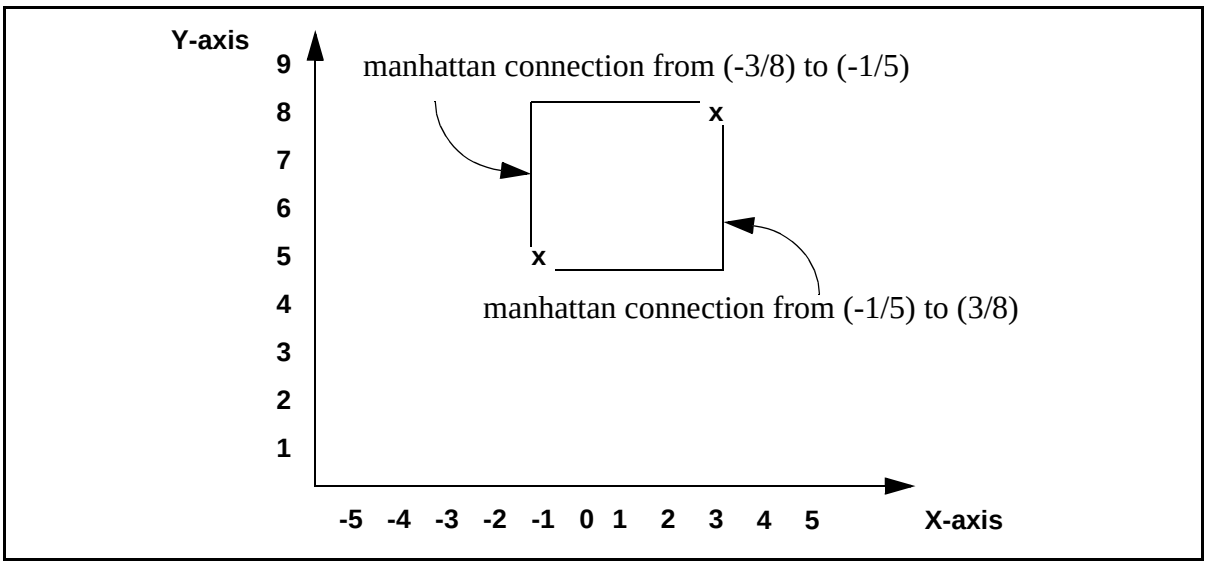



**Figure 13—Illustration of manhattan point-to-point connection**

*Example*

```
POLYGON {
    POINT_TO_POINT = direct;
    COORDINATES { -1 5 3 5 3 8 -1 8 }
}
```

```
POLYGON {
    POINT_TO_POINT = manhattan;
    COORDINATES { -1 5 3 8 }
}
```

Both objects describe the same rectangle.

## 9.36 Predefined geometric models using TEMPLATE

A *template* declaration (see Section 8.8) can be used to predefine particular geometric moddels.

The templates RECTANGLE and LINE shall be predefined as follows:

```
TEMPLATE RECTANGLE {
    POLYGON {
       POINT_TO_POINT = manhattan;
       COORDINATES { <left> <bottom> <right> <top> }
    }
}
TEMPLATE LINE {
    POLYLINE {
       POINT_TO_POINT = direct;
       COORDINATES { <x_start> <y_start> <x_end> <y_end> }
    }
}
```

*Example 1*

The following example shows the usage of the predefined templates *rectangle* and *line*.

```
// same rectangle as in previous example
RECTANGLE {left = -1; bottom = 5; right = 3; top = 8; }
//or
RECTANGLE {-1 5 3 8 }

// diagonals through the rectangle
LINE {x_start = -1; y_start = 5; x_end = 3; y_end = 8; }
LINE {x_start = 3; y_start = 5; x_end = -1; y_end = 8; }
//or
LINE { -1 5 3 8 }
LINE { 3 5 -1 8 }
```

*Example 2*

The following example shows user-defined template declarations.

```
TEMPLATE HORIZONTAL_LINE {
    POLYLINE {
       POINT_TO_POINT = direct;
       COORDINATES { <left> <y> <right> <y> }
    }
}
```

```
TEMPLATE VERTICAL_LINE {
    POLYLINE {
       POINT_TO_POINT = direct;
       COORDINATES { <x> <bottom> <x> <top> }
    }
}
```

*Example 3*

The following example shows the usage of the user-defined templates from Example 2.

```
// lines bounding the rectangle
HORIZONTAL_LINE { y = 5; left = -1; right = 3; }
HORIZONTAL_LINE { y = 8; left = -1; right = 3; }
VERTICAL_LINE { x = -1; bottom = 5; top = 8; }
VERTICAL_LINE { x = 3; bottom = 5; top = 8; }
//or
HORIZONTAL_LINE { 5 -1 3 }
HORIZONTAL_LINE { 8 -1 3 }
VERTICAL_LINE { -1 5 8 }
VERTICAL_LINE { 3 5 8 }
```

## 9.37 Geometric transformation

A *geometric transformation* shall be defined as shown in Syntax 68.

geometric_transformation ::=
  shift
  | rotate
  | flip
  | repeat
shift ::=
  **SHIFT {** *x*_number *y*_number **}**
rotate ::=
  **ROTATE =** number **;**
flip ::=
  **FLIP =** number **;**
repeat ::=
  **REPEAT** [ **=** unsigned_integer ] **{** geometric_transformation { geometric_transformation } **}**

*Syntax 68—Geometric transformation*

The SHIFT statement defines the horizontal and vertical offset measured between the coordinates of the geometric model and the actual placement of the object. Eventually, a layout tool only supports integer numbers. The numbers are in units of DISTANCE. If the SHIFT statement is not defined, both values default to 0.

The ROTATE statement defines the angle of rotation in degrees measured between the orientation of the object described by the coordinates of the geometric model and the actual placement of the object measured in counterclockwise direction, specified by a number between 0 and 360. Eventually, a layout tool can only support angles which are multiple of 90 degrees. The default is 0. The object shall rotate around its origin.

The FLIP describes a transformation of the specified coordinates by flipping the object around an axis specified by a number between 0 and 180. The number represents the angle of the flipping direction in degrees. Eventually, a layout tool can only support angles which are multiple of 90 degrees. The axis is orthogonal to the flipping

direction. The axis shall go through the origin of the object. For example, `0` means flip in horizontal direction, axis is vertical whereas `90` means flip in vertical direction, axis is horizontal.

The purpose of the `REPEAT` statement is to describe the replication of a physical object in a regular way, for example `SITE` (see 9.28). The `REPEAT` statement can also appear within a `geometric_model`. The `unsigned` number defines the total number of replications. The number `1` means, the object appears just once. If this number is not given, the `REPEAT` statement defines a rule for an arbitrary number of replications. `REPEAT` statements can also be nested.

*Examples*

The following example replicates an object three times along the horizontal axis in a distance of `7` units.

```
REPEAT = 3 {
    SHIFT { 7 0 }
}
```

The following example replicates an object five times along a 45-degree axis in a distance of 4 units.

```
REPEAT = 5 {
    SHIFT { 4 4 }
}
```

The following example replicates an object two times along the horizontal axis and four times along the vertical axis in a horizontal distance of 5 units and a vertical distance of 6 units.

```
REPEAT = 2 {
    SHIFT { 5 0 }
    REPEAT = 4 {
       SHIFT { 0 6 }
    }
}
```

NOTE—The order of nested `REPEAT` statements does not matter. The following example gives the same result as the previous example.

```
REPEAT = 4 {
    SHIFT { 0 6 }
    REPEAT = 2 {
       SHIFT { 5 0 }
    }
}
```

Rules and restrictions:

- A physical object can contain a `geometric_transformation` statement of any kind, but no more than one of a specific kind.
- The `geometric_transformation` statements shall apply to all `geometric_models` within the context of the object.
- The `geometric_transformation` statements shall refer to the origin of the object, i.e., the point with coordinates `{ 0 0 }`. Therefore, the result of a combined transformation shall be independent of the order in which each individual transformation is applied.

These are demonstrated in Figure 14.

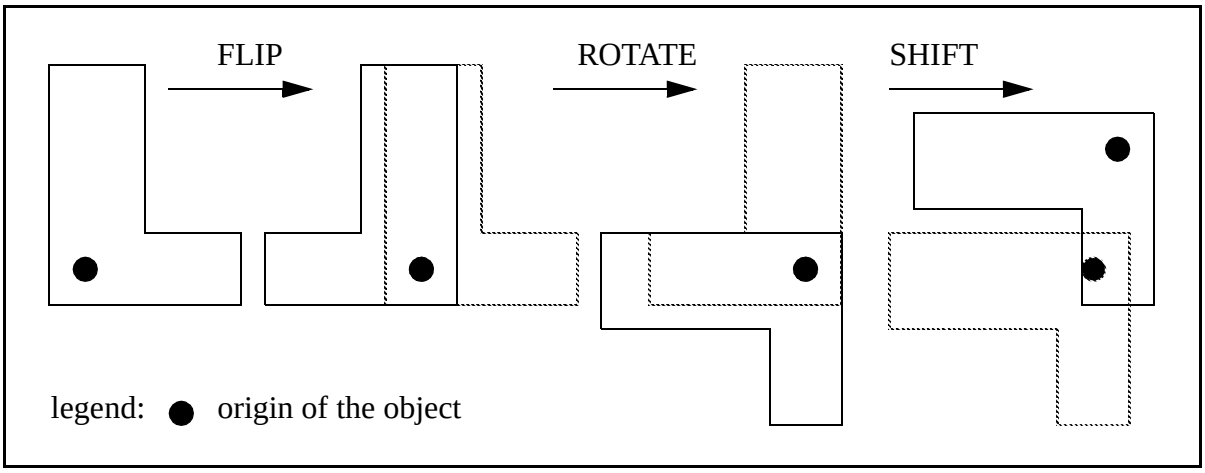



**Figure 14—Illustration of FLIP, ROTATE, and SHIFT**

## 9.38 ARTWORK statement

An *artwork* statement shall be defined as shown in Syntax 69.

```
artwork ::=
      ARTWORK = artwork_identifier ;
    | ARTWORK = artwork_identifier { { artwork_item } }
    | artwork_template_instantiation
artwork_item ::=
      geometric_transformation
    | pin_assignment
```

*Syntax 69—ARTWORK statement*

The `ARTWORK` statement creates a reference between the cell in the library and the original cell imported from a physical layout database, e.g., GDSII [**put reference to GDSII here**].

The `geometric_transformations` define the operations for transformation from the artwork geometry to the actual cell geometry. In other words, the artwork is considered as the original object whereas the cell is the transformed object.

The imported cell can have pins with different names. The LHS of the `pin_assignment` describes the pin names of the original cell, the RHS describes the pin names of the cell in this library. See 7.10 for the syntax of `pin_assignment`.

*Example*

```
CELL my_cell {
    PIN A { /* fill in pin items */ }
    PIN Z { /* fill in pin items */ }
    ARTWORK = \GDS2$!@#$ {
       SHIFT { HORIZONTAL = 0; VERTICAL = 0; }
       ROTATE = 0;
       \GDS2$!@#$A = A;
       \GDS2$!@#$B = B;
```

```
        }
    }
```

# 10. Constructs for modeling of functional behavior

**Add lead-in text**

## 10.1 FUNCTION statement

A *function* statement shall be defined as shown in Syntax 70.

```
function ::=
      FUNCTION { function_item { function_item } }
    | function_template_instantiation
function_item ::=
      all_purpose_item
    | behavior
    | structure
    | statetable
```

*Syntax 70—FUNCTION statement*

The purpose of the function statement is to describe a canonical specification of a digital electronic circuit implemented by a cell. A cell can contain at most one function statement.

The function statement can contain a *behavior* statement (see Section 10.4) or a set of one or more *statetable* statements (see Section 10.6). The purpose of the behavior and statetable statements in this context is to formally specify the logic state of a cell as a response to a given stimulus.

The function statement can also contain a specification for implementation using the *structure* statement (see Section 10.5).

## 10.2 TEST statement

A *test* statement shall be defined as shown in Syntax 71.

```
test ::=
      TEST { test_item { test_item } }
    | test_template_instantiation
test_item ::=
      all_purpose_item
    | behavior
    | statetable
```

*Syntax 71—TEST statement*

The purpose of the test statement is to describe the interface between a cell and a test algorithm applied to the cell. A cell can contain at most one test statement.

The test statement can contain a *behavior* statement (see Section 10.4) or a set of one or more *statetable* statements (see Section 10.6). The purpose of the behavior and statetable statements in this context is to model the interface between a cell and a test algorithm as a virtual digital circuit.

A test algorithm consists of a virtual input pattern and a virtual expected output pattern. The test statement does not specify the test algorithm per se, but the mapping of the virtual pattern into a stimulus applicable to the device under test, i.e., the cell. This is further explained in Section 10.3.

## 10.3 Declaration of pin variables

Both the variables involved in the test statement and the signals involved in the function statement shall be considered as *pin variables* (see Section 7.9).

Pin variables shall be declared as pins or pingroups of the cell with *pintype* annotation value *digital*. The annotation values for *direction* and *view* shall specify whether a pin can be used as a signal for function or as a variable for test, according to the following Table 68.

**Table 68—Annotations for PINs involved in FUNCTION and TEST**

| category | DIRECTION | VIEW |
|---|---|---|
| input signal for function | *input* | *functional* or *both* |
| output signal for function | *output* | *functional* or *both* |
| bidirectional signal for function | *both* | *functional* or *both* |
| internal signal for function | *none* | *none* |
| primary input variable for test | *input* | *none* |
| primary output variable for test | *output* | *none* |
| primary bidirectional variable for test | *both* | *none* |
| internal variable for test | *none* | *none* |

An pin attribute value can be used to specify a test method related to a variable. See Table 58, “PIN or PINGROUP attributes for memory BIST,” for specification of a particular test method.

A *primary input variable* for the test statement can hold a state of a virtual input pattern. A *primary output variable* for the test statemen can hold the state of a virtual expected output pattern. A *primary bidirectional variable* for the test statement can hold the state of a virtual input or output pattern, depending on the mode of the test algorithm. An *internal variable* for the test statement communicates neither with the test algorithm nor with the device under test.

An *input signal* of the cell can be *controlled* or *non-controlled* by the test algorithm. An *output signal* of the cell can be *observed* or *non-observed* by the test algorithm. A *bidirectional signal* of the cell can be controlled or non-controlled in input mode and observed or non-observed in output mode. An *internal signal* of the cell communicates neither with the test algorithm nor with the environment of the cell.

The relationship between pin variables involved in the test statement and in the function statement is illustrated in the following figure 15. The information flow depicted therein shall be established by a *behavior* statement (see Section 10.4) and/or by a set of *statetable* statements (see Section 10.6).

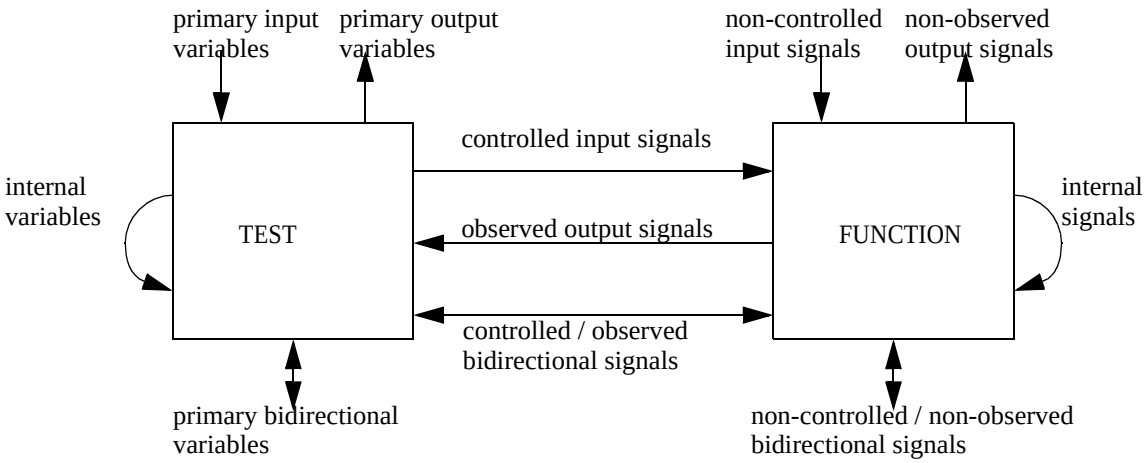



**Figure 15—Relationship between FUNCTION and TEST**

## 10.4 BEHAVIOR statement

A *behavior* statement shall be defined as shown in Syntax 72.

```
behavior ::=
     BEHAVIOR { behavior_item { behavior_item } }
   | behavior_template_instantiation
behavior_item ::=
      boolean_assignment
   | control_statement
   | primitive_instantiation
   | behavior_item_template_instantiation
boolean_assignment ::=
   pin_variable = boolean_expression ;
control_statement ::=
   primary_control_statement { alternative_control_statement }
primary_control_statement ::=
   @ control_expression { boolean_assignment { boolean_assignment } }
alternative_control_statement ::=
   : control_expression { boolean_assignment { boolean_assignment } }
control_expression ::=
     ( vector_expression )
   | ( boolean_expression )
primitive_instantiation ::=
     primitive_identifier [ identifier ] { pin_value { pin_value } }
   | primitive_identifier [ identifier ] { boolean_assignment { boolean_assignment } }
```

*Syntax 72—BEHAVIOR statement*

A *control statement* consists of a *primary control statement*, optionally followed by one or more *alternative control statements*. A *primary control statement* is identified by the *at* character followed by a *control expression*. An alternative control statement is identified by the *colon* character followed by a *control expression*. A control expression can be either a *boolean expression* (see Section 10.9) or a *vector expression* (see Section 10.12). The order of alternativs control statements shall specify the order of priority. If the main control statement does not evaluate true, the first alternative control statement is evaluated. If an alternative control statement does not evaluate true, the next alternative control statement is evaluated.

A *boolean assignment* assigns the evaluation result of a *boolean expression* to a *pin variable* (see Section 7.9). A boolean assignment with a behavior statement as a parent shall be considered a *continuous assignment*, i.e. the boolean expression is evaluated continuously.

A boolean assignment with a control statement as parent shall be considered a *conditional assignment*, i.e., the boolean expression is only evaluated when the associated control expression evaluates true. When a boolean expression is not evaluated, a pin variable shall hold its previously assigned value.

If the control expression is a boolean expression, the conditional assignment shall be called *level-sensitive* or *triggered by state*. If the control expression is a vector expression, the conditional assignment shall be called *edge-sensitive* or *triggered by event*.

A *primitive instantiation* establishes a reference to a predefined function statement within a *primitive* declaration (see Section 9.11). A continuous assignment of a boolean expression to a pin variable can be given by a boolean assignment within the primitive instantiation, wherein the pin variable shall be a declared pin within the primitive declaration. Alternatively, a continuous assignment of a pin value to a pin variable can be given by a set of pin values, wherein the order of pin values shall correspond to the order of pin declarations within the primitive declaration.

A *behavior item* is further subjected to the following rules:

a) An information flow graph involving one or more continuous assignments and/or level-sensitive conditional assignments can not contain a loop. The usage of a pin with *direction* annotation value *both* as a primary input and as a primary output in an information flow graph shall not be considered as a loop.
b) An information flow graph involving one or more edge-sensitive conditional assignments can contain a loop. The value of a pin variable immediately before the triggering event shall be considered for evaluation of a boolean expression. The evaluation result shall be assigned to a pin variable immediately after the triggering event.
c) An information flow graph established by boolean assignments can involve an *implicitly declared variable*, i.e., the LHS of a boolean assignment has not been declared as a pin variable. An implicitly declared variable can only be used in the context of its parent statement. An implicitly declared variable involved in a continuous assignment can not be used in the context of a conditional assignment and vice-versa.

## 10.5 STRUCTURE statement

A *structure* statement shall be defined as shown in Syntax 73.

```
structure ::=
    STRUCTURE { named_cell_instantiation { named_cell_instantiation } }
  | structure_template_instantiation
```

*Syntax 73—STRUCTURE statement*

The purpose of a structure statement is to specify a structural implementation, i.e., a netlist of a compound cell. A complete or a partial netlist can be specified. The components of the netlist can be cells and/or primitives. A structure statement shall not substitute a behavior statement or a statetable statement. The connectivity graph established by a structure statement is complementary to the information flow graph established by a behavior statement or by a statetable statement.

** need to extend “pin assignment” definition to include hierarchical pin.port **

## 10.6 STATETABLE statement

A *statetable* statement shall be defined as shown in Syntax 74.

```
statetable ::=
    STATETABLE [ identifier ]
        { statetable_header statetable_row { statetable_row } }
    | statetable_template_instantiation
statetable_header ::=
    input_pin_variable { input_pin_variable } : output_pin_variable { output_pin_variable } ;
statetable_row ::=
    statetable_control_values : statetable_data_values ;
statetable_control_values ::=
    statetable_control_value { statetable_control_value }
statetable_control_value ::=
      boolean_value
    | symbolic_bit_literal
    | edge_value
statetable_data_values ::=
    statetable_data_value { statetable_data_value }
statetable_data_value ::=
      boolean_value
    | ( [ ! ] input_pin_variable )
    | ( [ ~ ] input_pin_variable )
```

*Syntax 74—STATETABLE statement*

A statetable shall specify the state of a set of *output pin variables* dependent on the state of a set of *input pin variables*. Sequential behavior, i.e., next state as a function of previous state shall be modeled by a pin variable which appears both as input and output pin variable within the statetable header. A pin variable with *direction* annotation value *both* can also appear as input and output pin variable within the statetable header. However, the state of the output pin variable does not depend on the state of the corresponding input pin variable, unless there is sequential behavior.

In each *statetable row*, a *statetable control value* shall be associated with a particular input pin variable, and a *statetable data value* shall be associated with a particular output variable. The association is given by the position at which the pin variables appear in the header. Each statetable row shall have the same number of items as the statetable header. The delimiting *colon* in each statetable row shall in the same position as in the statetable header.

A *statetable control value* shall be compatible with the *datatype* of the corresponding input pin variable. A *statetable data value* shall be compatible with the datatype of the corresponding output pin variable. An input pin variable enclosed by parentheses shall specify that the value of the input pin variable be assigned to the output pin variable. Such input pin variable need not appear in the statetable header. A preceding *exclamation mark* shall indicate that the logically inverted value be assigned to the output variable. A preceding *tilde* shall indicate that the bitwise inverted value be assigned to the output variable.

## 10.7 NON_SCAN_CELL statement

A *non_scan_cell* statement shall be defined as shown in Syntax 75.

A non-scan cell statement applies for a scan cell. A scan cell is a cell with extra pins for testing purpose. The *unnamed cell instantiation* within the non-scan cell statement specifies a cell that is functionally equivalent to the scan cell, if the extra pins are not used. The cell without extra pins is referred to as non-scan cell. The name of the non-scan cell is given by the *cell identifier.*

```
non_scan_cell ::=
    NON_SCAN_CELL { unnamed_cell_instantiation { unnamed_cell_instantiation } }
  | NON_SCAN_CELL = unnamed_cell_instantiation
  | non_scan_cell_template_instantiation
```

*Syntax 75—NON_SCAN_CELL statement*

The pin mapping is given either by order, using *pin value*, or by name, using *pin assignment*. In the former case, the pin values shall refer to pin names of the scan cell. The order of the pin values corresponds to the pin declarations within the non-scan cell. In the latter case, the pin names of the non-scan cell shall appear at the LHS of the assignment, and the pin names of the scan cell shall appear at the RHS of the assignment. The order of the pin assignments is arbitrary.

*Example*

```
// declaration of a non-scan cell
CELL myNonScanFlop {
    PIN D { DIRECTION=input;  SIGNALTYPE=data; }
    PIN C { DIRECTION=input;  SIGNALTYPE=clock; POLARITY=rising_edge; }
    PIN Q { DIRECTION=output; SIGNALTYPE=data; }
}
// declaration of a scan cell
CELL myScanFlop {
   PIN CK { DIRECTION=input; SIGNALTYPE=clock; }
   PIN DI { DIRECTION=input; SIGNALTYPE=data; }
   PIN SI { DIRECTION=input; SIGNALTYPE=scan_data; }
   PIN SE { DIRECTION=input; SIGNALTYPE=scan_enable; POLARITY=high; }
   PIN DO { DIRECTION=output; SIGNALTYPE=data; }
   // put NON_SCAN_CELL statement here
}
```

The non-scan cell statement with pin mapping by order looks as follows:

```
   NON_SCAN_CELL { myNonScanFlop { DI CK DO } }
// corresponding pins by order:    D  C  Q
```

The non-scan cell statement with pin mapping by name looks as follows:

```
   NON_SCAN_CELL { myNonScanFlop { Q=DO; D=DI; C=CK; } }
```

## 10.8 RANGE statement

A *range* statement shall be defined as shown in Syntax 76.

```
range ::=
    RANGE { index_value : index_value }
```

*Syntax 76—RANGE statement*

The range statement shall be used to specify a valid address space for elements of a vector- or matrix-pin.

If no range statement is specified, the valid address space *A* is given by the following mathematical relationship:

$$0 \le A \le 2^B - 1$$

$$B = \begin{cases} 1 + \text{LSB} - \text{MSB} & \text{if}(\text{LSB} > \text{MSB}) \\ 1 + \text{MSB} - \text{LSB} & \text{if}(\text{LSB} \le \text{MSB}) \end{cases}$$

where

*A* is an unsigned integer representing the address space within a vector- or matrix-pin,
*B* is the *bitwidth* of the vector-or matrix-pin,

and

MSB is the left-most bit within the vector- or matrix-pin,
LSB is the right-most bit within the vector or- matrix-pin,

in accordance with Section 7.8.

The index values within a range statement shall be bound by the address space *a*, otherwise the range statement shall not be considered valid.

*Example*

```
PIN [5:8] myVectorPin { RANGE { 3 : 13 } }
```

| | |
|---|---|
| bitwidth: | $B = 4$ |
| default address space: | $0 \le A \le 15$ |
| address space defined by range statement: | $3 \le A \le 13$ |

## 10.9 Boolean expression

A *boolean expression* shall be defined as shown in Syntax 77.

```
boolean_expression ::=
      ( boolean_expression )
    | pin_variable
    | boolean_value
    | boolean_unary boolean_expression
    | boolean_expression boolean_binary boolean_expression
    | boolean_expression ? boolean_expression :
        { boolean_expression ? boolean_expression : }
        boolean_expression
boolean_unary ::=
      ! | ~ | & | ~& | | | ~| | ^ | ~^
boolean_binary ::=
      & | && | | | || | ^ | ~^ | != | == | >= | <= | > | < | + | - | * | / | % | >> | <<
```

*Syntax 77—Boolean expression*

The purpose of a boolean expression is to specify a boolean operation involving pin variables as operands. The evaluation result of a boolean expression shall be a boolean value.

## 10.10 Boolean value system

### 10.10.1 Scalar boolean value

A *scalar boolean value* shall be described by an *alphanumerical bit literal* (see Section 6.7). A scalar boolean value shall represent a *logical value* and optionally a *drive strength*. The set of logical values shall be *false*, *true* and *unknown*. The set of drive strengths shall be *strong*, *weak*, and *zero*. The symbols used for scalar boolean values and their meaning shall be defined as shown in Table 69.

**Table 69—Scalar boolean values**

| symbol | logical value | drive strength | symbol for reduced value | comment |
|---|---|---|---|---|
| `0` | false | strong | `0` | |
| `1` | true | strong | `1` | |
| `X` or `x` | unknown | strong | `X` or `x` | |
| `L` or `l` | false | weak | `0` | |
| `H` or `h` | true | weak | `1` | |
| `W` or `w` | unknown | weak | `X` or `x` | |
| `Z` or `z` | undefined | zero | `X` or `x` | use for high impedance |
| `U` or `u` | undefined | undefined | `X` or `x` | use for uninitialized signal in simulation |

A *boolean expression* (see Section 10.9) can evaluate to a scalar boolean value represented by an *alphanumeric bit literal*. For evaluation of a boolean expression, a scalar boolean value shall be reduced to a value 0, 1, or X within a 3-value system, unless an *alphabetic bit literal* (L, H, W, Z, U) is explicitely specified as evaluation result in the boolean expression.

### 10.10.2 Vectorized boolean value

A *vectorized boolean value* shall be described either by a *based literal* (see Section 6.8) or by an *integer* (see Section 6.5). A vectorized boolean value can be mapped into a vector of *alphanumerical bit literals*. The number of bit literals shall be called *bitwidth*.

An *octal digit* can be mapped into a three bit vector of *bit literals*, as shown in Table 70.

**Table 70—Mapping between octal base and binary base**

| Octal | Binary (bit literal) | Numerical value |
|---|---|---|
| **0** | **000** | 0 |
| **1** | **001** | 1 |
| **2** | **010** | 2 |
| **3** | **011** | 3 |

**Table 70—Mapping between octal base and binary base (Continued)**

| Octal | Binary (bit literal) | Numerical value |
|---|---|---|
| **4** | **100** | 4 |
| **5** | **101** | 5 |
| **6** | **110** | 6 |
| **7** | **111** | 7 |

A *hexadecimal digit* can be mapped into a four bit vector of *bit literals*, as shown in Table 71.

**Table 71—Mapping between hexadecimal base and binary base**

| Hexadecimal | Binary (bit literal) | Numerical value |
|---|---|---|
| **0** | **0000** | 0 |
| **1** | **0001** | 1 |
| **2** | **0010** | 2 |
| **3** | **0011** | 3 |
| **4** | **0100** | 4 |
| **5** | **0101** | 5 |
| **6** | **0110** | 6 |
| **7** | **0111** | 7 |
| **8** | **1000** | 8 |
| **9** | **1001** | 9 |
| **a** or **A** | **1010** | 10 |
| **b** or **B** | **1011** | 11 |
| **c** or **C** | **1100** | 12 |
| **d** or **D** | **1101** | 13 |
| **e** or **E** | **1110** | 14 |
| **f** or **F** | **1111** | 15 |

An alphabetic bit literal shall be mapped according to the following rules:

a) An alphabetic bit literal in octal base shall be mapped into three subsequent occurrences of the same bit literal in binary base.
b) An alphabetic bit literal in hexadecimal base shall be mapped into four subsequent occurrences of the same bit literal in binary base.

*Example*

**'o2xw0u** is equivalent to **'b010_xxx_www_000_uuu**
**'hLux** is equivalent to **'bLLLL_uuuu_xxxx**

An *integer* can be represented by a vector of bit literals, according to the following mathematical relationship.

unsigned integer $$N = \sum_{p=0}^{B-1} s(p) \cdot 2^p$$

signed integer $$N = \sum_{p=0}^{B-2} s(p) \cdot 2^p - S \cdot 2^{B-1}$$

where

*N* is the integer.
*B* is the bitwidth of the vector of bit literals.
*p* is the position of a bit within the vector, counted from 0 to *B*-1.
*s*(*p*) is the scalar value (zero or one) of the bit at position *p*.
*S* is the scalar value (zero or one) of the MSB, i.e., the bit at position *B*-1.

The bitwidth *B* of a vectorized boolean variable restricts the range of a corresponding integer *N* as follows:

unsigned integer $$0 \le N \le 2^B - 1$$

signed integer $$-2^{B-1} \le N \le 2^{B-1} - 1$$

A *vector pin* (see Section 9.7) can be used as a *pin variable* holding a vectorized boolean value. The position of a bit is related to an index within the pin declaration as follows:

$$p = \begin{cases} \text{LSB} - i & \text{if}(\text{LSB} > \text{MSB}) \\ i - \text{LSB} & \text{if}(\text{LSB} \le \text{MSB}) \end{cases}$$

where

*i* is the index within a vector pin.
LSB is the rightmost index within a vector pin. The corresponding position is 0.
MSB is the leftmost index within a vector pin. The corresponding position is *B*-1.

*Example:*

```
PIN [5:8] my_vector_pin;
```

| bit[index] | position | comment |
|---|---|---|
| my_vector_pin[5] | 3 | MSB |
| my_vector_pin[6] | 2 | |
| my_vector_pin[7] | 1 | |
| my_vector_pin[8] | 0 | LSB |

### 10.10.3 Non-assignable boolean value

A *non-assignable boolean value* shall be described by a *symbolic bit literal* (see Section 6.7), as shown in Table 72.

**Table 72—Symbolic boolean values**

| symbol | logical value | drive strength | comment |
|---|---|---|---|
| ? | arbitrary | arbitrary | use for “don’t care” |
| * | subject to random change | arbitrary | signal is not monitored |

A symbolic bit literal or a based literal containing a symbolic bit lieteral can not be assigned to a pin variable as a boolean value. A symbolic bit literal can be used within a statetable control value, but not within a statetable data value.

Within the context of a vectorized boolean value, a symbolic bit literal shall be mapped according to the following rules:

a) A symbolic bit literal in octal base shall be mapped into three subsequent occurrences of the same bit literal in binary base.
b) A symbolic bit literal in hexadecimal base shall be mapped into four subsequent occurrences of the same bit literal in binary base.

## 10.11 Boolean operations and operators

### 10.11.1 Logical operation

The operators for a *logical operation* shall be defined as shown in Table 73

**Table 73—Logical Operation**

| Operator | Description |
|---|---|
| **!** | logical *inversion* |
| **&&** | logical *and* |
| **\|\|** | logical *or* |

A boolean expression involving a logical *inversion*, *and*, *or* (see Table 73), *nand*, *nor*, *exor*, *exnor* (see Table 74) shall be evaluated according to the rules of boolean algebra ** do we need a reference to a textbook on boolean algebra here? **.

The result of the evaluation shall be *true*, *false*, or *unknown*.

If an alphabetic bit literal is used as operand, only the logical value, not the drive strength, shall be considered for evaluation. An *undefined* logical value within an operand shall be considered *unknown*.

If a vectorized boolean value is used as operand, the logical value of the operand shall be obtained by applying a logical *or* to all bits of the operand.

### 10.11.2 Bitwise operation

The operators for a *bitwise operation* shall be defined as shown in Table 74

**Table 74—Bitwise Operation**

| Operator | Description |
|---|---|
| **~** | bit-wise *inversion* |
| **&** | bit-wise *and* |
| **\|** | bit-wise *or* |
| **^** | bit-wise exclusive or (*exor*) |
| **~&** | bit-wise and with inversion (*nand*) |
| **~\|** | bit-wise or with inversion (*nor*) |
| **~!** | bit-wise exclusive or with inversion (*exnor*) |

A *bit-wise inversion* shall invert each bit of a vectorized boolean value.

The operators for bit-wise operations, except bit-wise inversion, can be used as *boolean unary* or as *boolean binary* operators.

A *boolean unary* operator for the operation *and*, *or*, *exor*, *nand*, *nor*, or *exnor* shall reduce a vectorized boolean value to a scalar boolean value by applying a logical *and*, *or*, *exor*, *nand*, *nor*, or *exnor* to all bits of the operand.

A *boolean binary* operator for the operation *and*, *or*, *exor*, *nand*, *nor*, or *exnor* shall apply a *logical and*, *or*, *exor*, *nand*, *nor*, or *exnor* to each corresponding bit of two vectorized boolean values. The operands shall be LSB-aligned. If the operands have different bitwidths, the missing bits of the operand with smaller bitwidth shall be considered *undefined*. The result of the operation shall be a vectorized boolean value.

A bit-wise operation involving only scalar boolean values or single bit vectorized boolean values as operands shall be considered equivalent to the corresponding logical operation.

### 10.11.3 Conditional operation

The symbols used for a *conditional operation* shall be defined as shown in Table 75

**Table 75—Conditional Operation**

| Symbol | Description |
|---|---|
| **?** | operator for a condition |

**Table 75—Conditional Operation**

| Symbol | Description |
|---|---|
| : | delimiter between alternatives |

If the boolean sub-expression to the left of the condition operator evaluates true, the boolean sub-expression to the right of the condition operator shall be evaluated. Otherwise, the boolean expression to the right of the delimiter between alternatives shall be evaluated. If multiple conditions and alternatives exist within a boolean expression, the evaluation shall proceed from the left to the right.

### 10.11.4 Integer arithmetic operation

The operators for an *integer arithmetic operation* shall be defined as shown in Table 76.

**Table 76—Integer Arithmetic Operation**

| Operator | Description |
|---|---|
| + | add |
| - | subtract |
| * | multiply |
| / | divide |
| % | modulus |

A boolean expression involving an *integer arithmetic operation* with operands represented as *integer* shall be evaluated according to the rules of integer arithmetic ** do we need a reference to a textbook on integer arithmetic here? **.

If an operand is represented as a *based literal*, the operand shall be converted into an integer according to Section 10.10.2. This conversion is well-defined, if each bit has the logical value *true* or *false*. The MSB of a based literal shall be interpreted according to the *datatype* annotation value (see Section 9.9.7) of a pin variable associated with the based literal.

An operand represented as a *bit literal* shall be treated in the same way as a single bit *binary based literal*.

If a bit literal or a bit of a based literal has the logical value *unknown*, the conversion into an integer is not well-defined. In this case, an application can optionally perform a partial evaluation of the boolean expression, by replacing the value *unknown* with the value *true* or *false*.

### 10.11.5 Shift operation

The operators for a *shift operation* shall be defined as shown in Table 77

**Table 77—Shift Operation**

| Operator | Description |
|---|---|
| << | shift left |
| >> | shift right |

A shift operation shall involve two operands. The LHS operand shall be a vectorized boolean value, represented by an integer, by a based literal, or, as a trivial case, by a bit literal. The RHS operand shall be an unsigned integer *N* in the range between zero and the bitwidth of the LHS operand, specifying the number of positions by which the bits of the LHS operand are to be shifted.

For *shift left*, N bits of the LHS operand shall be replaced with the logical value *unknown*, starting from the LSB. For *shift right*, N bits of the LHS operand shall be replaced with the logical value *unknown*, starting from the MSB.

### 10.11.6 Comparison operation

The operators for a *comparison operation* shall be defined as shown in Table 78

**Table 78—Comparison Operation**

| Operator | Description |
|---|---|
| == | equal |
| != | non equal |
| > | greater |
| < | less |
| >= | greater or equal |
| <= | lesser or equal |

A comparison involving operands represented as *integer* shall be evaluated according to the rules of integer arithmetic ** do we need a reference to a textbook on integer arithmetic here? **.

If an operand is represented as a *based literal*, the operand shall be converted into an integer according to Section 10.10.2. This conversion is well-defined, if each bit has the logical value *true* or *false*. The MSB of a based literal shall be interpreted according to the *datatype* annotation value (see Section 9.9.7) of a pin variable associated with the based literal.

If a bit of a based literal has the logical value *unknown*, the conversion into an integer is not well-defined. In this case, an application can optionally perform a partial comparison, by replacing the value *unknown* with the value *true* or *false*.

If the operands are integers or the conversion from based literal to integer is well-defined, a comparison shall evaluate *true* or *false*. If the conversion from based literal to integer is not well-defined, a comparison can evaluate *unknown*.

A comparison between scalar boolean values or single bit vectorized boolean values shall consider both the logical value and the drive strength as criterion for comparison

The *equal* comparison considering drive strength shall be evaluated according to the following Table 79

**Table 79—Equal comparison considering drive strength**

| logical value (true, false, unknown, or undefined) | drive strength (strong, weak, zero, or undefined) | result |
|---|---|---|
| same for both operands | same for both operands | true |
| same for both operands | different for each operand | false |
| different for each operand | arbitrary | false |

The *non-equal* comparison shall evaluate true, if the equal comparison evaluates false, and vice-versa.

Note: To compare scalar boolean values or single bit vectorized boolean values considering the logical value only, the *exor* operation can be used instead of the non-equal comparison, and the *exnor* operation can be used instead of the equal comparison.

The *greater* comparison considering drive strength shall be evaluated according to the following Table 80

**Table 80—Greater comparison considering drive strength**

| logical value LHS operand | logical value RHS operand | drive strength | result |
|---|---|---|---|
| true | false | arbitrary | true |
| true | unknown | arbitrary | unknown |
| false | true | arbitrary | false |
| false | unknown | arbitrary | false |
| unknown | true | arbitrary | unknown |
| unknown | false | arbitrary | unknown |
| unknown | unknown | arbitrary | unknown |
| true | true | same for both operands | false |
| false | false | same for both operands | false |
| true | true | different for each operand | unknown |
| false | false | different for each operand | unknown |

The *lesser* comparision shall be evaluated in the same way as the greater comparison, when the LHS operand and the RHS operand switch places.

The *greater-or-equal* comparison shall be evaluated as logical *or* between *greater* comparison and *equal* comparison.

The *lesser-or-equal* comparison shall be evaluated as logical *or* between *lesser* comparison and *equal* comparison.

### 10.11.7 Operator priorities

The binding priority of operations in a boolean expression shall be from the strongest to the weakest in the following order:

*a)* operation enclosed by *parentheses*
b) *boolean unary* (!, ~, &, ~&, |, ~|, ^, ~^)
c) *exor* (^), *exnor* (~^), *comparison* (>, <, >=, <=, ==, !=), *shift* (<<, >>)
d) *and* (&, &&), *nand* (~&), *multiply* (*), *divide* (/), *modulus* (%)
e) *or* (|, ||), *nor* (~|), *add* (+), *subtract* (-)
f) operator and delimiter for *conditional operation* (?, :)

When operations of the same binding priority are subsequently encountered in a boolean expression, the evaluation shall proceed from the left to the right.

## 10.12 Vector expression

A *vector expression* shall be defined as shown in Syntax 78.

```
vector_expression ::=
      ( vector_expression )
    | vector_unary boolean_expression
    | vector_expression vector_binary vector_expression
    | boolean_expression ? vector_expression :
        { boolean_expression ? vector_expression : }
        vector_expression
    | boolean_expression control_and vector_expression
    | vector_expression control_and boolean_expression
vector_unary ::=
    edge_literal
vector_binary ::=
     & | && | | | || | -> | ~> | <-> | <~> | &> | <&>
control_and ::=
    & | &&
```

*Syntax 78—Vector expression*

The purpose of a vector expression to specify a sequence of events. In a static application context, the vector expression shall be evaluated against a *proposed* sequence of events. In a dynamic application context, a vector expression shall be evaluated against a *monitored* sequence of events.

A vector expression shall evaluate true, when the specified sequence of events is satisfied or detected, i.e., the vector expression matches a proposed or monitored sequence of events. The true evaluation of a vector expression constitutes an event by itself, which can be used as a trigger within the context of a behavior statement (see Section 10.4).

## 10.13 Operators for event specification

The term *event* is used synonymously to *contents of an arbitrary vector expression*.

### 10.13.1 Specification of a single event

An *edge literal* (see Section 6.9) shall be used as a *vector unary* operator to specify a *single event*. The operand shall be a *boolean expression*. A single event on the operand shall be interpreted according to the following Table 81.

**Table 81—Specification of a single event**

| row | edge literal | event on operand |
|---|---|---|
| 1 | *first*_bit_literal *second*_bit_literal | value before is *first*_bit_literal, value after is *second*_bit_literal |
| 2 | *first*_based_literal *second*_based_literal | value before is *first*_based_literal, value after is *second*_based_literal |
| 3 | **??** | value before and after the event is *arbitrary* |
| 4 | **?*** | state of operand is *random* after the event |
| 5 | ***?** | state of operand is *random* before the event |
| 6 | **?!** | operand changes from arbitrary value to arbitrary different value |
| 7 | **?~** | every binary digit of the operand changes from arbitrary value to arbitrary different value |
| 8 | **?-** | operand does not change its value |

An edge literal consisting of two consecutive alphanumerical bit literals (row 1) can be used for a scalar operand. An edge literal consisting of two consecutive based literals (row 2) can be used for a scalar operand or for a vectorized operand, as long as the bitwidth of the operator is compatible ith the bitwidth of the operand. An edge literal consisting of two consecutive symbolic bit literals (row 3, 4, 5) can be used for either a scalar or a vectorized operand. A symbolic edge literal (row 6, 7, 8) can be used for either a scalar or a vectorized operand.

An edge literal (row 8 in particular) can specify the same value before and after the event. Such a specification shall be interpreted as event by exclusion, i.e., an event happens, but not on the operand.

An *arbitrary* value shall be comprised within the set of applicable values for the operand, i.e., a scalar operand or a binary digit of a vectorized operand can have a value specified by an alphanumerical bit literal, an operand with datatype *unsigned* can have an arbitrary *unsigned integer* value within the range of specified bitwidth, an operand with datatype *signed* can have an arbitrary *signed integer* value within the range of specified bitwidth.

A *random* value shall be interpreted as an arbitrary value subjected to random change. In a dynamic application context, an event on a variable is not monitored while the variable is in random value state.

A single event can be described by a timing diagram as illustrated in the following figure 16.

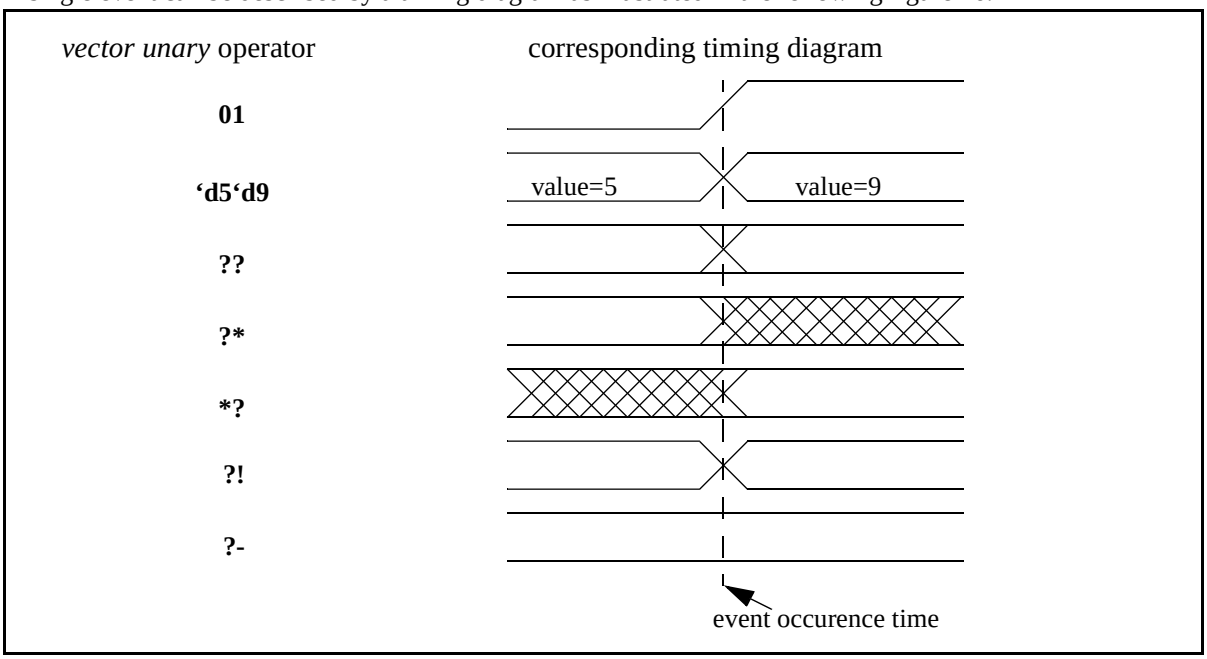



**Figure 16—Timing diagrams for single events**

The specification of a single event by itself does not imply any transition time. A single event can happen instantaneously. The transition time in figure 16 is only for the purpose of illustrating the difference between **??** and **?!**.

The operator **??** shall be considered *neutral operator*, since a specified single event involving **??** on an arbitrary operand always matches a proposed single event on any operand. A single event involving the neutral operator shall be considered *neutral single event*.

### 10.13.2 Temporal order of events

A *vector binary* operator shall be used to specify a temporal order between events, thus establishing an *event sequence*. Each operand shall be a vector expression. The operation result shall be another vector expression.

The vector expression shall be evaluated against a proposed or monitored event sequence. The proposed or monitored event sequence shall be established as follows:

a) A primary event sequence shall be established by representing in temporal order all single events on a set of pin variables. The set of pin variables shall be specified either by the *scope* annotation (see Section 9.9.15) within a pin declaration or by the *monitor* annotation (see Section 9.17.9) within a vector declaration. The elapsed time between subsequently occuring single events can vary between arbitrarily large and arbitrarily small values.

Note: In a dynamic application context, "all" single events can be eventually reduced to "the *N* latest relevant" single events, where *N* is large enough to contain the specified vector expression.

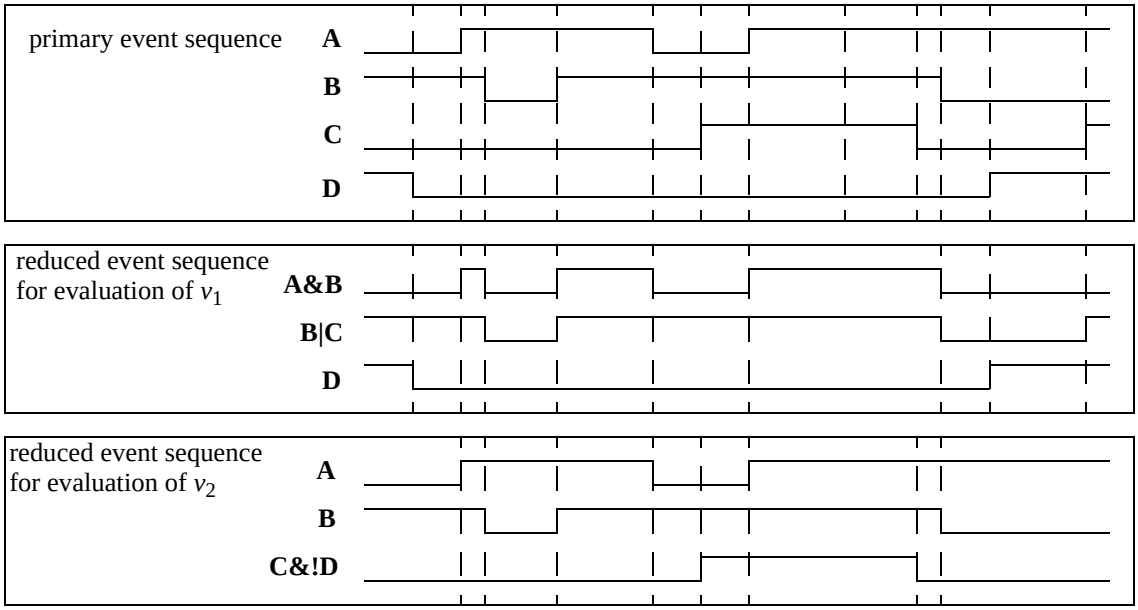
b) The single events on pin variables involved in the vector expression shall be reduced to single events on boolean expressions wherein the pin variables are involved. Other single events on these pin variables shall be disregarded. The single events on pin variables not involved in the vector expression shall be not be reduced.

*Example:*

A set of pin variables applicable for two vector expressions $v_1$and $v_2$ is **A**, **B**, **C**, **D**.
The vector expression $v_1$ reads **(01 (A&B) -> 10 (B|C))**.
The vector expression $v_2$ reads **(1? A -> 10 (C & ! D))**.

Therefore, the primary event sequence represents the single events on **A**, **B**, **C** and **D**.
The reduced event sequence for evaluation of $v_1$ represents the single events on **(A&B)**, **(B|C)** and **D**.
The reduced event sequence for evaluation of $v_2$ represents the single events on **A**, **B** and **(C & ! D)**.

The following picture shows sample event sequences.




The temporal order concept does not specify or imply a particular time interval between consecutive single event. Mathematically, each time interval shall be greater than zero, but it can be arbitrarily close to zero. Two single events can occur simultaneously, i.e., at the same time, either by implication or by co-incidence.

The following rules shall apply for the temporal order of events.

a) A value change of a boolean expression and a single event on a pin variable causing this value change shall be considered simultaneous by implication.
b) A value change of a vectorized pin variable and a corresponding value change of any part of the vectorized pin variable shall be considered simultaneous by implication.
c) Within the context of a *behavior* statement, the assignment of a boolean expression to a pin variable as a consequence of a value change of the boolean expression shall trigger an advancement in time.
d) Within the context of a *control* statement as part of a behavior statement, the assignment of a boolean expression to a pin variable as a consequence of a value change of a control expression shall trigger an advancement in time.
e) Single events on arbitrary independent pin variables can occur simultaneously by co-incidence.
f) In the context of a *vector* statement, all pin variables shall be considered independent, even though a causal dependency between some pin variables can exist in the context of a *behavior* statement.

It is possible that the application does not support a monitor capable of detecting simultaneously occuring events by co-incidence. In this case, the temporal order of such events is not predictable.

*Example:*

A behavior statement contains the boolean assignment **Z = A&B**.
The single event **(01 (A&B))** is caused by the single event **(01 A)**.
The single events **(01 (A&B))** and **(01 A)** are considered to occur simultaneously by implication.
Within the context of the behavior statement, the single event **(01 Z)** is considered to occur after the single event **(01 (A&B))**.
Outside the context of the behavior statement, the variables **A** and **Z** are considered independent. The numerical value of the measured propagation delay from **A** to **Z** could be greater than zero, lesser than zero, or zero. Therefore, the single events **(01 A)** and **(01 Z)** could occur simultaneously by co-incidence.

### 10.13.3 Canonical specification of an event

The operators in the following Table 82 shall be used for a canonical specification of a vector expression.

**Table 82—Canonical specification of an event**

| symbol | operator name | explanation |
|---|---|---|
| -> | immediately followed by | LHS event occurs before RHS event,<br>no event can occur in-between |
| ~> | eventually followed by | LHS event occurs before RHS event,<br>an arbitrary number of events can occur in-between |
| **&&** or **&** | simultaneous occurence | LHS event and RHS event occur at the same time |
| \|\| or \| | alternative occurence | Either LHS event or RHS event occur |
| **&>** | closely followed by | LHS event occurs immediately before RHS event,<br>or both events occur at the same time |

The semantic meaning of the operators is furthermore detailed as follows:

The *immediately followed by* operator applied to a sequence of single events shall specify that the latest single event within the LHS vector expression immediately precedes the earliest single event within the RHS vector expression.

The *eventually followed by* operator applied to a sequence of single events shall specify that the latest single event within the LHS vector expression occurs earlier than the earliest single event within the RHS vector expression.

The *simultaneous occurence* operator applied to a sequence of single events shall specify that each Nth latest single event within the LHS vector expression occurs at the same time as each Nth latest single event within the RHS vector expression.

This rule can be formulated as follows:

a) Product involving *immediately followed by* and *simultaneously occuring* operator
$( v^M_1 \text{ -> } v^N_1 )$ **&** $( v^M_2 \text{ -> } v^N_2 ) = (v^M_1 \text{ \& } v^M_2) \text{ -> } (v^N_1 \text{ \& } v^N_2)$

where $v^M_i$ and $v^N_i$, respectively, are vector expressions describing a sequence of M single events each and N single events each, respectively, ordered by the *immediately followed by* operator.

If the LHS and RHS vector expressions comprise a different number of subsequently occuring single events, the shorter vector expression shall be left-extended with neutral single events.

b) Product involving sequences of events with different length
$( v^{M}_{1} \text{ -> } v^{N}_{1} )$ **&** $v^{N}_{2} = v^{M}_{1} \text{ -> } ( v^{N}_{1} \text{ & } v^{N}_{2} )$

A set of mathematical rules for evaluation of a compound vector expression shall be established, wherein the the symbols $v_i$ represent vector expressions within the compound vector expression.

c) Associativity for *immediately followed by* operator
$v_1 \text{ -> } v_2 \text{ -> } v_3 = ( v_1 \text{ -> } v_2 ) \text{ -> } v_3 = v_1 \text{ -> } ( v_2 \text{ -> } v_3 )$

d) Associativity for *eventually followed by* operator
$v_1 \text{ ~> } v_2 \text{ ~> } v_3 = ( v_1 \text{ ~> } v_2 ) \text{ ~> } v_3 = v_1 \text{ ~> } ( v_2 \text{ ~> } v_3 )$

e) Mixed associativity for *immediately followed by* and *eventually followed by* operator
$v_1 \text{ -> } v_2 \text{ ~> } v_3 = ( v_1 \text{ -> } v_2 ) \text{ ~> } v_3 = v_1 \text{ -> } ( v_2 \text{ ~> } v_3 )$
$v_1 \text{ ~> } v_2 \text{ -> } v_3 = ( v_1 \text{ ~> } v_2 ) \text{ -> } v_3 = v_1 \text{ ~> } ( v_2 \text{ -> } v_3 )$

f) Assocativity for *simultaneous occurence* operator
$v_1 \text{ & } v_2 \text{ & } v_3 = ( v_1 \text{ & } v_2 ) \text{ & } v_3 = v_1 \text{ & } ( v_2 \text{ & } v_3 )$

g) Commutativity for *simultaneous occurence* operator
$v_1 \text{ & } v_2 = v_2 \text{ & } v_1$

h) Reduction rule for *simultaneous occurence* operator
$v_1 \text{ & } v_1 = v_1$

i) Assocativity for *alternative occurence* operator
$v_1 \mid v_2 \mid v_3 = ( v_1 \mid v_2 ) \mid v_3 = v_1 \mid ( v_2 \mid v_3 )$

j) Commutativity for *alternative occurence* operator
$v_1 \mid v_2 = v_2 \mid v_1$

k) Reduction rule for *alternative occurence* operator
$v_1 \mid v_1 = v_1$

l) Distributivity between *immediately followed by* operator and *alternative occurence* operator
$( v_1 \mid v_2 ) \text{ -> } v_3 = ( v_1 \text{ -> } v_3 ) \mid ( v_2 \text{ -> } v_3 )$
$v_1 \text{ -> } ( v_2 \mid v_3 ) = ( v_1 \text{ -> } v_2 ) \mid ( v_1 \text{ -> } v_3 )$

m) Distributivity between *eventually followed by* operator and *alternative occurence* operator
$( v_1 \mid v_2 ) \text{ ~> } v_3 = ( v_1 \text{ ~> } v_3 ) \mid ( v_2 \text{ ~> } v_3 )$
$v_1 \text{ ~> } ( v_2 \mid v_3 ) = ( v_1 \text{ ~> } v_2 ) \mid ( v_1 \text{ ~> } v_3 )$

n) Distributivity between *simultaneous occurence* operator and *alternative occurence* operator
$( v_1 \mid v_2 ) \text{ & } v_3 = ( v_1 \text{ & } v_3 ) \mid ( v_2 \text{ & } v_3 )$

The *closely followed by* operator shall be mathematically defined as follows:

o) $v_1 \text{ &> } v_2 = ( v_1 \text{ & } v_2 ) \mid ( v_1 \text{ -> } v_2 )$

Therefore, the *closely followed by* operator applied to a sequence of single events shall specify that the latest single event within the LHS vector expression immediately precedes the earliest single event within the RHS vector expression, or, each Nth latest single event within the LHS vector expression occurs at the same time as each Nth latest single event within the RHS vector expression.

A general vector expression can be mathematically formulated as a canonical “sum of products”.

$$v^{p}_{j} = v_{j(1)} \ldots \mathrm{op}_{j(i)} v_{j(i)} \ldots \mathrm{op}_{j(m)} v_{j(m)} = \prod_{i=1}^{m_j} \mathrm{op}_{j(i)} v_{j(i)}$$

$$\mathrm{op}_{j(i)} = \text{-> } \mid \text{ ~> } \mid \text{ &}$$

$$v^{s} = v^{p}_{1} \ldots \mid \ldots v^{p}_{j} \ldots \mid \ldots v^{p}_{n} = \sum_{j=1}^{n} v^{p}_{j}$$

where $v^s$ is a vector expression in "sum" form applying the *alternative occurence* operator to vector expressions $v^p_j$, and each $v^p_j$ is a vector expression in "product" form applying the operators *immediately followed by, eventually followed by,* or *simultaneous occurence* to single events $v_{j(i)}$. The usage of the symbols $op_{j(i)}$, Π and Σ for *vector binary* operators is only for mathematical representation, it is not a syntax feature for a vector expression. Also, the first operator $op_{j(1)}$ is irrelevant when converting the mathematical representation into a vector expression.

*Example:*

$$v^p_1 = \prod_{i=1}^{m_1} op_{1(i)} v_{1(i)} \qquad m_1 = \mathbf{3} \qquad op_{1(1)} = \mathbf{nil} \quad op_{1(2)} = \text{->} \quad op_{1(3)} = \text{->}$$

$$v_{1(1)} = \mathbf{(01\ A)} \quad v_{1(2)} = \mathbf{(10\ A)} \quad v_{1(3)} = \mathbf{(10\ B)}$$

$$v^p_2 = \prod_{i=1}^{m_2} op_{2(i)} v_{2(i)} \qquad m_2 = \mathbf{3} \qquad op_{2(1)} = \mathbf{nil} \quad op_{2(2)} = \text{->} \quad op_{2(3)} = \text{->}$$

$$v_{2(1)} = \mathbf{(01\ B)} \quad v_{2(2)} = \mathbf{(10\ B)} \quad v_{2(3)} = \mathbf{(10\ A)}$$

$$v^s = \sum_{j=1}^{2} v^p_j = (01\ A) \text{ -> } (10\ A) \text{ -> } (10\ B) \mid (01\ B) \text{ -> } (10\ B) \text{ - > } (10\ A)$$

### 10.13.4 Specification of a completely permutable event

*Permutation* operations shall be defined for events *immediately followed by each other*, for events *eventually followed by each other*, and for events *closely followed by each otheer*. The operands, i.e., arbitrary vector expressions $v_i$, shall be subjected to alternative event sequences with completely permutable temporal order.

The symbols for permutation operators are shown in the following Table 83.

**Table 83—Specification of a completely permutable event**

| symbol | operator name | explanation |
|---|---|---|
| <-> | permutation of events immediately followed by each other | LHS event immediately followed by RHS event<br>or<br>RHS event immediately followed by LHS event |
| <~> | permutation of events eventually followed by each other | LHS event eventually followed by RHS event<br>or<br>RHS event eventually followed by LHS event |
| <**&**> | permutation of events closely followed by each other | LHS event immediately followed by RHS event<br>or<br>RHS event eventually followed by LHS event<br>or<br>LHS event and RHS event occur simultaneously |

The *permutation* operator for two events *immediately followed by each other* shall be mathematically defined as follows:

p) $v_1 \text{ <-> } v_2 = ( v_1 \text{ -> } v_2 ) \mid ( v_2 \text{ -> } v_1 )$

The *permutation* operator for two events *eventually followed by each other* shall be mathematically defined as follows:

q) $v_1 \text{ <~> } v_2 = ( v_1 \text{ ~> } v_2 ) \mid ( v_2 \text{ ~> } v_1 )$

The *permutation* operator for two events *closely followed by each other* shall be mathematically defined as follows:

r) $v_1 \text{ <\&> } v_2 = ( v_1 \text{ \&> } v_2 ) \mid ( v_2 \text{ \&> } v_1 )$

The definition of a *permutation* operator for $N$ events ($N \geq 2$) shall be extended for $N$+1 events in the following way:

$$\prod_{k=1}^{N} \text{<->} v_k = \sum_{j=1}^{N!} \prod_{i=1}^{N} \text{->} v_{j(i)} = \sum_{j=1}^{N!} v_j^{p(N)} \quad \text{where} \quad v_{j(i)} \subset (v_k) \quad \text{with} \quad v_j^{p(N)} = \prod_{i=1}^{N} \text{->} v_{j(i)}$$

$$\prod_{k=1}^{N+1} \text{<->} v_k = \sum_{j=1}^{N!} \sum_{k=1}^{N+1} \left( \prod_{i=1}^{k-1} \text{->} v_{j(i)} \right) \text{->} v_{j(N+1)} \left( \prod_{i=k}^{N} \text{->} v_{j(i)} \right) = \sum_{j=1}^{(N+1)!} v_j^{p(K+1)} \quad \text{with} \quad v_j^{p(N+1)} = \prod_{i=1}^{N+1} \text{->} v_{j(i)}$$

If the operator <-> is globally replaced by <~> or <**&**>, respectively, the operator -> shall be globally replaced by ~> or **&**>, respectively.

A vector expression with $N$ operands $v_k$ subjected to a *permutation* operator (i.e., <-> or <~> or <**&**>) is equivalent to a vector expression with $N$! sum terms wherein each sum term represents a particular permutation of $v_k$. Each sum term consists of $N$ product terms, i.e., a sequence of $N$ events $v_{j(i)}$ subjected to a corresponding *followed by* operator (i.e., -> or ~> or **&**>). There are $N$! such sequences of events. The ($N$+1)th operand can be inserted in N+1 places within each sum term. Therefore a vector expression with $N$+1 operands $v_k$ subjected to a *permutation* operator is equivalent to a vector expression with ($N$+1)! sum terms, each of which consists of $N$+1 product terms.

As each permutation operator is defined for $N$=2 events, the definition can be immediatley extended to $N$=3 events.

Permutation of 3 immediately followed events:

$v_1 \text{ <-> } v_2 \text{ <-> } v_3 =$
$(v_1 \text{ -> } v_2 \text{ -> } v_3) \mid (v_1 \text{ -> } v_3 \text{ -> } v_2) \mid (v_3 \text{ -> } v_1 \text{ -> } v_2) \mid (v_2 \text{ -> } v_1 \text{ -> } v_3) \mid (v_2 \text{ -> } v_3 \text{ -> } v_1) \mid (v_3 \text{ -> } v_2 \text{ -> } v_1)$

Permutation of 3 eventually followed events:

$v_1 \text{ <~> } v_2 \text{ <~> } v_3 =$
$(v_1 \text{ ~> } v_2 \text{ ~> } v_3) \mid (v_1 \text{ ~> } v_3 \text{ ~> } v_2) \mid (v_3 \text{ ~> } v_1 \text{ ~> } v_2) \mid (v_2 \text{ ~> } v_1 \text{ ~> } v_3) \mid (v_2 \text{ ~> } v_3 \text{ ~> } v_1) \mid (v_3 \text{ ~> } v_2 \text{ ~> } v_1)$

Permutation of 3 closely followed events:

$v_1 \text{ <\&> } v_2 \text{ <\&> } v_3 =$
$(v_1 \text{ \&>} v_2 \text{ \&>} v_3) \mid (v_1 \text{ \&>} v_3 \text{ \&>} v_2) \mid (v_3 \text{ \&>} v_1 \text{ \&>} v_2) \mid (v_2 \text{ \&>} v_1 \text{ \&>} v_3) \mid (v_2 \text{ \&>} v_3 \text{ \&>} v_1) \mid (v_3 \text{ \&>} v_2 \text{ \&>} v_1)$

From $N$=3 events, the definition can be extended to $N$=4 events, and so forth.

### 10.13.5 Specification of a conditional event

A conditional event shall be defined by a condition operator with a vector expression and a boolean expression as operands.

The symbols for condition operators are shown in the following Table 83.

**Table 84—Specification a conditional event**

| symbol | operator name | comment |
|---|---|---|
| **&&** or **&** | control-and operator | overloaded symbol, also used for *logical and* (see Table 73) and *bitwise and* (Table 74) |
| **?** | condition operator | see also Table 75 |
| **:** | delimiter between alternatives | see also Table 75 |

A conditional event involving the control-and operator, an arbitrary vector expression *v* and an arbitrary boolean expression *b* shall be mathematically defined as follows:

s) $v$ **&** $b$ = (***1** $b$) -> $v$ -> (**1*** $b$)

The vector expression *v* shall be evaluated while *b* is true. Commutativity shall apply for the operands *v* and *b*.

t) $v$ **&** $b$ = $b$ **&** $v$

A conditional event involving the condition operator, the delimiter between alternatives, arbitrary vector expressions $v_1$ and $v_2$ and an arbitrary boolean expression *b* shall be mathematically defined as follows:

u) $b$ **?** $v_1$ **:** $v_2$ = $v_1$ **&** $b$ | $v_2$ **& !** $b$

If the boolean expression to the left of the condition operator evaluates true, the vector expression to the right of the condition operator shall be evaluated. Otherwise, the boolean expression to the right of the delimiter between alternatives shall be evaluated. If multiple conditions and alternatives exist, the evaluation shall proceed from the left to the right.

### 10.13.6 Operator priorities

The binding priority of operations in a vector expression shall be from the strongest to the weakest in the following order:

*a)* operation enclosed by *parentheses*
b) *vector unary* , i.e., *edge literal*
c) *permutation* operators (<->, <~>, <&>)
d) *and* operator (&, &&), to be interpreted as *simultaneous occurence* or as *control-and*
e) *followed-by* operators (->, ~>, &>)
f) *or* operator (|, ||), to be interpreted as *alternative*
g) operator and delimiter for *conditional operation* (?, :)

When operations of the same binding priority are subsequently encountered in a boolean expression, the evaluation shall proceed from the left to the right.

# 11. Constructs for electrical and physical modeling

**Add lead-in text**

## 11.1 Arithmetic expression

An *arithmetic expression* shall be defined as shown in Syntax 79.

```
arithmetic_expression ::=
      ( arithmetic_expression )
    | arithmetic_value
    | { boolean_expression ? arithmetic_expression : } arithmetic_expression
    | [ unary_arithmetic_operator ] arithmetic_operand
    | arithmetic_operand binary_arithmetic_operator arithmetic_operand
    | macro_arithmetic_operator ( arithmetic_operand { , arithmetic_operand } )
arithmetic_operand ::=
    arithmetic_expression
```

*Syntax 79—Arithmetic expression*

### 11.1.1 Unary arithmetic operator

An *unary arithmetic operator* shall be defined as shown in Syntax 80.

```
unary_arithmetic_operator ::=
      +
    | -
```

*Syntax 80—Unary arithmetic operator*

Table 85 defines the semantics of unary arithmetic operators.

**Table 85—Unary arithmetic operators**

| Operator | Description | Comment |
|---|---|---|
| + | Positive sign. | Neutral operator. |
| - | Negative sign. | |

### 11.1.2 Binary arithmetic operator

A *binary arithmetic operator* shall be defined as shown in Syntax 81.

```
binary_arithmetic_operator ::=
    +
  | -
  | *
  | /
  | **
  | %
```

*Syntax 81—Binary arithmetic operator*

Table 86 defines the semantics of binary arithmetic operators.

**Table 86—Binary arithmetic operators**

| Operator | Description | Comment |
|---|---|---|
| + | Addition | |
| - | Subtraction | |
| * | Multiplication | |
| / | Division | Result includes fractional part. |
| ** | Power | |
| % | Modulus | Remainder of division. |

### 11.1.3 Macro arithmetic operator

A *macro arithmetic operator* shall be defined as shown in Syntax 82.

```
macro_arithmetic_operator ::=
    abs
  | exp
  | log
  | min
  | max
```

*Syntax 82—Macro arithmetic operator*

Table 87 defines the semantics of macro arithmetic operators.

**Table 87—Macro arithmetic operators**

| Operator | Description | Comment |
|---|---|---|
| log | Natural logarithm. | 1 operand, `operand > 0`. |
| exp | Natural exponential. | 1 operand. |
| abs | Absolute value. | 1 operand. |
| min | Minimum. | `N>1` operands. |

**Table 87—Macro arithmetic operators (Continued)**

| Operator | Description | Comment |
|---|---|---|
| max | Maximum. | N>1 operands. |

The priority of operators in arithmetic expressions shall be from strongest to weakest in the following order:

a) unary arithmetic operator (+, -)
b) power (**)
c) multiplication (*), division (/), modulo division (%)
d) addition (+), subtraction (-)

*Examples for arithmetic expressions*

```
1.24
- Vdd
C1 + C2
MAX ( 3.5*C , -Vdd/2 , 0.0 )
(C > 10) ? Vdd**2 : 1/2*Vdd - 0.5*C
```

## 11.2 Arithmetic model

An *arithmetic model* shall be defined as a *trivial arithmetic_model*, a *partial arithmetic model*, or a *full arithmetic model*, as shown in Syntax 83.

```
arithmetic_model ::=
      trivial_arithmetic_model
    | partial_arithmetic_model
    | full_arithmetic_model
    | arithmetic_model_template_instantiation
```

*Syntax 83—Arithmetic model statement*

The purpose of an arithmetic model is to specify a measurable or a calculable quantity.

### 11.2.1 Trivial arithmetic model

A *trivial arithmetic model* shall be defined as shown in Syntax 84.

```
trivial_arithmetic_model ::=
      nonescaped_identifier [ name_identifier ] = arithmetic_value ;
    | nonescaped_identifier [ name_identifier ] = arithmetic_value { { model_qualifier } }
```

*Syntax 84—Trivial arithmetic model*

No mathematical operation is necessary to evaluate a trivial arithmetic model. The arithmetic value associated with the arithmetic model represents the evaluation result. One or more *model qualifier* statements can be associated with a trivial arithmetic model.

### 11.2.2 Partial arithmetic model

A *partial arithmetic model* shall be defined as shown in Syntax 85.

```
partial_arithmetic_model ::=
    nonescaped_identifier [ name_identifier ] { { partial_arithmetic_model_item } }
partial_arithmetic_model_item ::=
      model_qualifier
    | table
    | trivial_min-max
```

*Syntax 85—Partial arithmetic model*

A partial arithmetic model does not specify a mathematical operation or an arithmetic value. Therefore it can not be mathematically evaluated.

The purpose of a partial arithmetic model is to specify one or more *model qualifier* statements, a *table* statement, or a *trivial min-max* statement. The specification contained within a partial arithmetic model can be inherited by another arithmetic model of the same type, according to the following rules:

a) If the partial arithmetic model has no name, the specification shall be inherited by all arithmetic models of the same type appearing within the same parent statement or within a descendant of the same parent statement.
b) If the partial arithmetic model has a name, the specification shall be only inherited by an arithmetic model containing a reference to the partial arithmetic model, using the *model reference annotation* (see **event_reference??).
c) An arithmetic model can override an inherited specification by its own specification.

### 11.2.3 Full arithmetic model

A *full arithmetic model* shall be defined as shown in Syntax 86.

```
full_arithmetic_model ::=
    nonescaped_identifier [ name_identifier ] { { model_qualifier } model_body { model_qualifier } }
model_body ::=
      header-table-equation [ trivial_min-max ]
    | min-typ-max
    | arithmetic_submodel { arithmetic_submodel }
```

*Syntax 86—Full arithmetic model*

The *model body* specifies mathematical data associated with the arithmetic model. The data is represented either by a *header-table-equation* statement, or by a *min-typ-max* statement, or by one or more *arithmetic submodel* statements.

The mathematical operation or the arithmetic value for evaluation of the arithmetic model can be contained within one or more arithmetic submodels (see 11.4.3). The selection of an applicable submodel is controlled by the semantics of the keyword that identifies the type of the arithmetic submodel.

## 11.3 HEADER, TABLE, and EQUATION

A *header table equation* statement shall be defines as shown in Syntax 87.

```
header-table-equation ::=
    header table
  | header equation
```

*Syntax 87—Header table equation*

A header-table-equation statement specifies a procedure for evaluation of the mathematical data.

### 11.3.1 HEADER statement

A *header* statement shall be defined as shown in Syntax 88.

```
header ::=
    HEADER { partial_arithmetic_model { partial_arithmetic_model } }
```

*Syntax 88—HEADER statement*

Each partial arithmetic model within the header statement shall represent a *dimension* of an arithmetic model.

### 11.3.2 TABLE statement

A *table* statement shall be defined as shown in Syntax 89.

```
table ::=
    TABLE { arithmetic_value { arithmetic value } }
```

*Syntax 89—TABLE statement*

A table statement within a partial arithmetic model shall define the set of legal values for an arithmetic model that inherits the specification of the partial arithmetic model.

A table statement within a full arithmetic model shall represent a lookup table. If the model body contains a table statement, each dimension within the header statement shall also contain a table statement.

The mathematical relation between a lookup table and its dimensions shall be established as follows:

$$S = \prod_{i=1}^{N} S(i) \qquad N \geq 1$$

$$S \geq 1$$

$$0 \leq P \leq S - 1$$

$$P = \sum_{i=1}^{N} P(i) \prod_{k=1}^{i-1} S(k) \qquad S(i) \geq 1$$

$$0 \leq P(i) \leq S(i) - 1$$

where

*N* denotes the number of dimensions
*S* denotes the size of the lookup table, i.e., the number of arithmetic values within the lookup table
*P* denotes the position of an arithmetic value within the lookup table
*S*(*i*) denotes the size of a dimension, i.e., the number of arithmetic values in the table within a dimension
*P*(*i*) denotes the position of an arithmetic value within a dimension

A dimension can be either discrete or continuous. In the latter case, interpolation and extrapolation of table values is allowed, and the arithmetic values in this dimension shall appear in strictly monotonous ascending order.

### 11.3.3 EQUATION statement

An *equation* statement shall be defined as shown in Syntax 90.

```
equation ::=
      EQUATION { arithmetic_expression }
    | equation_template_instantiation
```

*Syntax 90—EQUATION statement*

The arithmetic expression within the equation statement shall represent the mathematical operation for evaluation of the arithmetic model.

Each dimension shall be involved in the arithmetic expression. The arithmetic expression shall refer to a dimension by name, if a name identifier exists or by type otherwise. Consequently, the type or the name of a dimension shall be unique.

## 11.4 Statements related to arithmetic model

**Add lead-in text**

### 11.4.1 Model qualifier

A *model qualifier* statement shall be defined as shown in Syntax 91.

```
model_qualifier ::=
      annotation
    | annotation_container
    | event_reference
    | from-to
    | auxiliary_arithmetic_model
    | violation
```

*Syntax 91—Model qualifier statement*

### 11.4.2 Auxiliary arithmetic model

An *auxiliary arithmetic model* shall be defined as shown in Syntax 92.

```
auxiliary_arithmetic_model ::=
      nonescaped_identifier = arithmetic_value ;
    | nonescaped_identifier [ = arithmetic_value ] { auxiliary_qualifier { auxiliary_qualifier } }
auxiliary_qualifier
      annotation
    | annotation_container
    | event_reference
    | from-to
```

*Syntax 92—Auxiliary arithmetic model*

An auxiliary arithmetic model can be considered as a special case of either a trivial arithmetic model or a partial arithmetic model, since the rule for *auxiliary qualifier* is a true subset of the rule for *model qualifier.* In particular, the items *auxiliary arithmetic model* and *violation* are disallowed in the rule for auxiliary qualifier.

### 11.4.3 Arithmetic submodel

An *arithmetic submodel* shall be defined as shown in Syntax 93.

```
arithmetic_submodel ::=
      nonescaped_identifier = arithmetic_value ;
    | nonescaped_identifier { [ violation ] min-max }
    | nonescaped_identifier { header-table-equation [ trivial_min-max ] }
    | nonescaped_identifier { min-typ-max }
    | arithmetic_submodel_template_instantiation
```

*Syntax 93—Arithmetic submodel*

### 11.4.4 MIN-MAX statement

A *min-max* statement shall be defined as shown in Syntax 94.

```
min-max ::=
      min [ max ]
    | max [ min ]
min ::=
      MIN = arithmetic_value ;
    | MIN = arithmetic_value { violation }
    | MIN { [ violation ] header-table-equation }
max ::=
      MAX = arithmetic_value ;
    | MAX = arithmetic_value { violation }
    | MAX { [ violation ] header-table-equation }
```

*Syntax 94—MIN-MAX statement*

### 11.4.5 MIN-TYP-MAX statement

A *min-typ-max* statement shall be defined as shown in Syntax 95.

```
min-typ-max ::=
    [ min-max ] typ [ min-max ]
typ ::=
      TYP = arithmetic_value ;
    | TYP { header-table-equation }
```

*Syntax 95—MIN-TYP-MAX statement*

### 11.4.6 Trivial MIN-MAX statement

A *trivial min-max* statement shall be defined as shown in Syntax 96

A trivial min-max statement defines the legal range of values for an arithmetic model. The arithmetic value associated with the *trivial min* statement represent the smallest legal number. The arithmetic value associated with the *trivial max* statement represents the greatest legal number. Per default, the range includes between negative and positive infinity.

```
trivial_min-max ::=
      trivial_min [ trivial_max ]
    | trivial_max [ trivial_min ]
trivial_min ::=
    MIN = arithmetic_value ;
trivial_max ::=
    MAX = arithmetic_value ;
```

*Syntax 96—Trivial MIN-MAX statement*

A trivial min-max statement within a dimension of a full arithmetic model defines the range of validity of a particular dimension. An application tool can still evaluate the header-table-equation statement outside the range of validity, however, the accuracy of the evaluation can not be guaranteed.

The following semantic restrictions shall apply:

a) A partial arithmetic model that is not a dimension of a lookup table can either contain a trivial min-max statement or a table statement but not both.
b) If a syntax rule allows both partial arithmetic model and full arithmetic model, a trivial min-max statement shall be interpreted as a min-typ-max statement, if the arithmetic model contains neither a header-table-equation statement nor a arithmetic submodel and no other arithmetic model can inherit the trivial min-max statement.

Rule a) is established because a trivial min-max statement would be redundant or eventually contradictory to a table statement, since the table statement already defines a discrete set of legal values.

Rule b) is established because the syntax rule for trivial min-max statement is a true subset of the syntax rule for min-typ-max statement.

### 11.4.7 Arithmetic model container

An *arithmetic model container* shall be defined as shown in Syntax 97.

```
arithmetic_model_container ::=
    arithmetic_model_container_identifier { arithmetic_model  { arithmetic_model } }
```

*Syntax 97—Arithmetic model container*

### 11.4.8 LIMIT statement

A *limit statement* shall be defined as shown in Syntax 98.

```
limit ::=
    LIMIT { limit_item { limit_item } }
limit_item ::=
    limit_arithmetic_model
limit_arithmetic_model ::=
    nonescaped_identifier [ name_identifier ] { { model_qualifier } limit_arithmetic_model_body }
limit_arithmetic_model_body ::=
      limit_arithmetic_submodel { limit_arithmetic_submodel }
    | min_max
limit_arithmetic_submodel ::=
    nonescaped_identifier { [ violation ] min-max }
```

*Syntax 98—LIMIT statement*

### 11.4.9 Event reference statement

An *event reference* statement shall be defined as shown in Syntax 99.

```
event_reference ::=
    PIN_reference_single_value_annotation [ EDGE_NUMBER_single_value_annotation ]
```

*Syntax 99—Event reference statement*

### 11.4.10 FROM and TO statements

A *from* or *to* statement shall be defined as shown in Syntax 100.

```
from-to ::=
      from [to]
    | [ from ] to
from ::=
    FROM { from-to_item { from-to_item } }
from-to_item ::=
      event_reference
    | THRESHOLD_arithmetic_model
to ::=
    TO { from-to_item { from-to_item } }
```

*Syntax 100—FROM and TO statements*

The event referred by the from-statement and the to-statement, respectively, shall be called *from-event* and *to-event*, respectively.

The from-and to-statements are subjected to the semantic restriction shown in Syntax 101.

```
SEMANTICS FROM {
  CONTEXT {
    TIME DELAY RETAIN SLEWRATE PULSSEWIDTH
    SETUP HOLD RECOVERY REMOVAL NOCHANGE ILLEGAL SKEW
  }
}
SEMANTICS TO {
  CONTEXT {
    TIME DELAY RETAIN SLEWRATE PULSSEWIDTH
    SETUP HOLD RECOVERY REMOVAL NOCHANGE ILLEGAL SKEW
  }
}
```

*Syntax 101— Semantic restriction*

### 11.4.11 EARLY and LATE statements

An *early* or a *late* statement shall be defined as shown in Syntax 102.

### 11.4.12 VIOLATION statement

A *violation* statement shall be defined as shown in Syntax 103.

```
early-late ::=
    early late
early ::=
    EARLY { early-late_item { early-late_item } }
early-late_item ::=
     DELAY_arithmetic_model
    | RETAIN_arithmetic_model
    | SLEWRATE_arithmetic_model
late ::=
    LATE { early-late_item { early-late_item } }
```

*Syntax 102—EARLY and LATE statements*

```
violation ::=
     VIOLATION { violation_item { violation_item } }
    | violation_template_instantiation
violation_item ::=
     MESSAGE_TYPE_single_value_annotation
    | MESSAGE_single_value_annotation
    | behavior
```

*Syntax 103—VIOLATION statement*

A violation statement is subjected to the semantic restriction shown in Semantics 60.

```
SEMANTICS VIOLATION {
  CONTEXT {
    SETUP HOLD RECOVERY REMOVAL SKEW NOCHANGE ILLEGAL
    LIMIT.arithmetic_model
    LIMIT.arithmetic_model.MIN
    LIMIT.arithmetic_model.MAX
    LIMIT.arithmetic_model.arithmetic_submodel
    LIMIT.arithmetic_model.arithmetic_submodel.MIN
    LIMIT.arithmetic_model.arithmetic_submodel.MAX
  }
}
```

*Semantics 60—VIOLATION restriction*

A violation statement can contain a behavior statement, as shown in Semantics 61.

```
SEMANTICS VIOLATION.BEHAVIOR {
  CONTEXT {
    VECTOR.arithmetic_model
    VECTOR.LIMIT.arithmetic_model
    VECTOR.LIMIT.arithmetic_model.MIN
    VECTOR.LIMIT.arithmetic_model.MAX
    VECTOR.LIMIT.arithmetic_model.arithmetic_submodel
    VECTOR.LIMIT.arithmetic_model.arithmetic_submodel.MIN
    VECTOR.LIMIT.arithmetic_model.arithmetic_submodel.MAX
  }
}
```

*Semantics 61—VIOLATION.BEHAVIOR restriction*

The *violation* statement can contain a *message-type* annotation and a *message* annotation.

A *message_type* annotation shall be defined as shown in Semantics 62.

```
KEYWORD MESSAGE_TYPE = single_value_annotation {
  CONTEXT = VIOLATION ;
  VALUETYPE = identifier ;
  VALUES { information warning error }
}
```

*Semantics 62—MESSAGE_TYPE annotation*

A *message* annotation shall be defined as shown in Semantics 63.

```
KEYWORD MESSAGE = single_value_annotation {
  CONTEXT = VIOLATION ;
  VALUETYPE = quoted_string ;
}
```

*Semantics 63—MESSAGE annotation*

## 11.5 Annotations for arithmetic models

**Add lead-in text**

### 11.5.1 UNIT annotation

A *unit* annotation shall be defined as shown in Semantics 64.

```
KEYWORD UNIT = annotation {
  CONTEXT = arithmetic_model ;
  VALUETYPE = quantity_value ;
  DEFAULT = 1 ;
}
```

*Semantics 64—UNIT annotation*

### 11.5.2 CALCULATION annotation

A *calculation* annotation shall be defined as shown in Semantics 65.

```
KEYWORD CALCULATION = annotation {
  CONTEXT = library_specific_object.arithmetic_model ;
  VALUES { absolute incremental }
  DEFAULT = absolute ;
}
```

*Semantics 65—CALCULATION annotation*

The meaning of the annotation values is shown in Table 88.

**Table 88—Calculation annotations**

| Annotation value | Description |
|---|---|
| `absolute` | The arithmetic model data is complete within itself. |
| `incremental` | The arithmetic model data shall be combined with other arithmetic model data. |

### 11.5.3 INTERPOLATION annotation

A *interpolation* annotation shall be defined as shown in Semantics 66.

```
KEYWORD INTERPOLATION = single_value_annotation {
  CONTEXT = HEADER.arithmetic_model ;
  VALUES { linear fit ceiling floor }
  DEFAULT = fit ;
}
```

*Semantics 66—INTERPOLATION annotation*

The interpolation annotation shall apply for a dimension of a lookup table with a continuous range of values. Every dimension in a lookup table can have its own interpolation annotation.

The meaning of the annotation values is shown in Table 89.

**Table 89—Interpolation annotations**

| Annotation value | Description |
|---|---|
| `linear` | Linear interpolation shall be used. |
| `ceiling` | The next greater value in the table shall be used. |
| `floor` | The next lesser value in the table shall be used. |
| `fit` | Linear or higher-order interpolation shall be used. |

The mathematical operations for *floor*, *ceiling*, and *linear* are specified as follows:

floor $y(x) = y(x^-)$

ceiling $y(x) = y(x^+)$

linear $$y(x) = \frac{(x - x^-) \cdot y(x^+) + (x^+ - x) \cdot y(x^-)}{x^+ - x^-}$$

where

$x$ denotes the value in a dimension subjected to interpolation.
$x^-$ and $x^+$ denote two subsequent values in the table associated with that dimension.
$x^-$ denotes the value to the left of $x$, such that $x^- < x$, or else $x^-$ denotes the smallest value in the table.
$x^+$ denotes the value to the right of $x$, such that $x < x^+$, or else $x^+$ denotes the largest value in the table.
$y$ denotes the evaluation result of the arithmetic model.

The mathematical operation for *fit* can be chosen by the application, as long as the following conditions are satisfied:

$y(x)$ is a continuous function of order $N>0$.
The first $N$-1 derivatives of $y(x)$ are continuous.
$y(x)$ is bound by $y(x^-)$ and $y(x^+)$.
In case of monotony, $y(x)$ is also bound by linear interpolation applied to the left and the right neighbor of $x$.
In case of monotonous derivative, $y(x)$ is also bound by linear interpolation applied to $x$ itself.

These conditions are illustrated in Figure 17.

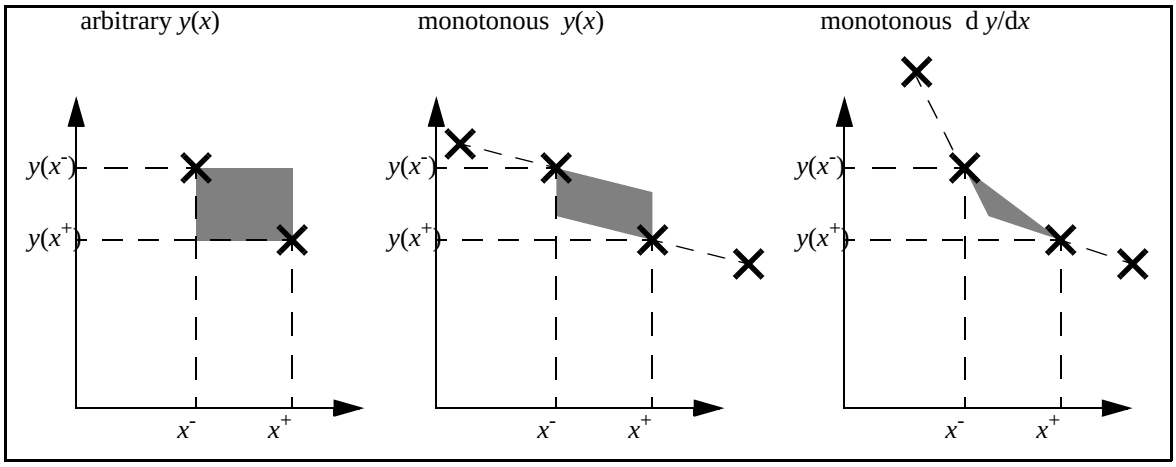



**Figure 17—Bounding regions for y(x) with INTERPOLATION=fit**

### 11.5.4 DEFAULT annotation

A *default* annotation shall be defined as shown in Semantics 67.

```
KEYWORD DEFAULT = single_value_annotation {
  CONTEXT { arithmetic_model KEYWORD }
  VALUETYPE = all_purpose_value ;
}
```

*Semantics 67—DEFAULT annotation*

### 11.5.5 MODEL annotation

A *model* annotation shall be defined as shown in Semantics 68.

```
KEYWORD MODEL = single_value_annotation {
  CONTEXT = arithmetic_model ;
  VALUETYPE = identifier ;
}
```

*Semantics 68—MODEL annotation*

## 11.6 TIME

**Is this (and some 35 other constucts after this) a *statement*, an *annotation*, *or some 'other grouping'*??
**and should their label(s), therefore, be *Syntax*, *Semantics*, *or some* `new_name`??

**If these constructs are really *statements*, they need to be converted in (true BNF) syntax boxes**

A *time* statement shall be defined as shown in Syntax 104.

```
KEYWORD TIME = arithmetic_model {
  VALUETYPE = number ;
}
TIME { UNIT = 1e-9; }
```

*Syntax 104—TIME statement*

A time statement can have a from-to statement as model qualifier.

### 11.6.1 TIME in context of a VECTOR declaration

A time statement can be a child or a grandchild of a vector declaration. In particular, the parent of the time statement can be a limit statement. In the context of a limit statement, the time statement shall specify a smallest required time or a largest allowed time interval. Otherwise, the time statement shall specify an actually measured time interval.

If the vector declaration involves a vector expression, from-to statements featuring event reference statements shall be used as model qualifier. The time statement shall model the measured time interval between the referred events.

If the vector declaration involves a boolean expression, the time statement applies to a time interval during which the boolean expression is true. A from-to statement shall not be used as model qualifier.

### 11.6.2 TIME in context of a HEADER statement

A time statement can be child of a header statement, thus representing a dimension of an arithmetic model.

If the arithmetic model is not a child of a limit statement, the time dimension shall be used to describe a quantity changing over time, which can be visualized by a waveform.

If the arithmetic model is a child of a vector declaration, either a from statement or a to statement can be used as model qualifier to define a temporal relationship between a referred event and the time dimension.

If the arithmetic model is a child of a limit statement, the time dimension shall be used to describe a dependency between a limit for a measured quantity and the expected lifetime of an electronic circuit. A from-to statement shall not be used as model qualifier.

### 11.6.3 TIME as auxiliary arithmetic model

A time statement can be a child of an arithmetic model, thus representing an auxiliary arithmetic model.

A *measurement* annotation (see 11.29.1) shall be used in conjunction with the time statement. The time statement shall specify the time interval during which the measurement is taken.

If the parent arithmetic model is a child of a vector declaration, a from-to statement can be used to define a temporal relationship between one or two events in the vector expression and the time interval.

## 11.7 FREQUENCY

A *frequency* statement shall be defined as shown in Syntax 105.

```
KEYWORD FREQUENCY = arithmetic_model {
  VALUETYPE = number ;
}
FREQUENCY { UNIT = 1e9; MIN = 0; }
```

*Syntax 105—FREQUENCY statement*

### 11.7.1 FREQUENCY in context of a VECTOR declaration

A frequency statement can be a child or a grandchild of a vector declaration. In particular, the parent of the frequency statement can be a limit statement. In the context of a limit statement, the frequency statement shall specify a smallest required occurrence frequency or a largest allowed occurrence frequency of the vector. Otherwise, the frequency statement shall specify an actually measured occurrence frequency of the vector.

### 11.7.2 FREQUENCY in context of a HEADER statement

A frequency statement can be child of a header statement, thus representing a dimension of an arithmetic model.

If the arithmetic model is a child of a vector declaration, the frequency dimension shall represent the occurrence frequency of the vector.

If the arithmetic model is not a child of a vector declaration, the frequency dimension shall be used to describe a spectral properties of the arithmetic model in the frequency domain.

### 11.7.3 FREQUENCY as auxiliary arithmetic model

A frequency statement can be a child of an arithmetic model, thus representing an auxiliary arithmetic model.

A *measurement* annotation (see 11.29.1) shall be used in conjunction with the frequency statement. The frequency statement shall specify the repetition frequency of the measurement.

A frequency statement can substitute a time statement in the capacity of an auxiliary arithmetic model, if no from-to statement is used as model qualifier. In this case, the measurement repetition frequency $f$ and the measurement time interval $t$ can be related by the equation $f = 1 / t$.

## 11.8 DELAY

A *delay* statement shall be defined using ALF language as shown in Syntax 106.

```
KEYWORD DELAY = arithmetic_model {
  SI_MODEL = TIME ;
}
```

*Syntax 106—DELAY statement*

### 11.8.1 DELAY in context of a VECTOR declaration

A delay statement can be a child or a grandchild of a vector declaration involving a vector expression. A delay statement shall have a from-to statement featuring event references as model qualifier. The delay statement shall define the measured time interval between a from-event and a to-event. Both events shall be part of the vector expression. A causal relationship between the from-event and the to-event is implied.

A delay statement with an incomplete model qualifier featuring only a from statement or only a to statement can be used to specify an incremental time interval to be added to another time interval. The calculation annotation (see 11.5.2) shall be used in conjunction with such an incomplete model qualifier.

### 11.8.2 DELAY in context of a library-specific object declaration

A delay statement can be a child of a library-specific object which can be a parent of a vector. Possible parents of a vector include library, sublibrary, cell and wire. Within such a context, a delay statement can not have an event reference within a from-to statement as model qualifier. A from-to statement can only feature threshold statements. The specification given by the threshold statements shall be inherited by delay statements which are child of a vector.

## 11.9 RETAIN

A *retain* statement shall be defined as shown in Syntax 107.

```
KEYWORD RETAIN = arithmetic_model {
  SI_MODEL = TIME ;
}
```

*Syntax 107—RETAIN statement*

A retain statement can be a child or a grandchild of a vector declaration involving a vector expression. A retain statement can be used as a substitution for a delay statement in the case where the vector expression features more than one possible to-event. Retain represents the time interval between the from-event and the earliest to-event. Later to-events can be involved in a delay statement.

Retain in conjunction with delay is illustrated in Figure 18.

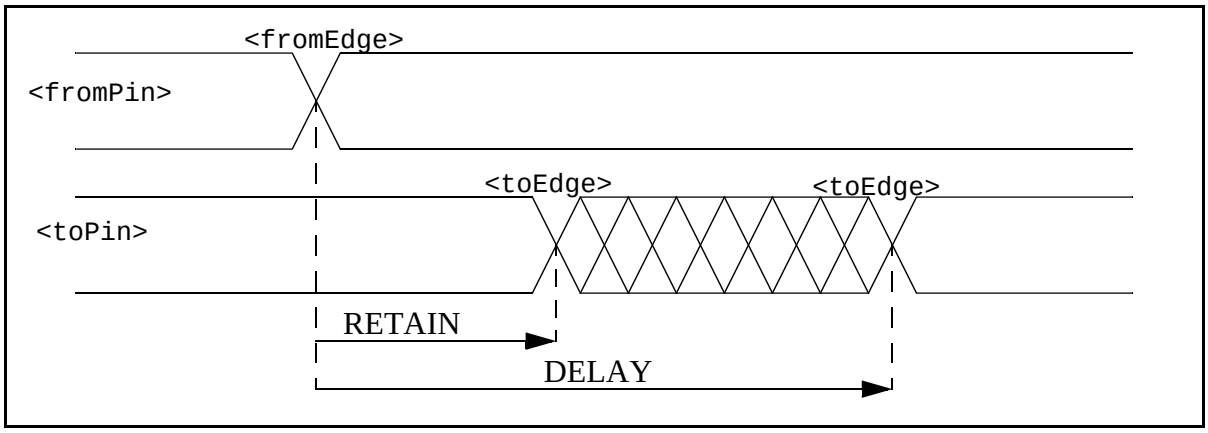



**Figure 18—RETAIN and DELAY**

## 11.10 SLEWRATE

A *slewrate* statement shall be defined as shown in Syntax 108.

```
KEYWORD SLEWRATE = arithmetic_model {
  SI_MODEL = TIME ;
}
SLEWRATE { MIN = 0; }
```

*Syntax 108—SLEWRATE statement*

Slewrate shall define the duration of a single event, measured between two reference transition points. If the parent of the slewrate statement is a limit statement, the slewrate statement defines a minimum required or a maximum allowed duration of an event. Otherwise, slewrate defines the actually measured duration of an event.

### 11.10.1 SLEWRATE in context of a VECTOR declaration

A slewrate statement can be a child or a grandchild of a vector declaration. Slewrate can also be a dimension of an arithmetic model in the context of a vector.

The slewrate statement can have an event reference statement and a from-to statement without event reference as model qualifier. The from-and the to-statement can involve threshold statements.

### 11.10.2 SLEWRATE in context of a PIN declaration

A slewrate statement can be a child or a grandchild of a pin declaration. In this context, no from-to statement and no event-reference statement is allowed as model qualifier.

The slewrate statement can have a rise statement or a fall statement as arithmetic submodel.

### 11.10.3 SLEWRATE in context of a library-specific object declaration

A slewrate statement can be a child of a library-specific object which can be a parent of a vector. Possible parents of a vector include library, sublibrary, cell and wire. Within such a context, a slewrate statement can not have an event reference as model qualifier. A from-to statement with threshold statements can be used as model qualifier.

The specification given by the threshold statements can be inherited by slewrate statements which are child of a vector.

The slewrate statement can have a rise statement or a fall statement as arithmetic submodel.

## 11.11 SETUP and HOLD

A *setup* or *hold* statement shall be defined as shown in Syntax 109.

```
KEYWORD SETUP = arithmetic_model {
  SI_MODEL = TIME ;
}
KEYWORD HOLD = arithmetic_model {
  SI_MODEL = TIME ;
}
```

*Syntax 109—SETUP and HOLD statements*

### 11.11.1 SETUP in context of a VECTOR declaration

A setup statement can be a child of a vector declaration. Setup represents the minimal required time interval between a signal event and a synchronization event such that the signal is already stable when the synchronization event occurs. The signal event and the synchronization event shall be represented as a from-event and a to-event, respectively, within a from-to statement.

### 11.11.2 HOLD in context of a VECTOR declaration

A hold statement can be a child of a vector declaration.Hold represents the minimal required time interval between a synchronization event and a signal event such that the synchronization event occurs while the signal is still stable. The synchronization event and the signal event shall be represented as a from-event and a to-event, respectively, within a from-to statement.

### 11.11.3 SETUP and HOLD in context of the same VECTOR declaration

A setup and a hold statement can be a child of the same vector, provided the vector expression features at least one synchronization event and two signal events related to the synchronization event. The sum of the time intervals represented by setup and hold represents a minimum required stability interval for the signal. This interval shall be greater than zero.

Setup in conjunction with hold is illustrated in Figure 19.

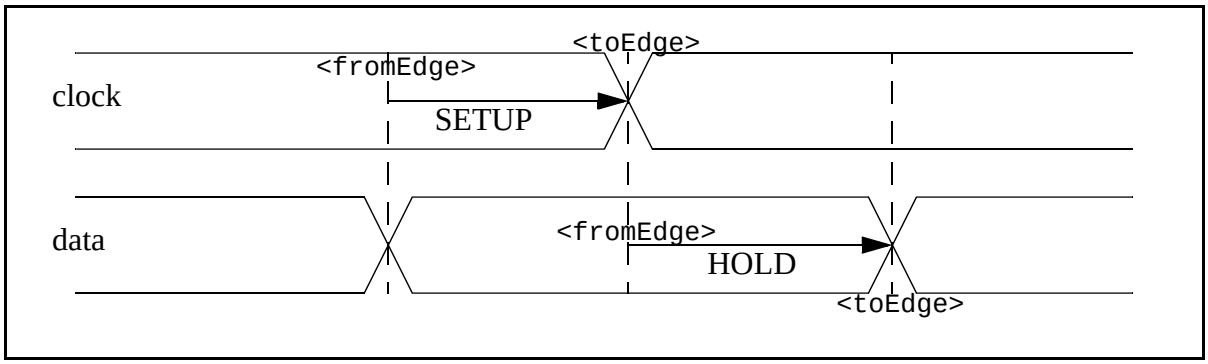



**Figure 19—SETUP and HOLD**

## 11.12 RECOVERY and REMOVAL

A *recovery* or *removal* statement shall be defined as shown in Syntax 110.

```
KEYWORD RECOVERY = arithmetic_model {
  SI_MODEL = TIME ;
}
KEYWORD REMOVAL = arithmetic_model {
  SI_MODEL = TIME ;
}
```

*Syntax 110—RECOVERY and REMOVAL statements*

### 11.12.1 RECOVERY in context of a VECTOR declaration

A recovery statement can be a child of a vector declaration. Recovery represents the minimal required time interval between a controlling event with higher priority and a controlling event with lower priority such that the signal with higher priority is already inactive when the event on the signal with lower priority occurs. The event with higher priority and the event with lower priority shall be represented as a from-event and a to-event, respectively, within a from-to statement.

### 11.12.2 REMOVAL in context of a VECTOR declaration

A removal statement can be a child of a vector declaration. Removal represents the minimal required time interval between a controlling event with lower priority and a controlling event with higher priority such that the signal with higher priority is still active when the event with lower priority occurs. The event with higher priority and the event with lower priority shall be represented as a from-event and a to-event, respectively, within a from-to statement.

### 11.12.3 RECOVERY and REMOVAL in context of the same VECTOR declaration

A recovery and a removal statement can be a child of the same vector, provided the vector expression features at least one event with lower priority and two alternative events with higher priority. The sum of the time intervals represented by recovery and removal represents a minimum required stability interval for the signal with higher priority. This interval shall be greater than zero.

Recovery in conjunction with removal is illustrated in Figure 20.

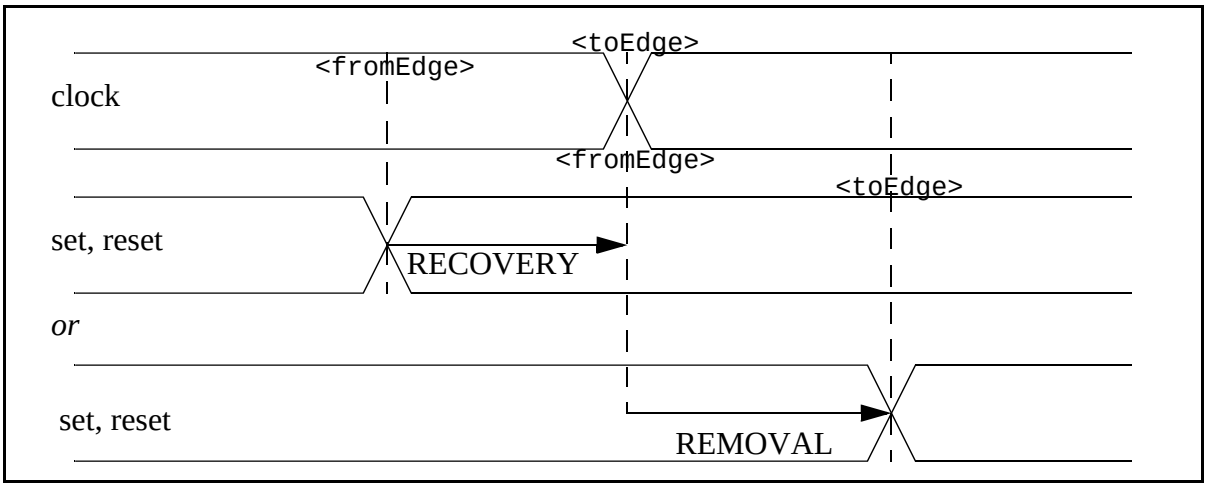



**Figure 20—RECOVERY and REMOVAL**

## 11.13 NOCHANGE and ILLEGAL

A *nochange* or an *illegal* statement shall be defined as shown in Syntax 111.

```
KEYWORD NOCHANGE = arithmetic_model {
  SI_MODEL = TIME ;
}
KEYWORD ILLEGAL = arithmetic_model {
  SI_MODEL = TIME ;
}
NOCHANGE { MIN = 0; }
ILLEGAL { MIN = 0; }
```

*Syntax 111—NOCHANGE and ILLEGAL statements*

### 11.13.1 NOCHANGE in context of a VECTOR declaration

A nochange statement can be a child of a vector declaration.

If the vector declaration involves a boolean expression, nochange shall specify a minimum required time interval during which the boolean expression is true. Nochange as a partial arithmetic model shall indicate a requirement for the boolean expression to be forever true.

If the vector declaration involves a vector expression, nochange as a partial arithmetic model shall indicate a requirement for the vector expression to be observed as specified. An optional from-to statement as model qualifier can indicate a requirement for the part of the vector expression within the time interval between the from-event and the to-event to be observed as specified. Nochange as a full arithmetic model or as a trivial arithmetic model shall furthermore specify a minimum required duration of the vector expression or part thereof.

### 11.13.2 ILLEGAL in context of a VECTOR declaration

An illegal statement can be a child of a vector declaration.

If the vector declaration involves a boolean expression, illegal shall specify a maximum allowed time interval during which the boolean expression is true. Illegal as a partial arithmetic model shall indicate a requirement for the boolean expression to be never true.

If the vector declaration involves a vector expression, illegal as a partial arithmetic model shall indicate that the vector expression is not allowed to occur. An optional from-to statement as model qualifier can indicate that a part of the vector expression within the time interval between the from-event and the to-event is not allowed to occur. Illegal as a full arithmetic model or as a trivial arithmetic model shall furthermore specify a maximum tolerated duration of the vector expression or part thereof.

## 11.14 SKEW

A *skew* statement shall be defined as shown in Syntax 112.

```
KEYWORD SKEW = arithmetic_model {
  SI_MODEL = TIME ;
}
SKEW { MIN = 0; }
```

*Syntax 112—SKEW statement*

A skew statement can be a child of a vector declaration.

### 11.14.1 SKEW involving two signals

A skew statement can specify a maximum allowed time interval between a from-event and a to-event. In this case, a from-to statement is mandatory as model qualifier. The vector declaration shall specify a vector expression such that the to-event cannot occur before the from-event.

### 11.14.2 SKEW involving multiple signals

A skew statement can specify a maximum allowed time separation between multiple events. In this case, a multi-value annotation containing pin references is mandatory as model qualifier. Optionally, this multi-value annotation can be accompanied by another multi-value annotation containing a matching number of edge numbers. The vector declaration shall specify a vector expression such that all events can occur simultaneously.

## 11.15 PULSEWIDTH

A *pulsewidth* statement shall be defined as shown in Syntax 113.

```
KEYWORD PULSEWIDTH = arithmetic_model {
  SI_MODEL = TIME ;
}
PULSEWIDTH { MIN = 0; }
```

*Syntax 113—PULSEWIDTH statement*

A pulsewidth statement shall define the time interval between two consecutive events on the same signal. If the parent of the pulsewidth statement is a limit statement, pulsewidth defines a minimum required or a maximum allowed duration of the time interval. Otherwise, pulsewidth defines the actually measured time interval.

### 11.15.1 PULSEWIDTH in context of a VECTOR declaration

A pulsewidth statement can be a child of a vector declaration. Pulsewidth can also be a dimension of an arithmetic model in the context of a vector.

The pulsewidth statement can have an event-reference statement and a from-to statement without event reference as model qualifier. The from-and the to-statement can involve threshold statements. The event reference shall refer to the first of two consecutive events.

### 11.15.2 PULSEWIDTH in context of a PIN declaration

A pulsewidth statement can be a child or a grandchild of a pin declaration. In this context, no from-to statement and no event-reference statement is allowed as model qualifier.

The pulsewidth statement can have a rise statement and/or a fall statement as arithmetic submodel. The switching direction indicated by rise or fall shall refer to the first of two consequtive events.

### 11.15.3 PULSEWIDTH in context of a library-specific object declaration

A pulsewidth statement can be a child of a library-specific object which can be a parent of a vector. Possible parents of a vector include library, sublibrary, cell and wire. Within such a context, a pulsewidth statement can not have an event reference as model qualifier. A from-to statement with threshold statements can be used as model qualifier. The specification given by the threshold statements can be inherited by pulsewidth statements which are child of a vector.

The pulsewidth statement can have a rise statement or a fall statement as arithmetic submodel. The switching direction indicated by rise or fall shall refer to the first of two consecutive events.

## 11.16 PERIOD

A *period* statement shall be defined as shown in Syntax 114.

```
KEYWORD PERIOD = arithmetic_model {
  SI_MODEL = TIME ;
}
PERIOD { MIN = 0; }
```

*Syntax 114—PERIOD statement*

A period statement can be a child or a grandchild of a vector. Period can also be a dimension of an arithmetic model in the context of a vector. Period shall define the time interval between two consecutive occurrences of a periodically repeating vector.

If the parent of the period statement is a limit statement, period defines a minimum required or a maximum allowed time interval. Otherwise, period defines the actually measured time interval.

## 11.17 JITTER

A *jitter* statement shall be defined as shown in Syntax 115.

```
KEYWORD JITTER = arithmetic_model {
  SI_MODEL = TIME ;
}
JITTER { MIN = 0; }
```

*Syntax 115—JITTER statement*

A jitter statement can be a child or a grandchild of a vector. Jitter can also be a dimension of an arithmetic model in the context of a vector. Jitter shall define the variability of a time interval between two consecutive occurrences of the periodically repeating vector.

If the parent of the jitter statement is a limit statement, jitter defines a minimum required or a maximum allowed variability of the time interval. Otherwise, jitter defines the actually measured variability of the time interval.

The measurement annotation (see 11.29.1) is applicable as model qualifier.

## 11.18 THRESHOLD

A *threshold* statement shall be defined using ALF language as shown in Syntax 116.

```
KEYWORD THRESHOLD = arithmetic_model {
  CONTEXT { PIN FROM TO }
}
THRESHOLD { MIN = 0; MAX = 1; }
```

*Syntax 116—THRESHOLD statement*

The `THRESHOLD` represents a reference voltage level for timing measurements, normalized to the signal voltage swing and measured with respect to the logic `0` voltage level, as shown in Figure 21.

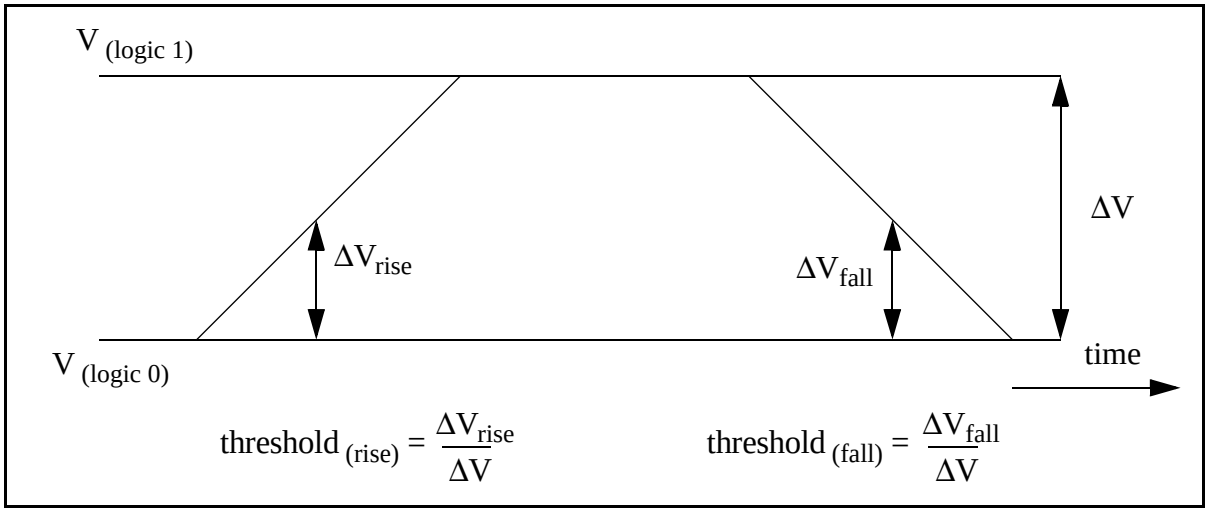



**Figure 21—THRESHOLD measurement definition**

The voltage levels for logic `1` and `0` represent a full voltage swing.

Different threshold data for RISE and FALL can be specified or else the data shall apply for both rising and falling transitions.

The THRESHOLD statement has the form of an arithmetic model. If the submodel keywords RISE and FALL are used, it has the form of an arithmetic model container.

The THRESHOLD statement can appear in the context of a FROM or TO container. In this case, it specifies the applicable reference for the start and end point of the timing measurement, respectively.

The THRESHOLD statement can also appear in the context of a PIN. In this case, it specifies the applicable reference for the start or end point of timing measurements indicated by the PIN annotation inside a FROM or TO container, unless a THRESHOLD is specified explicitly inside the FROM or TO container.

If both the RISE and FALL thresholds are specified and the switching direction of the applicable pin is clearly indicated in the context of a VECTOR, the RISE or FALL data shall be applied accordingly.

If thresholds are needed for exact definition of the model data, the FROM and TO containers shall each contain an arithmetic model for THRESHOLD.

FROM and TO containers with THRESHOLD definitions, yet without PIN annotations, can appear within unnamed timing model definitions in the context of a VECTOR, CELL, WIRE, SUBLIBRARY, or LIBRARY object for the purpose of specifying global threshold definitions for all timing models within scope of the definition. The following priorities apply:

a) THRESHOLD in the HEADER of the timing model
b) THRESHOLD in the FROM or TO statement within the timing model
c) THRESHOLD for timing model definition in the context of the same VECTOR
d) THRESHOLD within the PIN definition
e) THRESHOLD for timing model definition in the context of the same CELL or WIRE
f) THRESHOLD for timing model definition in the context of the same SUBLIBRARY
g) THRESHOLD for timing model definition in the context of the same LIBRARY
h) THRESHOLD for timing model definition outside LIBRARY

## 11.19 Annotations related to timing data

**Add lead-in text**

### 11.19.1 PIN reference annotation

If the timing measurements or timing constraints, respectively, apply semantically for two pins ~~(see 11.9.1.1)~~, the FROM and TO containers shall each contain the PIN annotation.

Otherwise, if the timing measurements or timing constraints apply semantically only to one pin ~~(see 11.9.1.3)~~, the PIN annotation shall be outside the FROM or TO container.

The semantic restrictions shown in Semantics 69 shall apply.

### 11.19.2 EDGE_NUMBER annotation

A *edge_number* annotation shall be defined as shown in Semantics 70.

The EDGE_NUMBER annotation within the context of a timing model shall specify the edge where the timing measurement applies. The timing model shall be in the context of a VECTOR. The EDGE_NUMBER shall have an

```
SEMANTICS PIN = single_value_annotation {
  CONTEXT {
    FROM TO SLEWRATE PULSEWIDTH
    CAPACITANCE RESISTANCE INDUCTANCE VOLTAGE CURRENT
  }
}
SEMANTICS SKEW.PIN = multi_value_annotation ;
```

*Semantics 69—PIN restriction*

```
KEYWORD EDGE_NUMBER = annotation {
  CONTEXT { FROM TO SLEWRATE PULSEWIDTH SKEW }
  VALUETYPE = unsigned_integer ;
  DEFAULT = 0;
}
SEMANTICS EDGE_NUMBER = single_value_annotation {
  CONTEXT { FROM TO SLEWRATE PULSEWIDTH }
}
SEMANTICS SKEW.EDGE_NUMBER = multi_value_annotation ;
```

*Semantics 70—EDGE_NUMBER annotation*

unsigned value pointing to exactly one of subsequent `vector_single_event` expressions applicable to the referenced pin. The `EDGE_NUMBER` shall be counted individually for each pin which appears in the `VECTOR`, starting with zero (`0`).

If the timing measurements or timing constraints apply semantically to two pins ~~(see 11.9.1.1)~~, the `EDGE_NUMBER` annotation shall be legal inside the `FROM` or `TO` container in conjunction with the `PIN` annotation.

Otherwise, if the timing measurements or timing constraints apply semantically only to one pin ~~(see 11.9.1.3)~~, the `EDGE_NUMBER` annotation shall be legal outside the `FROM` or `TO` container in conjunction with the `PIN` annotation.

## 11.20 PROCESS

A *process* statement shall be defined as shown in Syntax 117.

```
KEYWORD PROCESS = arithmetic_model {
  VALUETYPE = identifier ;
}
PROCESS { DEFAULT = nom; TABLE { nom snsp snwp wnsp wnwp } }
```

*Syntax 117—PROCESS statement*

The following identifiers can be used as predefined process corners:

`?n?p`process definition with transistor strength

where `?` can be

`s`strong
`w`weak

The possible process name combinations are shown in Table 90.

**Table 90—Predefined process names**

| Process name | Description |
|---|---|
| `snsp` | Strong NMOS, strong PMOS. |
| `snwp` | Strong NMOS, weak PMOS. |
| `wnsp` | Weak NMOS, strong PMOS. |
| `wnwp` | Weak NMOS, weak PMOS. |

## 11.21 DERATE_CASE

A *derate_case* statement shall be defined as shown in Syntax 118.

```
KEYWORD DERATE_CASE = arithmetic_model {
  VALUETYPE = identifier ;
}
DERATE_CASE { DEFAULT = nom;
  TABLE { nom bccom wccom bcind wcind bcmil wcmil }}
}
```

*Syntax 118—DERATE_CASE statement*

The following identifiers can be used as predefined derating cases:

`nom`nominal case
`bc?`prefix for best case
`wc?`prefix for worst case

where `?` can be

`com`suffix for commercial case
`ind`suffix for industrial case
`mil`suffix for military case

The possible derating case combinations are defined in Table 91.

**Table 91—Predefined derating cases**

| Derating case | Description |
|---|---|
| `bccom` | Best case commercial. |
| `bcind` | Best case industrial. |

**Table 91—Predefined derating cases (Continued)**

| Derating case | Description |
|---|---|
| bcmil | Best case military. |
| wccom | Worst case commercial. |
| wcind | Worst case military. |
| wcmil | Worst case military. |

## 11.22 TEMPERATURE

A *temperature* statement shall be defined as shown in Syntax 119.

```
KEYWORD TEMPERATURE = arithmetic_model {
  VALUETYPE = number ;
}
TEMPERATURE { MIN = -273; }
```

*Syntax 119—TEMPERATURE statement*

TEMPERATURE can be used as argument in the HEADER of an arithmetic model for timing or electrical data. It can also be used as an arithmetic model with DERATE_CASE as argument, in order to describe what temperature applies for the specified derating case.

## 11.23 PIN-related arithmetic models for electrical data

Arithmetic models for electrical data can be associated with a pin of a cell. Their meaning is illustrated in Figure 22.

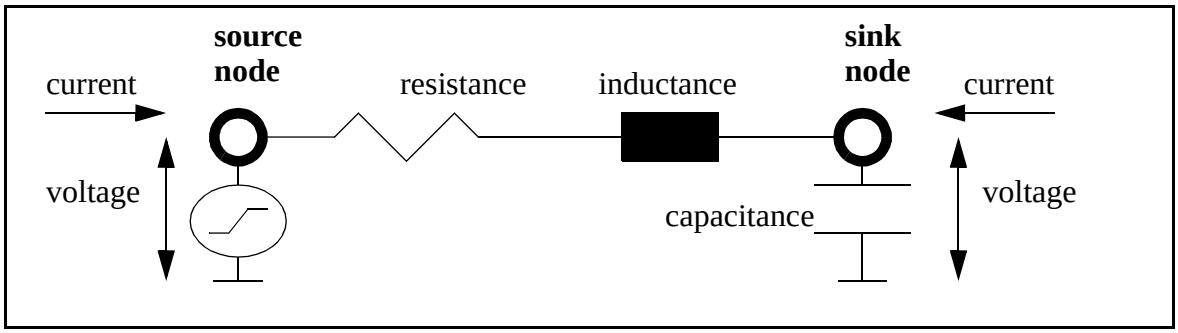



**Figure 22—General representation of electrical models around a pin**

A pin is represented as a source node and a sink node. For pins with DIRECTION=input, the source node is externally accessible. For pins with DIRECTION=output, the sink node is externally accessible.

### 11.23.1 CAPACITANCE, RESISTANCE, and INDUCTANCE

A *capacitance*, *resistance*, or *inductance* statement shall be defined as shown in Syntax 120.

```
KEYWORD CAPACITANCE = arithmetic_model {
  VALUETYPE = number ;
}
KEYWORD RESISTANCE = arithmetic_model {
  VALUETYPE = number ;
}
KEYWORD INDUCTANCE = arithmetic_model {
  VALUETYPE = number ;
}
CAPACITANCE { UNIT = 1e-12; MIN = 0; }
RESISTANCE { UNIT = 1e3; MIN = 0; }
INDUCTANCE { UNIT = 1e-6; MIN = 0; }
```

*Syntax 120—CAPACITANCE, RESISTANCE, and INDUCTANCE statements*

RESISTANCE and INDUCTANCE apply between the source and sink node. CAPACITANCE applies between the sink node and ground. By default, the values for resistance, inductance and capacitance shall be zero (0).

### 11.23.2 VOLTAGE and CURRENT

A *voltage* or *current* statement shall be defined as shown in Syntax 121.

```
KEYWORD VOLTAGE = arithmetic_model {
  VALUETYPE = number ;
}
KEYWORD CURRENT = arithmetic_model {
  VALUETYPE = number ;
}
VOLTAGE { UNIT = 1; }
CURRENT { UNIT = 1e-3; }
```

*Syntax 121—VOLTAGE and CURRENT statements*

VOLTAGE and CURRENT can be measured at either source or sink node, depending on which node is externally accessible. However, a voltage source can only be connected to a source node. The sense of measurement for voltage shall be from the node to ground. The sense of measurement for current shall be *into* the node.

### 11.23.3 Context-specific semantics

An arithmetic model for VOLTAGE, CURRENT, SLEWRATE, RESISTANCE, INDUCTANCE, and CAPACITANCE can be associated with a PIN in one of the following ways.

a) A model in the context of a PIN

*Example*

```
PIN my_pin {
    CAPACITANCE = 0.025;
```

b) A model in the context of a CELL, WIRE, or VECTOR with PIN annotation

*Example*

```
VOLTAGE = 1.8 { PIN = my_pin; }
```

The model in the context of a PIN shall be used if the data is completely confined to the pin. That means, no argument of the model shall make reference to any pin, since such reference implies an external dependency. A model with dependency only on environmental data not associated with a pin (e.g., TEMPERATURE, PROCESS, and DERATE_CASE) can be described within the context of the PIN.

A model with dependency on external data applied to a pin (e.g., load capacitance) shall be described outside the context of the PIN, using a PIN annotation. In particular, if the model involves a dependency on logic state or logic transition of other PINs, the model shall be described within the context of a VECTOR.

Figure 23 illustrates electrical models associated with input and output pins.

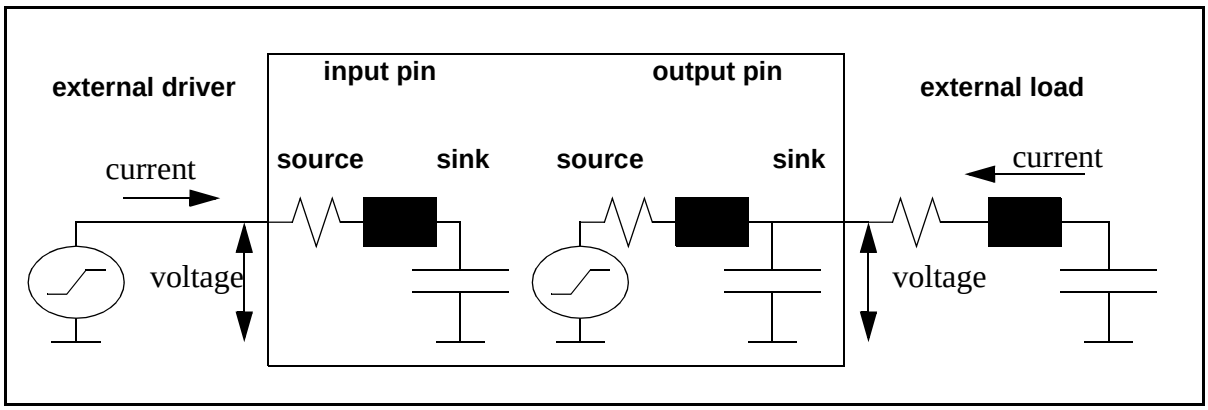



**Figure 23—Electrical models associated with input and output pins**

Table 92 and Table 93 define how models are associated with the pin, depending on the context.

**Table 92—Direct association of models with a PIN**

| Model | Model in context of PIN | Model in context of CELL, WIRE, and VECTOR with PIN annotation |
|---|---|---|
| CAPACITANCE | Pin self-capacitance. | Externally controlled capacitance at the pin, e.g., voltage-dependent. |
| INDUCTANCE | Pin self-inductance. | Externally controlled inductance at the pin, e.g., voltage-dependent. |
| RESISTANCE | Pin self-resistance. | Externally controlled resistance at the pin, e.g., voltage-dependent, in the context of a VECTOR for timing-arc specific driver resistance. |
| VOLTAGE | Operational voltage measured at pin. | Externally controlled voltage at the pin. |
| CURRENT | Operational current measured into pin. | Externally controlled current into pin. |
| *SAME_PIN_TIMING_ MEASUREMENT* | For model definition, default, etc.; not for the timing arc. | In context of VECTOR for timing arc, other context for definition, default, etc. |
| *SAME_PIN_TIMING_ CONSTRAINT* | For model definition, default, etc.; not for the timing arc. | In context of VECTOR for timing arc, other context for definition, default, etc. |

**Table 93—External association of models with a PIN**

| Model / context | LIMIT within PIN or with PIN annotation | Model argument with PIN annotation |
|---|---|---|
| CAPACITANCE | Min or max limit for applicable load. | Load for model characterization. |
| INDUCTANCE | Min or max limit for applicable load. | Load for model characterization. |
| RESISTANCE | Min or max limit for applicable load. | Load for model characterization. |
| VOLTAGE | Min or max limit for applicable voltage. | Voltage for model characterization. |
| CURRENT | Min or max limit for applicable current. | Current for model characterization. |
| *SAME_PIN_TIMING_MEASUREMENT* | Currently applicable for min or max limit for SLEWRATE. | Stimulus with SLEWRATE for model characterization. |
| *SAME_PIN_TIMING_CONSTRAINT* | N/A, since the keyword means a min or max limit by itself. | N/A |

*Example*

```
CELL my_cell {
    PIN pin1 { DIRECTION=input; CAPACITANCE = 0.05; }
    PIN pin2 { DIRECTION=output; LIMIT { CAPACITANCE { MAX=1.2; } } }
    PIN pin3 { DIRECTION=input; }
    PIN pin4 { DIRECTION=input; }
    CAPACITANCE {
       PIN=pin3;
       HEADER { VOLTAGE { PIN=pin4; } }
       EQUATION { 0.25 + 0.34*VOLTAGE }
    }
}
```

The capacitance on `pin1` is `0.05`. The maximum allowed load capacitance on `pin2` is `1.2`. The capacitance on `pin3` depends on the voltage on pin4.

## 11.24 POWER and ENERGY

A *power* or an *energy* statement shall be defined as shown in Syntax 122.

```
KEYWORD POWER = arithmetic_model {
  VALUETYPE = number ;
}
KEYWORD ENERGY = arithmetic_model {
  VALUETYPE = number ;
}
POWER { UNIT = 1e-3; }
ENERGY { UNIT = 1e-12; }
```

*Syntax 122—POWER and ENERGY statements*

The purpose of power calculation is to evaluate the electrical power supply demand and electrical power dissipation of an electronic circuit. In general, both power supply demand and power dissipation are the same, due to the

energy conservation law. However, there are scenarios where power is supplied and dissipated locally in different places. The power models in ALF shall be specified in such a way that the total power supply and dissipation of a circuit adds up correctly to the same number.

*Example*

> A capacitor C is charged from 0 volt to V volt by a switched DC source. The energy supplied by the source is $C*V^2$. The energy stored in the capacitor is $1/2*C*V^2$. Hence the dissipated energy is also $1/2*C*V^2$. Later the capacitor is discharged from V volt to 0 volt. The supplied energy is 0. The dissipated energy is $1/2*C*V^2$. A supply-oriented power model can associate the energy $E_1=C*V^2$ with the charging event and $E_2=0$ with the discharging event. The total energy is $E=E_1+E_2=C*V^2$. A dissipation-oriented power model can associate the energy $E_3=1/2*C*V^2$ with both the charging and discharging event. The total energy is also $E=2*E_3=C*V^2$.

In many cases, it is not so easy to decide when and where the power is supplied and where it is dissipated. The choice between a supply-oriented and dissipation-oriented model or a mixture of both is subjective. Hence the ALF language provides no means to specify, which modeling approach is used. The choice is up to the model developer, as long as the energy conservation law is respected.

POWER and/or ENERGY models shall be in the context of a CELL or within a VECTOR. The total energy and/or power of a cell shall be calculated by combining the data of all models within the scope of the CELL or the VECTORs within the cell.

The data for POWER and/or ENERGY shall be positive when energy is actually supplied to the CELL and/or dissipated within the CELL. The data shall be negative when energy is actually supplied or restored by the CELL.

## 11.25 FLUX and FLUENCE

A *flux* or *fluence* statement shall be defined as shown in Syntax 123.

```
KEYWORD FLUX = arithmetic_model {
  VALUETYPE = number ;
}
KEYWORD FLUENCE = arithmetic_model {
  VALUETYPE = number ;
}
FLUX { UNIT = 1e-3; }
FLUENCE { UNIT = 1e-12; }
```

*Syntax 123—FLUX and FLUENCE statements*

The purpose of hot electron calculation is to evaluate the damage done to the performance of an electronic device due to the hot electron effect. The hot electron effect consists in accumulation of electrons trapped in the gate oxide of a transistor. The more electrons are trapped, the more the device slows down. At a certain point, the performance specification no longer is met and the device is considered to be damaged.

FLUX and/or FLUENCE models shall be in the context of a CELL or within a VECTOR. Total fluence and/or flux of a cell shall be calculated by combining the data of all models within the scope of the CELL or the VECTORs within the cell.

Both FLUX and FLUENCE are measures for hot electron damage. FLUX relates to FLUENCE in the same way as POWER relates to ENERGY.

## 11.26 DRIVE_STRENGTH

A *drive_strength* statement shall be defined as shown in Syntax 124.

```
KEYWORD DRIVE_STRENGTH = arithmetic_model {
  VALUETYPE = number ;
}
DRIVE_STRENGTH { MIN = 0; }
```

*Syntax 124—DRIVE_STRENGTH statement*

DRIVE_STRENGTH is a unit-less, abstract measure for the drivability of a PIN. It can be used as a substitute of driver RESISTANCE. The higher the DRIVE_STRENGTH, the lower the driver RESISTANCE. However, DRIVE_STRENGTH can only be used within a coherent system of calculation models, since it does not represent an absolute quantity, as opposed to RESISTANCE. For example, the weakest driver of a library can have drive strength 1, the next stronger driver can have drive strength 2 and so forth. This does not necessarily mean the resistance of the stronger driver is exactly half of the resistance of the weaker driver.

An arithmetic model for conversion from DRIVE_STRENGTH to RESISTANCE can be given to relate the quantity DRIVE_STRENGTH across technology libraries.

*Example*

```
SUBLIBRARY high_speed_library {
    RESISTANCE {
       HEADER { DRIVE_STRENGTH } EQUATION { 800 / DRIVE_STRENGTH }
    }
    CELL high_speed_std_driver {
       PIN Z { DIRECTION = output; DRIVE_STRENGTH = 1; }
    }
}
SUBLIBRARY low_power_library {
    RESISTANCE {
       HEADER { DRIVE_STRENGTH } EQUATION { 1600 / DRIVE_STRENGTH }
    }
    CELL low_power_std_driver {
       PIN Z { DIRECTION = output; DRIVE_STRENGTH = 1; }
    }
}
```

Drive strength 1 in the high speed library corresponds to 800 ohm. Drive strength 1 in the low power library corresponds to 1600 ohm.

NOTE—Any particular arithmetic model for RESISTANCE in either library shall locally override the conversion formula from drive strength to resistance.

## 11.27 SWITCHING_BITS

A *switching_bits* statement shall be defined as shown in Syntax 125.

The quantity SWITCHING_BITS applies only for bus pins. The range is from 0 to the width of the bus. Usually, the quantity SWITCHING_BITS is not calculated by an arithmetic model, since the number of switching bits on a bus depends on the functional specification rather than the electrical specification. However,

```
KEYWORD SWITCHING_BITS = arithmetic_model {
  VALUETYPE = unsigned_integer ;
}
```

*Syntax 125—SWITCHING_BITS statement*

SWITCHING_BITS can be used as argument in the HEADER of an arithmetic model to calculate electrical quantities, for instance, energy consumption.

*Example*

```
CELL my_rom {
    PIN [3:0] addr { DIRECTION=input;  SIGNALTYPE=address; }
    PIN [7:0] dout { DIRECTION=output; SIGNALTYPE=data; }
    VECTOR ( ?! addr -> ?! dout ) {
       ENERGY {
          HEADER {
             SWITCHING_BITS addr_bits { PIN = addr; }
             SWITCHING_BITS dout_bits { PIN = dout; }
          }
          EQUATION { 0.45*LOG(addr_bits) + 2.6*dout_bits }
       }
    }
}
```

The energy consumption of my_rom depends on the number of switching data bits and on the logarithm of the number of switching address bits.

## 11.28 NOISE and NOISE_MARGIN

A *noise* or *noise_margin* statement shall be defined as shown in Syntax 126.

```
KEYWORD NOISE = arithmetic_model {
  VALUETYPE = number ;
}
KEYWORD NOISE_MARGIN = arithmetic_model {
  VALUETYPE = number ;
}
NOISE { MIN = 0; }
NOISE_MARGIN { MIN = 0; MAX = 1; }
```

*Syntax 126—NOISE and NOISE_MARGIN statements*

### 11.28.1 NOISE margin

*Noise margin* is defined as the maximal allowed difference between the ideal signal voltage under a well-specified operation condition and the actual signal voltage normalized to the ideal voltage swing. This is illustrated in Figure 24.

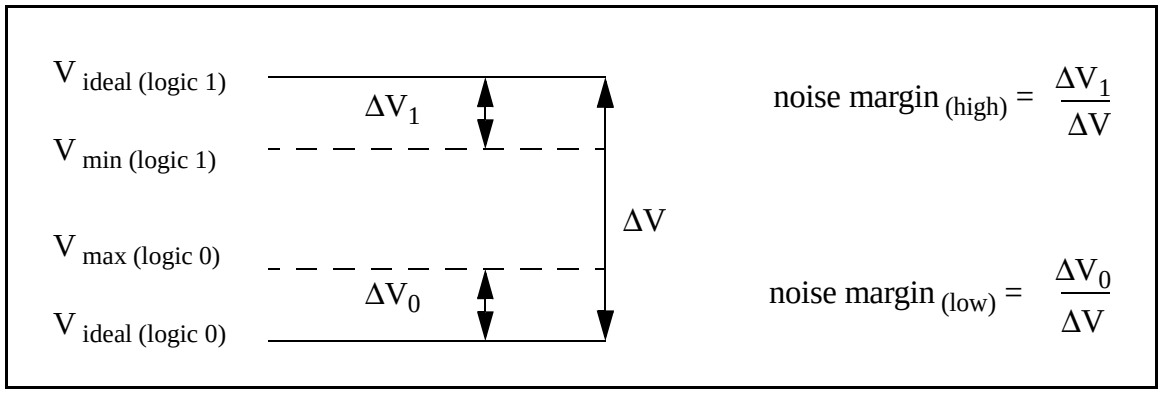



**Figure 24—Definition of noise margin**

NOISE_MARGIN is a pin-related quantity. It can appear either in the context of a PIN statement or in the context of a VECTOR statement with PIN annotation. It can also appear in the global context of a CELL, SUBLIBRARY, or LIBRARY statement.

If a NOISE_MARGIN statement appears in multiple contexts, the following priorities apply:

a) NOISE_MARGIN with PIN annotation in the context of the VECTOR, NOISE_MARGIN with PIN annotation in the context of the CELL, or NOISE_MARGIN in the context of the PIN
b) NOISE_MARGIN without PIN annotation in the context of the CELL
c) NOISE_MARGIN in the context of the SUBLIBRARY
d) NOISE_MARGIN in the context of the LIBRARY
e) NOISE_MARGIN outside the LIBRARY

### 11.28.2 NOISE

Noise is defined as the actual measured noise against which the noise margin is compared.

## 11.29 Annotations and statements related to electrical models

**Add lead-in text**

### 11.29.1 MEASUREMENT annotation

A *measurement* annotation shall be defined as shown in Semantics 71.

```
KEYWORD MEASUREMENT = single_value_annotation {
  VALUETYPE = identifier ;
  VALUES {
    transient static average absolute_average rms peak
  }
  CONTEXT {
    ENERGY POWER CURRENT VOLTAGE FLUX FLUENCE JITTER
  }
}
```

*Semantics 71—MEASUREMENT annotation*

Arithmetic models can have a MEASUREMENT annotation. This annotation indicates the type of measurement used for the computation in arithmetic model.

The meaning of the annotation values is shown in Table 94.

**Table 94—MEASUREMENT annotation**

| Annotation value | Description |
|---|---|
| transient | Measurement is a transient value. |
| static | Measurement is a static value. |
| average | Measurement is an average value. |
| absolute_verage | Measurement is an average over absolute values. |
| rms | Measurement is the root mean square value. |
| peak | Measurement is a peak value. |

Their mathematical definitions are shown in Figure 25.

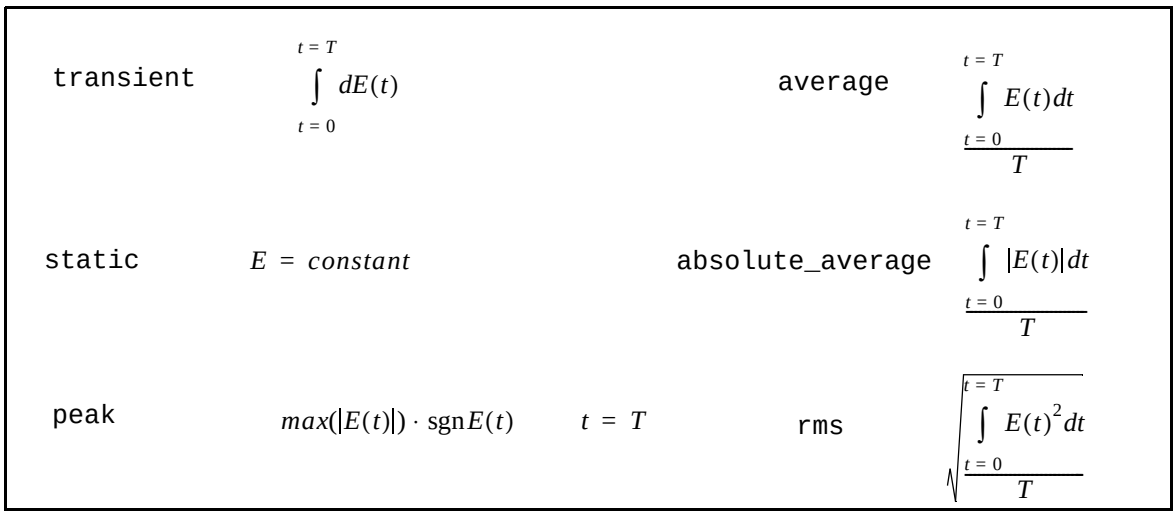



**Figure 25—Mathematical definitions for MEASUREMENT annotations**

Arithmetic models with certain values of MEASUREMENT annotation can also have *either* TIME *or* FREQUENCY as auxiliary arithmetic models.

The semantics are defined in Table 95.

**Table 95—Semantic interpretation of MEASUREMENT, TIME, or FREQUENCY**

| MEASUREMENT annotation | Semantic meaning of TIME | Semantic meaning of FREQUENCY |
|---|---|---|
| transient | Integration of analog measurement is done during that time window. | Integration of analog measurement is repeated with that frequency. |
| static | N/A | N/A |
| average | Average value is measured over that time window. | Average value measurement is repeated with that frequency. |
| absolute_average | Absolute average value is measured over that time window. | Absolute average value measurement is repeated with that frequency. |
| rms | Root-mean-square value is measured over that time window. | Root-mean-square measurement is repeated with that frequency. |
| peak | Peak value occurs at that time (only within context of VECTOR). | Observation of peak value is repeated with that frequency. |

In the case of average and rms, the interpretation FREQUENCY = 1 / TIME is valid. Either one of these annotations shall be mandatory. The values for average measurements and for rms measurements scale linearly with FREQUENCY and 1 / TIME, respectively.

In the case of transient and peak, the interpretation FREQUENCY = 1 / TIME is not valid. Either one of these annotations shall be optional. The values do not necessarily scale with TIME or FREQUENCY. The TIME or FREQUENCY annotations for transient measurements are purely informational.

### 11.29.2 TIME to peak measurement

For a model in the context of a VECTOR, with a peak measurement, the TIME annotation shall define the time between a reference event within the vector_expression and the instant when the peak value occurs.

For that purpose, either the FROM or the TO statement shall be used in the context of the TIME annotation, containing a PIN annotation and, if necessary, a THRESHOLD and/or an EDGE_NUMBER annotation.

If the FROM statement is used, the start point shall be the reference event and the end point shall be the occurrence time of the peak, as shown in Figure 26.

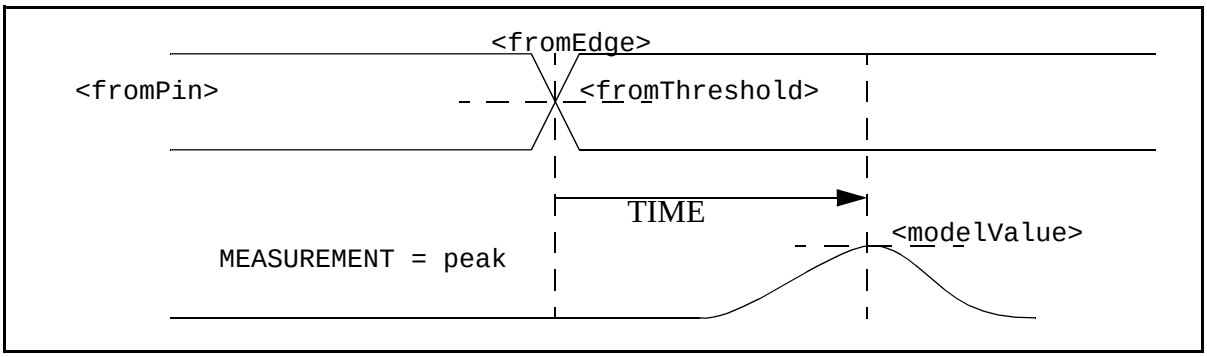


**Figure 26—Illustration of time to peak using FROM statement**

If the TO statement is used, the start point shall be the occurrence time of the peak and the end point shall be the reference event, as shown in Figure 27.

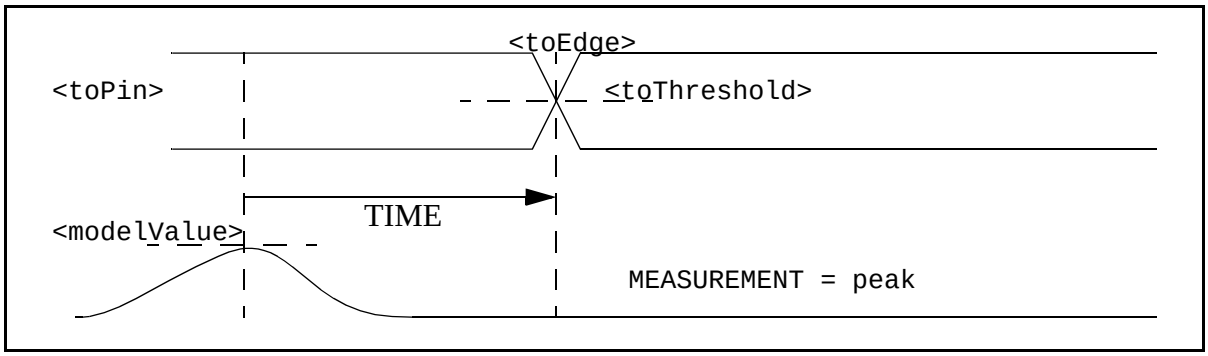



**Figure 27—Illustration of time to peak using TO statement**

### 11.29.3 COMPONENT annotation

A *component* annotation shall be defined as shown in Semantics 72.

```
KEYWORD COMPONENT = single_value_annotation {
  CONTEXT = CURRENT ;
  VALUETYPE = identifier ;
}
```

*Semantics 72—COMPONENT annotation*

### 11.29.4 FLOW annotation

A *flow* annotation shall be defined as shown in Semantics 73.

```
KEYWORD FLOW = single_value_annotation {
  CONTEXT = CURRENT ;
  VALUETYPE = identifier ;
  VALUES { IN OUT }
}
```

*Semantics 73—FLOW annotation*

## 11.30 CONNECTIVITY

A *connectivity* statement shall be defined as shown in Syntax 127.

A *driver* or *receiver* statement shall be defined as shown in Syntax 128.

Connectivity can also be described as a lookup table model. This description is usually more compact than the description using the BETWEEN statements.

```
KEYWORD CONNECTIVITY = arithmetic_model {
  VALUETYPE = boolean ;
  VALUES { 1 0 ? }
}
```

*Syntax 127—CONNECTIVITY statement*

```
KEYWORD DRIVER = arithmetic_model {
  VALUETYPE = identifier ;
  CONTEXT = CONNECTIVITY.HEADER
}
KEYWORD RECEIVER = arithmetic_model {
  VALUETYPE = identifier ;
  CONTEXT = CONNECTIVITY.HEADER
}
```

*Syntax 128— DRIVER and RECEIVER statements*

The connectivity model can have the arguments shown in Table 96 in the HEADER.

**Table 96—Arguments for connectivity**

| Argument | Value type | Description |
|---|---|---|
| DRIVER | *identifier* | Dimension of connectivity function. |
| RECEIVER | *identifier* | Dimension of connectivity function. |

Each dimension shall contain a TABLE.

The connectivity model specifies the allowed and disallowed connections amongst drivers or receivers in one-dimensional tables or between drivers and receivers in two-dimensional tables.The boolean literals in the table refer to the CONNECT_RULE as shown in Table 97.

**Table 97—Boolean literals in non-interpolateable tables**

| Boolean literal | Description |
|---|---|
| 1 | CONNECT_RULE is *True*. |
| 0 | CONNECT_RULE is *False*. |
| ? | CONNECT_RULE does not apply. |

## 11.31 SIZE

A *size* statement shall be defined as shown in Syntax 129.

```
KEYWORD SIZE = arithmetic_model {
  VALUETYPE = unsigned_number ;
}
```

*Syntax 129—SIZE statement*

## 11.32 AREA

A *area* statement shall be defined as shown in Syntax 130.

```
KEYWORD AREA = arithmetic_model {
  VALUETYPE = unsigned_number ;
}
```

*Syntax 130—AREA statement*

## 11.33 WIDTH

A *width* statement shall be defined as shown in Syntax 131.

```
KEYWORD WIDTH = arithmetic_model {
  VALUETYPE = unsigned_number ;
}
```

*Syntax 131—WIDTH statement*

Width can be associated with a *routing segment* (see Section 9.33.2). Width shall be measured orthogonal to the routing direction.

## 11.34 HEIGHT

A *height* statement shall be defined as shown in Syntax 132.

```
KEYWORD HEIGHT = arithmetic_model {
  VALUETYPE = unsigned_number ;
}
```

*Syntax 132—HEIGHT statement*

## 11.35 LENGTH

A *length* statement shall be defined as shown in Syntax 133.

```
KEYWORD LENGTH = arithmetic_model {
  VALUETYPE = unsigned_number ;
}
```

*Syntax 133—LENGTH statement*

Length can be associated with a *routing segment* (see Section 9.33.2). Length shall be measured parallel to the routing direction. Length can also be associated with two parallel routing segments. In this case, length shall represent the distance between two lines which are orthogonal to the routing segments, cross both routing segments and are as far apart from each other as possible.

## 11.36 DISTANCE

A *distance* statement shall be defined as shown in Syntax 134.

```
KEYWORD DISTANCE = arithmetic_model {
  VALUETYPE = unsigned_number ;
}
```

*Syntax 134—DISTANCE statement*

Distance can be associated with two parallel *routing segments* (see Section 9.33.2). Distance shall be measured orthogonal to the routing direction.

## 11.37 OVERHANG

A *overhang* statement shall be defined as shown in Syntax 135.

```
KEYWORD OVERHANG = arithmetic_model {
  VALUETYPE = unsigned_number ;
}
```

*Syntax 135—OVERHANG statement*

## 11.38 PERIMETER

A *perimeter* statement shall be defined as shown in Syntax 136.

```
KEYWORD PERIMETER = arithmetic_model {
  VALUETYPE = unsigned_number ;
}
```

*Syntax 136—PERIMETER statement*

## 11.39 EXTENSION

An *extension* statement shall be defined as shown in Syntax 137.

```
KEYWORD EXTENSION = arithmetic_model {
  VALUETYPE = unsigned_number ;
}
```

*Syntax 137—EXTENSION statement*

## 11.40 THICKNESS

A *thickness* statement shall be defined as shown in Syntax 138.

```
KEYWORD THICKNESS = arithmetic_model {
  VALUETYPE = unsigned_number ;
}
```

*Syntax 138—THICKNESS statement*

## 11.41 DENSITY

A *density* statement shall be defined as shown in Syntax 139.

```
KEYWORD DENSITY = arithmetic_model {
  VALUETYPE = unsigned_number ;
  MIN = 0;
  MAX = 1;
}
```

*Syntax 139—DENSITY statement*

## 11.42 Annotations for physical models

**Add lead-in text**

### 11.42.1 CONNECT_RULE annotation

A *connect_rule* annotation shall be defined as shown in Semantics 74.

```
KEYWORD CONNECT_RULE = single_value_annotation {
  VALUETYPE = identifier ;
  VALUES { must_short can_short cannot_short }
  CONTEXT = CONNECTIVITY;
}
```

*Semantics 74—CONNECT_RULE annotation*

The meaning of the annotation values is shown in Table 98.

**Table 98—CONNECT_RULE annotation**

| Annotation value | Description |
|---|---|
| must_short | Electrical connection required. |
| can_short | Electrical connection allowed. |
| cannot_short | Electrical connection disallowed. |

It is not necessary to specify more than one rule between a given set of objects. If one rule is specified to be *True*, the logical value of the other rules can be implied shown in Table 99.

**Table 99—Implications between connect rules**

| must_short | cannot_short | can_short |
|---|---|---|
| *False* | *False* | *True* |
| *False* | *True* | *False* |
| *True* | *False* | N/A |

### 11.42.2 BETWEEN annotation

A *between* annotation shall be defined as shown in Semantics 75.

```
KEYWORD BETWEEN = multi_value_annotation {
  VALUETYPE = identifier ;
  CONTEXT { DISTANCE LENGTH OVERHANG CONNECTIVITY }
}
```

*Semantics 75—BETWEEN annotation*

If the `BETWEEN` statement contains only one identifier, than the `CONNECTIVITY` shall apply between multiple instances of the same object.

The `BETWEEN` statement within `DISTANCE` or `LENGTH` shall identify the objects for which the measurement applies.

If the `BETWEEN` statement contains only one identifier, than the `DISTANCE` or `LENGTH`, respectively, shall apply between multiple instances of the same object, as shown in the following example and Figure 28.

*Example*

```
DISTANCE = 4 { BETWEEN { object1 object2 } }
LENGTH = 2 { BETWEEN { object1 object2 } }
```

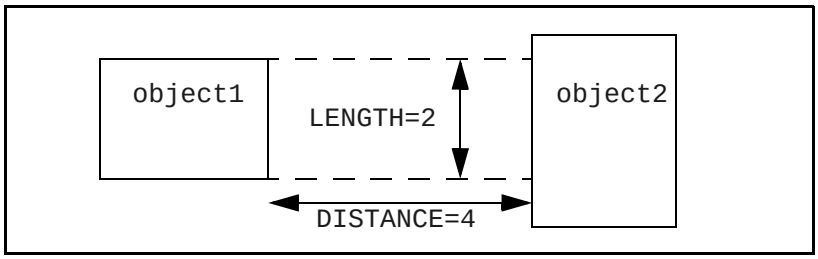



**Figure 28—Illustration of LENGTH and DISTANCE**

### 11.42.3 DISTANCE-MEASUREMENT annotation

A *distance_measurement* annotation shall be defined as shown in Semantics 76.

```
KEYWORD DISTANCE_MEASUREMENT = single_value_annotation {
  VALUETYPE = identifier ;
  VALUES { euclidean horizontal vertical manhattan }
  DEFAULT = euclidean ;
  CONTEXT = DISTANCE ;
}
```

*Semantics 76—DISTANCE_MEASUREMENT annotation*

The mathematical definitions for distance measurements between two points with differential coordinates $\Delta x$ and $\Delta y$ are:

- *euclidean* distance = $(\Delta x^2 + \Delta y^2)^{1/2}$
- *horizontal* distance = $\Delta x$
- *vertical* distance = $\Delta y$
- *manhattan* distance = $\Delta x + \Delta y$

### 11.42.4 REFERENCE annotation container

A *reference* annotation shall be defined as shown in Semantics 77.

```
KEYWORD REFERENCE = annotation_container {
  CONTEXT = DISTANCE ;
}
SEMANTICS REFERENCE.identifier = single_value_annotation {
  VALUETYPE = identifier ;
  VALUES { center origin near_edge far_edge }
  DEFAULT = origin ;
}
```

*Semantics 77—REFERENCE annotation*

The meaning of the annotation values is illustrated in Figure 29.

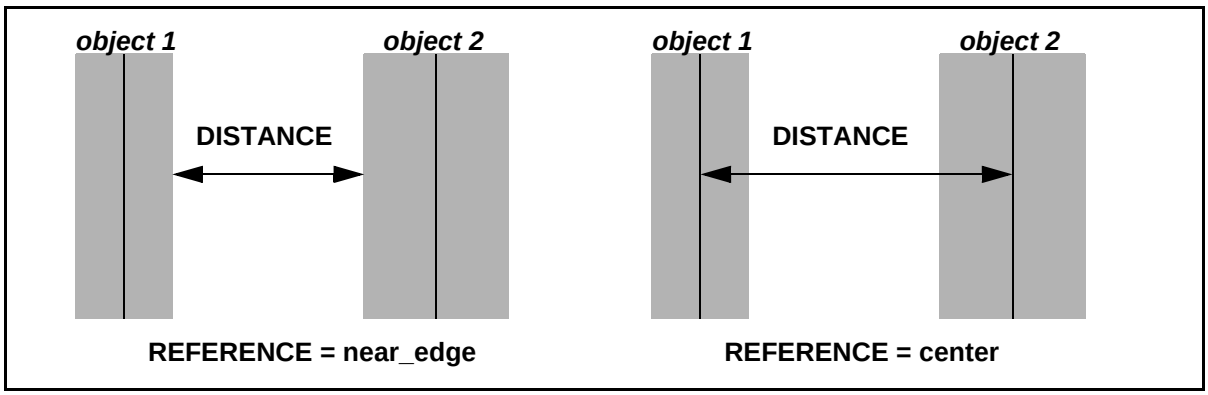


**Figure 29—Illustration of REFERENCE for DISTANCE**

### 11.42.5 ANTENNA reference annotation

An *antenna* annotation shall be defined as shown in Semantics 78.

```
SEMANTICS ANTENNA = annotation {
  VALUETYPE = identifier ;
  CONTEXT { PIN.SIZE PIN.AREA PIN.PERIMETER }
}
```

*Semantics 78—ANTENNA annotation*

In hierarchical design, a `PIN` with physical `PORT`s can be abstracted. Therefore, an arithmetic model for `SIZE`, `AREA`, `PERIMETER`, etc. for certain antenna rules can be precalculated. An `ANTENNA` statement within the arithmetic model enables references to the set of antenna rules for which the arithmetic model applies

*Example*

```
CELL cell1 {
    PIN pin1 {
       AREA poly_area = 1.5 {
          LAYER = poly;
          ANTENNA { individual_m1 individual_via1 }
       }
       AREA m1_area = 1.0 {
          LAYER = metal1;
          ANTENNA { individual_m1 }
       }
       AREA via1_area = 0.5 {
          LAYER = via1;
          ANTENNA { individual_via1 }
       }
    }
}
```

The area `poly_area` is used in the rules `individual_m1` and `individual_via1`.
The area `m1_area` is used in the rule `individual_m1` only.
The area `via1_area` is used in the rule `individual_via1` only.

The case with diffusion is illustrated in the following example:

```
CELL my_diode {
    CELLTYPE = special; ATTRIBUTE { DIODE }
    PIN my_diode_pin {
       AREA = 3.75 {
          LAYER = diffusion;
          ANTENNA { rule1_for_diffusion rule2_for_diffusion }
       }
    }
}
```

### 11.42.6 PATTERN reference annotation

A *pattern* annotation shall be defined as shown in Semantics 79.

```
SEMANTICS PATTERN = single_value_annotation {
  VALUETYPE = identifier ;
  CONTEXT {
    LENGTH WIDTH HEIGHT SIZE AREA THICKNESS
    PERIMETER EXTENSION
  }
}
```

*Semantics 79—PATTERN annotation*

Reference to a PATTERN shall be legal within arithmetic models, if the pattern and the model are within the scope of the same parent object.

## 11.43 Arithmetic submodels for timing and electrical data

The arithmetic submodels shown in Table 100 are only applicable in the context of electrical modeling.

**Table 100—Submodels applicable for timing and electrical modeling**

| Object | Description |
|---|---|
| HIGH | Applicable for electrical data measured at a logic high state of a pin. |
| LOW | Applicable for electrical data measured at a logic low state of a pin. |
| RISE | Applicable for electrical data measured during a logic low to high transition of a pin. |
| FALL | Applicable for electrical data measured during a logic high to low transition of a pin. |

## 11.44 Arithmetic submodels for physical data

The arithmetic submodels shown in Table 101 are only applicable in the context of physical modeling.

**Table 101—Submodels applicable for physical modeling**

| Object | Description |
|---|---|
| HORIZONTAL | Applicable for layout measurements in 0 degree, i.e., horizontal direction. |
| VERTICAL | Applicable for layout measurements in 90 degree, i.e., vertical direction. |
| ACUTE | Applicable for layout measurements in 45 degree direction. |
| OBTUSE | Applicable for layout measurements in 135 degree direction. |

# Annex A

(informative)

# Syntax rule summary

This summary replicates the syntax detailed in the preceding clauses. If there is any conflict, in detail or completeness, the syntax presented in the clauses shall considered as the normative definition.

**The current ordering is as each item appears in its subchapter; this needs to be updated to be complete.**

## A.1 ALF meta-language

ALF_statement ::= (see 5.1)
ALF_type [ALF_name ] [ **=** ALF_value ] ALF_statement_termination

ALF_type ::=
non_escaped_identifier [ index ]
| **@**
| **:**

ALF_name ::=
identifier [ index ]
| control_expression

ALF_value ::=
identifier
| number
| arithmetic_expression
| boolean_expression
| control_expression

ALF_statement_termination ::=
**;**
| **{** { ALF_value | **:** | **;** } **}**
| **{** { ALF_statement } **}**

## A.2 Lexical definitions

character ::= (see 6.1)
whitespace
| letter
| digit
| special

whitespace ::=
space | vertical_tab | horizontal_tab | new_line | carriage_return | form_feed

letter ::=
uppercase | lowercase

uppercase ::=
**A** | **B** | **C** | **D** | **E** | **F** | **G** | **H** | **I** | **J** | **K** | **L** | **M** | **N** | **O** | **P** | **Q** | **R** | **S** | **T** | **U** | **V** | **W**
| **X** | **Y** | **Z**

lowercase ::=
**a** | **b** | **c** | **d** | **e** | **f** | **g** | **h** | **i** | **j** | **k** | **l** | **m** | **n** | **o** | **p** | **q** | **r** | **s** | **t** | **u** | **v** | **w** | **x** | **y** | **z**

digit ::=
**0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

special ::=
**&** | **|** | **^** | **~** | **+** | **-** | ***** | **/** | **%** | **?** | **!** | **:** | **;** | **,** | **”** | **’** | **@** | **=** | **** | **.** | **$** | **_** | **#**
| **(** | **)** | **<** | **>** | **[** | **]** | **{** | **}**

comment ::= (see 6.2)
in_line_comment
| block_comment

in_line_comment ::=
**//**{character}new_line
| **//**{character}carriage_return

block_comment ::=
**/***{character}***/**

delimiter ::= (see 6.3)
**(** | **)** | **[** | **]** | **{** | **}** | **:** | **;** | **,**

operator ::= (see 6.4)
arithmetic_operator
| boolean_operator
| relational_operator
| shift_operator
| event_sequence_operator
| meta_operator

arithmetic_operator ::=
**+** | **-** | ***** | **/** | **%** | ******

boolean_operator ::=
**&&** | **||** | **~&** | **~|** | **^** | **~^** | **~** | **!** | **&** | **|**

relational_operator ::=
**==** | **!=** | **>=** | **<=** | **>** | **<**

shift_operator ::=
**<<** | **>>**

event_sequence_operator ::=
**->** | **~>** | **<->** | **<~>** | **&>** | **<&>**

meta_operator ::=
**=** | **?** | **@**

number ::= (see 6.5)
signed_integer | signed_real | unsigned_integer | unsigned_real

signed_number ::=
signed_integer | signed_real

unsigned_number ::=
unsigned_integer | unsigned_real

integer ::=
signed_integer | unsigned_integer

signed_integer ::=
sign unsigned_integer

unsigned_integer ::=
digit { [ **_** ] digit }

real ::=
signed_real | unsigned_real

signed_real ::=
sign unsigned_real

unsigned_real ::=
mantisse [ exponent ]

| unsigned_integer exponent

sign ::=

**+** | **-**

mantisse ::=

**.** unsigned_integer

| unsigned_integer **.** [ unsigned_integer ]

exponent ::=

**E** [ sign ] unsigned_integer

| **e** [ sign ] unsigned_integer

quantity_symbol ::= (see 6.6)

unity { letter } | K { letter } | M E G { letter } | G { letter }

| M { letter } | U { letter } | N { letter } | P { letter } | F { letter }

unity ::=

**1**

K ::=

**K** | **k**

M ::=

**M** | **m**

E ::=

**E** | **e**

G ::=

**G** | **g**

U ::=

**U** | **u**

N ::=

**N** | **n**

P ::=

**P** | **p**

F ::=

**F** | **f**

bit_literal ::= (see 6.7)

alphanumeric_bit_literal

| symbolic_bit_literal

alphanumeric_bit_literal ::=

numeric_bit_literal

| alphabetic_bit_literal

numeric_bit_literal ::=

**0** | **1**

alphabetic_bit_literal ::=

**X** | **Z** | **L** | **H** | **U** | **W**

| **x** | **z** | **l** | **h** | **u** | **w**

symbolic_bit_literal ::=

**?** | *****

based_literal ::= (see 6.8)

binary_based_literal | octal_based_literal | decimal_based_literal | hexadecimal_based_literal

binary_based_literal ::=

binary_base bit_literal { [ **_** ] bit_literal }

binary_base ::=

**'B** | **'b**

octal_based_literal ::=

octal_base octal_digit { [ _ ] octal_digit }

octal_base ::=
**'O** | **'o**

octal_digit ::=
bit_literal | **2** | **3** | **4** | **5** | **6** | **7**

decimal_based_literal ::=
decimal_base digit { [ _ ] digit }

decimal_base ::=
**'D** | **'d**

hexadecimal_based_literal ::=
hexadecimal_base hexadecimal_digit { [ _ ] hexadecimal_digit }

hexadecimal_base ::=
**'H** | **'h**

hexadecimal_digit ::=
octal | **8** | **9**
| **A** | **B** | **C** | **D** | **E** | **F**
| **a** | **b** | **c** | **d** | **e** | **f**

edge_literal ::= (see 6.9)
bit_edge_literal
| based_edge_literal
| symbolic_edge_literal

bit_edge_literal ::=
bit_literal bit_literal

based_edge_literal ::=
based_literal based_literal

symbolic_edge_literal ::=
**?~** | **?!** | **?-**

quoted_string ::= (see 6.10)
**"** { character } **"**

identifier ::= (see 6.11)
non_escaped_identifier
| escaped_identifier
| placeholder_identifier
| hierarchical_identifier

non_escaped_identifier ::= (see 6.11.1)
letter { letter | digit | **_** | **$** | **#** }

escaped_identifier ::= (see 6.11.2)
backslash escapable_character { escapable_character }

escapable_character ::=
letter | digit | special

placeholder_identifier ::= (see 6.11.3)
**<** non_escaped_identifier **>**

hierarchical_identifier ::= (see 6.11.4)
identifier [ \ ] **.** identifier

*keyword*_identifier ::= (see 6.12)
letter { [ **_** ] letter }

## A.3 Auxiliary definitions

all_purpose_value ::= (see 7.1)
number

| identifier
| quoted_string
| bit_literal
| based_literal
| edge_value
| pin_variable
| control_expression

quantity_value ::= (see 7.2)
unsigned_number | quantity_symbol

string_value ::= (see 7.3)
quoted_string | identifier

arithmetic_value ::= (see 7.4)
number | identifier | bit_literal | based_literal

boolean_value ::= (see 7.5)
alphanumeric_bit_literal | based_literal | integer

edge_value ::= (see 7.6)
**(** edge_literal **)**

index_value ::= (see 7.7)
unsigned_integer | identifier

index ::= (see 7.8)
single_index | multi_index

single_index ::=
**[** index_value **]**

multi_index ::=
**[** index_value **:** index_value **]**

pin_variable ::= (see 7.9)
*pin_variable*_identifier [ index ]

pin_value ::=
pin_variable | boolean_value

pin_assignment ::= (see 7.10)
pin_variable **=** pin_value **;**

annotation ::= (see 7.11)
single_value_annotation
| multi_value_annotation

single_value_annotation ::=
*annotation*_identifier **=** annotation_value **;**

annotation_value ::=
number
| identifier
| quoted_string
| bit_literal
| based_literal
| edge_value
| pin_variable
| control_expression
| boolean_expression
| arithmetic_expression

multi_value_annotation ::=
*annotation*_identifier **{** annotation_value { annotation_value } **}**

annotation_container ::= (see 7.12)
*annotation_container*_identifier **{** annotation { annotation } **}**

attribute ::= (see 7.13)

**ATTRIBUTE {** identifier { identifier } **}**

property ::= (see 7.14)

**PROPERTY** [ identifier ] **{** annotation { annotation } **}**

include ::= (see 7.15)

**INCLUDE** quoted_string **;**

revision ::= (see 7.17)

**ALF_REVISION** string_value

generic_object ::= (see 7.18)

alias_declaration
| constant_declaration
| class_declaration
| keyword_declaration
| semantics_declaration
| group_declaration
| template_declaration

library_specific_object ::= (see 7.19)

library
| sublibrary
| cell
| primitive
| wire
| pin
| pingroup
| vector
| node
| layer
| via
| rule
| antenna
| site
| array
| blockage
| port
| pattern
| region

all_purpose_item ::= (see 7.20)

generic_object
| include_statement
| associate_statement
| annotation
| annotation_container
| arithmetic_model
| arithmetic_model_container
| *all_purpose_item*_template_instantiation

## A.4 Generic definitions

alias_declaration ::= (see 8.1)

**ALIAS** *alias*_identifier **=** *original*_identifier **;**

constant_declaration ::= (see 8.2)

**CONSTANT** *constant*_identifier **=** constant_value **;**

constant_value ::=

number | based_literal

class_declaration ::= (see 8.3)

**CLASS** *class*_identifier **;**
| **CLASS** *class*_identifier **{** { all_purpose_item } **}**

keyword_declaration ::= (see 8.4)
**KEYWORD** *keyword*_identifier **=** *syntax_item*_identifier **;**
| **KEYWORD** *keyword*_identifier **=** *syntax_item*_identifier **{** { keyword_item } **}**

keyword_item ::=
*VALUETYPE*_single_value_annotation
| *VALUES*_multi_value_annotation
| *DEFAULT*_single_value_annotation
| *CONTEXT*_annotation

semantics_declaration ::= (see 8.6)
**SEMANTICS** *semantics*_identifier **=** *syntax_item*_identifier **;**
| **SEMANTICS** *semantics*_identifier [ **=** *syntax_item*_identifier ] **{** { semantics_item } **}**

semantics_item ::=
*VALUES*_multi_value_annotation
| *DEFAULT*_single_value_annotation
| *CONTEXT*_annotation

group_declaration ::= (see 8.7)
**GROUP** *group*_identifier **{** all_purpose_value { all_purpose_value } **}**
| **GROUP** *group*_identifier **{** *left*_index_value **:** *right*_index_value **}**

template_declaration ::= (see 8.8)
**TEMPLATE** *template*_identifier **{** ALF_statement { ALF_statement } **}**

template_instantiation ::= (see 8.9)
static_template_instantiation
| dynamic_template_instantiation

static_template_instantiation ::=
*template*_identifier [ **= STATIC** ] **;**
| *template*_identifier [ **= STATIC** ] **{** { all_purpose_value } **}**
| *template*_identifier [ **= STATIC** ] **{** { annotation } **}**

dynamic_template_instantiation ::=
*template*_identifier **= DYNAMIC {** { dynamic_template_instantiation_item } **}**

dynamic_template_instantiation_item ::=
annotation
| arithmetic_model
| arithmetic_assignment

arithmetic_assignment ::=
identifier **=** arithmetic_expression **;**

## A.5 Library definitions

library ::= (see 9.1)
**LIBRARY** *library*_identifier **;**
| **LIBRARY** *library*_identifier **{** { library_item } **}**
| *library*_template_instantiation

library_item ::=
sublibrary
| sublibrary_item

sublibrary ::=
**SUBLIBRARY** *sublibrary*_identifier **;**
| **SUBLIBRARY** *sublibrary*_identifier **{** { sublibrary_item } **}**
| *sublibrary*_template_instantiation

sublibrary_item ::=
all_purpose_item
| cell
| primitive
| wire
| layer
| via
| rule
| antenna
| array
| site
| region

cell ::= (see 9.3)
**CELL** *cell*_identifier **;**
| **CELL** *cell*_identifier **{** { cell_item } **}**
| *cell*_template_instantiation

cell_item ::=
all_purpose_item
| pin
| pingroup
| primitive
| function
| non_scan_cell
| test
| vector
| wire
| blockage
| artwork
| pattern
| region

named_cell_instantiation ::= (see 9.4)
*cell*_identifier *instance*_identifier **;**
| *cell*_identifier *instance*_identifier **{** pin_value { pin_value } **}**
| *cell*_identifier *instance*_identifier **{** pin_assignment { pin_assignment } **}**

unnamed_cell_instantiation ::=
*cell*_identifier **{** pin_value { pin_value } **}**
| *cell*_identifier **{** pin_assignment { pin_assignment } **}**

pin ::= (see 9.7)
scalar_pin | vector_pin | matrix_pin

scalar_pin ::=
**PIN** *pin*_identifier **;**
| **PIN** *pin*_identifier **{** { scalar_pin_item } **}**
| *scalar_pin*_template_instantiation

scalar_pin_item ::=
all_purpose_item
| port

vector_pin ::=
**PIN** multi_index *pin*_identifier **;**
| **PIN** multi_index *pin*_identifier **{** { vector_pin_item } **}**
| *vector_pin*_template_instantiation

vector_pin_item ::=
all_purpose_item
| range

matrix_pin ::=
**PIN** *first*_multi_index *pin*_identifier *second*_multi_index **;**
| **PIN** *first*_multi_index *pin*_identifier *second*_multi_index **{** { matrix_pin_item } **}**
| *matrix_pin*_template_instantiation

matrix_pin_item ::=
vector_pin_item

pingroup ::= (see 9.8)
simple_pingroup | vector_pingroup

simple_pingroup ::=
**PINGROUP** *pingroup*_identifier **{** members { all_purpose_item } **}**
| *simple_pingroup*_template_instantiation

members ::=
**MEMBERS {** *pin*_identifier *pin*_identifier { *pin*_identifier } **}**

vector_pingroup ::=
| **PINGROUP [** index_value **:** index_value **]** *pingroup*_identifier
**{** members { vector_pingroup_item } **}**
| *vector_pingroup*_template_instantiation

vector_pingroup_item ::=
all_purpose_item
| range

primitive ::= (see 9.11)
**PRIMITIVE** *primitive*_identifier **{** { primitive_item } **}**
| **PRIMITIVE** *primitive*_identifier **;**
| *primitive*_template_instantiation

primitive_item ::=
all_purpose_item
| pin
| pingroup
| function
| test

wire ::= (see 9.12)
**WIRE** *wire*_identifier **{** wire_items **}**
| **WIRE** *wire*_identifier **;**
| *wire*_template_instantiation

wire_item ::=
all_purpose_item
| node

node ::= (see 9.15)
**NODE** *node*_identifier **;**
| **NODE** *node*_identifier **{** { node_item } **}**
| *node*_template_instantiation

node_item ::=
all_purpose_item

vector ::= (see 9.16)
**VECTOR** control_expression **;**
| **VECTOR** control_expression **{** { vector_item } **}**
| *vector*_template_instantiation

vector_item ::=
all_purpose_item

layer ::= (see 9.18)
**LAYER** *layer*_identifier **;**
| **LAYER** *layer*_identifier **{** { layer_item } **}**
| *layer*_template_instantiation

layer_item ::=
all_purpose_item
via ::= (see 9.20)
**VIA** *via*_identifier **;**
| **VIA** *via*_identifier **{** { via_item } **}**
| *via*_template_instantiation
via_item ::=
all_purpose_item
| pattern
| artwork
via_instantiation ::= (see 9.21)
*via*_identifier *instance*_identifier **;**
| *via*_identifier *instance*_identifier **{** { geometric_transformation } **}**
rule ::= (see 9.23)
**RULE** *rule*_identifier **;**
| **RULE** *rule*_identifier **{** { rule_item } **}**
| *rule*_template_instantiation
rule_item ::=
all_purpose_item
| pattern
| region
| via_instantiation
antenna ::= (see 9.24)
**ANTENNA** *antenna*_identifier **;**
| **ANTENNA** *antenna*_identifier **{** { antenna_item } **}**
| *antenna*_template_instantiation
antenna_item ::=
all_purpose_item
blockage ::= (see 9.25)
**BLOCKAGE** *blockage*_identifier **;**
| **BLOCKAGE** *blockage*_identifier **{** { blockage_item } **}**
| *blockage*_template_instantiation
blockage_item ::=
all_purpose_item
| pattern
| region
| rule
| via_instantiation
port ::= (see 9.26)
**PORT** *port*_identifier **;{** { port_item } **}**
| **PORT** *port*_identifier **;**
| *port*_template_instantiation
port_item ::=
all_purpose_item
| pattern
| region
| rule
| via_instantiation
site ::= (see 9.28)
**SITE** *site*_identifier **;**
| **SITE** *site*_identifier **{** { site_item } **}**
| *site*_template_instantiation
site_item ::=
all_purpose_item

| *WIDTH*_arithmetic_model
| *HEIGHT*_arithmetic_model

array ::= (see 9.30)
**ARRAY** *array*_identifier **;**
| **ARRAY** *array*_identifier **{** { array_item } **}**
| *array*_template_instantiation

array_item ::=
all_purpose_item
| geometric_transformation

pattern ::= (see 9.32)
**PATTERN** *pattern*_identifier **;**

| **PATTERN** *pattern*_identifier **{** { pattern_item } **}**
| *pattern*_template_instantiation

pattern_item ::=
all_purpose_item
| geometric_model
| geometric_transformation

geometric_model ::= (see 9.35)
nonescaped_identifier [ *geometric_model*_identifier ]
**{** geometric_model_item { geometric_model_item } **}**
| *geometric_model_t*emplate_instantiation

geometric_model_item ::=
*POINT_TO_POINT*_single_value_annotation
| coordinates

coordinates ::=
**COORDINATES {** point { point } **}**

point ::=
*x*_number *y*_number

geometric_transformation ::= (see 9.37)
shift
| rotate
| flip
| repeat

shift ::=
**SHIFT {** *x*_number *y*_number **}**

rotate ::=
**ROTATE =** number **;**

flip ::=
**FLIP =** number **;**

repeat ::=
**REPEAT** [ **=** unsigned_integer ] **{** geometric_transformation { geometric_transformation } **}**

artwork ::= (see 9.38)
**ARTWORK =** *artwork*_identifier **;**
| **ARTWORK =** *artwork*_identifier **{** { artwork_item } **}**
| *artwork*_template_instantiation

artwork_item ::=
geometric_transformation
| pin_assignment

## A.6 Function definitions

function ::= (see 10.1)
**FUNCTION {** function_item { function_item } **}**
| *function*_template_instantiation
function_item ::=
all_purpose_item
| behavior
| structure
| statetable
test ::= (see 10.2)
**TEST {** test_item { test_item } **}**
| *test*_template_instantiation
test_item ::=
all_purpose_item
| behavior
| statetable

behavior ::= (see 10.4)
**BEHAVIOR {** behavior_item { behavior_item }s **}**
| *behavior*_template_instantiation
behavior_item ::=
boolean_assignment
| control_statement
| primitive_instantiation
| *behavior_item*_template_instantiation
boolean_assignment ::=
pin_variable **=** boolean_expression **;**
control_statement ::=
primary_control_statement { alternative_control_statement }
primary_control_statement ::=
**@** control_expression **{** boolean_assignment { boolean_assignment } **}**
alternative_control_statement ::=
**:** control_expression **{** boolean_assignment { boolean_assignment } **}**
primitive_instantiation ::=
*primitive*_identifier [ identifier ] **{** pin_value { pin_value } **}**
| *primitive*_identifier [ identifier ] **{** boolean_assignment { boolean_assignment } **}**
structure ::= (see 10.5)
**STRUCTURE {** named_cell_instantiation { named_cell_instantiation } **}**
| *structure*_template_instantiation
statetable ::= (see 10.6)
**STATETABLE** [ identifier ]
**{** statetable_header statetable_row { statetable_row } **}**
| *statetable*_template_instantiation
statetable_header ::=
*input*_pin_variable { *input*_pin_variable } **:** *output*_pin_variable { *output*_pin_variable } **;**
statetable_row ::=
statetable_control_values **:** statetable_data_values **;**
statetable_control_values ::=
statetable_control_value { statetable_control_value }
statetable_control_value ::=
boolean_value
| symbolic_bit_literal

| edge_value

statetable_data_values ::=
statetable_data_value { statetable_data_value }

statetable_data_value ::=
boolean_value
| **(** [ **!** ] *input*_pin_variable **)**
| **(** [ **~** ] *input*_pin_variable **)**

non_scan_cell ::= (see 10.7)
**NON_SCAN_CELL {** unnamed_cell_instantiation { unnamed_cell_instantiation } **}**
| **NON_SCAN_CELL =** unnamed_cell_instantiation
| *non_scan_cell*_template_instantiation

range ::= (see 10.8)
**RANGE {** index_value **:** index_value **}**

boolean_expression ::= (see 10.9)
**(** boolean_expression **)**
| pin_variable
| boolean_value
| boolean_unary boolean_expression
| boolean_expression boolean_binary boolean_expression
| boolean_expression **?** boolean_expression **:**
{ boolean_expression **?** boolean_expression **:** }
boolean_expression

boolean_unary ::=
**!**
| **~**
| **&**
| **~&**
| |
| **~**|
| ^
| **~**^

boolean_binary ::=
**&**
| **&&**
| |
| ||
| ^
| **~**^
| **!=**
| **==**
| **>=**
| **<=**
| **>**
| **<**
| **+**
| **-**
| ***
| /
| **%**
| **>>**
| **<<**

vector_expression ::= (see 10.12)
**(** vector_expression **)**
| vector_unary boolean_expression
| vector_expression vector_binary vector_expression
| boolean_expression **?** vector_expression **:**
{ boolean_expression **?** vector_expression **:** }
vector_expression
| boolean_expression control_and vector_expression
| vector_expression control_and boolean_expression

vector_unary ::=
edge_literal

vector_binary ::=
**&**
| **&&**
| **|**
| **||**
| **->**
| **~>**
| **<->**
| **<~>**
| **&>**
| **<&>**

control_and ::=
**&** | **&&**

control_expression ::=
**(** vector_expression **)**
| **(** boolean_expression **)**

## A.7 Arithmetic definitions

arithmetic_expression ::= (see 11.1)
**(** arithmetic_expression **)**
| arithmetic_value
| { boolean_expression **?** arithmetic_expression **:** } arithmetic_expression
| [ unary_arithmetic_operator ] arithmetic_operand
| arithmetic_operand binary_arithmetic_operator arithmetic_operand
| macro_arithmetic_operator **(** arithmetic_operand { **,** arithmetic_operand } **)**

arithmetic_operand ::=
arithmetic_expression

unary_arithmetic_operator ::= (see 11.1.1)
**+**
| **-**

binary_arithmetic_operator ::= (see 11.1.2)
**+**
| **-**
| *****
| **/**
| ******
| **%**

macro_arithmetic_operator ::= (see 11.1.3)
**abs**
| **exp**
| **log**
| **min**
| **max**

arithmetic_model ::= (see 11.2)
trivial_arithmetic_model
| partial_arithmetic_model
| full_arithmetic_model
| *arithmetic_model*_template_instantiation

trivial_arithmetic_model ::= (see 11.2.1)
nonescaped_identifier [ *name*_identifier ] **=** arithmetic_value **;**
| nonescaped_identifier [ *name*_identifier ] **=** arithmetic_value **{** { model_qualifier } **}**

partial_arithmetic_model ::= (see 11.2.2)
nonescaped_identifier [ *name*_identifier ] **{** { partial_arithmetic_model_item } **}**

partial_arithmetic_model_item ::=
model_qualifier
| table
| trivial_min-max

full_arithmetic_model ::= (see 11.2.3)
nonescaped_identifier [ *name*_identifier ] **{** { model_qualifier } model_body { model_qualifier } **}**

model_body ::=
header-table-equation [ trivial_min-max ]
| min-typ-max
| arithmetic_submodel { arithmetic_submodel }

header-table-equation ::= (see 11.3)
header table
| header equation

header ::= (see 11.3.1)
**HEADER {** partial_arithmetic_model { partial_arithmetic_model } **}**

table ::= (see 11.3.2)
**TABLE {** arithmetic_value { arithmetic value } **}**

equation ::= (see 11.3.3)
**EQUATION {** arithmetic_expression **}**
| *equation*_template_instantiation

model_qualifier ::= (see 11.4.1)
annotation
| annotation_container
| event_reference
| from-to
| auxiliary_arithmetic_model
| violation

auxiliary_arithmetic_model ::= (see 11.4.2)
nonescaped_identifier **=** arithmetic_value **;**
| nonescaped_identifier [ **=** arithmetic_value ] **{** auxiliary_qualifier { auxiliary_qualifier } **}**

auxiliary_qualifier
annotation
| annotation_container
| event_reference
| from-to

arithmetic_submodel ::= (see 11.4.3)

nonescaped_identifier **=** arithmetic_value **;**
| nonescaped_identifier **{** [ violation ] min-max **}**
| nonescaped_identifier **{** header-table-equation [ trivial_min-max ] **}**
| nonescaped_identifier **{** min-typ-max **}**
| *arithmetic_submodel*_template_instantiation

min-max ::= (see 11.4.4)
min [ max ]
| max [ min ]

min ::=
**MIN =** arithmetic_value **;**
| **MIN =** arithmetic_value **{** violation **}**
| **MIN {** [ violation ] header-table-equation **}**

max ::=
**MAX =** arithmetic_value **;**
| **MAX =** arithmetic_value **{** violation **}**
| **MAX {** [ violation ] header-table-equation **}**

min-typ-max ::= (see 11.4.5)
[ min-max ] typ [ min-max ]

typ ::=
**TYP =** arithmetic_value **;**
| **TYP {** header-table-equation **}**

trivial_min-max ::= (see 11.4.6)
trivial_min [ trivial_max ]
| trivial_max [ trivial_min ]

trivial_min ::=
**MIN =** arithmetic_value **;**

trivial_max ::=
**MAX =** arithmetic_value **;**

arithmetic_model_container ::= (see 11.4.7)
*arithmetic_model_container*_identifier **{** arithmetic_model { arithmetic_model } **}**

limit ::= (see 11.4.8)
**LIMIT {** limit_item { limit_item } **}**

limit_item ::=
limit_arithmetic_model

limit_arithmetic_model ::=
nonescaped_identifier [ *name*_identifier ] **{** { model_qualifier } limit_arithmetic_model_body **}**

limit_arithmetic_model_body ::=
limit_arithmetic_submodel { limit_arithmetic_submodel }
| min_max

limit_arithmetic_submodel ::=
nonescaped_identifier **{** [ violation ] min-max **}**

event_reference ::= (see 11.4.9)
*PIN_reference*_single_value_annotation [ *EDGE_NUMBER*_single_value_annotation ]

from-to ::= (see 11.4.10)
from [to]
| [ from ] to

from ::=
**FROM {** from-to_item { from-to_item } **}**

from-to_item ::=
event_reference
| *THRESHOLD*_arithmetic_model

to ::=

**TO {** from-to_item { from-to_item } **}**

early-late ::= (see 11.4.11)

early late

early ::=

**EARLY {** early-late_item { early-late_item } **}**

early-late_item ::=

*DELAY*_arithmetic_model
| *RETAIN*_arithmetic_model
| *SLEWRATE*_arithmetic_model

late ::=

**LATE {** early-late_item { early-late_item } **}**

violation ::= (see 11.4.12)

**VIOLATION {** violation_item { violation_item } **}**
| *violation*_template_instantiation

violation_item ::=

*MESSAGE_TYPE*_single_value_annotation
| *MESSAGE*_single_value_annotation
| behavior

# Annex B

(informative)

# Semantics rule summary

This summary replicates the semantics detailed in the preceding clauses. If there is any conflict, in detail or completeness, the semantics presented in the clauses shall considered as the normative definition.

**The current ordering is as each item appears in its subchapter; this needs to be updated to be complete.**

**I kept the font/formatting as it is from the original semantics sections; let me know if you want to change this (how it appears in print)**

## B.1 Library definitions

```
KEYWORD INFORMATION = annotation_container {                  (see 9.2.1)
  CONTEXT { LIBRARY SUBLIBRARY CELL WIRE PRIMITIVE }
}
KEYWORD PRODUCT = single_value_annotation {
  VALUETYPE = string_value; DEFAULT = “”; CONTEXT = INFORMATION;
}
KEYWORD TITLE = single_value_annotation {
  VALUETYPE = string_value; DEFAULT = “”; CONTEXT = INFORMATION;
}
KEYWORD VERSION = single_value_annotation {
  VALUETYPE = string_value; DEFAULT = “”; CONTEXT = INFORMATION;
}
KEYWORD AUTHOR = single_value_annotation {
  VALUETYPE = string_value; DEFAULT = “”; CONTEXT = INFORMATION;
}
KEYWORD DATETIME = single_value_annotation {
  VALUETYPE = string_value; DEFAULT = “”; CONTEXT = INFORMATION;
}
KEYWORD CELLTYPE = single_value_annotation {                  (see 9.5.1)
  CONTEXT = CELL;
  VALUETYPE = identifier;
  VALUES {
    buffer combinational multiplexor flipflop latch
    memory block core special
  }
}
KEYWORD SWAP_CLASS = annotation {                             (see 9.5.2)
  CONTEXT = CELL;
  VALUETYPE = identifier;
}
KEYWORD RESTRICT_CLASS = annotation {                         (see 9.5.3)
  CONTEXT { CELL CLASS }
  VALUETYPE = identifier;
}
```

```
KEYWORD SCAN_TYPE = single_value_annotation {                (see 9.5.4)
  CONTEXT = CELL;
  VALUETYPE = identifier;
  VALUES { muxscan clocked lssd control_0 control_1 }
}
KEYWORD SCAN_USAGE = single_value_annotation {               (see 9.5.5)
  CONTEXT = CELL;
  VALUETYPE = identifier;
  VALUES { input output hold }
}
KEYWORD BUFFERTYPE = single_value_annotation {               (see 9.5.6)
  CONTEXT = CELL;
  VALUETYPE = identifier;
  VALUES { input output inout internal }
  DEFAULT = internal;
}
KEYWORD DRIVERTYPE = single_value_annotation {               (see 9.5.7)
  CONTEXT = CELL;
  VALUETYPE = identifier;
  VALUES { predriver slotdriver both }
}
KEYWORD PARALLEL_DRIVE = single_value_annotation {           (see 9.5.8)
  CONTEXT = CELL;
  VALUETYPE = unsigned;
  DEFAULT = 1;
}
KEYWORD PLACEMENT_TYPE = single_value_annotation {           (see 9.5.9)
  CONTEXT = CELL;
  VALUETYPE = identifier;
  VALUES { pad core ring block connector }
  DEFAULT = core;
}
KEYWORD VIEW = single_value_annotation {                     (see 9.9.1)
  CONTEXT { PIN PINGROUP }
  VALUETYPE = identifier;
  VALUES { functional physical both none }
  DEFAULT = both
}
KEYWORD PINTYPE = single_value_annotation {                  (see 9.9.2)
  CONTEXT = PIN;
  VALUETYPE = identifier;
  VALUES { digital analog supply }
  DEFAULT = digital;
}
KEYWORD DIRECTION = single_value_annotation {                (see 9.9.3)
  CONTEXT = PIN;
  VALUETYPE = identifier;
  VALUES { input output both none }
}
KEYWORD SIGNALTYPE = single_value_annotation {               (see 9.9.4)
  CONTEXT = PIN;
  VALUETYPE = identifier;
```

```
  VALUES {
    data scan_data address control select tie clear set
    enable out_enable scan_enable scan_out_enable
    clock master_clock slave_clock
    scan_master_clock scan_slave_clock
  }
  DEFAULT = data;
}
KEYWORD ACTION = single_value_annotation {                    (see 9.9.5)
  CONTEXT = PIN;
  VALUETYPE = identifier;
  VALUES { asynchronous synchronous }
}
KEYWORD POLARITY = single_value_annotation {                  (see 9.9.6)
  CONTEXT = PIN;
  VALUETYPE = identifier;
  VALUES { high low rising_edge falling_edge double_edge }
}
KEYWORD DATATYPE = single_value_annotation {                  (see 9.9.7)
  CONTEXT { PIN PINGROUP }
  VALUETYPE = identifier;
  VALUES { signed unsigned }
 }
KEYWORD INITIAL_VALUE = single_value_annotation {             (see 9.9.8)
  CONTEXT = CELL;
  VALUETYPE = boolean_value;
}
KEYWORD SCAN_POSITION = single_value_annotation {             (see 9.9.9)
  CONTEXT = PIN;
  VALUETYPE = unsigned;
  DEFAULT = 0;
}
KEYWORD STUCK = single_value_annotation {                    (see 9.9.10)
  CONTEXT = PIN;
  VALUETYPE = identifier;
  VALUES { stuck_at_0 stuck_at_1 both none }
  DEFAULT = both;
}
KEYWORD SUPPLYTYPE = annotation {                            (see 9.9.11)
  CONTEXT = PIN;
  VALUETYPE = identifier;
  VALUES { power ground reference }
}
KEYWORD SIGNAL_CLASS = annotation {                          (see 9.9.12)
  CONTEXT { PIN PINGROUP }
  VALUETYPE = identifier;
}
KEYWORD SUPPLY_CLASS = annotation {                          (see 9.9.13)
  CONTEXT { PIN PINGROUP CLASS }
  VALUETYPE = identifier;
}
```

```
KEYWORD DRIVETYPE = single_value_annotation {              (see 9.9.14)
  CONTEXT = PIN;
  VALUETYPE = identifier;
  VALUES {
    cmos nmos pmos cmos_pass nmos_pass pmos_pass
    ttl open_drain open_source
  }
  DEFAULT = cmos;
}
KEYWORD SCOPE = single_value_annotation {                  (see 9.9.15)
  CONTEXT = PIN;
  VALUETYPE = identifier;
  VALUES { behavior measure both none }
  DEFAULT = both;
}
KEYWORD CONNECT_CLASS = single_value_annotation {          (see 9.9.16)
  CONTEXT = PIN;
  VALUETYPE = identifier;
}
KEYWORD SIDE = single_value_annotation {                   (see 9.9.17)
  CONTEXT { PIN PINGROUP }
  VALUETYPE = identifier;
  VALUES { left right top bottom inside }
}
KEYWORD ROW = annotation {                                 (see 9.9.18)
  CONTEXT { PIN PINGROUP }
  VALUETYPE = unsigned;
}
KEYWORD COLUMN = annotation {
  CONTEXT { PIN PINGROUP }
  VALUETYPE = unsigned;
}
KEYWORD ROUTING_TYPE = single_value_annotation {           (see 9.9.19)
  CONTEXT { PIN PORT }
  VALUETYPE = identifier;
  VALUES { regular abutment ring feedthrough }
  DEFAULT = regular;
}
KEYWORD PULL = single_value_annotation {                   (see 9.9.20)
  CONTEXT = PIN;
  VALUETYPE = identifier;
  VALUES { up down both none }
  DEFAULT = none;
}
KEYWORD SELECT_CLASS = annotation {
  CONTEXT = WIRE;                                          (see 9.14.1)
  VALUETYPE = identifier;
}
KEYWORD NODETYPE = single_value_annotation {               (see 9.15.1)
  CONTEXT = NODE;
  VALUETYPE = identifier;
```

```
  VALUES { power ground source sink
    driver receiver interconnect }
}
KEYWORD NODE_CLASS = annotation {                          (see 9.15.2)
  CONTEXT = NODE;
  VALUETYPE = identifier;
}
KEYWORD PURPOSE = annotation {                             (see 9.17.1)
  CONTEXT { VECTOR CLASS }
  VALUETYPE = identifier ;
  VALUES { bist test timing power noise reliability }
}
KEYWORD OPERATION = single_value_annotation {              (see 9.17.2)
  CONTEXT = VECTOR;
  VALUETYPE = identifier;
  VALUES {
    read write read_modify_write refresh load
    start end iddq
  }
}
KEYWORD LABEL = single_value_annotation {                  (see 9.17.3)
  CONTEXT = VECTOR;
  VALUETYPE = string_value;
}
KEYWORD EXISTENCE_CONDITION = single_value_annotation {    (see 9.17.4)
  CONTEXT { VECTOR CLASS }
  VALUETYPE = boolean_expression;
  DEFAULT = 1;
}
KEYWORD EXISTENCE_CLASS = annotation {                     (see 9.17.5)
  CONTEXT { VECTOR CLASS }
  VALUETYPE = identifier;
}
KEYWORD
CHARACTERIZATION_CONDITION = single_value_annotation {     (see 9.17.6)
  CONTEXT { VECTOR CLASS }
  VALUETYPE = boolean_expression;
}
KEYWORD CHARACTERIZATION_VECTOR = single_value_annotation { (see 9.17.7)
  CONTEXT { VECTOR CLASS }
  VALUETYPE = control_expression;
}
KEYWORD CHARACTERIZATION_CLASS = annotation {
  CONTEXT { VECTOR CLASS }                                 (see 9.17.8)
  VALUETYPE = identifier;
}
KEYWORD LAYERTYPE = single_value_annotation {              (see 9.19.1)
  CONTEXT = LAYER;
  VALUETYPE = identifier;
  VALUES {
    routing cut substrate dielectric reserved abstract
```

```
  }
}
KEYWORD PITCH = single_value_annotation {                    (see 9.19.2)
  CONTEXT = LAYER;
  VALUETYPE = unsigned_number;
}
KEYWORD PREFERENCE = single_value_annotation {               (see 9.19.3)
  CONTEXT = LAYER;
  VALUETYPE = identifier;
  VALUES { horizontal vertical acute obtuse }
}
KEYWORD VIATYPE = single_value_annotation {                  (see 9.22.1)
  CONTEXT = VIA;
  VALUETYPE = identifier;
  VALUES { default non_default partial_stack full_stack }
  DEFAULT = default;
}
KEYWORD PORT_VIEW = single_value_annotation {                (see 9.27.1)
  CONTEXT = PORT;
  VALUETYPE = identifier;
  VALUES { physical electrical both none }
  DEFAULT = both;
}
KEYWORD ORIENTATION_CLASS = annotation {                     (see 9.29.1)
  CONTEXT { SITE CELL }
  VALUETYPE = IDENTIFIER;
}
KEYWORD SYMMETRY_CLASS = annotation {                        (see 9.29.2)
  CONTEXT { SITE CELL }
  VALUETYPE = identifier;
}
KEYWORD ARRAYTYPE = single_value_annotation {                (see 9.31.1)
  CONTEXT = ARRAY;
  VALUETYPE = identifier;
  VALUES { floorplan placement
    global_routing detailed_routing }
}
KEYWORD SHAPE = single_value_annotation {                    (see 9.33.2)
  CONTEXT = PATTERN;
  VALUETYPE = identifier;
  VALUES { line tee cross jog corner end }
  DEFAULT = line;
}
KEYWORD VERTEX = single_value_annotation {                   (see 9.33.3)
  CONTEXT = PATTERN;
  VALUETYPE = identifier;
  VALUES { round linear }
  DEFAULT = linear;
}

KEYWORD POINT_TO_POINT = single_value_annotation {             (see 9.35)
  CONTEXT { POLYLINE RING POLYGON }
```

```
  VALUETYPE = identifier;
  VALUES { direct manhattan }
  DEFAULT = direct;
}
```

## B.2 Arithmetic definitions

```
SEMANTICS VIOLATION {                                        (see 11.4.12)
  CONTEXT {
    SETUP HOLD RECOVERY REMOVAL SKEW NOCHANGE ILLEGAL
    LIMIT.arithmetic_model
    LIMIT.arithmetic_model.MIN
    LIMIT.arithmetic_model.MAX
    LIMIT.arithmetic_model.arithmetic_submodel
    LIMIT.arithmetic_model.arithmetic_submodel.MIN
    LIMIT.arithmetic_model.arithmetic_submodel.MAX
  }
}
SEMANTICS VIOLATION.BEHAVIOR {
  CONTEXT {
    VECTOR.arithmetic_model
    VECTOR.LIMIT.arithmetic_model
    VECTOR.LIMIT.arithmetic_model.MIN
    VECTOR.LIMIT.arithmetic_model.MAX
    VECTOR.LIMIT.arithmetic_model.arithmetic_submodel
    VECTOR.LIMIT.arithmetic_model.arithmetic_submodel.MIN
    VECTOR.LIMIT.arithmetic_model.arithmetic_submodel.MAX
  }
}
KEYWORD MESSAGE_TYPE = single_value_annotation {
  CONTEXT = VIOLATION ;
  VALUETYPE = identifier ;
  VALUES { information warning error }
}
KEYWORD MESSAGE = single_value_annotation {
  CONTEXT = VIOLATION ;
  VALUETYPE = quoted_string ;
}
KEYWORD UNIT = annotation {                                   (see 11.5.1)
  CONTEXT = arithmetic_model ;
  VALUETYPE = quantity_value ;
  DEFAULT = 1 ;
}
KEYWORD CALCULATION = annotation {                            (see 11.5.2)
  CONTEXT = library_specific_object.arithmetic_model ;
  VALUES { absolute incremental }
  DEFAULT = absolute ;
}

KEYWORD INTERPOLATION = single_value_annotation {             (see 11.5.3)
  CONTEXT = HEADER.arithmetic_model ;
```

```
  VALUES { linear fit ceiling floor }
  DEFAULT = fit ;
}
KEYWORD DEFAULT = single_value_annotation {                (see 11.5.4)
  CONTEXT { arithmetic_model KEYWORD }
  VALUETYPE = all_purpose_value ;
}
SEMANTICS PIN = single_value_annotation {                  (see 11.19.1)
  CONTEXT {
    FROM TO SLEWRATE PULSEWIDTH
    CAPACITANCE RESISTANCE INDUCTANCE VOLTAGE CURRENT
  }
}
SEMANTICS SKEW.PIN = multi_value_annotation ;
KEYWORD EDGE_NUMBER = annotation {                         (see 11.19.2)
  CONTEXT { FROM TO SLEWRATE PULSEWIDTH SKEW }
  VALUETYPE = unsigned_integer ;
  DEFAULT = 0;
}
SEMANTICS EDGE_NUMBER = single_value_annotation {
  CONTEXT { FROM TO SLEWRATE PULSEWIDTH }
}
SEMANTICS SKEW.EDGE_NUMBER = multi_value_annotation ;
KEYWORD MEASUREMENT = single_value_annotation {            (see 11.29.1)
  VALUETYPE = identifier ;
  VALUES {
    transient static average absolute_average rms peak
  }
  CONTEXT {
    ENERGY POWER CURRENT VOLTAGE FLUX FLUENCE JITTER
  }
}
KEYWORD CONNECT_RULE = single_value_annotation {           (see 11.42.1)
  VALUETYPE = identifier ;
  VALUES { must_short can_short cannot_short }
  CONTEXT = CONNECTIVITY;
}
KEYWORD BETWEEN = multi_value_annotation {                 (see 11.42.2)
  VALUETYPE = identifier ;
  CONTEXT { DISTANCE LENGTH OVERHANG CONNECTIVITY }
}
KEYWORD DISTANCE_MEASUREMENT = single_value_annotation {   (see 11.42.3)
  VALUETYPE = identifier ;
  VALUES { euclidean horizontal vertical manhattan }
  DEFAULT = euclidean ;
  CONTEXT = DISTANCE ;
}
KEYWORD REFERENCE = annotation_container {                 (see 11.42.4)
  CONTEXT = DISTANCE ;
}
SEMANTICS REFERENCE.identifier = single_value_annotation {
  VALUETYPE = identifier ;
```

```
  VALUES { center origin near_edge far_edge }
  DEFAULT = origin ;
}
SEMANTICS ANTENNA = annotation {                          (see 11.42.5)
  VALUETYPE = identifier ;
  CONTEXT { PIN.SIZE PIN.AREA PIN.PERIMETER }
}
SEMANTICS PATTERN = single_value_annotation {             (see 11.42.6)
  VALUETYPE = identifier ;
  CONTEXT {
    LENGTH WIDTH HEIGHT SIZE AREA THICKNESS
    PERIMETER EXTENSION
  }
}
```

# Annex C

(informative)

# Bibliography

[B1] Ratzlaff, C. L., Gopal, N., and Pillage, L. T., “RICE: Rapid Interconnect Circuit Evaluator,” *Proceedings of 28th Design Automation Conference*, pp. 555–560, 1991.

[B2] SPICE 2G6 User’s Guide.

[B3] Standard Delay Format Specification, Version 3.0, Open Verilog International, May 1995.

[B4] The IEEE Standard Dictionary of Electrical and Electronics Terms, Sixth Edition.

## A

## B



## C

## S



## T



## U



## V



## W