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 7

October 24, 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 information 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

Table 1—Target applications and models supported by ALF (Continued)

Application	Functional model	Performance model	Physical model
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.

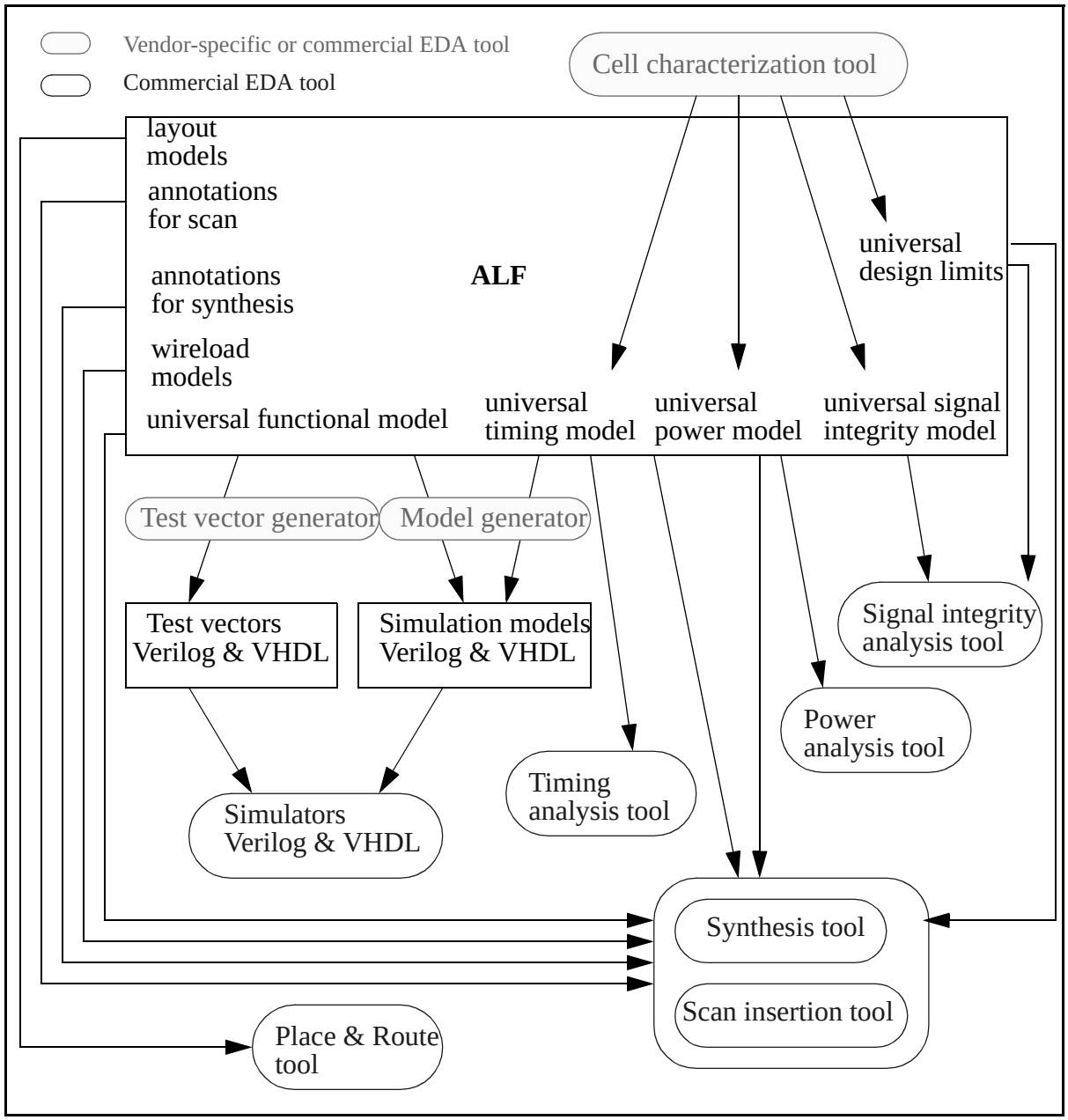


Figure 1—ALF and its target applications

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 (Description of functional and physical implementation) defines syntax and semantics of the control expression language used in this standard
— Clause 11 (Description of electrical and physical measurements) 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 10.4). 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. Name is mandatory. 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. Name is mandatory. Value does not apply. 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. Name is optional. Value is mandatory, if atomic.
Arithmetic submodel	Describe an arithmetic model under a specific measurement condition.	Statement is atomic or compound. Name does not apply. Value is mandatory, if atomic. Category of parent is exclusively *arithmetic model.*
Arithmetic model container	Provide a context for an arithmetic model.	Statement is compound. Name and value do not apply. 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. Name is optional. Value does not apply.
Annotation	Provide a qualifier or a set of qualifiers for an ALF statement.	Statement is atomic, semi-compound or compound. Name does not apply. Value is mandatory, if atomic or compound. Value does not apply, if semi-compound. Category of child is exclusively *annotation.*
Annotation container	Provide a context for an annotation.	Statement is compound. Name and value do not apply. 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.

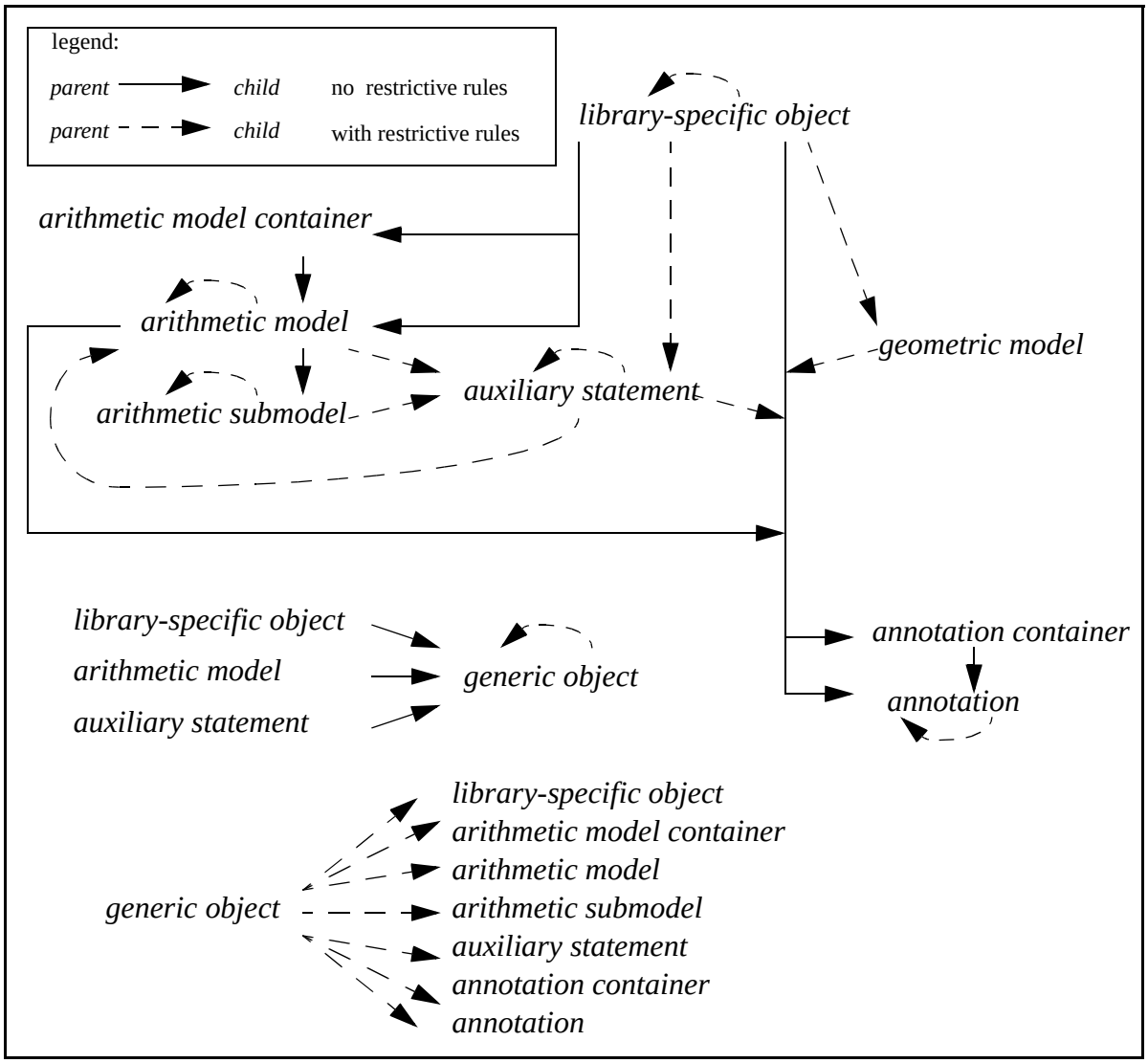


Figure 2—Parent/child relationship between ALF statements

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 8.6.
GROUP	Group declaration	See 8.8.
KEYWORD	Keyword declaration	See 8.3.
SEMANTICS	Semantics declaration	See 8.4.
TEMPLATE	Template declaration	See 8.9.

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 9.8.
WIRE	Wire declaration	See 9.9.
PIN	Pin declaration	See 9.5.
PINGROUP	Pin group declaration	See 9.6.
VECTOR	Vector declaration	See 9.13.
NODE	Node declaration	See 9.11.
LAYER	Layer declaration	See 9.15.
VIA	Via declaration	See 9.17.
RULE	Rule declaration	See 9.19.
ANTENNA	Antenna declaration	See 9.20.
SITE	Site declaration	See 9.24.
ARRAY	Array declaration	See 9.26.
BLOCKAGE	Blockage declaration	See 9.21.
PORT	Port declaration	See 9.22.
PATTERN	Pattern declaration	See 9.28.
REGION	Region declaration	See 9.30.

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

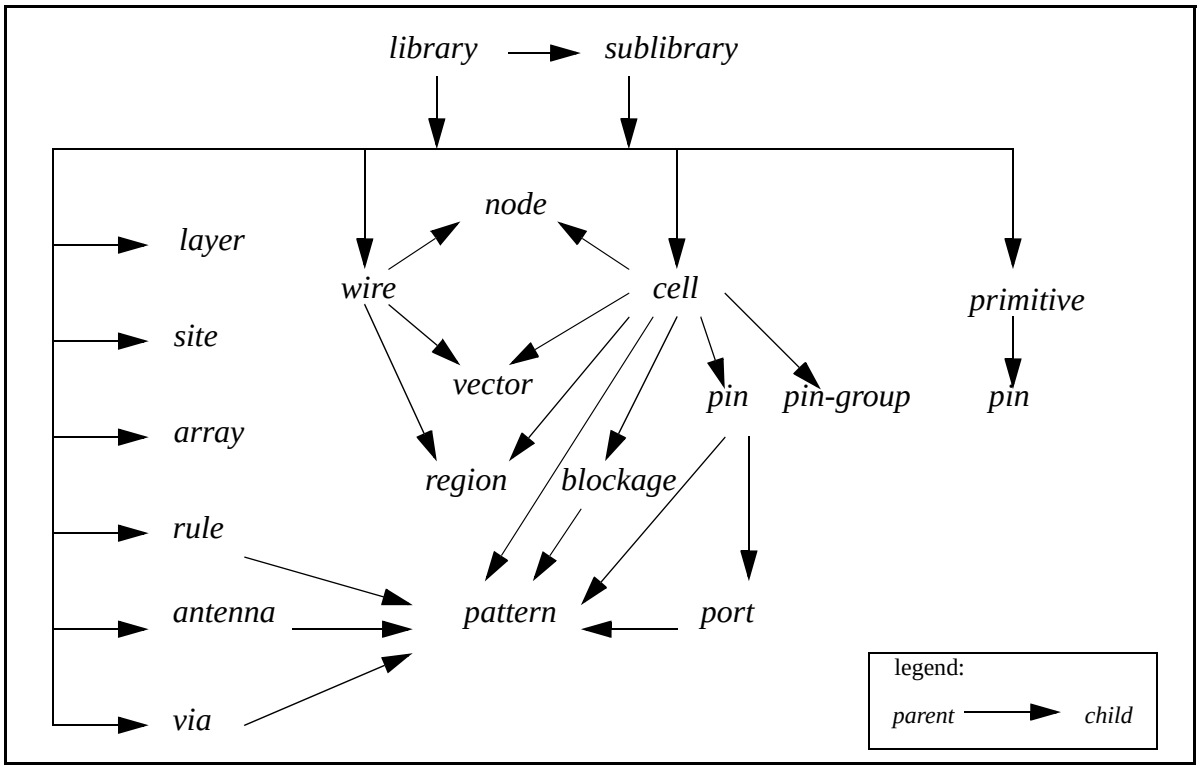


Figure 3—Parent/child relationship amongst library-specific objects

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 11.12.
TO	To statement	N/A	Compound	See 11.12.

Table 5—Singular statements (Continued)

Keyword	Item	Value	Complexity	Section
VIOLATION	Violation statement	N/A	Compound	See 11.10.
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.

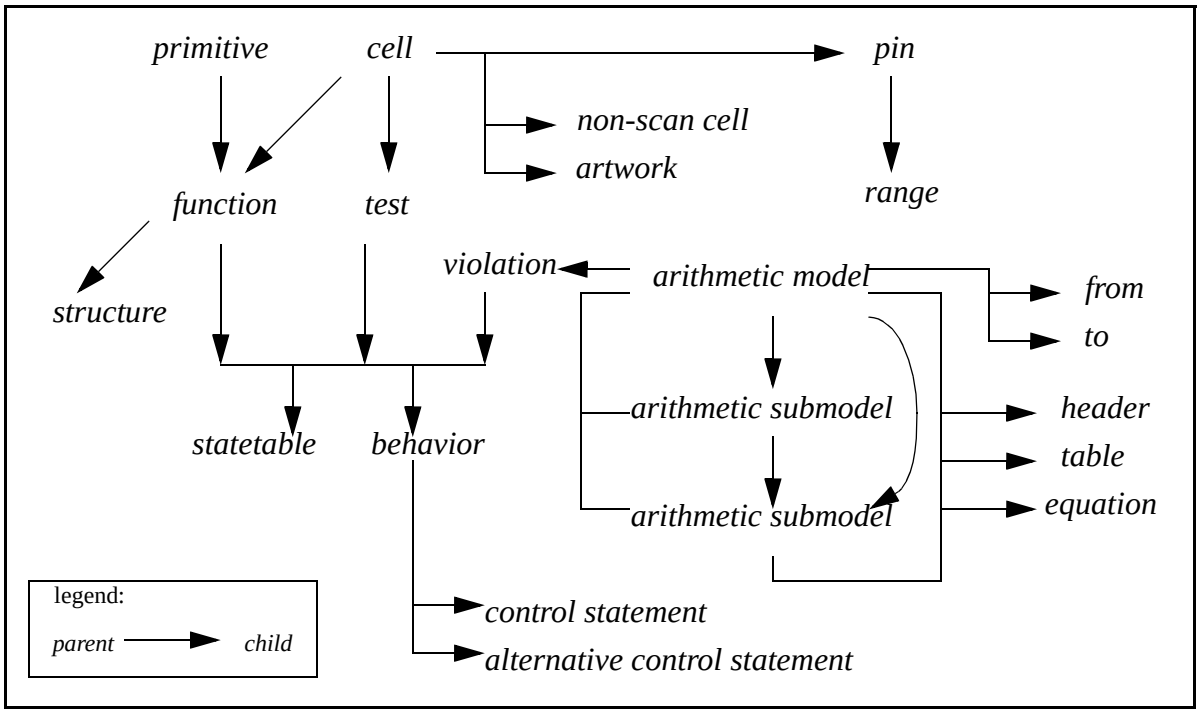


Figure 4—Parent/child relationship involving singular statements and plural statements

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 8.10.
Via instantiation	Mandatory	N/A	See 9.20.
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 8.10.
Boolean assignment	See 10.4.

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

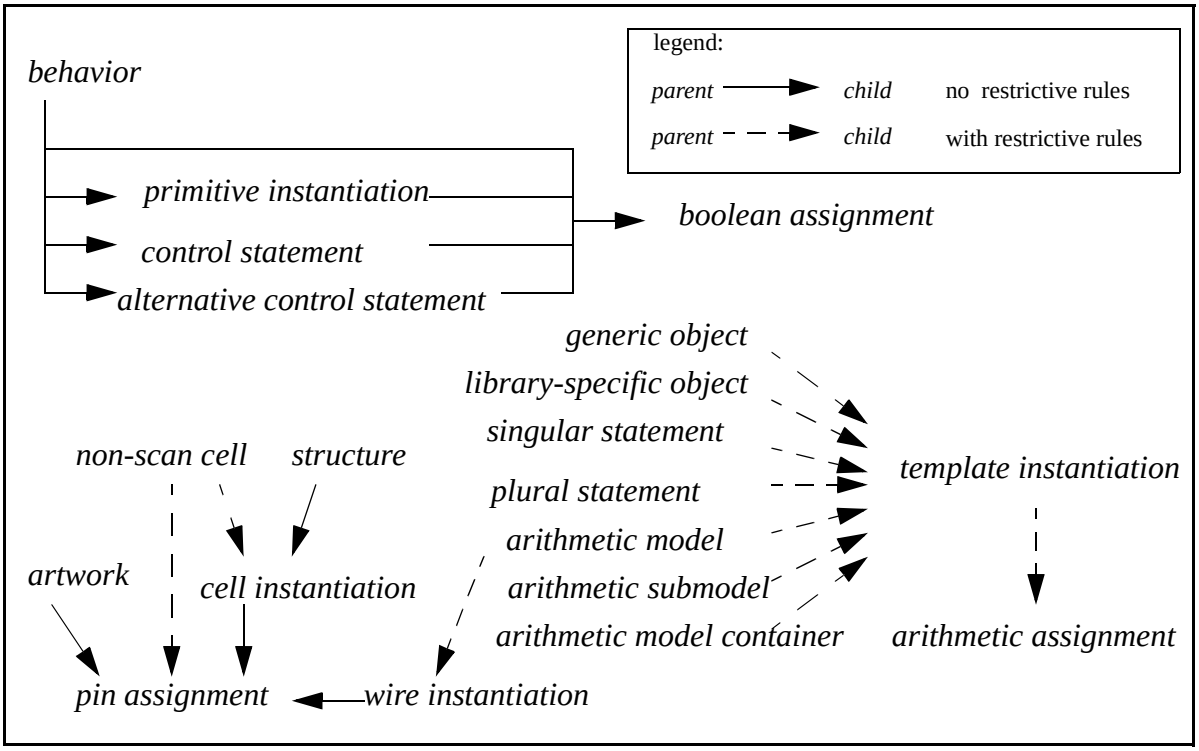


Figure 5—Parent/child relationship involving instantiation and assignment statements

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 11.3.
Arithmetic submodel	See 11.7.
Arithmetic model container	See 11.8.
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 9.2.2.

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 11.8.2.
`MIN`	Arithmetic submodel, also operator within *arithmetic expression.*	See 11.7, 11.2.3.
`MAX`	Arithmetic submodel, also operator within *arithmetic expression.*	See 11.4.4, 11.2.3.
`TYP`	Arithmetic submodel.	See 11.5.
`DEFAULT`	Annotation.	See 11.9.4.
`ABS`	Operator within *arithmetic expression*.	See 11.2.3.
`EXP`	Operator within *arithmetic expression*.	See 11.2.3.

Table 11—Keywords related to arithmetic model (Continued)

Keyword	Item / category	Section
LOG	Operator within *arithmetic expression*.	See 11.2.3.

The definitions of other predefined keywords, especially in the category arithmetic model, can be self-described in ALF using the *keyword declaration* statement (see 8.3).

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 10.11.4.
-	Minus	Both	See 10.11.4.
*	Multiply	Binary	See 10.11.4.
/	Divide	Binary	See 10.11.4.
%	Modulo	Binary	See 10.11.4.
**	Power	Binary	See 11.2.2.

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 10.11.1.
&&	Logical and	Binary	See 10.11.1.
||	Logical or	Binary	See 10.11.1.
~	bit-wise inversion	Unary	See 10.11.2.
&	bit-wise and	Both	See 10.11.2.
~&	bit-wise nand	Both	See 10.11.2.
|	bit-wise or	Both	See 10.11.2.
~|	bit-wise nor	Both	See 10.11.2.
^	Exclusive or	Both	See 10.11.2.
~^	Exclusive nor	Both	See 10.11.2.

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 10.11.6.
!=	Not equal	Binary	See 10.11.6.
>	Greater	Binary	See 10.11.6.
<	Lesser	Binary	See 10.11.6.
>=	Greater or equal	Binary	See 10.11.6.
<=	Lesser or equal	Binary	See 10.11.6.

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 10.11.5.
>>	Shift right	Binary	See 10.11.5.

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 10.13.3.
~>	Eventually followed by	Binary	See 10.13.3.
<->	Immediately following each other	Binary	See 10.13.4.
<~>	Eventually following each other	Binary	See 10.13.4.
&>	Simultaneous or immediately followed by	Binary	See 10.13.3.
<&>	Simultaneous or immediately following each other	Binary	See 10.13.4.

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, 8.10, 10.4.
?	Condition	Binary	See 10.13.5.
@	Control	Unary	See 10.4.

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 Multiplier prefix symbol

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

The purpose of a multiplier prefix symbol is the specification of a multiplier for the base unit associated with an *arithmetic model* (see Section 11.3). Only the leading characters of the multiplier prefix symbol shall be used for identification of the corresponding number. Optional subsequent letters can be used to indicate the base unit. For example, “pF” can be used to denote “picofarad”, “MegaHz” can be used to denote “megahertz”, etc.

```
multiplier_prefix_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—Multiplier prefix symbol

A multiplier prefix symbol shall relate to the International System of Units and Measurements [** reference needed **] as shown in Table 21.

Table 21—Multiplier prefix symbol and corresponding SI-prefix

Lexical token	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

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 8.9), which is to be replaced by an actual parameter in a *template instantiation* statement (see 8.10).

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 8.3.

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 Vector expression macro

A *vector expression macro* shall be defined as shown in Syntax 18.

```
vector_expression_macro ::=
    # . non_escaped_identifier
```

Syntax 18—Vector expression macro

A *vector expression macro* shall be used as a substitution for a predefined vector expression (see Section 10.12). The *alias* declaration (see Section 8.1) shall be used to establish the substitution mechanism.

6.14 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.

6.15 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 19.

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

Syntax 19—All purpose value

7.2 Multiplier prefix value

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

```
multiplier_prefix_value ::=
    unsigned_number | multiplier_prefix_symbol
```

Syntax 20—Multiplier prefix value

The multiplier prefix value shall be represented either as an *unsigned number* (see Section 6.5) or a *multiplier prefix symbol* (see Section 6.6).

7.3 String value

A *string value* shall be defined as shown in Syntax 21.

```
string_value ::=
    quoted_string | identifier
```

Syntax 21—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 22.

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

Syntax 22—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 23.

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

Syntax 23—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 24.

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

Syntax 24—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 25.

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

Syntax 25—Index value

An index value shall represent a particular position within a *vector pin* (see 9.5). 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 26.

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

Syntax 26—Index

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.

7.9 Pin variable and pin value

A *pin variable* and a *pin value* shall be defined as shown in Syntax 27.

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

Syntax 27—Pin variable, pin-port variable and pin value

A *pin variable* shall represent one of the following:

the name of a declared *pin* (see Section 9.5) in conjunction with an optional *index* (see Section 7.8),
the name of a declared *pingroup* (see Section 9.6) in conjunction with an optional index,
the name of a declared *node* (see Section 9.11), or
the name of a declared *port* (see Section 9.22) as a child of a scalar pin.

A *pin value* can be a *pin variable* or a *boolean value* (see Section 7.5).

7.10 Pin assignment

A *pin assignment* shall be defined as shown in Syntax 28.

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

Syntax 28—Pin assignment

A *pin assignment* shall represent an association between a pin variable and a pin value. The following rules define the compatibility between a pin variable and a pin value.

a) The bitwidth of the pin value shall be equal to the bitwidth of the pin variable.
b) A bit literal or a based literal representing a single bit can be assigned to a scalar pin.
c) A based literal or an unsigned integer, representing a binary number can be assigned to a pingroup, to a vector pin, or to a one-dimensional slice of a matrix pin.

7.11 Annotation

An *annotation* shall be divided into the subcategories *single value annotation* and *multi value annotation*, as shown in Syntax 29

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, 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.

```
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 29—Annotation

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 30

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

Syntax 30—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 31.

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

Syntax 31—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 32.

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

Syntax 32—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 33.

include ::=
INCLUDE quoted_string **;**

Syntax 33—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 and FORMAT annotation

An *associate* statement shall be defined as shown in Syntax 34.

```
associate ::=
    ASSOCIATE quoted_string ;
  | ASSOCIATE quoted_string { FORMAT_single_value_annotation }
```

Syntax 34—ASSOCIATE statement

The associate statement shall specify a relationship of the parent of the associate statement with an object described in a file referenced by the quoted string. The format annotation shall specify the format of the associated file. In contrast to the *include* statement (see Section 7.15), the ALF parser is not expected to read the associated file. The formal specification of the semantic validity of the association is beyond the scope of this standard.

Using a *keyword* declaration (see Section 8.3) in conjunction with a *context* annotation (see Section 8.5.4), a *valuetype* annotation (see Section 8.5.1), a *values* annotation (see Section 8.5.2), and a *default* annotation (see Section 8.5.3), the *format* annotation shall be defined as shown in Semantics 1.

```
KEYWORD FORMAT = single_value_annotation {
  CONTEXT = ASSOCIATE;
  VALUETYPE = identifier;
  VALUES { vhdl verilog c \c++ alf }
  DEFAULT = alf;
}
```

Semantics 1—FORMAT annotation

The meaning of the annotation values is specified in the following Table 23.

Table 23—FORMAT annotation values

Annotation value	Description
`vhdl`	The associated file is in a format specified by the IEEE 1076 std.
`verilog`	The associated file is in a format specified by the IEEE 1364 std.
`c`	The associated file is in a format specified by theANSI [** reference needed **]] std.
`\c++`	The associated file is in a format specified by the [** reference needed **] std.
`alf`	The associated file is in a format specified by this standard

Note: The format annotation value does not specify the format version of the associated file. An application that can read the associated file can obtain the version either from the associated file itself or by other means of version control.

7.17 REVISION statement

A *revision statement* shall be defined as shown in Syntax 35

```
revision ::=
    ALF_REVISION string_value
```

Syntax 35—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 24

Table 24—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.
"P1603.2002-10-24"	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 36.

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

Syntax 36—Generic object

The syntax items introduced in Syntax 36 are defined in Section 8.

7.19 Library-specific object

A *library-specific object* shall be defined as shown in Syntax 37.

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

Syntax 37—Library-specific object

The syntax items introduced in Syntax 37 are defined in Section 9.

7.20 All purpose item

An *all purpose item* shall be defined as shown in Syntax 38.

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

Syntax 38—All purpose item

The syntax items introduced in Syntax 38 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 39.

```
alias_declaration ::=
    ALIAS alias_identifier = original_identifier ;
  | ALIAS vector_expression_macro = ( vector_expression )
```

Syntax 39—ALIAS declaration

The alias declaration shall specify an alias *identifier* (see Section 6.11) or a *vector expression macro* (see Section 6.13).

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

The vector expression macro can be used as a substitution of a vector expression.

Example

```
ALIAS reset = clear;
ALIAS #.rising_edge = ( 01 clock );
```

8.2 CONSTANT declaration

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

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

Syntax 40—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 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
    | REFERENCETYPE_annotation
    | SI_MODEL_single_value_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 25.

Table 25—Syntax item identifier

Syntax item 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 11.3).
`arithmetic_submodel`	The keyword shall specify an *arithmetic submodel* (see 11.7).
`arithmetic_model_container`	The keyword shall specify an *arithmetic model container* (see 11.8).

8.4 SEMANTICS declaration

Semantics shall be declared as shown in Syntax 42—.

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. The hierarchical identifier can be composed of 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.

```
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
    | REFERENCETYPE_annotation
    | SI_MODEL_single_value_annotation
```

Syntax 42—SEMANTICS declaration

A *semantic item* can be used to qualify the contents of the semantics declaration. One or more annotations (see 8.5) can be used to qualify the contents of the semantics declaration.

8.5 Annotations and rules related to a KEYWORD or a SEMANTICS declaration

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

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 26.

Table 26—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`, `signed_integer`, `unsigned_integer`, `signed_real`, `unsigned_real`, `identifier`, `quoted_string`, `edge_value`, `pin_variable`, `control_expression`, `boolean_expression`, `arithmetic_expression`.	`identifier`	See Syntax 29, definition of *annotation value.*
`annotation_container`	N/A	N/A	An *annotation container* (see Syntax 30) has no value.
`arithmetic_model`	`number`, `signed_integer`, `unsigned_integer`, `signed_real`, `unsigned_real`, `identifier`, `bit_literal`, `based_literal`.	`number`	See Syntax 22, definition of *arithmetic value.*

Table 26—VALUETYPE annotation (Continued)

Syntax item identifier	Set of legal values for VALUETYPE	Default value for VALUETYPE	Comment
`arithmetic_submodel`	N/A	N/A	An *arithmetic sub-model* (see 11.7) shall always have the same `value-type` as its parent arithmetic model.
`arithmetic_model_container`	N/A	N/A	An *arithmetic model container* (see 11.8) 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.

The valuetype annotation can be partially self-described as shown in Semantics 2.

```
KEYWORD VALUETYPE = single_value_annotation {
  CONTEXT = KEYWORD;
}
```

Semantics 2—Partial self-description of VALUETYPE annotation

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*. It shall be 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. The *values* annotation and the *valuetype* annotation shall be compatible.

The values annotation can be partially self-described as shown in Semantics 3.

```
KEYWORD VALUES = multi_value_annotation {
  CONTEXT { KEYWORD SEMANTICS }
}
```

Semantics 3—Partial self-description of VALUES annotation

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.

A partial self-description of the default annotation is given in Semantics 4.

```
KEYWORD DEFAULT = single_value_annotation {
  CONTEXT { KEYWORD SEMANTICS arithmetic_model }
}
```

Semantics 4—Partial self-description of DEFAULT annotation

A default annotation shall also be applicable for an *arithmetic model* (see 11.3 and 11.9.4).

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 be a *single value annotation* or a *multi value annotation*. It 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.

A hierarchical identifier can be used to specify the ALF type of a legal parent of the statement, constraint by the ALF type of the grandparent or by the ALF type of the great-grandparent etc.

A partial self-description of the context annotation is given in Semantics 5.

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

Semantics 5—Partial self-description of CONTEXT annotation

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 } }
```

The following example shows the use of an hierarchical identifier.

```
KEYWORD PrimitivePinQualifier = annotation { CONTEXT = PRIMITIVE.PIN ; }
```

8.5.5 REFERENCETYPE annotation

The *referencetype* annotation shall be a *single value annotation* or a *multi value annotation*. The referencetype annotation shall be legal if the syntax item identifier in the keyword declaration is *annotation*, *single value annotation* or *multi value annotation*.

A partial self-description of the referencetype annotation is given in Semantics 6.

```
KEYWORD REFERENCETYPE = annotation {
  CONTEXT { KEYWORD SEMANTICS }
  VALUES { CLASS LIBRARY SUBLIBRARY CELL PIN PINGROUP
    PRIMITIVE WIRE NODE VECTOR LAYER VIA RULE ANTENNA
    BLOCKAGE PORT SITE ARRAY PATTERN REGION
    arithmetic_model arithmetic_submodel }
}
```

Semantics 6—Partial self-description of REFERENCETYPE annotation

The purpose of the referencetype annotation is to specify the ALF type of a referenced object. An object shall be referenced by its ALF name or eventually by a hierarchical identifier involving the ALF name of the parent of the object.

Example:

```
CLASS myClass;
KEYWORD myReference = single_value_annotation {
    REFERENCETYPE = CLASS;
}
CELL myCell {
    myReference = myClass;
}
```

In this example, the annotation “myReference” refers to an object of the ALF type “CLASS” with the ALF name “myClass”.

8.5.6 SI_MODEL annotation

The *SI-model* annotation shall be a *single value annotation*. It shall specify a relation of a declared keyword with the International System of Units and Measurements [** reference needed **]. The SI-model annotation is only applicable for a keyword declaring an *arithmetic model* (see Section 11.3).

A self-description of the SI-model annotation is given in Semantics 7.

```
KEYWORD SI_MODEL = single_value_annotation {
  CONTEXT = KEYWORD;
  VALUETYPE = identifier;
  VALUES {
    TIME FREQUENCY CURRENT VOLTAGE POWER ENERGY
    RESISTANCE CAPACITANCE INDUCTANCE
    DISTANCE AREA
  }
}
```

Semantics 7—SI model annotation

The set of legal annotation values is shown in the following Table 27.

Table 27—SI_MODEL annotation

annotation value	mathematical symbol	base unit	relationship with other quantity	Reference to arithmetic model declaration
TIME	*t*	Second		see Section 11.11.1
FREQUENCY	*f*	Hertz	1 / *t*	see
CURRENT	*I*	Ampere		see
VOLTAGE	*V*	Volt		see
RESISTANCE	*R*	Ohm	*V* / *I*	see

Table 27—SI_MODEL annotation (Continued)

annotation value	mathematical symbol	base unit	relationship with other quantity	Reference to arithmetic model declaration
CAPACITANCE	*C*	Farad	*I* / (d*V* / d*t*)	see
INDUCTANCE	*L*	Henry	*V* / (d*I* / d*t*)	see
ENERGY	*E*	Joule		see
POWER	*P*	Watt	*I V*, d*E* / d*t*	see
DISTANCE	*d*	Meter		see
AREA	*A*	Square meter	d^2	see

8.5.7 Rules for legal usage of KEYWORD and SEMANTICS declaration

The following rules shall apply for legal use of annotations within a keyword or a semantics declaration.

a) A keyword declaration can not overwrite, redefine, or otherwise invalidate a syntax rule.
b) A semantics declaration shall relate to a keyword declaration or a syntax rule. A semantics declaration shall be compatible with a related keyword declaration or a related syntax rule.

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; }
}
```

8.6 CLASS declaration

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

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. Such a reference is made by a *class reference* annotation (see Section 8.7).

```
class_declaration ::=
      CLASS class_identifier ;
    | CLASS class_identifier { { class_item } }
class_item ::=
      all_purpose_item
    | geometric_model
    | geometric_transformation
```

Syntax 43—CLASS declaration

The semantics specified by a *class item* within a class declaration shall be inherited by the statement containing the reference. A class item can be an *all purpose item* (see Section 7.20), a *geometric model* (see Section 10.16) or a *geometric transformation* (see Section 10.18).

8.7 Annotations related to a CLASS declaration

This subsection specifies how other objects can make a reference to a class by using either a *general* class reference annotation or a *specific* class reference annotation.

8.7.1 General CLASS reference annotation

A general *class reference* annotation shall be defined as shown in Semantics 8.

```
SEMANTICS CLASS = annotation {
  CONTEXT {
    library_specific_object
    arithmetic_model
  }
  VALUETYPE = identifier;
  REFERENCETYPE = CLASS;
}
```

Semantics 8—CLASS reference annotation

Note: A class declaration itself can not contain a general class reference annotation. This avoids circular 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”
```

Note: It is possible that a reference to multiple classes can result in the inheritance of semantically incompatible attributes. It is expected that an ALF compiler or an ALF interpreter detects such semantic incompatibility. However, the behavior of an application as a consequence of this detection is not specified by this standard, since the desired behavior can depend on the nature of the application.

8.7.2 USAGE annotation

The *usage* annotation shall be defined as shown in Semantics 9.

```
KEYWORD USAGE = annotation {
  CONTEXT = CLASS;
  VALUETYPE = identifier;
  VALUES {
    SWAP_CLASS RESTRICT_CLASS
    SIGNAL_CLASS SUPPLY_CLASS CONNECT_CLASS
    SELECT_CLASS NODE_CLASS
    EXISTENCE_CLASS CHARACTERIZATION_CLASS
    ORIENTATION_CLASS SYMMETRY_CLASS
  }
}
```

Semantics 9—USAGE annotation

The usage annotation shall specify, which specific class reference annotation can be legally used to make a reference to the class.

The set of legal annotation values is shown in the following Table 28.

Table 28—USAGE annotation

annotation value	definition of specific class reference annotation
SWAP_CLASS	see Section 9.4.3
RESTRICT_CLASS	see
SIGNAL_CLASS	see
SUPPLY_CLASS	see
CONNECT_CLASS	see
SELECT_CLASS	see
NODE_CLASS	see
EXISTENCE_CLASS	see
CHARACTERIZATION_CLASS	see
ORIENTATION_CLASS	see
SYMMETRY_CLASS	see

Note: Knowing the ALF type of a legal parent of a specific class reference annotation, the ALF parser can evaluate the contents of the class declaration for semantic correctness. If the usage annotation is not present, the ALF paraser can evaluate the contents of the class declaration for semantic correctness only when encountering a reference to the class.

8.8 GROUP declaration

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

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

Syntax 44—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];} }
}
```

8.9 TEMPLATE declaration

A *template* shall be declared as shown in Syntax 45.

template_declaration ::=
TEMPLATE *template*_identifier **{** ALF_statement { ALF_statement } **}**

Syntax 45—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 8.10) 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).

8.10 TEMPLATE instantiation

A *template* shall be instantiated in form of a *static template instantiation* or a *dynamic template instantiation*, as shown in Syntax 46

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 an all-purpose value, or alternatively, replacement by reference, using an annotation (see 7.11). A dynamic template instantiation shall support replacement by reference only, using an annotation and/or an arithmetic model (see 7.11 and 11.3) and/or an arithmetic assignment.

```
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 46—TEMPLATE instantiation

In the case of replacement by reference, the reference shall be established by a non-escaped identifier matching the placeholder identifier without the angular brackets. 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 legal. 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 legal.
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 47.

```
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 47—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 related to a LIBRARY or a SUBLIBRARY declaration

Add lead-in text

9.2.1 LIBRARY reference annotation

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

The purpose of a library reference annotation is to establish an association between a library or a sublibrary and an *arithmetic model* (see Section 11.3).

```
SEMANTICS LIBRARY = annotation {
  VALUETYPE = identifier;
  CONTEXT = arithmetic_model;
  REFERENCETYPE { LIBRARY SUBLIBRARY }
}
```

Semantics 10—LIBRARY reference annotation

A hierarchical identifier can be used to specify a reference to a sublibrary as a child of a library.

9.2.2 INFORMATION annotation container

An *information* annotation container shall be defined as shown in Semantics 11.

```
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 11—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 29.

Table 29—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 48.

```
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 48—CELL declaration

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

9.4 Annotations related to a CELL declaration

This section defines annotations and attribute values related to a cell declaration.

9.4.1 CELL reference annotation

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

```
SEMANTICS CELL = annotation {
  VALUETYPE = identifier;
  CONTEXT = arithmetic_model;
  REFERENCETYPE = CELL;
}
```

Semantics 12—CELL reference annotation

The purpose of a cell reference annotation is to establish an association between a cell and an *arithmetic model* (see Section 11.3).

A hierarchical identifier can be used to specify a reference to a cell as a child of a library or a sublibrary.

9.4.2 CELLTYPE annotation

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

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

Semantics 13—CELLTYPE annotation

The *celltype* shall divide cells into categories, as specified in Table 30.

Table 30—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.

Table 30—CELLTYPE annotation values (Continued)

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

9.4.3 SWAP_CLASS annotation

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

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

Semantics 14—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 9.4.4) authorizes usage of the cell and the cells to be swapped are compatible from an application standpoint.

9.4.4 RESTRICT_CLASS annotation

A *restrict-class* annotation shall be defined as shown in Semantics 15.

```
KEYWORD RESTRICT_CLASS = annotation {
  CONTEXT { CELL CLASS }
  VALUETYPE = identifier;
  REFERENCETYPE = CLASS;
}
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 15—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 15 is specified in Table 31.

Table 31—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 15.

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".

9.4.5 SCAN_TYPE annotation

A *scan_type* annotation shall be defined as shown in Semantics 16.

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

Semantics 16—SCAN_TYPE annotation

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

Table 32—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.

9.4.6 SCAN_USAGE annotation

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

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

Semantics 17—SCAN_USAGE annotation

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

Table 33—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.

Table 33—SCAN_USAGE annotations for a CELL object (Continued)

Annotation value	Description
`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.

9.4.7 BUFFERTYPE annotation

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

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

Semantics 18—BUFFERTYPE annotation

It can take the *values* shown in Table 34.

Table 34—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.

9.4.8 DRIVERTYPE annotation

A *drivertype* annotation shall be defined as shown in Semantics 19.

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

Semantics 19—DRIVERTYPE annotation

It can take the *values* shown in Table 35.

Table 35—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`.

9.4.9 PARALLEL_DRIVE annotation

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

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

Semantics 20—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`.

9.4.10 PLACEMENT_TYPE annotation

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

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

Semantics 21—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 36.

Table 36—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.

9.4.11 SITE reference annotation for a CELL

A *site* reference annotation in the context of a cell shall be defined as shown in Semantics 72.

```
SEMANTICS CELL.SITE = single_value_annotation;
```

Semantics 22—SITE reference annotation

The purpose of a site reference annotation in the context of a cell is to specify a legal placement location for the cell.

9.4.12 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 37 can be used within a `CELL` with `CELLTYPE=memory`.

Table 37—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

Table 37—Attribute values for a CELL with CELLTYPE=memory (Continued)

Attribute item	Description
synchronous	operation synchronized with a clock signal

The attributes shown in Table 38 can be used within a CELL with CELLTYPE=block.

Table 38—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.
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 39 can be used within a CELL with CELLTYPE=core.

Table 39—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 40 can be used within a `CELL` with `CELLTYPE=special`.

Table 40—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).

9.5 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
    | pattern
    | 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*.

The order of pin declarations within a cell declaration shall reflect the order in which pins are referenced, when the cell is instantiated in a netlist. The *view* annotation (see Section 9.7.3) shall further specify which pin is visible in a netlist.

Note: The order of pin declarations is irrelevant, if pin reference by name is used.

A scalar pin can be associated with a general electrical signal. However, a vector pin or a matrix pin can only be associated with a digital signal. 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]`.

9.6 PINGROUP declaration

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

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* annotation.

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.

```
pingroup ::=
    simple_pingroup | vector_pingroup
simple_pingroup ::=
     PINGROUP pingroup_identifier
      { MEMBERS_multi_value_annotation { all_purpose_item } }
    | simple_pingroup_template_instantiation
vector_pingroup ::=
    | PINGROUP multi_index pingroup_identifier
        { MEMBERS_multi_value_annotation { vector_pingroup_item } }
    | vector_pingroup_template_instantiation
vector_pingroup_item ::=
     all_purpose_item
    | range
```

Syntax 50—PINGROUP declaration

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.

9.7 Annotations related to a PIN or a PINGROUP declaration

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

9.7.1 PIN reference annotation

A *pin reference* annotation shall be defined as shown in .

```
SEMANTICS PIN = annotation {
  VALUETYPE = pin_variable;
  CONTEXT { arithmetic_model FROM TO }
  REFERENCETYPE { PIN PINGROUP PORT NODE }
}
```

Semantics 23—PIN reference annotation

The purpose of a pin reference annotation is to establish an association between a pin, a pingroup, a *port* (see Section 9.22) or a *node* (see Section 9.11) and an *arithmetic model* (see Section 11.3) or a *from-to* statement (see Section 11.12). In this context, the pin, pingroup, port or node is used as a reference point related to a timing measurement or an electrical measurement.

A hierarchical identifier can be used to specify a reference to a pin, a pingroup, a port or a node as a child of a cell, a pin or a wire.

9.7.2 MEMBERS annotation

A *members* annotation shall be defined as shown in Semantics 24.

The purpose of the members annotation is to specify the constituent pins of a pingroup.

9.7.3 VIEW annotation

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

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

```
KEYWORD MEMBERS = multi_value_annotation {
  CONTEXT = PINGROUP;
  VALUETYPE = identifier;
  REFERENCETYPE = PIN;
}
```

Semantics 24—MEMBERS annotation

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

Semantics 25—VIEW annotation

It can take the values shown in Table 41.

Table 41—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.

9.7.4 PINTYPE annotation

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

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

Semantics 26—PINTYPE annotation

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

It can take the values shown in Table 42.

Table 42—PINTYPE annotations for a PIN object

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

9.7.5 DIRECTION annotation

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

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

Semantics 27—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 43.

Table 43—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`.

9.7.6 SIGNALTYPE annotation

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

```
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 28—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 44

Table 44—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 44—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.

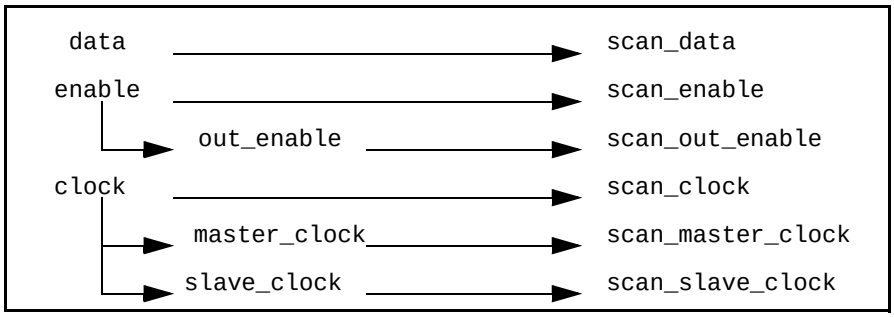


Figure 6—Scheme for construction of composite signaltype values

The composite `SIGNALTYPE` values are defined in Table 45

Table 45—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 45—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 44 and Table 45, 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`.

9.7.7 ACTION annotation

An *action* annotation shall be defined as shown in Semantics 29.

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

Semantics 29—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 46.

Table 46—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 47. The rule applies also to any composite `SIGNALTYPE` values based on the fundamental values.

Table 47—ACTION applicable in conjunction with SIGNALTYPE values

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

Table 47—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

9.7.8 POLARITY annotation

A *polarity* annotation shall be defined as shown in Semantics 30.

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

Semantics 30—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 48.

Table 48—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 49..

Table 49—POLARITY applicable in conjunction with SIGNALTYPE values

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

Table 49—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,

9.7.9 CONTROL_POLARITY annotation container

A *control polarity* annotation container shall be defined as shown in Semantics 31.

```
KEYWORD CONTROL_POLARITY = annotation_container {
  CONTEXT = PIN ;
}
SEMANTICS
CONTROL_POLARITY.identifier = single_value_annotation {
  VALUETYPE = identifier ;
  VALUES { high low rising_edge falling_edge double_edge }
}
```

Semantics 31—Control polarity annotation container

The control polarity annotation container can be used in the context of a pin with signaltype value *control* or *clock*.

The purpose of the control polarity annotation container is to specify the active state or the active edge of an input signal in association with a particular mode of operation. The name of the mode of operation is given by the annotation identifier.

Example:

```
PIN ModeSel1 {
    DIRECTION = input; SIGNALTYPE = control;
    CONTROL_POLARITY { normal=high; scan=low; hold=low; }
}
PIN ModeSel2 {
    DIRECTION = input; SIGNALTYPE = control;
    CONTROL_POLARITY { scan=high; hold=low; }
}
```

```
// corresponding truth table:
// ModeSel1 ModeSel2 mode of operation
// 0        0        hold
// 0        1        scan
// 1        ?        normal
```

9.7.10 DATATYPE annotation

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

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

Semantics 32—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 50.

Table 50—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.

9.7.11 INITIAL_VALUE annotation

An *initial value* annotation shall be defined as shown in Semantics 33.

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

Semantics 33—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 74).

9.7.12 SCAN_POSITION annotation

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

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

Semantics 34—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.

9.7.13 STUCK annotation

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

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

Semantics 35—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 51.

Table 51—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.
`both`	Pin can exhibit a faulty static high or low state.
`none`	Pin can not exhibit a faulty static state.

9.7.14 SUPPLYTYPE annotation

A *supplytype* annotation shall be defined as shown in Semantics 36.

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

Semantics 36—SUPPLYTYPE annotation

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

Table 52—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 42).

9.7.15 SIGNAL_CLASS annotation

A *signal-class* annotation shall be defined as shown in Semantics 37.

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

Semantics 37—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 arbitrary *signaltype* value (see Section 9.7.6).

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 9.22).

```
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; }
}
```

9.7.16 SUPPLY_CLASS annotation

A *supply-class* annotation shall be defined as shown in Semantics 38.

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

Semantics 38—SUPPLY_CLASS annotation

The value shall be the name of a declared CLASS.

The purpose of the supply-class annotation is to specify a relation between a pin and a power supply system, represented by the refered class.

The supply-class annotation shall apply for a pin with any *signaltype* value (see Section 9.7.6) or *supplytype* value (see Section 9.7.14).

The supply-class annotation shall also apply for a class with *usage* value *connect-class* (see Section 9.7.19). The latter class shall represent a global net related to a power supply system.

The supply-class annotation shall also apply for the arithmetic models *power* and *energy* (see Section 11.11.15).

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; }
}
```

9.7.17 DRIVETYPE annotation

A *drivetype* annotation shall be defined as shown in Semantics 39.

```
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 39—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 (see Table 74).

The drivetype annotation can take the values shown in Table 53.

Table 53—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 53—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.

9.7.18 SCOPE annotation

A *scope* annotation shall be defined as shown in Semantics 40.

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

Semantics 40—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 9.13) or within a behavior statement (see Section 10.4).

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

Table 54—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 54—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.

9.7.19 CONNECT_CLASS annotation

A *connect_class* annotation shall be defined as shown in Semantics 41.

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

Semantics 41—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 9.7.16. For application in conjunction with *connect-rule* see Section 11.20.1.

9.7.20 SIDE annotation

A *side* annotation shall be defined as shown in Semantics 42.

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

Semantics 42—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 55.

Table 55—SIDE annotations for a PIN object

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

Table 55—SIDE annotations for a PIN object (Continued)

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

9.7.21 ROW and COLUMN annotation

A *row* annotation and a *column* annotation shall be defined as shown in Semantics 43.

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

Semantics 43—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.

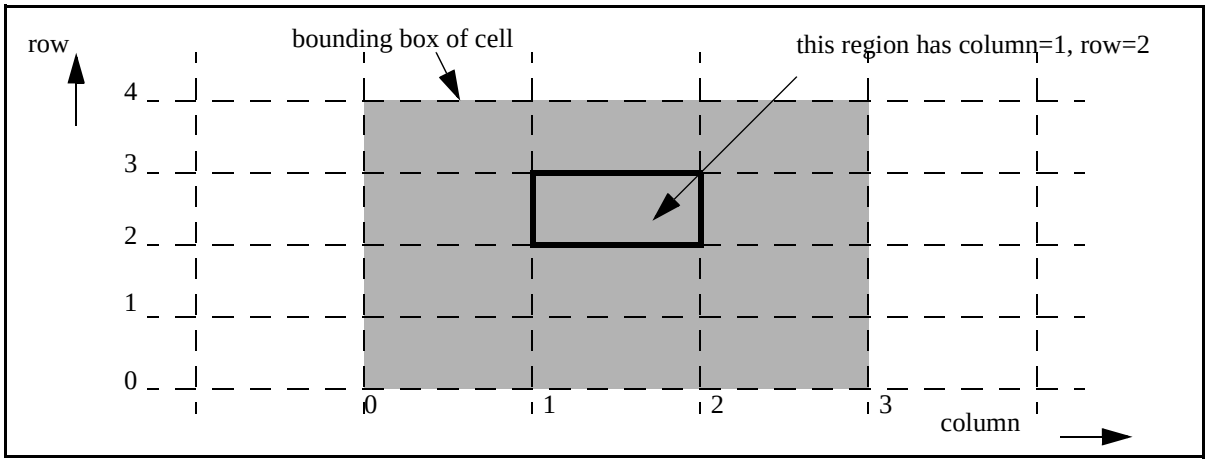


Figure 7—ROW and COLUMN relative to a bounding box of a CELL

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.

9.7.22 ROUTING_TYPE annotation

A *routing-type* annotation shall be defined as shown in Semantics 44.

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

Semantics 44—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 56.

Table 56—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

9.7.23 PULL annotation

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

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

Semantics 45—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 57.

Table 57—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	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.

9.7.24 ATTRIBUTE values for a PIN or a PINGROUP

The attribute values shown in Table 58 are applicable for a pin or a pingroup with the following characteristics.

Table 58—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 attribute values shown in Table 59 are applicable for a *pin* or a *pingroup* of a cell with *celltype* value *memory* in conjunction with a specific *signaltype* value.

Table 59—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 59—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 attribute values shown in Table 60 are applicable for a pair of signals.

Table 60—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 attribute values shown in Table 61 shall be defined for memory BIST.

Table 61—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 61—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).

9.8 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).

9.9 WIRE declaration

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

```
wire ::=
      WIRE wire_identifier { { 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.

9.10 Annotations related to a WIRE declaration

Add lead-in text

9.10.1 WIRE reference annotation

A *wire reference* annotation shall be defined as shown in .

```
SEMANTICS WIRE = annotation {
  VALUETYPE = identifier;
  CONTEXT = arithmetic_model;
  REFERENCETYPE = WIRE;
}
```

Semantics 46—WIRE reference annotation

The purpose of a wire reference annotation is to establish an association between a vector and an *arithmetic model* (see Section 11.3).

A hierarchical identifier can be used to specify a reference to a wire as a child of a cell or a sublibrary or a library.

9.10.2 WIRETYPE annotation

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

```
KEYWORD WIRETYPE = single_value_annotation {
  CONTEXT = WIRE;
  VALUETYPE = identifier;
  VALUES { estimated extracted interconnect load }
}
```

Semantics 47—WIRETYPE annotation

The purpose of the wiretype annotation is to define a purpose and a usage model for the wire statement.

The wiretype annotation can take the values shown in Table 62.

Table 62—WIRETYPE annotations for a WIRE object

Annotation value	Description
`estimated`	The wire declaration contains a statistical wireload model, i.e., a model for estimation of R, L, C values for a net, without a structural description of a circuit.

Table 62—WIRETYPE annotations for a WIRE object (Continued)

Annotation value	Description
`extracted`	The wire declaration contains a structural description of a circuit, i.e. a netlist, related to the parent object, i.e. a cell. The R, L, C components represent extracted parasitics from a physical implementation of the cell.
`interconnect`	The wire declaration contains a structural description of a circuit, representing a model for interconnect analysis. A general R, L, C interconnect network is expected to be reduced to the specified circuit for analysis purpose.
`load`	The wire declaration contains a structural description of a circuit, which is to be connected as a load to a device, i.e., a cell, for characterization or test. A wire instantiation (see Section 11.11) shall be used to describe such a connection.

An R, L, C component within the context of the wire declaration shall be described as *arithmetic model* (see Section 11). A related electrical measurement, e.g., voltage, current, noise, shall also be described as arithmetic model.

9.10.3 SELECT_CLASS annotation

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

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

Semantics 48—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 selecting a set of wire objects by an application. The user of the application can select a set of related wire objects by specifying the name of a class rather than specifying the name of each wire object.

The semantics of the select class shall be under the responsibility of the library provider. The library provider can define a select class based on criteria such as range of wire length, range of die size, accuracy requirements for delay calculation etc.

The select class annotation is orthogonal to the wiretype annotation, as illustrated in the following example.

Example:

```
CLASS short_wire { USAGE = SELECT_CLASS ; }
CLASS long_wire { USAGE = SELECT_CLASS ; }
WIRE pre_layout_small {
    WIRETYPE = estimated; SELECT_CLASS = short_wire;
    // put statistical wireload model here
```

```
}
WIRE post_layout_small {
    WIRETYPE = interconnect; SELECT_CLASS = short_wire;
    // put interconnect analysis model here
}
WIRE pre_layout_large {
    WIRETYPE = estimated; SELECT_CLASS = long_wire;
    // put statistical wireload model here
}
WIRE post_layout_large {
    WIRETYPE = interconnect; SELECT_CLASS = long_wire;
    // put interconnect analysis model here
}
```

9.11 NODE declaration

A *node* shall be declared as shown in Syntax 53.

node ::=
NODE *node*_identifier **;**
| **NODE** *node*_identifier **{** { node_item } **}**
| *node*_template_instantiation
node_item ::=
all_purpose_item

Syntax 53—NODE statement

The purpose of a node declaration is to specify an electrical node in the context of a *wire* declaration (see Section 9.9) or in the context of a *cell* declaration (see Section 9.3).

9.12 Annotations related to a NODE declaration

9.12.1 NODE reference annotation

A *node reference* annotation shall be defined as shown in .

```
SEMANTICS NODE = multi_value_annotation {
  VALUETYPE = pin_variable;
  CONTEXT = arithmetic_model;
  REFERENCETYPE { PIN PORT NODE }
}
```

Semantics 49—PIN reference annotation

The purpose of a node reference annotation is to establish an association between a pin, a pingroup, a *port* (see Section 9.22) or a *node* (see Section 9.11) and an *arithmetic model* (see Section 11.3). In this context, the pin, pingroup, port or node is used to specify the connectivity of an electrical component within a structural circuit.

A hierarchical identifier can be used to specify a reference to a pin, a port or a node as a child of a cell, a pin or a wire.

9.12.2 NODETYPE annotation

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

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

Semantics 50—NODETYPE annotation

The *values* shall have the semantic meaning shown in Table 63.

Table 63—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. In case of an ideal driver, the source node is collapsed with a driver node . The collapsed node shall have the nodetype value *driver.*
`sink`	The node is a virtual end point of signal propagation. In case of an ideal receiver, the sink node is collapsed with a receiver node . The collapsed node shall have the nodetype value *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 node.
`ground`	The node supports electrical current for a falling signal at a source or a driver node and a reference for logic a low signal at a sink or a receiver node
`interconnect`	The node serves for connecting purpose only.

A circuit wherein all nodes are interconnected by either a resistance or an inductance or a voltage source is called a DC-connected net.

The meaning of the nodetype annotation values in context of a DC-connected net is illustrated in the following figure 8.

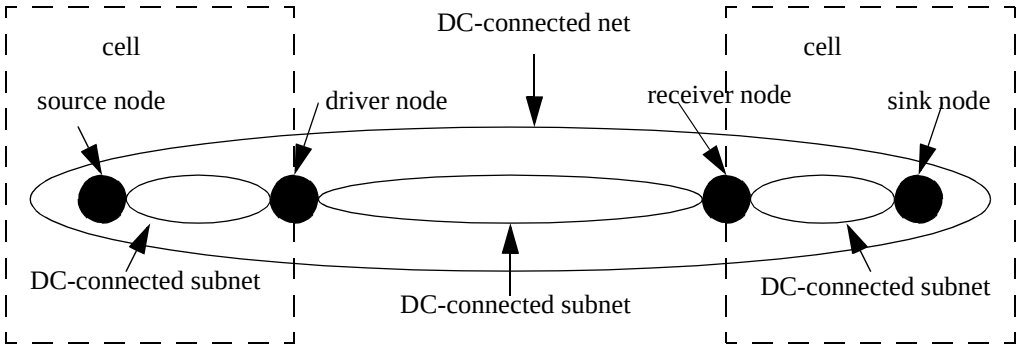


Figure 8—NODETYPE in context of a DC-connected net

The nodetype annotation specifies a way of separating a DC-connected net into three DC-connected subnets. The DC-connected subnet between a *source* node and a *driver* node is considered a model of an internal interconnect within a cell. The driver node shall be considered an output pin of the cell. The DC-connected subnet between a *receiver* node and a *sink* node is considered a model of an internal interconnect within another cell. The driver node shall be considered an input pin of the cell. The DC-connected subnet between a *driver* node and a *receiver* node is considered a model of the external interconnect between two cells. The association of an *interconnect* node with either cell or with the interconnect between the cells is infered by the connectivity within the DC-connected net. A *power* or a *ground* node which is not part of the DC-connected net is considered global.

9.12.3 NODE_CLASS annotation

A *node class* annotation shall be defined as shown in Semantics 51.

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

Semantics 51—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 cell in the case where an association can not be infered by the connectivity within a DC-connected net.

Example:

```
WIRE CrosstalkAccrossPowerDomains {
    CLASS aggressor { USAGE = NODE_CLASS; }
    CLASS victim { USAGE = NODE_CLASS; }
    NODE vdd1    { NODETYPE = power;  NODE_CLASS = aggressor; }
    NODE driver1 { NODETYPE = driver; NODE_CLASS = aggressor; }
    NODE vdd2    { NODETYPE = power;  NODE_CLASS = victim; }
    NODE driver2 { NODETYPE = driver; NODE_CLASS = victim; }
```

```
// put electrical components here
// put crosstalk model here
}
```

The node declarations in this example provide a context for a crosstalk model, where the noise magnitude at the victim's driver node can depend on the supply voltage at the aggressor's power node, the supply voltage at the victim's power node, the signal characteristics at the aggressor's driver node and other parameters. The crosstalk model itself is not shown here.

9.13 VECTOR declaration

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

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

Syntax 54—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 10.4) shall specify a stimulus related to characterization or test.

9.14 Annotations related to a VECTOR declaration

Add lead-in text

9.14.1 VECTOR reference annotation

A *vector reference* annotation shall be defined as shown in .

```
SEMANTICS VECTOR = single_value_annotation {
  VALUETYPE = control_expression;
  CONTEXT = arithmetic_model;
  REFERENCETYPE = VECTOR;
}
```

Semantics 52—VECTOR reference annotation

The purpose of a vector reference annotation is to establish an association between a vector and an *arithmetic model* (see Section 11.3).

9.14.2 PURPOSE annotation

A *purpose* annotation shall be defined as shown in Semantics 53.

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.

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

Semantics 53—PURPOSE annotation

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

Table 64—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 11.11.1 to Section 11.11.11)
`power`	The vector contains an arithmetic model related to power calculation (see Section 11.11.15)
`noise`	The vector contains an arithmetic model related to noise calculation (see Section 11.11.14)
`reliability`	The vector contains an arithmetic model related to reliability calculation (see Section 11.17.2, also Section 11.11.1 and Section 11.11.2)

9.14.3 OPERATION annotation

An *operation* annotation shall be defined as shown in Semantics 54.

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

Semantics 54—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 65.

Table 65—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.
`load`	Operation for supplying data to a control register.
`iddq`	Operation for supply current measurements in quiescent state.

9.14.4 LABEL annotation

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

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

Semantics 55—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 11.17.1) 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.

9.14.5 EXISTENCE_CONDITION annotation

An *existence-condition* annotation shall be defined as shown in Semantics 56.

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

Semantics 56—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.

9.14.6 EXISTENCE_CLASS annotation

An *existence-class* annotation shall be defined as shown in Semantics 57.

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

Semantics 57—EXISTENCE_CLASS annotation

The identifier shall be the name of a declared class.

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.

9.14.7 CHARACTERIZATION_CONDITION annotation

A *characterization-condition* annotation shall be defined as shown in Semantics 58.

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

Semantics 58—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.

9.14.8 CHARACTERIZATION_VECTOR annotation

A *characterization-vector* annotation shall be defined as shown in Semantics 59.

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

Semantics 59—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.

9.14.9 CHARACTERIZATION_CLASS annotation

A *characterization-class* annotation shall be defined as shown in Semantics 60.

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

Semantics 60—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.

9.14.10 MONITOR annotation

A *monitor* annotation shall be defined as shown in Semantics 61.

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

Semantics 61—MONITOR annotation

The purpose of the monitor annotation is to specify a set of *pin variables* (see Section 7.9) involved in the evaluation of a vector expression. Events on this set of pin variables need to be monitored for detection of a specified *event sequence* (see Section 10.13.2).

9.15 LAYER declaration

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

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

Syntax 55—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. The layer which is created first shall be declared first. A virtual layer, i.e. a layer that is not created by a manufacturing process, shall be declared last.

9.16 Annotations related to a LAYER declaration

Add lead-in text

9.16.1 LAYER reference annotation

A *layer reference* annotation shall be defined as shown in .

```
SEMANTICS LAYER = annotation {
  VALUETYPE = identifier;
  CONTEXT { arithmetic_model PATTERN ARRAY }
  REFERENCETYPE = LAYER;
}
```

Semantics 62—LAYER reference annotation

The purpose of a layer reference annotation is to establish an association between a layer and a *pattern* (see Section 9.28), an *array* (see Section 9.26) or an *arithmetic model* (see Section 11.3).

9.16.2 LAYERTYPE annotation

A *layertype* annotation shall be defined as shown in Semantics 63.

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

Semantics 63—LAYERTYPE annotation

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

Table 66—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.

9.16.3 PITCH annotation

A *pitch* annotation shall be defined as shown in Semantics 64.

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

Semantics 64—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.

9.16.4 PREFERENCE annotation

A *preference* annotation shall be defined as shown in Semantics 65.

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

Semantics 65—PREFERENCE annotation

The purpose of the preference annotation is to specify the prefered routing direction for a routing segment on a layer with *layertype* value *routing* (see Section 9.16.2).

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

Table 67—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.

9.17 VIA declaration

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

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

Syntax 56—VIA declaration

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

9.18 Annotations related to a VIA declaration

Add lead-in text

9.18.1 VIA reference annotation

A *via reference* annotation shall be defined as shown in .

```
SEMANTICS VIA = annotation {
  VALUETYPE = identifier;
  CONTEXT = arithmetic_model;
  REFERENCETYPE = VIA;
}
```

Semantics 66—VIA reference annotation

The purpose of a via reference annotation is to establish an association between a via and an *arithmetic model* (see Section 11.3).

9.18.2 VIATYPE annotation

Single subheader

A *viatype* annotation shall be defined as shown in Semantics 67.

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

Semantics 67—VIATYPE annotation

The *values* shall have the semantic meaning shown in Table 68.

Table 68—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.

9.19 RULE declaration

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

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

Syntax 57—RULE statement

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 9.28), a *region* (see Section 9.30), or a *via instantiation* (see Section 10.20). The electrical or physical contraint shall be described as arithmetic model (see Section 11.3).

9.20 ANTENNA declaration

An *antenna* shall be declared as shown in Syntax 58.

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

Syntax 58—ANTENNA declaration

An antenna declaration shall be used to define manufacturability constraints involving physical objects or *regions* (see Section 9.30), wherein the regions are created by physical objects. The physical objects shall be associated with a *layer* (see Section 9.15). Within the context of an antenna declaration, arithmetic models for *size* (see Section 11.19.1), *area* (see Section 11.19.2), *perimeter* (see Section 11.19.3) associated with a layer or with a region can be described. The arithmetic models can be combined, based on electrical *connectivity* (see Section 11.18.1) 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 considered. 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 9.

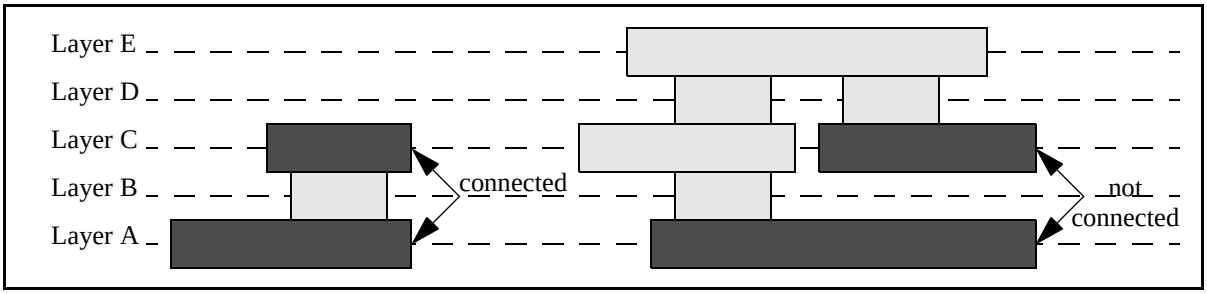


Figure 9—Connection between layers during manufacturing

The dark objects on layer A and layer C on the left side of figure 9 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 9 are not considered connected, because the connection involves layer D and E which do not yet exist when layer C is manufactured.

9.21 BLOCKAGE declaration

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

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 9.19). A rule within the context of a blockage shall only be applicable for a phys-

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

Syntax 59—BLOCKAGE statement

ical object within the blockage in relation to its environment. A physical object within the blockage can also be subjected to a more general rule, i.e. a rule that is declared outside the context of the blockage.

9.22 PORT declaration

A *port* shall be declared as shown in Syntax 60.

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

Syntax 60—PORT declaration

A port declaration shall be used in context of a *scalar pin* (see Section 9.5) 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 9.19). 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.

9.23 Annotations related to a PORT declaration

Add lead-in text

9.23.1 CONNECT_TYPE annotation

Single subheader

A *connect_type* annotation shall be defined as shown in Semantics 68.

```
KEYWORD CONNECT_TYPE = single_value_annotation {
  CONTEXT = PORT;
  VALUETYPE = identifier;
  VALUES { external internal }
  DEFAULT = external;
}
```

Semantics 68—PORT_VIEW annotation

The values shall have the semantic meaning shown in Table 69.

Table 69—CONNECT_TYPE annotation values

Annotation value	Description
`external`	A physical port of a block available for external connection
`internal`	A physical port inside a block

9.24 SITE declaration

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

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

Syntax 61—SITE declaration

A site declaration shall be used to specify a legal placement location for a *cell* (see Section 9.3).

9.25 Annotations related to a SITE declaration

Add lead-in text

9.25.1 SITE reference annotation

A *site* reference annotation shall be defined as shown in Semantics 69.

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

Semantics 69—SITE reference annotation

The purpose of a site reference annotation is to establish an association between a site and a *cell* (see Section 9.3) or an *array* (see Section 9.26). A cell or an array can inherit a site reference annotation from a *class* (see Section 8.6).

9.25.2 ORIENTATION_CLASS annotation

An *orientation class* annotation shall be defined as shown in Semantics 70.

```
KEYWORD ORIENTATION_CLASS = annotation {
  CONTEXT { SITE CELL }
  VALUETYPE = identifier;
  REFERENCETYPE = CLASS;
}
```

Semantics 70—ORIENTATION_CLASS annotation

The purpose of the orientation class annotation is to specify a legal placement orientation for a *cell* (see Section 9.3) on a site. The annotation value shall be the name of a declared *class* (see Section 8.6). The declared class can contain a *geometric transformation* statement (see Section 10.18). The geometric transformation shall indicate a transformation of coordinates from the cell as a standalone object to the cell placed on a site. The standalone cell is considered as the original object, whereas the cell placed on a site is the transformed object.

A cell can only be placed on a site, if a matching orientation class annotation value is found within both the cell declaration and the site declaration.

9.25.3 SYMMETRY_CLASS annotation

A *symmetry class* annotation shall be defined as shown in Semantics 71.

```
KEYWORD SYMMETRY_CLASS = multi_value_annotation {
  CONTEXT = SITE;
  VALUETYPE = identifier;
  REFERENCETYPE = CLASS;
}
```

Semantics 71—SYMMETRY_CLASS annotation

The purpose of the symmetry class annotation is to specify a symmetry between legal placement orientations of a cell (see Section 9.3) on a site.

A legal orientation is specified by the *orientation class* annnotation (see Section 9.25.2). If there is a set of common legal orientations for both cell and site with symmetry, the cell can be placed on the site using any orientation within that set.

Example

The site has legal orientations A and B. The cell has legal orientations A and B.

Case 1: A and B are not symmetrical.

```
CLASS A { PURPOSE = ORIENTATION_CLASS; }
CLASS B { PURPOSE = ORIENTATION_CLASS; }
```

```
SITE mySite { ORIENTATION_CLASS { A B } }
CELL myCell { ORIENTATION_CLASS { A 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: A and B are symmetrical.

```
CLASS A { PURPOSE { ORIENTATION_CLASS SYMMETRY_CLASS } }
CLASS B { PURPOSE { ORIENTATION_CLASS SYMMETRY_CLASS } }
SITE mySite { ORIENTATION_CLASS { A B } SYMMETRY_CLASS { A B } }
CELL myCell { ORIENTATION_CLASS { A B } }
```

When the site appears in either orientation A or B, the cell can be placed in either orientation A or B.

9.26 ARRAY declaration

An array shall be declared as shown in Syntax 62.

array ::=
ARRAY *array*_identifier **;**
| **ARRAY** *array*_identifier **{** { array_item } **}**
| *array*_template_instantiation
array_item ::=
all_purpose_item
| geometric_transformation

Syntax 62—ARRAY statement

An array declaration shall be used for the purpose to describe a grid for creating physical objects within design. A *geometric transformation* (see Section 10.18) can be used to define a transformation of coordinates from a basic constructive element of the array to an element placed within the array. The basic constructive element is considered the original object, whereas the element placed within the array is the transformed object.

9.27 Annotations related to an ARRAY declaration

Add lead-in text

9.27.1 ARRAYTYPE annotation

An *arraytype* annotation shall be defined as shown in Semantics 72.

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

Semantics 72—ARRAYTYPE annotation

The *values* shall have the semantic meaning shown in Table 70.

Table 70—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.* 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*. 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 detailed routing.

9.27.2 LAYER reference annotation for ARRAY

A *layer* reference annotation in the context of an *array* shall be defined as shown in Semantics 73.

```
SEMANTICS ARRAY.LAYER = multi_value_annotation;
```

Semantics 73—LAYER reference annotation for ARRAY

The layer reference annotation shall be applicable for an array with *arraytype* value *detailed routing* (see Section 9.27.1). It shall specify a *layer* (see Section 9.15) with *layertype* value *routing* (see Section 9.16.2).

9.27.3 SITE reference annotation for ARRAY

A *site* reference annotation in the context of an *array* shall be defined as shown in Semantics 72.

```
SEMANTICS ARRAY.SITE = single_value_annotation;
```

Semantics 74—SITE reference annotation

The purpose of a site reference annotation in the context of an array is to specify the basic element from which the array is constructed.

The site reference annotation is applicable for an array with *arraytype* value *floorplan* or *placement* (see Section 9.27.1).

9.28 PATTERN declaration

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

The purpose of a pattern declaration is the description of a geometry formed by a physical object.

```
pattern ::=
    PATTERN pattern_identifier ;
  | PATTERN pattern_identifier { { pattern_item } }
  | pattern_template_instantiation
pattern_item ::=
    all_purpose_item
  | geometric_model
  | geometric_transformation
```

Syntax 63—PATTERN declaration

9.29 Annotations related to a PATTERN declaration

Add lead-in text

9.29.1 PATTERN reference annotation

A *pattern* reference annotation shall be defined as shown in .

```
SEMANTICS PATTERN = annotation {
  VALUETYPE = identifier ;
  CONTEXT = arithmetic_model ;
  REFERENCETYPE = PATTERN ;
}
```

Semantics 75—PATTERN reference annotation

The purpose of a pattern reference annotation is to establish an association between a pattern and an *arithmetic model* (see Section 11.3).

9.29.2 SHAPE annotation

A *shape* annotation shall be defined as shown in Semantics 76.

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

Semantics 76—SHAPE annotation

The shape annotation applies for a pattern associated with a layer with *layertype* value *routing* (see Section 9.16.2).

The *values* shall have the semantic meaning shown in Table 71.

Table 71—SHAPE annotation values

Annotation value	Description
`line`	A routing segment in prefered routing direction. Each end is connected with a via or with another routing segment.
`jog`	A routing segment in non-prefered routing direction. Each end is connected with a routing segment in prefered routing direction.
`tee`	An intersection point between two orthogonal routing segments. One of the routing segments ends at the intersection.
`cross`	An intersection point between two orthogonal routing segments. Both routing segments continue beyond the intersection.
`corner`	An intersection point between two orthogonal routing segments. Both routing segments end at the intersection.
`end`	An unconnected point of an open routing segment.

The meaning of the shape annotation values is further illustrated in Figure 10.

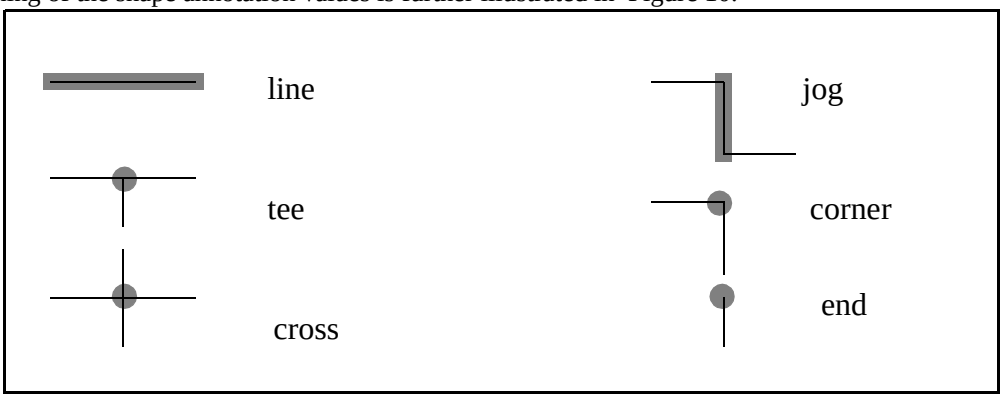


Figure 10—SHAPE annotation illustration

The *shape* annotation specifies whether a *pattern* is represented by a point or by a line. A pattern with shape annotation value *line* or *jog* is represented by a line. A pattern with shape annotation value *tee*, *cross*, *corner* or *end* is represented by a point.

9.29.3 VERTEX annotation

A *vertex* annotation shall be defined as shown in Semantics 77.

The vertex annotation applies for a pattern in conjunction with *shape* annotation value *tee*, *cross*, *corner*, or *end* (see Section 9.29.2).

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

Semantics 77—VERTEX annotation

The *values* shall have the semantic meaning shown in Table 72.

Table 72—VERTEX annotation values

Annotation value	Description
`angular`	The angle between intersecting routing segments shall be preserved.
`round`	The angle between intersecting routing segments shall be rounded.

The meaning of the vertex annotation values is further illustrated in Figure 11.

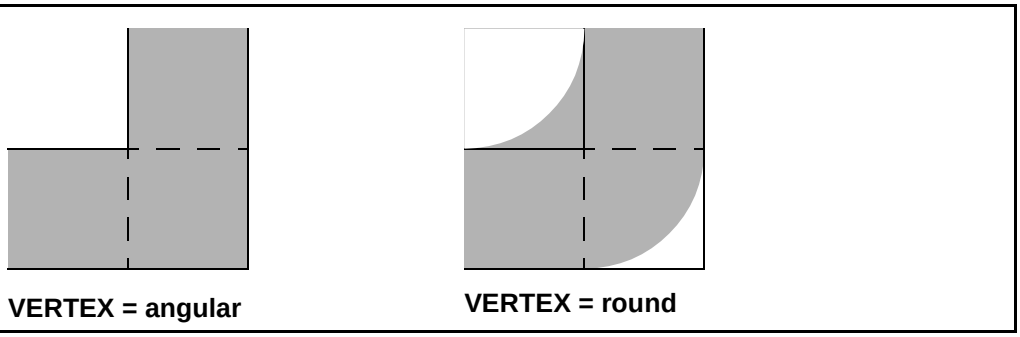


Figure 11—VERTEX annotation illustration

9.29.4 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 78—ROUTE annotation

The route annotation applies for a pattern with shape annotation value *line*, *jog*, or *tee* (see Section 9.29.2).

The purpose of a route annotation is to specify the actual routing direction for the pattern. This is illustrated in Figure 12..

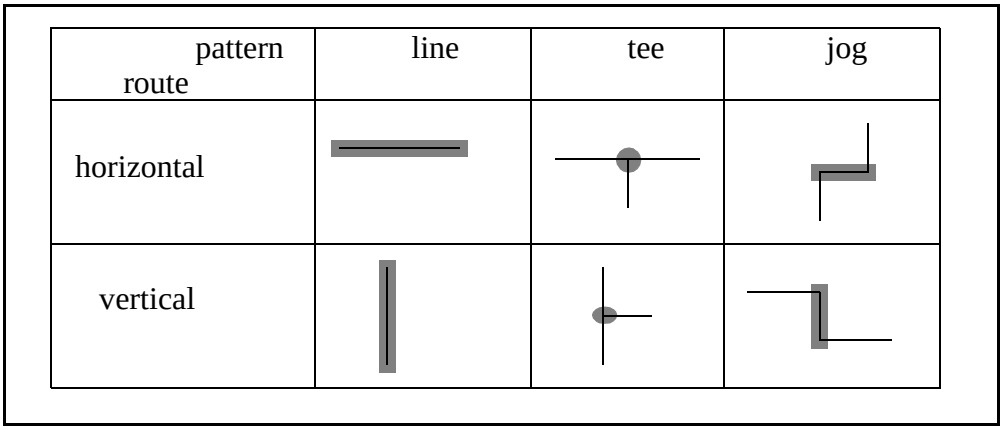


Figure 12—ROUTE annotation illustration

If the route annotation does not appear and a layer reference annotation (see Section 9.29.5) appears, the prefered routing direction specified by the *preference* annotation (see Section 9.16.4) within the layer declaration shall apply to infer the actual routing direction. If both route annotation and layer reference annotation appear, the route annotation shall take precedence.

9.29.5 LAYER reference annotation for PATTERN

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

```
SEMANTICS PATTERN.LAYER = single_value_annotation;
```

Semantics 79—LAYER reference annotation for PATTERN

The purpose of a layer reference annotation in the context of a pattern is to establish an association between a pattern and a *layer* (see Section 9.15). The physical object represented by the pattern shall reside on a layer. A pattern declaration without layer reference annotation shall be considered incomplete.

9.30 REGION declaration

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

```
region ::=
      REGION region_name_identifier ;
    | REGION region_name_identifier { { region_item } }
    | region_template_instantiation
region_item ::=
      all_purpose_item
    | geometric_model
    | geometric_transformation
    | BOOLEAN_single_value_annotation
```

Syntax 64—REGION declaration

The purpose of a region declaration is the description of a geometry. The geometry can be formed by intersection or union of physical objects. The geometry can also be described in abstract mathematical terms without being associated with a particular physical object.

The specification of geometries by one or more *geometric models* (see Section 10.16) and/or by a *boolean* annotation (see Section 9.31.2) shall be additive, i.e., the region shall be considered the union of the specified geometries. If a *geometric transformation* (see Section 10.18) is present, it shall apply to all specified geometries within the region.

9.31 Annotations related to a REGION declaration

9.31.1 REGION reference annotation

A *region* reference annotation shall be defined as shown in .

```
SEMANTICS REGION = annotation {
  VALUETYPE = identifier ;
  CONTEXT = arithmetic_model ;
  REFERENCETYPE = REGION ;
}
```

Semantics 80—PATTERN reference annotation

The purpose of a region reference annotation is to establish an association between a region and an *arithmetic model* (see Section 11.3).

9.31.2 BOOLEAN annotation

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

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

Semantics 81—BOOLEAN annotation

The purpose of the boolean annotation is to specify a region by a boolean operation (see Section 10.11). The name of a *pattern* (see Section 9.28) or the name of another region shall be considered a legal operand. The operators specified in Section 10.11.1, Table 78 and Section 10.11.2, Table 80 shall be considered legal operators.

10. Description of functional and physical implementation

Add lead-in text

10.1 FUNCTION statement

A *function* statement shall be defined as shown in Syntax 65.

```
function ::=
      FUNCTION { function_item { function_item } }
    | function_template_instantiation
function_item ::=
      all_purpose_item
    | behavior
    | structure
    | statetable
```

Syntax 65—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 66.

```
test ::=
      TEST { test_item { test_item } }
    | test_template_instantiation
test_item ::=
      all_purpose_item
    | behavior
    | statetable
```

Syntax 66—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 a pin variable

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 73.

Table 73—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 61, “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 13. 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).

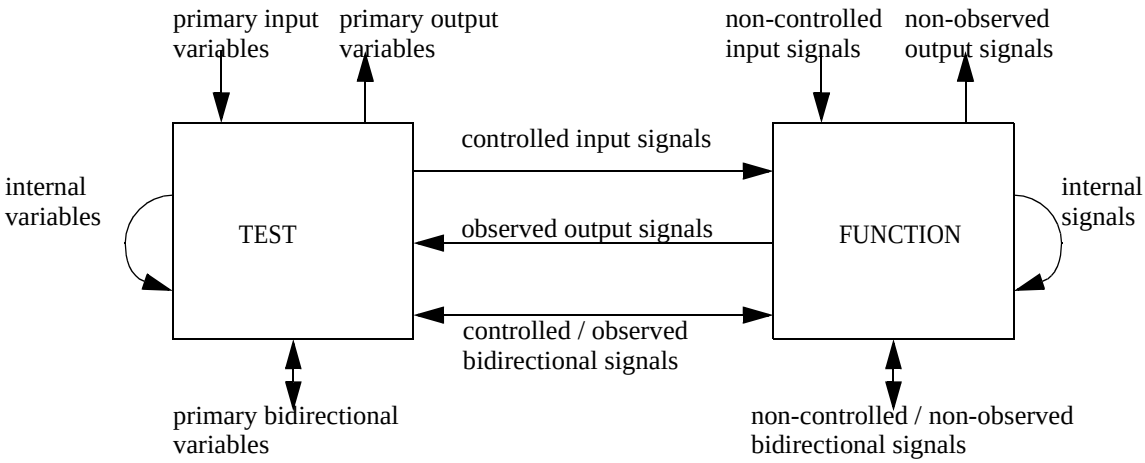


Figure 13—Relationship between FUNCTION and TEST

10.4 BEHAVIOR statement

A *behavior* statement shall be defined as shown in Syntax 67.

```
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 67—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 *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.

A *primitive instantiation* establishes a reference to a predefined function statement within a *primitive* declaration (see Section 9.8). 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 set of predefined primitve declarations is specified in Section 10.14.

10.5 STRUCTURE statement and CELL instantiation

A *structure* statement shall be defined as shown in Syntax 68.

```
structure ::=
      STRUCTURE { cell_instantiation { cell_instantiation } }
    | structure_template_instantiation
cell_instantiation ::=
      cell_reference_identifier cell_instance_identifier ;
    | cell_reference_identifier cell_instance_identifier { { cell_instance_pin_value } }
    | cell_reference_identifier cell_instance_identifier { { cell_instance_pin_assignment } }
    | cell_instantiation_template_instantiation
cell_instance_pin_assignment ::=
    cell_reference_pin_variable = cell_instance_pin_value ;
```

Syntax 68—STRUCTURE statement

The purpose of a structure statement is to specify a structural implementation of a compound cell, i.e., a netlist. A complete or a partial netlist can be specified. A component of a netlist can be a cell or a primitive. A structure statement shall not substitute a behavior statement or a statetable statement.

A *cell instantiation* shall specify the mapping between a cell reference and a cell instance within the structure statement. The mapping shall be established either by order or by name.

In case of mapping by order, a *pin value* (see Section 7.9) shall be associated with the cell instance. A corresponding pin variable associated with the cell reference shall be infered by the order of pin declarations within the cell reference.

If mapping by order is not possible without ambiguity, mapping shall be established by name, using *pin assignment* (see Section 7.10). The left-hand side of the pin assignment shall represent a pin variable associated with the cell reference. The right-hand side of the pin assignment shall represent a pin value associated with the cell instance.

10.6 STATETABLE statement

A *statetable* statement shall be defined as shown in Syntax 69.

```
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 69—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 70.

```
non_scan_cell ::=
    NON_SCAN_CELL = non_scan_cell_reference
  | NON_SCAN_CELL { non_scan_cell_reference { non_scan_cell_reference } }
  | non_scan_cell_template_instantiation
non_scan_cell_reference ::=
    non_scan_cell_identifier {  { scan_cell_pin_identifier }  }
  | non_scan_cell_identifier {  { non_scan_cell_pin_identifier = scan_cell_pin_identifier ; }  }
```

Syntax 70—NON_SCAN_CELL statement

A non-scan cell statement applies for a scan cell. A scan cell is a cell with extra pins for testing purpose. The *non-scan cell reference* 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 *non-scan cell identifier.*

The pin mapping is given either by order or by name. In case of pin mapping by order, 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 case of pin mapping by name, the pin names of the non-scan cell shall appear at the left-hand side, and the pin names of the scan cell shall appear at the right-hand side.

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 71.

range ::=
RANGE **{** index_value **:** index_value **}**

Syntax 71—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:

$$0 \le A \le 2^B - 1$$

$$B = \begin{cases} 1 + \mathrm{LSB} - \mathrm{MSB} & \text{if}(\mathrm{LSB} > \mathrm{MSB}) \\ 1 + \mathrm{MSB} - \mathrm{LSB} & \text{if}(\mathrm{LSB} \le \mathrm{MSB}) \end{cases}$$

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 72.

```
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 72—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 74.

Table 74—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 75.

Table 75—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
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 76.

Table 76—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

Table 76—Mapping between hexadecimal base and binary base (Continued)

Hexadecimal	Binary (bit literal)	Numerical value
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 9.5) 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:

$$p = \begin{cases} \text{LSB} - i & \text{if}(\text{LSB} > \text{MSB}) \\ i - \text{LSB} & \text{if}(\text{LSB} \leq \text{MSB}) \end{cases}$$

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 77.

Table 77—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 78

Table 78—Logical Operation

Operator	Description
!	logical *inversion*
&&	logical *and*
||	logical *or*

A boolean expression involving a logical *inversion*, *and*, *or* (see Table 78), *nand*, *nor*, *exor*, *exnor* (see Table 79) 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 79

Table 79—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 80

Table 80—Conditional Operation

Symbol	Description
?	operator for a condition
:	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 81.

Table 81—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 9.7.10) 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 82

Table 82—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 83

Table 83—Comparison Operation

Operator	Description
==	equal
!=	non equal
>	greater
<	less

Table 83—Comparison Operation

Operator	Description
>=	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 9.7.10) 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 84

Table 84—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 85

Table 85—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 73.

```
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_expression_macro
vector_unary ::=
    edge_literal
vector_binary ::=
      & | && | | | || | -> | ~> | <-> | <~> | &> | <&>
control_and ::=
    & | &&
```

Syntax 73—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 86.

Table 86—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

Table 86—Specification of a single event

row	edge literal	event on operand
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 14.

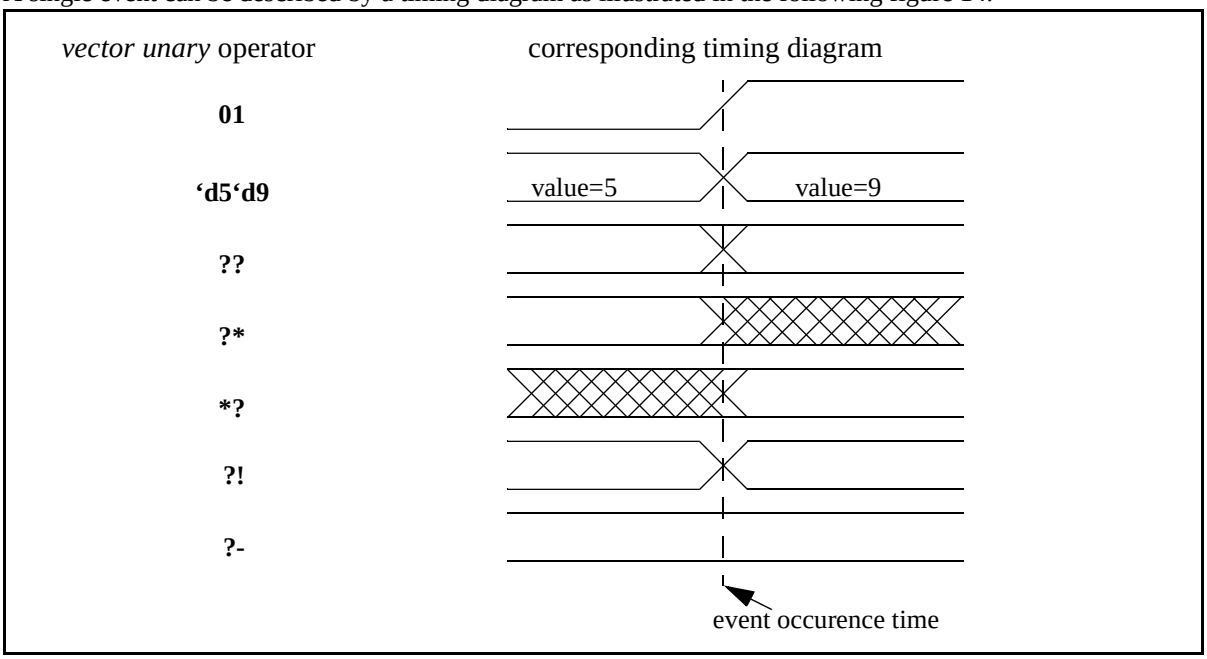


Figure 14—Timing diagrams for single events

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 14 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 within an event sequence

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 9.7.18) within a pin declaration or by the *monitor* annotation (see Section 9.14.10) 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.

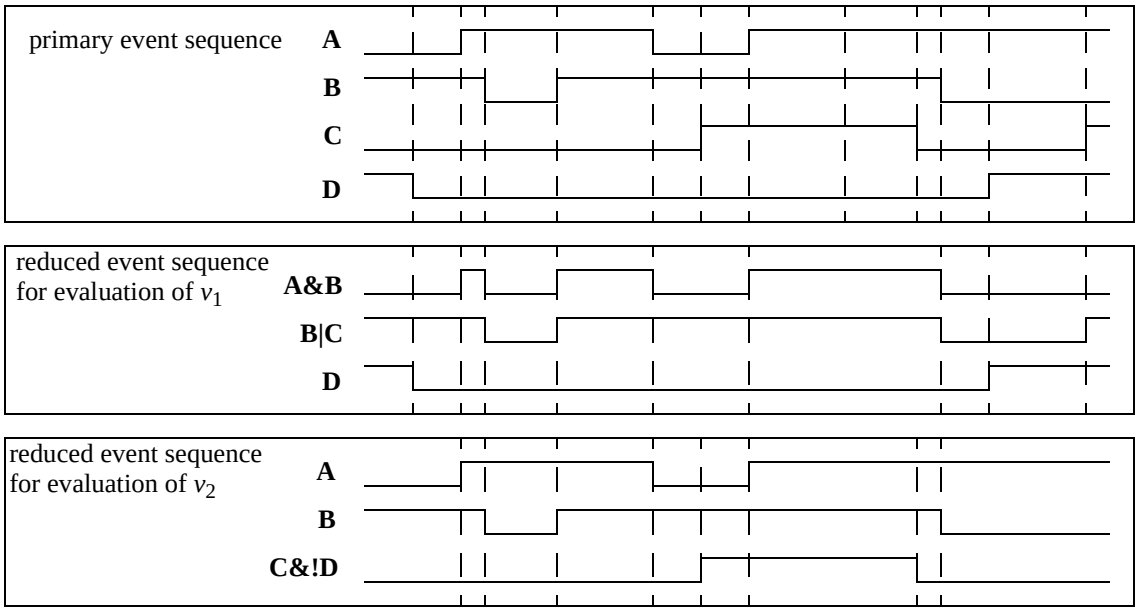
b) The single events on pin variables involved in the vector expression shall be reduced to single events on boolean expressions wherein the pin variables are involved. Other single events on these pin variables shall be disregarded. The single events on pin variables not involved in the vector expression 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 a sequence of events

The operators in the following Table 87 shall be used for a canonical specification of an event sequence.

Table 87—Canonical specification of an event

symbol	operator name	explanation
->	immediately followed by	LHS event occurs before RHS event, no event can occur in-between
~>	eventually followed by	LHS event occurs before RHS event, 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, 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
$(v^{M}_{1} \text{ -> } v^{N}_{1})$ **&** $(v^{M}_{2} \text{ -> } v^{N}_{2}) = (v^{M}_{1}$ **&** $v^{M}_{2}) \text{ -> } (v^{N}_{1}$ **&** $v^{N}_{2})$

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
$(v^{M}_{1} \text{ -> } v^{N}_{1})$ **&** $v^{N}_{2} = v^{M}_{1} \text{ -> } (v^{N}_{1}$ **&** $v^{N}_{2})$

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 -> v_2 -> v_3 = (v_1 -> v_2) -> v_3 = v_1 -> (v_2 -> v_3)

d) Associativity for *eventually followed by* operator
v_1 ~> v_2 ~> v_3 = (v_1 ~> v_2) ~> v_3 = v_1 ~> (v_2 ~> v_3)

e) Mixed associativity for *immediately followed by* and *eventually followed by* operator
v_1 -> v_2 ~> v_3 = (v_1 -> v_2) ~> v_3 = v_1 -> (v_2 ~> v_3)
v_1 ~> v_2 -> v_3 = (v_1 ~> v_2) -> v_3 = v_1 ~> (v_2 -> v_3)

f) Assocativity for *simultaneous occurence* operator
v_1 **&** v_2 **&** v_3 = (v_1 **&** v_2) **&** v_3 = v_1 **&** (v_2 **&** v_3)

g) Commutativity for *simultaneous occurence* operator
v_1 **&** v_2 = v_2 **&** v_1

h) Reduction rule for *simultaneous occurence* operator
v_1 **&** v_1 = v_1

i) Assocativity for *alternative occurence* operator
v_1 | v_2 | v_3 = (v_1 | v_2) | v_3 = v_1 | (v_2 | v_3)

j) Commutativity for *alternative occurence* operator
v_1 | v_2 = v_2 | v_1

k) Reduction rule for *alternative occurence* operator
v_1 | v_1 = v_1

l) Distributivity between *immediately followed by* operator and *alternative occurence* operator
(v_1 | v_2) -> v_3 = (v_1 -> v_3) | (v_2 -> v_3)
v_1 -> (v_2 | v_3) = (v_1 -> v_2) | (v_1 -> v_3)

m) Distributivity between *eventually followed by* operator and *alternative occurence* operator
(v_1 | v_2) ~> v_3 = (v_1 ~> v_3) | (v_2 ~> v_3)
v_1 ~> (v_2 | v_3) = (v_1 ~> v_2) | (v_1 ~> v_3)

n) Distributivity between *simultaneous occurence* operator and *alternative occurence* operator
(v_1 | v_2) **&** v_3 = (v_1 **&** v_3) | (v_2 **&** v_3)

The *closely followed by* operator shall be mathematically defined as follows:

o) v_1 **&>** v_2 = (v_1 **&** v_2) | (v_1 -> 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”.

$$v_j^p = v_{j(1)} \ldots \mathrm{op}_{j(i)} v_{j(i)} \ldots \mathrm{op}_{j(m)} v_{j(m)} = \prod_{i=1}^{m_j} \mathrm{op}_{j(i)} v_{j(i)}$$

$\mathrm{op}_{j(i)}$ = -> | ~> | **&**

$$v^s = v_1^p \ldots | \ldots v_j^p \ldots | \ldots v_n^p = \sum_{j=1}^{n} v_j^p$$

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 $\mathrm{op}_{j(i)}$, 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 $\mathrm{op}_{j(1)}$ is irrelevant when converting the mathematical representation into a vector expression.

Example:

$$v_1^p = \prod_{i=1}^{m_1} \mathrm{op}_{1(i)} v_{1(i)} \qquad m_1 = \mathbf{3} \qquad \mathrm{op}_{1(1)} = \mathbf{nil} \quad \mathrm{op}_{1(2)} = \mathbf{->} \quad \mathrm{op}_{1(3)} = \mathbf{->}$$
$$v_{1(1)} = \mathbf{(01\ A)} \quad v_{1(2)} = \mathbf{(10\ A)} \quad v_{1(3)} = \mathbf{(10\ B)}$$

$$v_2^p = \prod_{i=1}^{m_2} \mathrm{op}_{2(i)} v_{2(i)} \qquad m_2 = \mathbf{3} \qquad \mathrm{op}_{2(1)} = \mathbf{nil} \quad \mathrm{op}_{2(2)} = \mathbf{->} \quad \mathrm{op}_{2(3)} = \mathbf{->}$$
$$v_{2(1)} = \mathbf{(01\ B)} \quad v_{2(2)} = \mathbf{(10\ B)} \quad v_{2(3)} = \mathbf{(10\ A)}$$

$$v^s = \sum_{j=1}^{2} v_j^p = (01\ \mathrm{A}) \text{ -> } (10\ \mathrm{A}) \text{ -> } (10\ \mathrm{B}) \mid (01\ \mathrm{B}) \text{ -> } (10\ \mathrm{B}) \text{ - > } (10\ \mathrm{A})$$

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 88.

Table 88—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 or RHS event immediately followed by LHS event
<~>	permutation of events eventually followed by each other	LHS event eventually followed by RHS event or RHS event eventually followed by LHS event
<&>	permutation of events closely followed by each other	LHS event immediately followed by RHS event or RHS event eventually followed by LHS event or 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 88.

Table 89—Specification a conditional event

symbol	operator name	comment
&& or **&**	control-and operator	overloaded symbol, also used for *logical and* (see Table 78) and *bitwise and* (Table 79)
?	condition operator	see also Table 80
:	delimiter between alternatives	see also Table 80

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.

10.14 Predefined PRIMITIVE

This section defines the predefined primitive declarations, wherein the prefix “ALF_” is reserved for the name of such primitives.

10.14.1 Predefined PRIMITIVE ALF_BUF

The primitive *ALF_BUF* shall be defined as shown in .

```
PRIMITIVE ALF_BUF {
  PIN in { DIRECTION = input; }
  PIN [1:<bitwidth>] out { DIRECTION = output; }
  GROUP index { 1 : <bitwidth> }
  FUNCTION { BEHAVIOR { out[index] = in ; } }
}
```

Semantics 82—Predefined PRIMITIVE ALF_BUF

10.14.2 Predefined PRIMITIVE ALF_NOT

The primitive *ALF_NOT* shall be defined as shown in .

```
PRIMITIVE ALF_NOT {
  PIN in { DIRECTION = input; }
  PIN [1:<bitwidth>] out { DIRECTION = output; }
  GROUP index { 1 : <bitwidth> }
  FUNCTION { BEHAVIOR { out[index] = ! in ; } }
}
```

Semantics 83—Predefined PRIMITIVE ALF_NOT

10.14.3 Predefined PRIMITIVE ALF_AND

The primitive *ALF_AND* shall be defined as shown in .

```
PRIMITIVE ALF_AND {
  PIN out { DIRECTION = output; }
  PIN [1:<bitwidth>] in { DIRECTION = input; }
  FUNCTION { BEHAVIOR { out = & in ; } }
}
```

Semantics 84—Predefined PRIMITIVE ALF_AND

10.14.4 Predefined PRIMITIVE ALF_NAND

The primitive *ALF_NAND* shall be defined as shown in .

10.14.5 Predefined PRIMITIVE ALF_OR

The primitive *ALF_OR* shall be defined as shown in .

```
PRIMITIVE ALF_NAND {
  PIN out { DIRECTION = output; }
  PIN [1:<bitwidth>] in { DIRECTION = input; }
  FUNCTION { BEHAVIOR { out = ~& in ; } }
}
```

Semantics 85—Predefined PRIMITIVE ALF_NAND

```
PRIMITIVE ALF_OR {
  PIN out { DIRECTION = output; }
  PIN [1:<bitwidth>] in { DIRECTION = input; }
  FUNCTION { BEHAVIOR { out = | in ; } }
}
```

Semantics 86—Predefined PRIMITIVE ALF_OR

10.14.6 Predefined PRIMITIVE ALF_NOR

The primitive *ALF_NOR* shall be defined as shown in .

```
PRIMITIVE ALF_NOR {
  PIN out { DIRECTION = output; }
  PIN [1:<bitwidth>] in { DIRECTION = input; }
  FUNCTION { BEHAVIOR { out = ~| in ; } }
}
```

Semantics 87—Predefined PRIMITIVE ALF_NOR

10.14.7 Predefined PRIMITIVE ALF_XOR

The primitive *ALF_XOR* shall be defined as shown in .

```
PRIMITIVE ALF_XOR {
  PIN out { DIRECTION = output; }
  PIN [1:<bitwidth>] in { DIRECTION = input; }
  FUNCTION { BEHAVIOR { out = ^ in ; } }
}
```

Semantics 88—Predefined PRIMITIVE ALF_XOR

10.14.8 Predefined PRIMITIVE ALF_XNOR

The primitive *ALF_XNOR* shall be defined as shown in .

10.14.9 Predefined PRIMITIVE ALF_BUFIF1

The primitive *ALF_BUFIF1* shall be defined as shown in .

10.14.10 Predefined PRIMITIVE ALF_BUFIF0

The primitive *ALF_BUFIF0* shall be defined as shown in .

```
PRIMITIVE ALF_XNOR {
  PIN out { DIRECTION = output; }
  PIN [1:<bitwidth>] in { DIRECTION = input; }
  FUNCTION { BEHAVIOR { out = ~^ in ; } }
}
```

Semantics 89—Predefined PRIMITIVE ALF_XNOR

```
PRIMITIVE ALF_BUFIF1 {
  PIN out { DIRECTION = output; }
  PIN in  { DIRECTION = input; }
  PIN enable { DIRECTION = input; }
  FUNCTION { BEHAVIOR { out = (enable)? in : ‘bZ ; } }
}
```

Semantics 90—Predefined PRIMITIVE ALF_BUFIF1

```
PRIMITIVE ALF_BUFIF0 {
  PIN out { DIRECTION = output; }
  PIN in  { DIRECTION = input; }
  PIN enable { DIRECTION = input; }
  FUNCTION { BEHAVIOR { out = (! enable)? in : ‘bZ ; } }
}
```

Semantics 91—Predefined PRIMITIVE ALF_BUFIF0

10.14.11 Predefined PRIMITIVE ALF_NOTIF1

The primitive *ALF_NOTIF1* shall be defined as shown in .

```
PRIMITIVE ALF_NOTIF1 {
  PIN out { DIRECTION = output; }
  PIN in  { DIRECTION = input; }
  PIN enable { DIRECTION = input; }
  FUNCTION { BEHAVIOR { out = (enable)? ! in : ‘bZ ; } }
}
```

Semantics 92—Predefined PRIMITIVE ALF_NOTIF1

10.14.12 Predefined PRIMITIVE ALF_NOTFIF0

The primitive *ALF_NOTIF0* shall be defined as shown in .

```
PRIMITIVE ALF_NOTIF0 {
  PIN out { DIRECTION = output; }
  PIN in  { DIRECTION = input; }
  PIN enable { DIRECTION = input; }
  FUNCTION { BEHAVIOR { out = (! enable)? ! in : ‘bZ ; } }
}
```

Semantics 93—Predefined PRIMITIVE ALF_NOTIF0

10.14.13 Predefined PRIMITIVE ALF_MUX

The primitive *ALF_MUX* shall be defined as shown in .

```
PRIMITIVE ALF_MUX {
  PIN Q  { DIRECTION = output; }
  PIN [1:0] D { DIRECTION = input; }
  PIN S { DIRECTION = input; }
  FUNCTION {
    BEHAVIOR {
      Q = ! S & D[0] | S & D[1] | D[0] & D[1] ;
    }
  }
}
```

Semantics 94—Predefined PRIMITIVE ALF_MUX

10.14.14 Predefined PRIMITIVE ALF_LATCH

The primitive *ALF_LATCH* shall be defined as shown in .

```
PRIMITIVE ALF_LATCH {
  PIN Q  { DIRECTION = output; }
  PIN QN { DIRECTION = output; }
  PIN D  { DIRECTION = input; }
  PIN ENABLE { DIRECTION = input; }
  PIN CLEAR  { DIRECTION = input; }
  PIN SET    { DIRECTION = input; }
  PIN Q_CONFLICT  { DIRECTION = input; }
  PIN QN_CONFLICT { DIRECTION = input; }
  FUNCTION {
    BEHAVIOR {
      @ ( CLEAR && SET ) {
        Q = Q_CONFLICT ; QN = QN_CONFLICT ;
      } : ( CLEAR ) {
        Q = 0 ; QN = 1 ;
     } : ( SET ) {
        Q = 1 ; QN = 0 ;
      } : ( ENABLE ) {
        Q = D ; QN = ! D ;
      }
    }
  }
}
```

Semantics 95—Predefined PRIMITIVE ALF_LATCH

10.14.15 Predefined PRIMITIVE ALF_FLIPFLOP

The primitive *ALF_FLIPFLOP* shall be defined as shown in .

```
PRIMITIVE ALF_FLIPFLOP {
  PIN Q  { DIRECTION = output; }
  PIN QN { DIRECTION = output; }
  PIN D  { DIRECTION = input; }
  PIN CLOCK  { DIRECTION = input; }
  PIN CLEAR  { DIRECTION = input; }
  PIN SET    { DIRECTION = input; }
  PIN Q_CONFLICT  { DIRECTION = input; }
  PIN QN_CONFLICT { DIRECTION = input; }
  FUNCTION {
    BEHAVIOR {
      @ ( CLEAR && SET ) {
        Q = Q_CONFLICT ; QN = QN_CONFLICT ;
      } : ( CLEAR ) {
        Q = 0 ; QN = 1 ;
     } : ( SET ) {
        Q = 1 ; QN = 0 ;
      } : ( 01 CLOCK ) {
        Q = D ; QN = ! D ;
      }
    }
  }
}
```

Semantics 96—Predefined PRIMITIVE ALF_FLIPFLOP

10.15 WIRE instantiation

A *wire instantiation* shall be defined as shown in Syntax 74.

```
wire_instantiation ::=
    wire_reference_identifier wire_instance_identifier ;
  | wire_reference_identifier wire_instance_identifier { { wire_instance_pin_value } }
  | wire_reference_identifier wire_instance_identifier { { wire_instance_pin_assignment } }
  | wire_instantiation_template_instantiation
wire_instance_pin_assignment ::=
  wire_reference_pin_variable = wire_instance_pin_value ;
```

Syntax 74—WIRE instantiation

The purpose of a *wire instantiation* is to describe an electrical circuit for characterization or test. A reference of the electrical circuit shall be given by a wire declaration (see Section 9.9). A cell, subjected to characterization or test, can be connected with an instance of the electrical circuit.

The mapping between the wire reference and the wire instance shall be established either by order or by name.

In case of mapping by order, a *pin value* (see Section 7.9) shall be associated with the wire instance. A corresponding pin variable associated with the wire reference shall be infered by the order of node declarations within the wire reference.

If mapping by order is not possible without ambiguity, mapping shall be established by name, using *pin assignment* (see Section 7.10). The left-hand side of the pin assignment shall represent the name of a node associated

with the wire reference. The right-hand side of the pin assignment shall represent a pin value associated with the wire instance.

10.16 Geometric model

A *geometric model* shall be defined as shown in Syntax 75.

```
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 75—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 11.19.9).

The parent object of the geometric model can contain a *geometric transformation* (see Section 10.18) applicable to the geometric model.

Table 90 specifiies the meaning of predefined geometric model identifiers.

Table 90—Geometric model identifiers

Identifier	Description
`DOT`	Describes one point.
`POLYLINE`	Defined by `N>1` directly connected points, forming an open object.
`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 15.

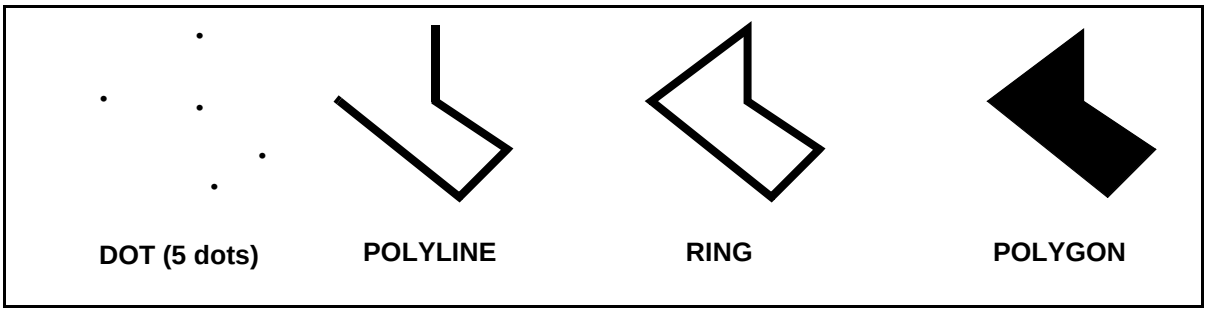


Figure 15—Illustration of geometric models

A *point_to_point* annotation shall be defined as shown in Semantics 97.

```
KEYWORD POINT_TO_POINT = single_value_annotation {
  CONTEXT { POLYLINE RING POLYGON }
  VALUETYPE = identifier;
  VALUES { direct manhattan }
  DEFAULT = direct;
}
```

Semantics 97—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 16. It specifies the shortest possible connection between points.

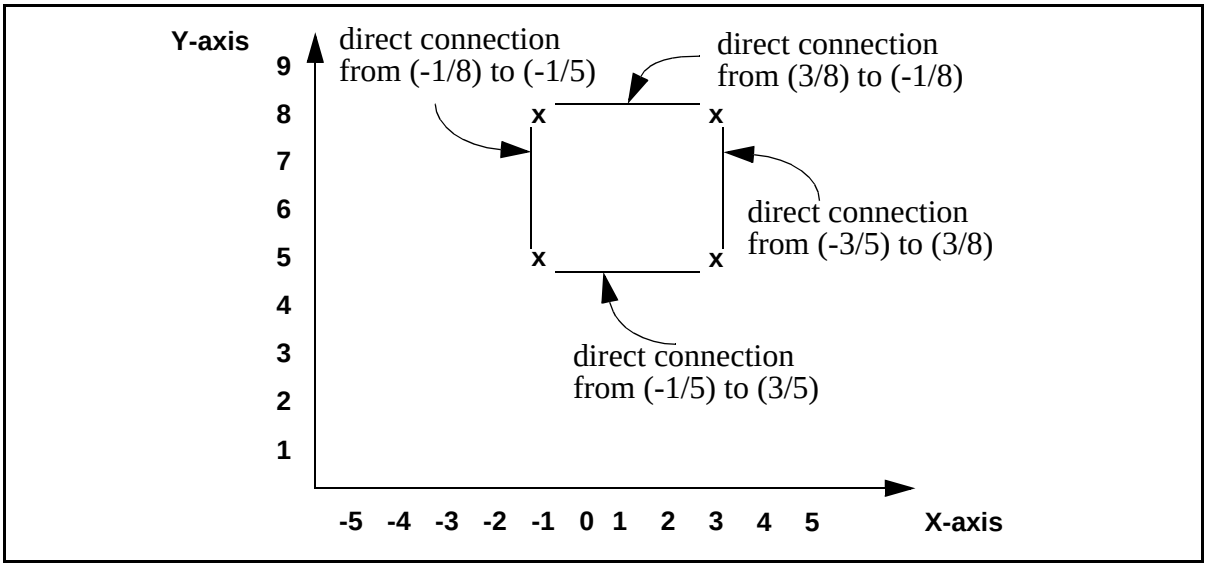


Figure 16—Illustration of direct point-to-point connection

The meaning of the annotation value *manhattan* is illustrated in Figure 17. 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.

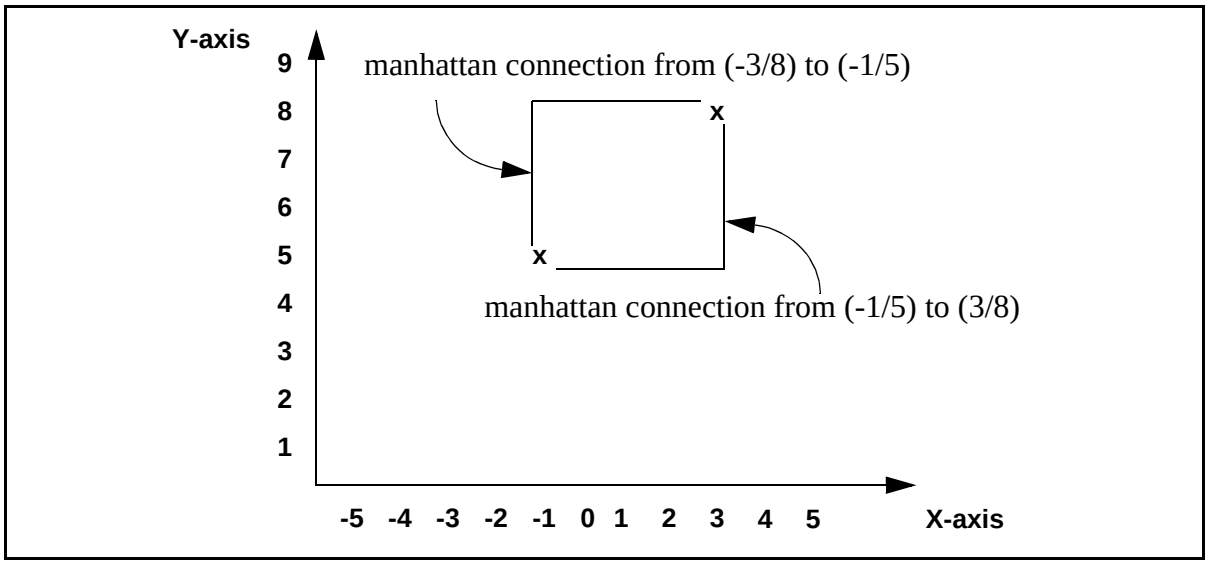


Figure 17—Illustration of manhattan point-to-point connection

Example 1

```
POLYGON {
    POINT_TO_POINT = direct;
    COORDINATES { -1 5 3 5 3 8 -1 8 }
}
```

Example 2

```
POLYGON {
    POINT_TO_POINT = manhattan;
    COORDINATES { -1 5 3 8 }
}
```

Both statements describe the same rectangle.

10.17 Predefined geometric models using TEMPLATE

A *template* declaration (see Section 8.9) can be used to describe particular geometric models. This section describes predefined geometric models.

10.17.1 Predefined TEMPLATE RECTANGLE

The template *rectangle* shall be predefined as shown in Semantics 98.

```
TEMPLATE RECTANGLE {
  POLYGON {
    POINT_TO_POINT = manhattan;
    COORDINATES { <left> <bottom> <right> <top> }
  }
}
```

Semantics 98—Predefined TEMPLATE RECTANGLE

10.17.2 Predefined TEMPLATE LINE

The template *line* shall be predefined as shown in Semantics 99.

```
TEMPLATE LINE {
  POLYLINE {
    POINT_TO_POINT = direct;
    COORDINATES { <x_start> <y_start> <x_end> <y_end> }
  }
}
```

Semantics 99—Predefined TEMPLATE LINE

10.18 Geometric transformation

A *geometric transformation* shall be defined as shown in Syntax 76.

```
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 76—Geometric transformation

A *geometric model* (see Section 10.16) shall be subjected to a *geometric transformation* if both statements appear in the same context, i.e., they have the same parent.

The following rules shall apply for the geometric transformations *shift*, *rotate* and *flip*:

— A number associated with a geometric transformation shall be measured in units of *distance* (see Section 11.19.9).
— A geometric transformation shall apply to the origin of a geometric model. Therefore, the result of subsequent transformations is independent of the order in which each individual transformation is applied.

— The direction of the transformation shall be from the geometric model to the actual object.

The *shift* statement shall define the horizontal and vertical offset measured between the coordinates within a declared geometric model and the actual coordinates of an object.

The *rotate* statement shall define the angle of rotation in degrees measured between the orientation of a defined geometric model and the actual orientation of an object. The angle shall be measured in counter-clockwise direction, specified by a number between `0` and `360`.

The *flip* statement shall define a mirror operation. The number shall represent the angle of the movement of the object in degrees. By definition, the movement is orthogonal to the mirror axis. Therefore, the number `0` specifies flip in horizontal direction, therefore the axis is vertical, whereas the number `90` specifies flip in vertical direction, therefore the axis is horizontal.

The geometric transformations *flip*, *rotate*, and *shift* are further illustrated in Figure 18.

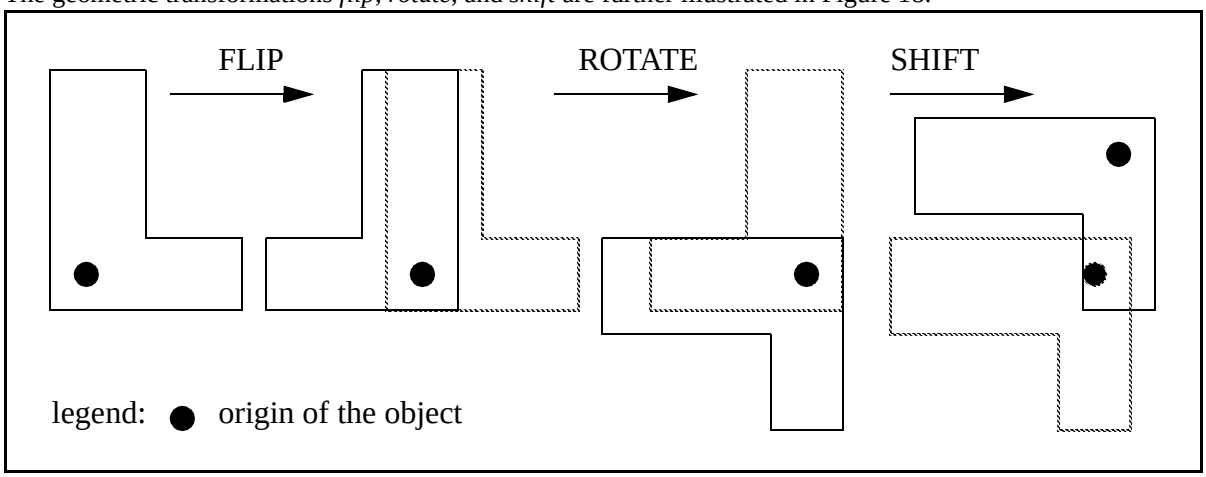


Figure 18—Illustration of FLIP, ROTATE, and SHIFT

The *repeat* statement shall describe the replication of an object. The unsigned integer shall define the total number of replications, including the original instance. Therefore, the number `1` means that the object appears once. A repeat statement without unsigned integer shall indicate an arbitrary number of replications.

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 }
    }
}
```

10.19 ARTWORK statement

An *artwork* statement shall be defined as shown in Syntax 77.

```
artwork ::=
    ARTWORK = artwork_identifier ;
  | ARTWORK = artwork_reference
  | ARTWORK { artwork_reference { artwork_reference } }
  | artwork_template_instantiation
artwork_reference ::=
    artwork_identifier { { geometric_transformation } { cell_pin_identifier } }
  | artwork__identifier
      { { geometric_transformation } { artwork_pin_identifier = cell_pin_identifier ; } }
```

Syntax 77—ARTWORK statement

The purpose of the *artwork* statement is to create a reference between an artwork described in a physical layout format, e.g., GDSII [**put reference to GDSII here**], and the cell described in the ALF.

A *geometric transformation* (see Section 10.18) can be used to define a transformation of coordinates from the artwork geometry to the cell geometry. The artwork is considered the original object whereas the cell is the transformed object.

The artwork statement can also etablish a mapping between a pin within the artwork and a pin of the cell. The name of the artwork pin shall appear on the left-hand side. The name of the cell pin shall appear on the right-hand side.

Example

```
CELL my_cell {
    PIN A { /* fill in pin items */ }
    PIN Z { /* fill in pin items */ }
    ARTWORK = \GDS2$!@#$ {
       SHIFT { 0 0 }
       ROTATE = 0;
       \GDS2$!@#$A = A;
       \GDS2$!@#$B = B;
```

```
        }
    }
```

10.20 VIA instantiation

A *via instantiation* shall be defined as shown in Syntax 78.

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

Syntax 78—VIA instantiation

The purpose of a via instantiation is to enable the definition of a design *rule* (see Section 9.22), a *blockage* (see Section 9.24) or a *port* (see Section 9.25) involving a declared *via* (see Section 9.17). A geometric transformation (see Section 10.18) can be used to describe a transformation of coordinates from a via declaration to the via instantiation. The declared via is considered the original object, whereas the instantiated via is the transformed object.

11. Description of electrical and physical measurements

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

The purpose of an arithmetic expression is the construction of an *arithmetic model* (see Section 11.3) or an *arithmetic assignment* (see Section 8.10).

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 operations and operators

11.2.1 Unary arithmetic operator

An *unary arithmetic operator* shall be defined as shown in Syntax 80.

```
unary_arithmetic_operator ::=
      +
    | -
```

Syntax 80—Unary arithmetic operator

Table 91 defines the semantics of unary arithmetic operators.

Table 91—Unary arithmetic operators

Operator	Description	Comment
+	Positive sign.	Neutral operator.
-	Negative sign.	

11.2.2 Binary arithmetic operator

A *binary arithmetic operator* shall be defined as shown in Syntax 81.

```
binary_arithmetic_operator ::=
    +
  | -
  | *
  | /
  | **
  | %
```

Syntax 81—Binary arithmetic operator

Table 92 defines the semantics of binary arithmetic operators.

Table 92—Binary arithmetic operators

Operator	Description	Comment
+	Addition	
-	Subtraction	
*	Multiplication	
/	Division	Result includes fractional part.
**	Power	
%	Modulus	Remainder of division.

11.2.3 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 93 defines the semantics of macro arithmetic operators.

Table 93—Macro arithmetic operators

Operator	Description	Comment
log	Natural logarithm.	1 operand, operand > 0
exp	Natural exponential.	1 operand

Table 93—Macro arithmetic operators (Continued)

Operator	Description	Comment
`abs`	Absolute value.	1 operand
`min`	Minimum.	`N` operands, N > 1
`max`	Maximum.	N operands, N > 1

11.2.4 Operator priorities

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 (`-`)

11.3 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

The purpose of an arithmetic model is to specify a measurable or a calculatable quantity.

A *trivial arithmetic model* shall be defined as shown in Syntax 84.

```
trivial_arithmetic_model ::=
      arithmetic_model_identifier [ name_identifier ] = arithmetic_value ;
    | arithmetic_model_identifier [ name_identifier ] = arithmetic_value
      { { arithmetic_model_qualifier } }
```

Syntax 84—Trivial arithmetic model

The purpose of a trivial arithmetic model is to specify a constant *arithmetic value* associated with the arithmetic model. Therefore, no mathematical operation is necessary to evaluate a trivial arithmetic model. A trivial arithmetic model can contain a singular or a plural *arithmetic model qualifier* (see Syntax 88).

A *partial arithmetic model* shall be defined as shown in Syntax 85.

The purpose of a partial arithmetic model is to specify a singluar or a plural *model qualifier* (see Syntax 88), or a *table* (see Syntax 91) or a *trivial min-max* statement (see Syntax 95). The specification contained within a partial arithmetic model can be inherited by another arithmetic model of the same type, according to the following rules:

```
partial_arithmetic_model ::=
    arithmetic_model_identifier [ name_identifier ] { { partial_arithmetic_model_item } }
partial_arithmetic_model_item ::=
      arithmetic_model_qualifier
    | table
    | trivial_min-max
```

Syntax 85—Partial arithmetic model

a) If the partial arithmetic model has no name, the specification shall be inherited by all arithmetic models of the same type appearing either within the same parent or within a descendant of the same parent.
b) If the partial arithmetic model has a name, the specification shall be only inherited by an arithmetic model containing a reference to the name, using the *model reference annotation* (see Section 11.9.5).
c) An arithmetic model can override an inherited specification by its own specification.

A partial arithmetic model does not specify a mathematical operation or an arithmetic value. Therefore it can not be mathematically evaluated.

A *full arithmetic model* shall be defined as shown in Syntax 86.

```
full_arithmetic_model ::=
    nonescaped_identifier [ name_identifier ]
    { { arithmetic_model_qualifier } arithmetic_model_body { arithmetic_model_qualifier } }
```

Syntax 86—Full arithmetic model

The purpose of a full arithmetic model is to specify mathematical data and a mathematical evaluation method associated with the arithmetic model. This specification resides in the *arithmetic model body* (see Syntax 87). A full arithmetic model can also contain a singular or a plural *arithmetic model qualifier* (see Syntax 88).

The *arithmetic model identifier* in Syntax 84, Syntax 85 and Syntax 86 shall be declared as a *keyword* (see Section 8.3) and provide specific semantics for the arithmetic model.

An *arithmetic model body* shall be defined as shown in Syntax 87.

```
arithmetic_model_body ::=
      header-table-equation [ trivial_min-max ]
    | min-typ-max
    | arithmetic_submodel { arithmetic_submodel }
```

Syntax 87—Arithmetic model body

The purpose of the arithmetic model body is to specify mathematical data associated with a full arithmetic model. The data is represented either by a *header-table-equation* statement (see Section 11.4), or by a *min-typ-max* statement (see Section 11.5), or by a singular or a plural *arithmetic submodel* (see Section 11.7).

An *arithmetic model qualifier* shall be defined as shown in Syntax 88.

The purpose of an arithmetic model qualifier is to specify semantics related to an arithmetic model.

An *inheritable arithmetic model qualifier*, i.e., an *annotation* (see Section 7.11), an *annotation container* (see Section 7.12) or a *from-to* statement (see Section 11.12) can be inherited by another arithmetic model using a *model reference annotation* (see Section 11.9.5).

```
arithmetic_model_qualifier ::=
      inheritable_arithmetic_model_qualifier
    | non_inheritable_arithmetic_model_qualifier
inheritable_arithmetic_model_qualifier ::=
      annotation
    | annotation_container
    | from-to
non_inheritable_arithmetic_model_qualifier ::=
      auxiliary_arithmetic_model
    | violation
```

Syntax 88—Arithmetic model qualifier

A *non-inheritable arithmetic model qualifier,* i.e., an *auxiliary arithmetic model* (see Section 11.6), a *violation* (see Section 11.10) or a *wire instantiation* (see Section 11.11) shall apply only for the arithmetic model under evaluation.

11.4 HEADER, TABLE, and EQUATION statements

A *header-table-equation* statement shall be defined as shown in Syntax 89.

```
header-table-equation ::=
      header table | header equation
```

Syntax 89—Header table equation

The purpose of a header-table-equation statement is to specify the mathematical data and a method for evaluation of the mathematical data associated with a full arithmetic model (see Syntax 86).

A *header* statement shall be defined as shown in Syntax 90.

```
header ::=
    HEADER { header_arithmetic_model { header_arithmetic_model } }
header_arithmetic_model ::=
    arithmetic_model_identifier [ name_identifier ] { { header_arithmetic_model_item } }
header_arithmetic_model_item ::=
      inheritable_arithmetic_model_qualifier
    | table
    | trivial_min-max
```

Syntax 90—HEADER statement

Each *header arithmetic model* shall represent a *dimension* of an arithmetic model.

Any *arithmetic model* (see Section 11.3) with a *header* as a parent shall be interpreted as a *header arithmetic model*. A declared *keyword* (see Section 8.3) for *arithmetic model* shall apply as identifier.

Note: The syntax for *header arithmetic model* is a true subset of the syntax for *arithmetic model*.

A *table* statement shall be defined as shown in Syntax 91.

A table statement within a *partial arithmetic model* shall define a discrete set of legal and applicable values. A table statement within a *full arithmetic model* shall represent a lookup table. If the arithmetic model body con-

```
table ::=
    TABLE { arithmetic_value { arithmetic value } }
```

Syntax 91—TABLE statement

tains a table statement, each *header arithmetic model* shall also contain a table statement. The table statement within the *header arithmetic model* shall represent the lookup index for a particular dimension.

The mathematical relation between a lookup table and its lookup indices 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
i denotes the index corresponding to the order of appearance of a dimension within the header statement
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.

An *equation* statement shall be defined as shown in Syntax 92.

```
equation ::=
    EQUATION { arithmetic_expression }
  | equation_template_instantiation
```

Syntax 92—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.

A full arithmetic model or any of its dimensions can inherit a set of legal values from a partial arithmetic model (see Syntax 85), represented by a table statement. Such a table statement can not substitute a lookup index within a dimension, and it can not pose a restriction on the evaluation of an arithmetic expression. The header-table-equation statement shall enable evaluation of the arithmetic model at least within the set or range of legal values, but possibly beyond.

11.5 MIN, MAX, and TYP statements

A *min-typ-max* statement shall be defined as shown in Syntax 93.

```
min-typ-max ::=
    min-max | [ min ] typ [ max ]
min-max ::=
    min | max | min max
min ::=
    trivial_min | non_trivial_min
max ::=
    trivial_max | non_trivial_max
typ ::=
    trivial_typ | non_trivial_typ
```

Syntax 93—MIN-TYP-MAX statement

The purpose of a min-typ-max statement is to represent one or more possible sets of mathematical data associated with an arithmetic model, rather than a single actual set.

Data associated with a *min* statement shall represent the smallest possible evaluation result under a given evaluation condition, i.e., actual evaluation results can be numerically greater.

Data associated with a *max* statement shall represent the greatest possible evaluation result under a given evaluation condition, i.e., actual evaluation results can be numerically smaller.

Data associated with a *typ* statement shall represent a typical evaluation result under a given evaluation condition, i.e., actual evaluation results can be numerically greater or smaller.

A *non-trivial min* or *max* or *typ* statement shall be defined as shown in Syntax 94.

```
non_trivial_min ::=
    MIN = arithmetic_value { violation }
  | MIN { [ violation ] header-table-equation }
non_trivial_max ::=
    MAX = arithmetic_value { violation }
  | MAX { [ violation ] header-table-equation }
non_trivial_typ ::=
    TYP { header-table-equation }
```

Syntax 94—Non-trivial MIN, MAX and TYP statements

By definition, a non-trivial *min* or *max* statement is associated with a *header-table-equation* statement (see Syntax 89) or a *violation* statement (see Section 11.10). A non-trivial *typ* statement is associated with a *header-table-equation* statement.

Note: A violation statement is a particular *arithmetic model qualifier* (see Syntax 88).

A *trivial min*, *max*, or *typ* statement shall be defined as shown in Syntax 95

By definition, a trivial *min*, *max*, or *typ* statement is associated with a constant arithmetic value.

A *trivial min-max* statement within a *partial arithmetic model* (see Syntax 85) shall define 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.

```
trivial_min-max ::=
     trivial_min |  trivial_max  | trivial_min trivial_max
trivial_min ::=
    MIN = arithmetic_value ;
trivial_max ::=
    MAX = arithmetic_value ;
trivial_typ ::=
     TYP = arithmetic_value ;
```

Syntax 95—Trivial MIN, MAX and TYP statements

A trivial min-max statement within a *header arithmetic model* (see Syntax 90) shall define the range of validity of a particular dimension. An application tool can evaluate the *header-table-equation* statement (see Syntax 89) outside the range of validity, however, the accuracy of the evaluation outside the range of validity is not guaranteed.

A trivial min-max statement shall be subjected to the following parsing rules:

a) Within a *partial arithmetic model* (see Syntax 85), a set of legal values defined by a *table* statement (see Syntax 91) shall take precedence over a range of legal values defined by a trivial min-max statement.
b) Within an *arithmetic model* (see Syntax 83) that can be interpreted as either a *partial arithmetic model* (see Syntax 85) or a *full arithmetic model* (see Syntax 86), the interpretation of a trivial min-max statement as a *min-typ-max statement* (see Syntax 95) shall take precedence. As a consequence, the interpretation of an arithmetic model as a full arithmetic model takes precedence.

The following Semantics 100 define the interpretation of *min*, *max*, *typ* as a particular *arithmetic submodel* (see Section 11.7).

```
SEMANTICS MIN = arithmetic_submodel {
  CONTEXT { arithmetic_model arithmetic_submodel }
}
SEMANTICS MAX = arithmetic_submodel {
  CONTEXT { arithmetic_model arithmetic_submodel }
}
SEMANTICS TYP = arithmetic_submodel {
  CONTEXT { arithmetic_model arithmetic_submodel }
}
```

Semantics 100—Interpretation of MIN, MAX, TYP as arithmetic submodel

This interpretation shall only apply in the context of a semantic rule, without invalidating a more restrictive syntax rule.

Note: The syntax rule for *min*, *max*, *typ* (see Syntax 93, Syntax 94, Syntax 95) is a true subset of the syntax rule for *arithmetic submodel* (see Syntax 97).

11.6 Auxiliary arithmetic model

An *auxiliary arithmetic model* shall be defined as shown in Syntax 96.

The purpose of an auxiliary arithmetic model is to serve as a *non-inheritable arithmetic model qualifier* (see Syntax 88) for another *arithmetic model* (see Syntax 83), called *principal arithmetic model*. The auxiliary arith-

```
auxiliary_arithmetic_model ::=
      arithmetic_model_identifier = arithmetic_value ;
    | arithmetic_model_identifier [ = arithmetic_value ]
      { inheritable_arithmetic_model_qualifier { inheritable_arithmetic_model_qualifier } }
```

Syntax 96—Auxiliary arithmetic model

metic model an be associated with a singular or plural *inheritable arithmetic model qualifier* (see Syntax 88), with a constant *arithmetic value*, or both.

Any *arithmetic model* (see Section 11.3) with another arithmetic model as a parent shall be interpreted as an *auxiliary arithmetic model*. A declared *keyword* (see Section 8.3) for *arithmetic model* shall apply as identifier.

Note: The syntax for *auxiliary arithmetic model* is a true subset of the syntax for *arithmetic model*.

A constant arithmetic value associated with an auxiliary arithmetic model shall indicate that an applicable dimension of the principal arithmetic model shall be evaluated under this constant arithmetic value or that the principal arithmetic model itself is characterized by this constant arithmetic value.

Note: The auxiliary arithmetic model is not a dimension of the principal arithmetic model.

11.7 Arithmetic submodel

An *arithmetic submodel* shall be defined as shown in Syntax 97.

```
arithmetic_submodel ::=
      arithmetic_submodel_identifier = arithmetic_value ;
    | arithmetic_submodel_identifier { [ violation ] min-max }
    | arithmetic_submodel_identifier { header-table-equation [ trivial_min-max ] }
    | arithmetic_submodel_identifier { min-typ-max }
    | arithmetic_submodel_template_instantiation
```

Syntax 97—Arithmetic submodel

The purpose of an arithmetic submodel is to serve as *arithmetic model body* (see Syntax 87), wherein the data associated with the *full arithmetic model* (see Syntax 83) is represented as one or more measurement-specific sets rather than a single set. The *arithmetic submodel identifier* shall be declared as a *keyword* (see Section 8.3) and provide specific semantics.

11.8 Arithmetic model container

11.8.1 General arithmetic model container

A general *arithmetic model container* shall be defined as shown in Syntax 98.

```
arithmetic_model_container ::=
      limit_arithmetic_model_container
    | early-late_arithmetic_model_container
    | arithmetic_model_container_identifier { arithmetic_model { arithmetic_model } }
```

Syntax 98—General arithmetic model container

The purpose of an arithmetic model container is to provide a context for an arithmetic model. The *arithmetic model container identifier* shall be a declared *keyword* (see Section 8.3) and provide specific semantics.

11.8.2 Arithmetic model container LIMIT

The arithmetic model container *limit* shall be defined as shown in Syntax 99.

```
limit_arithmetic_model_container ::=
    LIMIT { limit_arithmetic_model { limit_arithmetic_model } }
limit_arithmetic_model ::=
    arithmetic_model_identifier [ name_identifier ]
    { { arithmetic_model_qualifier } limit_arithmetic_model_body }
limit_arithmetic_model_body ::=
      limit_arithmetic_submodel { limit_arithmetic_submodel }
    | min-max
limit_arithmetic_submodel ::=
    arithmetic_submodel_identifier { [ violation ] min-max }
```

Syntax 99—Arithmetic model container LIMIT

The purpose of the arithmetic model container *limit* is to specify one or more quantifiable design limits. The design limit shall be represented as a *min-max* statement (see Section 11.5) in the context of a *limit arithmetic model* or a *limit arithmetic submodel*.

Any *arithmetic model* (see Section 11.3) with a *limit* as a parent shall be interpreted as a *limit arithmetic model*. A declared *keyword* (see Section 8.3) for *arithmetic model* shall apply as identifier. Any *arithmetic submodel* (see Section 11.7) with a *limit arithmetic model* as a parent shall be interpreted as a *limit arithmetic submodel*. A declared *keyword* (see Section 8.3) for *arithmetic submodel* shall apply as identifier.

Note: The syntax for *limit arithmetic model* is a true subset of the syntax for *arithmetic model*. The syntax for *limit arithmetic submodel* is a true subset of the syntax for *arithmetic submodel*.

The following Semantics 101 define the interpretation of *limit* as *arithmetic model container.*

```
SEMANTICS LIMIT = arithmetic_model_container;
```

Semantics 101—Arithmetic model container LIMIT

11.8.3 Arithmetic model container EARLY and LATE

The arithmetic model containers *early* and *late* shall be defined as shown in Syntax 100.

The purpose of the arithmetic model containers *early* and *late* is to specify an envelope of a timing waveform. The arithmetic model *delay* (see Section 11.11.3), *retain* (see Section 11.11.4) or *slewrate* (see Section 11.11.5) can be used to specify a timing waveform. The arithmetic model container *early* and *late* shall be associated with the leading and trailing part of the envelope, respectively. A partial specification of the envelope, i.e., only the leading part or only the trailing part, is possible.

The following Semantics 102 define the interpretation of *early* and *late* as arithmetic model container.

The arithmetic model containers *early* and *late* shall be children of a declared *vector* (see Section 9.13).

```
early-late_arithmetic_model_container ::=
      early_arithmetic_model_container
    | late_arithmetic_model_container
    | early_arithmetic_model_container late_arithmetic_model_container
early_arithmetic_model_container ::=
    EARLY { early-late_arithmetic_model { early-late_arithmetic_model } }
late_arithmetic_model_container ::=
    LATE { early-late_arithmetic_model { early-late_arithmetic_model } }
early-late_arithmetic_model ::=
      DELAY_arithmetic_model
    | RETAIN_arithmetic_model
    | SLEWRATE_arithmetic_model
```

Syntax 100—Arithmetic model container EARLY and LATE

```
SEMANTICS EARLY = arithmetic_model_container
 { CONTEXT = VECTOR; }
SEMANTICS LATE = arithmetic_model_container
 { CONTEXT = VECTOR; }
```

Semantics 102—Arithmetic model container EARLY and LATE

11.9 Generally applicable annotations for arithmetic models

Add lead-in text

11.9.1 UNIT annotation

A *unit* annotation shall be defined as shown in Semantics 103.

```
KEYWORD UNIT = single_value_annotation {
  CONTEXT = arithmetic_model ;
  VALUETYPE = multiplier_prefix_value ;
}
```

Semantics 103—UNIT annotation

The purpose of the unit annotation is to specify a *multiplier prefix value* (see Section 7.2) associated with the base unit of the arithmetic model. The base unit of an arithmetic model shall be specified by the *SI-model annotation* (see Section 8.5.6).

If the unit annotation is not present, a locally declared arithmetic model shall inherit the unit annotation of a globally declared arithmetic model of the same ALF type. If the ALF type of the globally declared arithmetic model is an SI-model annotation value, a locally declared arithmetic model with the same associated SI-model annotation value shall inherit the unit annotation as well.

Note: The *multiplier prefix value* specification given by the *unit annotation* applies to an *arithmetic model* declaration. Therefore it can be locally changed. The *SI-model annotation* applies to the *keyword* declaration (see Section 8.3) of an arithmetic model. Therefore it can not be changed.

Example:

The arithmetic model *delay* (see Section 11.11.3) has the SI-model annotation value *time*. Therefore *delay* can inherit the unit annotation value of the arithmetic model *time* (see Section 11.11.1).

11.9.2 CALCULATION annotation

A *calculation* annotation shall be defined as shown in Semantics 104.

```
KEYWORD CALCULATION = annotation {
  CONTEXT = library_specific_object.arithmetic_model ;
  VALUES { absolute incremental }
  DEFAULT = absolute ;
}
```

Semantics 104—CALCULATION annotation

The meaning of the annotation values is shown in Table 94.

Table 94—Calculation annotation

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.

The following rules for combination of arithmetic model data shall apply:

a) Data shall be combined by adding them together.
b) Data can only be combined, if the respective arithmetic models have the same type.
c) Data can only be combined, if a common semantic interpretation of the respective arithmetic models within their context exists.

Specifics of rule c) are described in sections for specific arithmetic models.

11.9.3 INTERPOLATION annotation

A *interpolation* annotation shall be defined as shown in Semantics 105.

```
KEYWORD INTERPOLATION = single_value_annotation {
  CONTEXT = HEADER.arithmetic_model ;
  VALUES { linear fit ceiling floor }
  DEFAULT = fit ;
}
```

Semantics 105—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 95.

Table 95—Interpolation annotation

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 19.

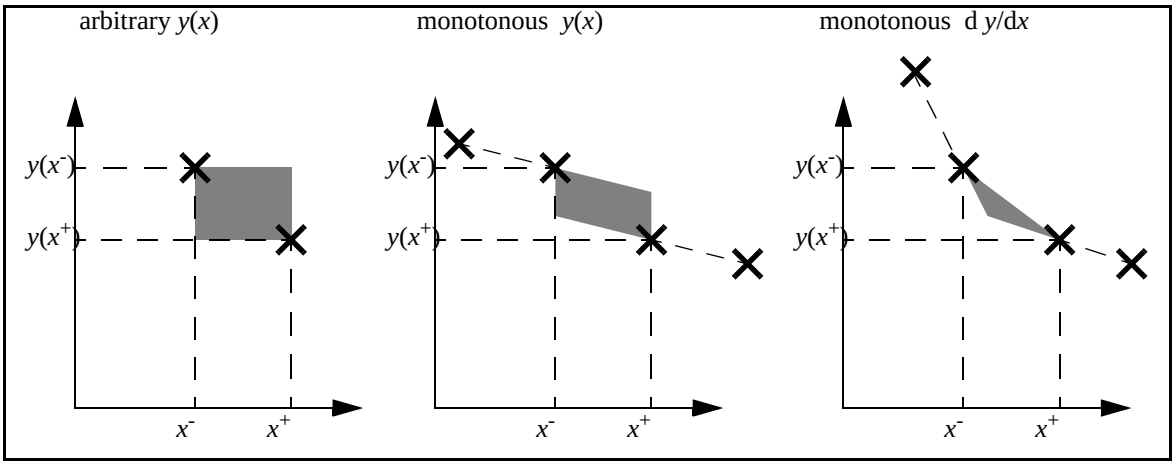


Figure 19—Bounding regions for y(x) with INTERPOLATION=fit

11.9.4 DEFAULT annotation

A *default* annotation (see Section 8.5.3) shall be applicable for an arithmetic model, unless the *keyword* declaration (see Section 8.3) for the arithmetic model contains already a default annotation.

The purpose of the default annotation is the specification of an evaluation result for a *full arithmetic model* (see Section 11.3, Syntax 86) or a *header arithmetic model* (see Section 11.4, Syntax 90) in case the arithmetic model can not be evaluated otherwise. A default annotation shall not apply for a *trivial arithmetic model* (see Section 11.3, Syntax 84). A default annotation for a *partial arithmetic model* (see Section 11.3, Syntax 85) shall serve as *inheritable arithmetic model qualifier* (see Section 11.3, Syntax 88), to be acquired by another full arithmetic model.

A default annotation value associated with a *header arithmetic model* or with a *partial arithmetic model* shall be an *arithmetic value* (see Section 7.4) compatible with the arithmetic model's *valuetype* (see Section 8.5.1). A default annotation value associated with a *full arithmetic model* shall be either an arithmetic value compatible with its valuetype, or, alternatively, an *identifier* refering to another arithmetic model or to an *arithmetic submodel* (see Section 11.7).

The following rules shall apply for the usage of the default annotation value:

a) If the application provides values for all header arithmetic models, no default annotation value shall be used for the evaluation of a full arithmetic model.
b) If the application provides values for some, but not all header arithmetic models, and the remaining header arithmetic models have associated default annotations, those default annotation values shall be used.
c) If application values for all header arithmetic models are missing and the full arithmetic model has an associated default annotation, this default annotation value shall be used.
d) If application values for all header arithmetic models are missing and the full arithmetic model has no associated default annotation, but all header arithmetic models have, those default annotation values shall be used.

In any other case, the evaluation of the full arithmetic model shall fail and result in an application error.

11.9.5 MODEL reference annotation

A *model* reference annotation shall be defined as shown in Semantics 106.

```
KEYWORD MODEL = single_value_annotation {
  CONTEXT = arithmetic_model ;
  VALUETYPE = identifier ;
  REFERENCETYPE { arithmetic_model arithmetic_submodel }
}
```

Semantics 106—MODEL reference annotation

The purpose of a model reference annotation is to acquire an *inheritable arithmetic model qualifier* (see Section 11.3, Syntax 88), an evaluation result (Syntax 91, Syntax 92) or both from another arithmetic model. The model reference annotation value shall be the ALF name of the referenced arithmetic model.

An evaluation result can also be acquired from a referenced *arithmetic submodel* (see Section 11.7). In this case, the model reference annotation value shall be a *hierarchical identifier* (see Section 6.11.4) composed of the ALF name of the parent arithmetic model and the ALF type of the arithmetic submodel.

A calculation graph can be established by using the model reference annotation within a *header arithmetic model* (see Section 11.4, Syntax 90). In this case, the evaluation of the arithmetic model containing the header arithmetic model depends on the evaluation of the referenced model. A circular reference shall not be allowed.

The model reference annotation shall further be legal under the following restrictions:

a) Both the referencing and the referenced arithmetic model have the same ALF type,
   or, alternatively:
b) the ALF type of either arithmetic model is an *SI-model annotation* value (see Section 8.5.6), and both arithmetic models have the same associated SI-model annotation value.
c) The semantics of any arithmetic model qualifier are compatible with the semantics of any acquired arithmetic model qualifier.

Examples:

Rule a): An arithmetic model of ALF type *time* (see Section 11.11.1) can refer to the arithmetic model of ALF type *time*.

Rule b): The arithmetic model *delay* (see Section 11.11.3) has the SI-model annotation value *time*. Therefore an arithmetic model of ALF type *delay* can refer to an arithmetic model of ALF type *time* and vice-versa.

Rule c): If both arithmetic models have an annotation of the same ALF type (e.g. *unit* annotation, see Section 11.9.1), the annotation values shall be the same.

11.10 VIOLATION statement, MESSAGE TYPE and MESSAGE annotation

A *violation* statement shall be defined as shown in Syntax 101.

The purpose of a violation statement is to specify the consequence of an evaluation of an *arithmetic model* (see Section 11.3) resulting in a violation of a design constraint or a design limit.

A violation statement shall be subjected to the restriction shown in Semantics 107.

```
violation ::=
    VIOLATION { violation_item { violation_item } }
  | violation_template_instantiation
violation_item ::=
    MESSAGE_TYPE_single_value_annotation
  | MESSAGE_single_value_annotation
  | behavior
```

Syntax 101—VIOLATION statement

```
SEMANTICS VIOLATION {
  CONTEXT {
    SETUP HOLD RECOVERY REMOVAL NOCHANGE ILLEGAL
    NOISE_MARGIN
    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 107—Restriction for VIOLATION statement

The purpose of the restriction is to specify a legal anchestor of a violation statement. Only an arithmetic model that serves the purpose of evaluating a design constraint or a design limit can be a legal anchestor of a violation statement.

A violation statement can contain a *message-type* annotation, a *message* annotation and a *behavior* statement (see Section 10.4).

The behavior statement shall be subjected to the restriction shown in Semantics 108.

```
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 108—Restriction for BEHAVIOR statement within VIOLATION

The purpose of the restriction is to provide a triggering event for the consequence of a violation. The evaluation of an arithmetic model with a vector as anchestor, and hence the consequence of a violation, is triggered by the evaluation of the vector expression ,which is the name of the vector.

A *message type* annotation shall be defined as shown in Semantics 109.

```
KEYWORD MESSAGE_TYPE = single_value_annotation {
  CONTEXT = VIOLATION ;
  VALUETYPE = identifier ;
  VALUES { information warning error }
}
```

Semantics 109—MESSAGE_TYPE annotation

The purpose of the message type annotation value is to classify the severity of a violation.

The meaning of the annotation values is shown in .

Table 96—MESSAGE_TYPE annotation

Annotation value	Description
`information`	The application tool shall issue an informative message when the violation is encountered.
`warning`	The application tool shall issue a warning message when the violation is encountered.
`error`	The application tool shall issue an error message when the violation is encountered.

A *message* annotation shall be defined as shown in Semantics 110.

```
KEYWORD MESSAGE = single_value_annotation {
  CONTEXT = VIOLATION ;
  VALUETYPE = quoted_string ;
}
```

Semantics 110—MESSAGE annotation

The purpose of the message annotation is to specify verbatim the text of the message issued by the application tool when a violation is encountered.

11.11 Arithmetic models for timing, power and signal integrity

11.11.1 TIME

The arithmetic model *time* shall be defined as shown in Semantics 111.

The purpose of the arithmetic model *time* is to specify a time interval in general.

— TIME in context of a declared *library* or *sublibrary* (see Section 9.1), a declared *cell* (see Section 9.3), or a declared *wire* (see Section 9.9)

A *partial arithmetic model* (see Syntax 85 within Section 11.3) can be used to globally specify an *inheritable arithmetic model qualifier* (see Syntax 88 within Section 11.3).

— TIME in context of a declared *vector* (see Section 9.13)

```
KEYWORD TIME = arithmetic_model {
  VALUETYPE = number ;
  SI_MODEL = TIME ;
}
SEMANTICS TIME {
  CONTEXT {
    LIBRARY SUBLIBRARY CELL WIRE
    VECTOR VECTOR.arithmetic_model_container
    HEADER
    arithmetic_model
  }
}
TIME { UNIT = NanoSeconds ; }
```

Semantics 111—Arithmetic model TIME

If the ALF name of the vector is a *vector expression* (see Section 10.12), a *from-to* statement (see Section 11.12) shall be used as *model qualifier.* The arithmetic model shall represent a measured time interval between two *single events* (see Section 10.13.1).

Otherwise, if the ALF name of the vector is a *boolean expression* (see Section 10.9), the arithmetic model shall represent a time interval during which the boolean expression is true. A from-to statement shall not be used as model qualifier.

As a child of the arithmetic model container *limit* (see Section 11.8.2), the arithmetic model shall specify a design limit for a time interval. Otherwise, the arithmetic model shall specify a measured time interval.

— TIME as *header arithmetic model* (see Syntax 89 in Section 11.4)

The header arithmetic model *time* shall represent a *dimension* of another arithmetic model. The dimension *time* shall generally describe a quantity changing over time, which can be visualized by a timing waveform.

If the grandparent or the great-grandparent of the header arithmetic model is a *vector* with a *vector expression* as ALF name, a *from* statement can be used as *model qualifier* to define a temporal relationship between a *single event* and the dimension time.

If the grandparent of the header arithmetic model is the arithmetic model container *limit*, the dimension *time* shall describe a dependency between a design limit and the expected lifetime of an electronic circuit, rather than a timing waveform.

Note: By definition, the parent of a *header arithmetic model* is always a *full arithmetic model*.

— TIME as *auxiliary arithmetic model* (see Syntax 96 in Section 11.6)

The auxiliary arithmetic model *time* shall be used in conjunction with a *measurement* annotation (see Section 11.13.7). The auxiliary arithmetic model shall specify the time interval during which the measurement is taken.

If the grandparent of the auxiliary arithmetic model is a *vector* with a *vector expression* as ALF name, a *from-to* statement can be used to define a temporal relationship between one or two single events in the vector expression and the time interval.

11.11.2 FREQUENCY

The arithmetic model *frequency* shall be defined as shown in Semantics 112.

```
KEYWORD FREQUENCY = arithmetic_model {
  VALUETYPE = unsigned_number ;
  SI_MODEL = FREQUENCY ;
}
SEMANTICS FREQUENCY {
  CONTEXT {
    LIBRARY SUBLIBRARY CELL WIRE
    VECTOR VECTOR.arithmetic_model_container
    HEADER
    arithmetic_model
  }
}
FREQUENCY { UNIT = GigaHertz; }
```

Semantics 112—Arithmetic model FREQUENCY

The purpose of the arithmetic model *frequency* is to specify a temporal frequency in general.

The arithmetic model *frequency* can be a child or a grandchild of a declared *library* or *sublibrary* (see Section 9.1), a declared *cell* (see Section 9.3), *wire* (see Section 9.9) or *vector* (see Section 9.13).

— FREQUENCY in context of a declared *vector* (see Section 9.13)

As a child or a grandchild of a declared vector with a *vector expression* (see Section 10.12) as ALF name, the arithmetic model shall specify a statistical occurence frequency of the vector.

As a child of the arithmetic model container *limit* (see Section 11.8.2), the arithmetic model shall specify a design limit for an occurence frequency. Otherwise, the arithmetic model shall specify a measured occurence frequency.

— FREQUENCY as *header arithmetic model* (see Syntax 89 in Section 11.4)

The header arithmetic model *frequency* shall represent a *dimension* of another arithmetic model.

If the grandparent or the great-grandparent of the header arithmetic model is a *vector* with a *vector expression* as ALF name, the dimension frequency shall represent the occurrence frequency of the vector.

If the grandparent or the great-grandparent of the header arithmetic model is not a *vector*, the frequency dimension shall be represent a spectral dependency of the arithmetic model.

— FREQUENCY as *auxiliary arithmetic model* (see Syntax 96 in Section 11.6)

A frequency statement can be a child of an arithmetic model, thus representing an auxiliary arithmetic model.

The auxiliary arithmetic model *frequency* shall be used in conjunction with a *measurement* annotation (see Section 11.13.7). The auxiliary arithmetic model shall specify the repetition frequency of the measurement.

The auxiliary arithmetic models *frequency* and *time* (see Section 11.11.1) can be used interchangably, unless a *from* or a *to* statement is associated with time. The measurement repetition frequency *f* and the measurement time interval *t* can be equated by $f = 1 / t$.

11.11.3 DELAY

The arithmetic model *delay* shall be defined as shown in Semantics 113.

```
KEYWORD DELAY = arithmetic_model { SI_MODEL = TIME ; }
SEMANTICS DELAY {
  CONTEXT {
    LIBRARY SUBLIBRARY CELL WIRE
    VECTOR VECTOR.EARLY VECTOR.LATE
  }
}
```

Semantics 113—Arithmetic model DELAY

The purpose of the arithmetic model *delay* is to specify a time interval, implying a causal relationship between two events. A *from-to* statement (see Section 11.12) shall be used as *model qualifier.*

— DELAY in context of a declared *vector* (see Section 9.13)

As a child or a grandchild of a declared vector with a *vector expression* (see Section 10.12) as ALF name, the arithmetic model *delay* shall specify a measured time interval between two *single events* (see Section 10.13.1), implying that the *from-event* is the cause of the *to-event*.

If the model qualifier features only a *from* or only a *to* statement, the arithmetic model delay shall be interpreted as a partial time interval specification. The *calculation* annotation (see 11.9.2) shall be used in conjunction a partial time interval specification. If the annotation value is *incremental*, the partial time interval shall be added to another time interval. If the annotation value is *absolute*, the partial time interval shall be used as a default and otherwise be substituted by a completely specified time interval.

— DELAY in context of a declared *library* or *sublibrary* (see Section 9.1), a declared *cell* (see Section 9.3), or a declared *wire* (see Section 9.9)

As a *partial arithmetic model* (see Syntax 85 within Section 11.3), *delay* can be used for global specification of a *model qualifier*. In particular, the arithmetic model *threshold* (see Section 11.11.13) within a *from-to* statement can be globally specified.

The global specification of a model qualifier shall be inherited by the arithmetic models *delay, retain* (see Section 11.11.4), *setup* and *hold* (see Section 11.11.6), *recovery* and *removal* (see Section 11.11.7) and *skew* (see Section 11.11.12) in the context of a *vector.*

11.11.4 RETAIN

The arithmetic model *retain* shall be defined as shown in Semantics 114.

The purpose of the arithmetic model *retain* is to specify a time interval, during which a cause has no observable effect. A *from-to* statement (see Section 11.12) shall be used as *model qualifier.*

```
KEYWORD RETAIN = arithmetic_model { SI_MODEL = TIME ; }
SEMANTICS RETAIN{
  CONTEXT {
    VECTOR VECTOR.EARLY VECTOR.LATE
  }
}
```

Semantics 114—Arithmetic model RETAIN

As a child or a grandchild of a declared vector with a *vector expression* (see Section 10.12) as ALF name, the arithmetic model *retain* shall specify a measured time interval between two *single events* (see Section 10.13.1), implying that the *to-event* is the earliest observable effect of the *from-event*.

The arithmetic models *retain* and *delay* with matching model qualifiers can be jointly used. In this case, retain shall represent the time interval between a cause (i.e., an input signal) and the earliest effect (i.e., initial change of an output signal), and delay shall represent the time interval between a cause and the latest effect (i.e., final change of an output signal). During the time interval between initial and final change, the output signal is considered unstable.

Retain in conjunction with delay is illustrated in Figure 20.

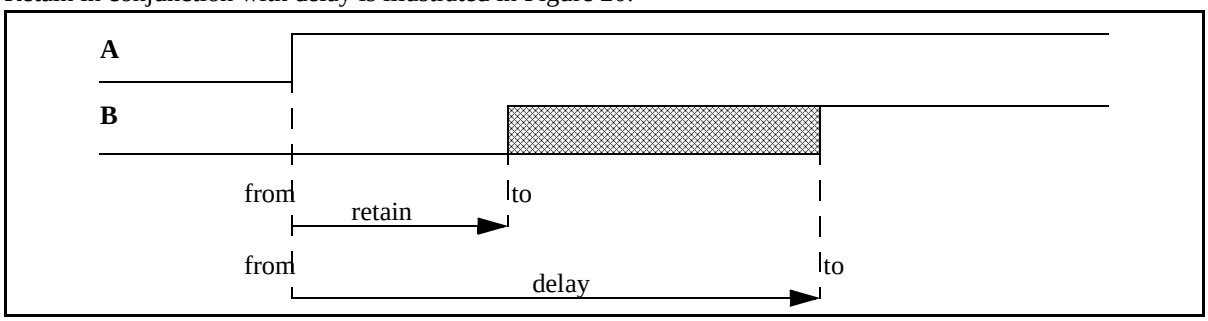


Figure 20—Illustration of RETAIN and DELAY

11.11.5 SLEWRATE

The arithmetic model *slewrate* statement shall be defined as shown in Semantics 115.

```
KEYWORD SLEWRATE = arithmetic_model { SI_MODEL = TIME ; }
SEMANTICS SLEWRATE {
  CONTEXT {
    LIBRARY LIBRARY.LIMIT
    SUBLIBRARY SUBLIBRARY.LIMIT
    CELL CELL.LIMIT
    PIN PIN.LIMIT
    WIRE WIRE.LIMIT
    VECTOR VECTOR.LIMIT VECTOR.EARLY VECTOR.LATE
    HEADER
  }
}
SLEWRATE { MIN = 0; }
```

Semantics 115—Arithmetic model SLEWRATE

The purpose of the arithmetic model *slewrate* is to specify the duration of a transient event, measured between two reference points. A reference point shall be specified by the arithmetic model *threshold* (see Section 11.11.13) within a *from-to* statement (see Section 11.12). No particular waveform shape shall be implied for the transient event.

— SLEWRATE in context of a declared *vector* (see Section 9.13)

If *slewrate* is a child or a grandchild of a declared *vector* with a *vector expression* (see Section 10.12) as ALF name, a *pin reference* annotation, eventually in conjunction with an *edge number* anotation, shall be used (see Section 11.13.2) to refer to a *single event* (see Section 10.13.1).

— SLEWRATE in context of a declared *pin* (see Section 9.5)

If *slewrate* is a child or a grandchild of a declared *pin*, the arithmetic submodel *rise* or *fall* (see Section 11.21) can be used as a substitute for a reference to a single event.

— SLEWRATE in context of a declared *library* or *sublibrary* (see Section 9.1), a declared *cell* (see Section 9.3), or a declared *wire* (see Section 9.9)

As a *partial arithmetic model* (see Syntax 85 within Section 11.3), *slewrate* can be used for global specification of a *model qualifier.* In particular, the arithmetic model *threshold* (see Section 11.11.13) within a *from-to* statement can be globally specified.

The global specification of a model qualifier shall be inherited by the arithmetic model *slewrate* in the context of a *vector.*

— SLEWRATE as *header arithmetic model* (see Syntax 89 in Section 11.4)

The header arithmetic model *slewrate* shall represent a *dimension* of another arithmetic model. The arithmetic model shall be in the context of a *vector.* A reference to a *single event* shall be used as *model qualifier.*

Slewrate is illustrated in the following .

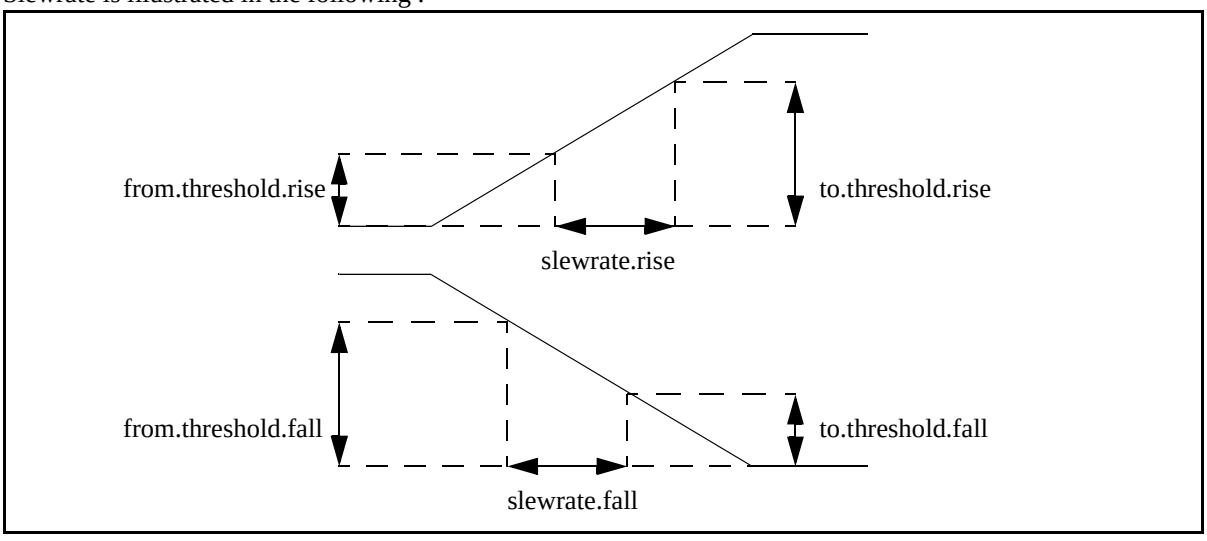


Figure 21—Illustration of SLEWRATE

11.11.6 SETUP and HOLD

The arithmetic models *setup* and *hold* shall be defined as shown in .

```
KEYWORD SETUP = arithmetic_model { SI_MODEL = TIME ; }
SEMANTICS SETUP { CONTEXT = VECTOR ; }
KEYWORD HOLD = arithmetic_model { SI_MODEL = TIME ; }
SEMANTICS HOLD { CONTEXT = VECTOR ; }
```

Semantics 116—Arithmetic models SETUP and HOLD

The purpose of the arithmetic models *setup* and *hold* is to specify timing constraints between a data signal and a clock signal. Each arithmetic model shall be a child of a declared *vector* (see Section 9.13) with a *vector expression* (see Section 10.12) as ALF name. A *from-to* statement (see Section 11.12) shall be used as *model qualifier.*

The arithmetic model *setup* shall represent the minimal required time interval during which a data signal needs to be stable before activation of a clock signal. This time interval can be positive, zero, or negative. The data signal shall be refered to within a *from* statement. The clock signal shall be refered to within a *to* statement.

The arithmetic model *hold* shall represent the minimal required time interval during which a data signal needs to be stable after activation of a clock signal. This time interval can be positive, zero, or negative. The clock signal shall be refered to within a *from* statement. The data signal shall be refered to within a *to* statement.

Co-dependent arithmetic models *setup* and *hold* can be described as children of the same *vector.* A corresponding timing diagram is illustrated in Figure 22.

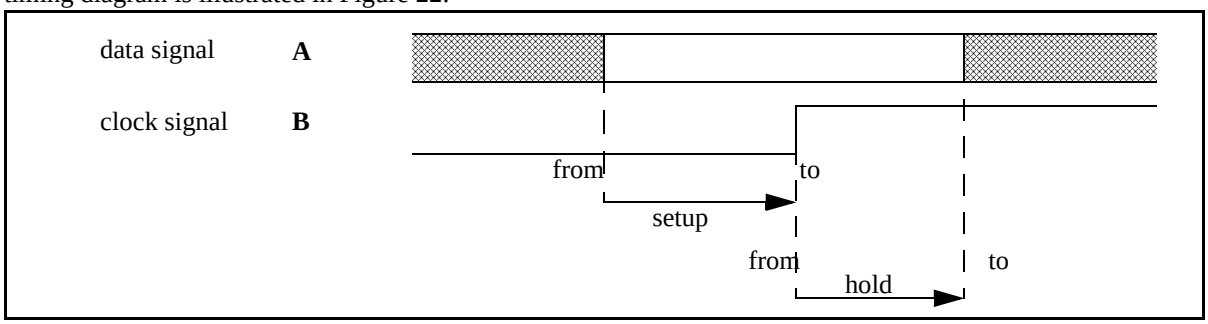


Figure 22—Illustration of SETUP and HOLD

11.11.7 RECOVERY and REMOVAL

The arithmetic models *recovery* and *removal* shall be defined as shown in Semantics 117.

```
KEYWORD RECOVERY = arithmetic_model { SI_MODEL = TIME ; }
SEMANTICS RECOVERY { CONTEXT = VECTOR ; }
KEYWORD REMOVAL = arithmetic_model { SI_MODEL = TIME ; }
SEMANTICS REMOVAL { CONTEXT = VECTOR ; }
```

Semantics 117—Arithmetic models RECOVERY and REMOVAL

The purpose of the arithmetic models *recovery* and *removal* is to specify timing constraints between a clock signal and an asynchronous control signal. Each arithmetic model shall be a child of a declared *vector* (see Section 9.13) with a *vector expression* (see Section 10.12) as ALF name. A *from-to* statement (see Section 11.12) shall be used as *model qualifier.*

The arithmetic model *recovery* shall represent the minimal required time interval between de-assertion of an asynchronous control signal and activation of a clock signal. This time interval can be positive, zero, or negative. The asynchronous control signal signal shall be refered to within a *from* statement. The clock signal shall be refered to within a *to* statement.

The arithmetic model *removal* shall represent the minimal required time interval between a suppressed activation of a clock signal and de-assertion of an asynchronous control signal. This time interval can be positive, zero, or negative. The clock signal shall be refered to within a *from* statement. The asynchronous control signal signal shall be refered to within a *to* statement.

Co-dependent arithmetic models *recovery* and *removal* can be described as children of the same *vector.* A corresponding timing diagram is illustrated in Figure 23.

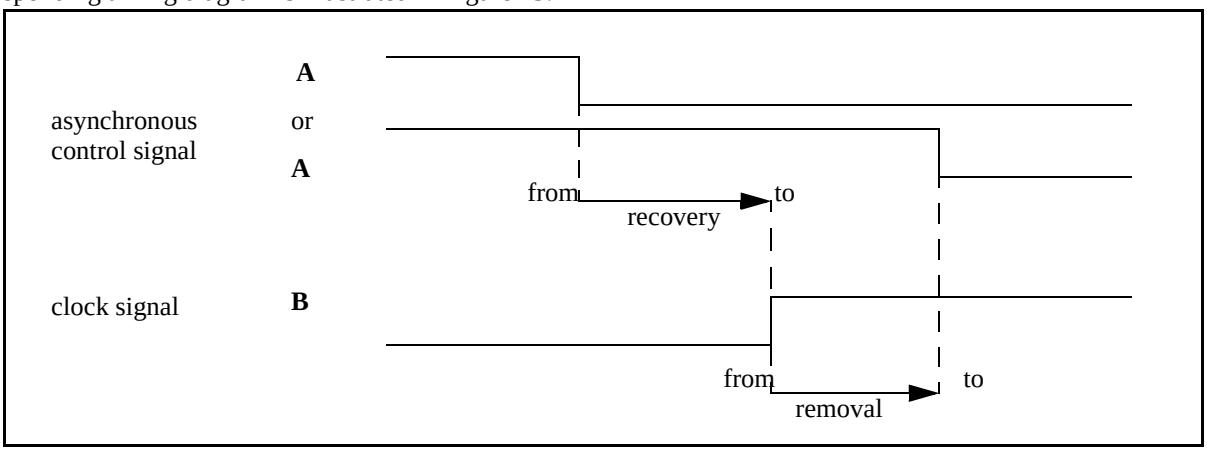


Figure 23—RECOVERY and REMOVAL

11.11.8 NOCHANGE and ILLEGAL

The arithmetic models *nochange* and *illegal* shall be defined as shown in Semantics 118.

```
KEYWORD NOCHANGE = arithmetic_model { SI_MODEL = TIME ; }
SEMANTICS NOCHANGE { CONTEXT = VECTOR ; }
NOCHANGE { MIN = 0; }
KEYWORD ILLEGAL = arithmetic_model { SI_MODEL = TIME ; }
SEMANTICS ILLEGAL { CONTEXT = VECTOR ; }
ILLEGAL { MIN = 0; }
```

Semantics 118—Arithmetic models NOCHANGE and ILLEGAL

The purpose of the arithmetic models *nochange* and *illegal* is to specify requirements for the duration of a logical state in the context of a declared *vector* (see Section 9.13).

If the ALF name of the vector is a *vector expression* (see Section 10.12), a *from-to* statement (see Section 11.12) can be used as *model qualifier.* The events occuring in-between the from-and to-events, including the from-and to-events themselves, shall be considered a *vector sub-expression.*

— NOCHANGE in the context of a declared *vector*

If the ALF name of the vector is a *boolean expression* (see Section 10.9), the arithmetic model *nochange* shall specify a minimum required time interval during which the boolean expression is true. A partial arithmetic model *nochange* shall indicate a requirement for the boolean expression to be forever true.

Otherwise, if the ALF name of the vector is a *vector expression* (see Section 10.12), the arithmetic model *nochange* shall specify a minimum required duration for the vector expression or for the *vector sub-expression* specified by the *from-to* statement. A partial arithmetic model *nochange* shall specify that the vector expression or the vector sub-expression is required to be observed as specified.

— ILLEGAL in the context of a declared *vector*

If the ALF name of the vector is a *boolean expression* (see Section 10.9), the arithmetic model *illegal* shall specify a maximum allowed time interval during which the boolean expression is true. A partial arithmetic model *illegal* shall indicate a requirement for the boolean expression to be never true.

Otherwise, if the ALF name of the vector is a *vector expression* (see Section 10.12), the arithmetic model *illegal* shall specify a maximum allowed duration for the vector expression or for the *vector sub-expression* specified by the *from-to* statement. A partial arithmetic model *illegal* shall specify that the vector expression or the vector sub-expression is not allowed to be observed as specified.

Nochange and *illegal* are illustrated in the following .

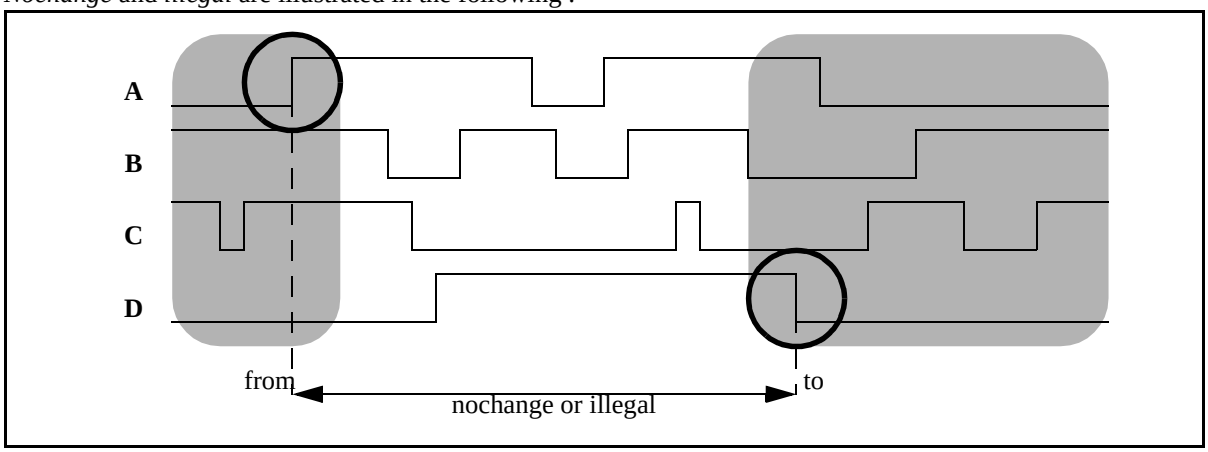


Figure 24—Illustration of NOCHANGE and ILLEGAL

If an actual event sequence involving the four signals **A**, **B**, **C** and **D** matches the beginning and the end of the timing diagram (underlaid in grey), including the *from*-and *to*-events (marked with circles), the actual event sequence in-between the *from*-and *to*-events shall be examined.

In the case of *nochange*, the actual event sequence is required to match the middle of the timing diagram, and eventually a minimal time interval between *from* and *to* is required.

In the case of *illegal*, the actual event sequence is required not to match the middle of the timing diagram, or eventually a maximum time interval between *from* and *to* is allowed.

11.11.9 PULSEWIDTH

The arithmetic model *pulsewidth* shall be defined as shown in Semantics 119.

The purpose of the arithmetic model *pulsewidth* is to specify the duration of a pulse, measured between two reference points. A reference point shall be specified by the arithmetic model *threshold* (see Section 11.11.13)

```
KEYWORD PULSEWIDTH=arithmetic_model { SI_MODEL = TIME; }
SEMANTICS PULSEWIDTH {
  CONTEXT {
    LIBRARY LIBRARY.LIMIT
    SUBLIBRARY SUBLIBRARY.LIMIT
    CELL CELL.LIMIT
    PIN PIN.LIMIT
    WIRE WIRE.LIMIT
    VECTOR VECTOR.LIMIT
    HEADER
  }
}
PULSEWIDTH { MIN = 0; }
```

Semantics 119—Arithmetic model PULSEWIDTH

within a *from-to* statement (see Section 11.12). No particular waveform shape shall be implied for the sequence of transient events.

For a *noise* waveform (see Section 11.11.14), i.e., a waveform that does not reach a constant logic value, pulsewidth shall be measured between the crossings of 50% magnitude.

— PULSEWIDTH in context of a declared *vector* (see Section 9.13)

If *pulsewidth* is a child or a grandchild of a declared *vector* with a *vector expression* (see Section 10.12) as ALF name, a *pin reference* annotation, eventually in conjunction with an *edge number* anotation, shall be used (see Section 11.13.2) to refer to a *single event* (see Section 10.13.1), representing the leading edge of the pulse.

— PULSEWIDTH in context of a declared *pin* (see Section 9.5)

If *pulsewidth* is a child or a grandchild of a declared *pin*, the arithmetic submodel *rise* or *fall* (see Section 11.21) can be used as a substitute for a reference to a single event.

— PULSEWIDTH in context of a declared *library* or *sublibrary* (see Section 9.1), a declared *cell* (see Section 9.3), or a declared *wire* (see Section 9.9)

As a *partial arithmetic model* (see Syntax 85 within Section 11.3), *pulsewidth* can be used for global specification of a *model qualifier.* In particular, the arithmetic model *threshold* (see Section 11.11.13) within a *from-to* statement can be globally specified. The global specification of a model qualifier shall be inherited by the arithmetic model *pulsewidth* in the context of a *vector.*

— PULSEWIDTH as *header arithmetic model* (see Syntax 89 in Section 11.4)

The header arithmetic model *pulsewidth* shall represent a *dimension* of another arithmetic model. The arithmetic model shall be in the context of a *vector.* A reference to a *single event* shall be used as *model qualifier.*

Pulsewidth is illustrated in the following .

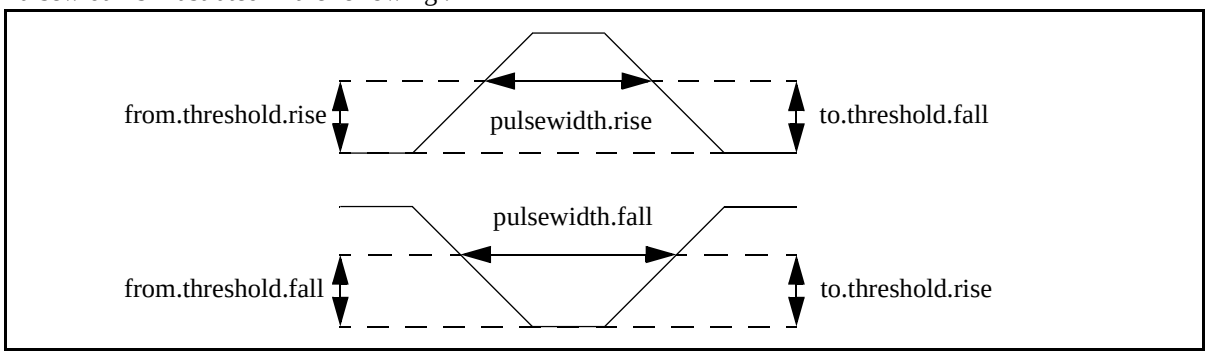


Figure 25—Illustration of PULSEWIDTH

11.11.10 PERIOD

The arithmetic model *period* shall be defined as shown in Semantics 120.

```
KEYWORD PERIOD = arithmetic_model { SI_MODEL = TIME ; }
SEMANTICS PERIOD {
  CONTEXT { VECTOR VECTOR.LIMIT HEADER }
}
PERIOD { MIN = 0; }
```

Semantics 120—Arithmetic model PERIOD

The purpose of the arithmetic model *period* is to specify a time interval between periodical repetitions of events.

The arithmetic model *period* shall be in the context of a declared *vector* (see Section 9.13) with a *vector expression* (see Section 10.12) as ALF name. The *vector expression* shall specify a primitive sequence of events .

The *header arithmetic model* (see Syntax 89 in Section 11.4) *period* shall represent a *dimension* of another arithmetic model, which shall be in the context of a *vector.*

Period is illustrated in the following .

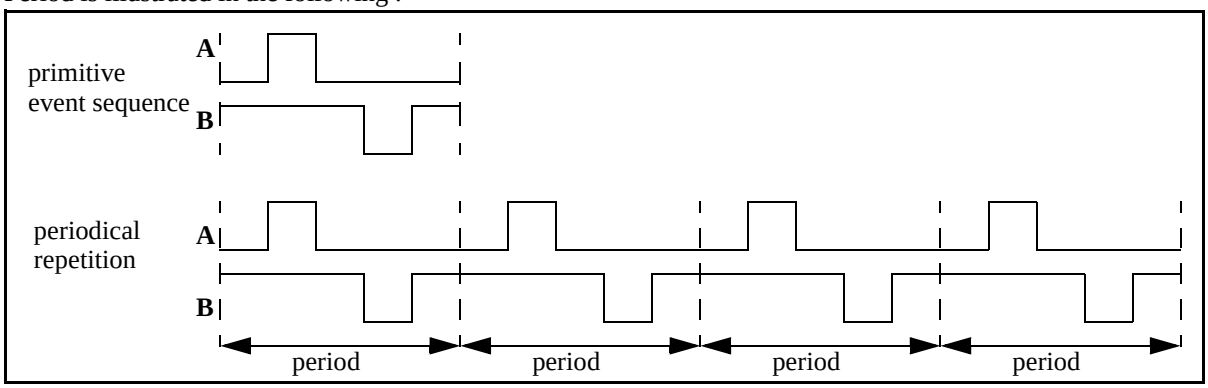


Figure 26—Illustration of PERIOD

A primitive event sequence involving two signals **A** and **B** is repeated periodically.

11.11.11 JITTER

The arithmetic model *jitter* shall be defined as shown in Semantics 121.

```
KEYWORD JITTER = arithmetic_model { SI_MODEL = TIME ; }
SEMANTICS JITTER {
  CONTEXT { VECTOR VECTOR.LIMIT HEADER }
}
JITTER { MIN = 0; }
```

Semantics 121—Arithmetic model JITTER

The purpose of the arithmetic model *jitter* is to specify the variability of a time interval between periodical repetitions of events. The *measurement* annotation (see Section 11.13.7) shall be applicable as *model qualifier.*

The arithmetic model *jitter* shall be in the context of a declared *vector* (see Section 9.13) with a *vector expression* (see Section 10.12) as ALF name. The *vector expression* shall specify a primitive sequence of events .

The *header arithmetic model* (see Syntax 89 in Section 11.4) *jitter* shall represent a *dimension* of another arithmetic model, which shall be in the context of a *vector.*

Jitter is illustrated in the following .

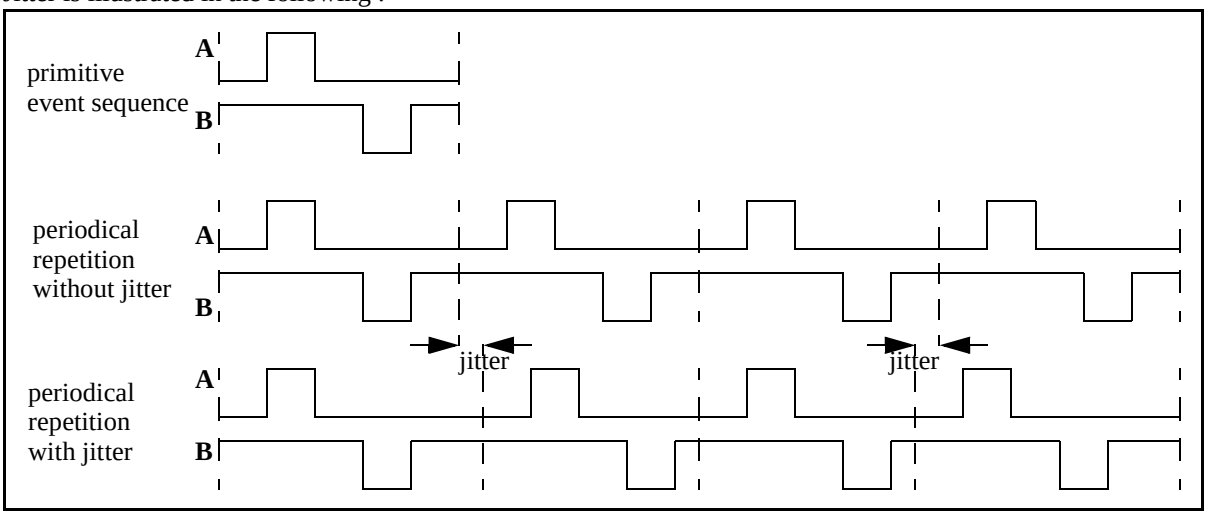


Figure 27—Illustration of JITTER

A primitive event sequence involving two signals **A** and **B** is repeated periodically. A timing diagram with and without jitter is shown.

11.11.12 SKEW

The arithmetic model *skew* shall be defined as shown in Semantics 122.

The purpose of the arithmetic model *skew* is to specify a non-negative temporal separation between multiple signals.

In the context of a declared *vector* (see Section 9.13) with a *vector expression* (see Section 10.12) as ALF name, a *pin reference* annotation, eventually in conjunction with a matching *edge number* anotation, shall be used (see

```
KEYWORD SKEW = arithmetic_model { SI_MODEL = TIME ; }
SEMANTICS SKEW {
  CONTEXT { VECTOR VECTOR.LIMIT HEADER }
}
SKEW { MIN = 0; }
```

Semantics 122—Arithmetic model SKEW

Section 11.13.5) to refer to multiple *single events* (see Section 10.13.1). The arithmetic model itself shall not specify a temporal order of the events. The temporal separation between events shall be considered for any order of events allowed by the vector expression. If the vector expression specifies *simultaneously occuring events* (see Section 10.13.3), but the arithmetic model skew specifies a non-zero temporal separation between these events, the skew shall take precedence, and the temporal separation shall be considered for an arbitrary permutation of order of occurence.

The *header arithmetic model skew* shall represent a *dimension* of another arithmetic model, which shall be in the context of a *vector.* A reference to multiple *single events* shall be used as *model qualifier.*

Skew is illustrated in the following Figure 28.

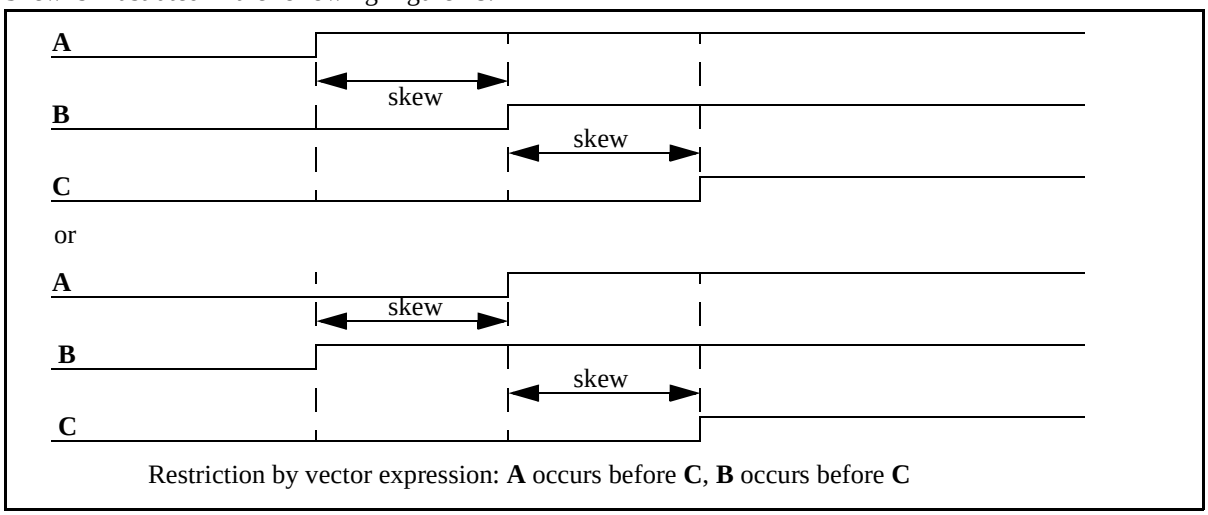


Figure 28—Illustration of SKEW

The arithmetic model skew involves three signals **A**, **B** and **C**, and the vector expression restricts **A** and **B** to occur before **C**.

11.11.13 THRESHOLD

The arithmetic model *threshold* shall be defined as shown in Semantics 123.

```
KEYWORD THRESHOLD = arithmetic_model {
  VALUETYPE = number ;
  CONTEXT { PIN FROM TO }
}
THRESHOLD { MIN = 0; MAX = 1; }
```

Semantics 123—Arithmetic model THRESHOLD

The purpose of the arithmetic model *threshold* is to specify a reference point for a timing measurement.

Threshold shall be a normalized quantity, according to the following mathematical definition.

$$\text{threshold.rise} = (vt_r - v_0) / (v_1 - v_0)$$
$$\text{threshold.fall} = (vt_f - v_0) / (v_1 - v_0)$$

where

v_0 is the nominal voltage level for the value logic zero,
v_1 is the nominal voltage level for the value logic one,
vt_r is a specified voltage level crossed during a rising transition,
vt_f is a specified voltage level crossed during a falling transition,

subject to the following restrictions:

$v_0 < v_1$
$v_0 \leq vt_r \leq v_1$ and $v_0 \leq vt_f \leq v_1$.

Threshold is illustrated in Figure 29.

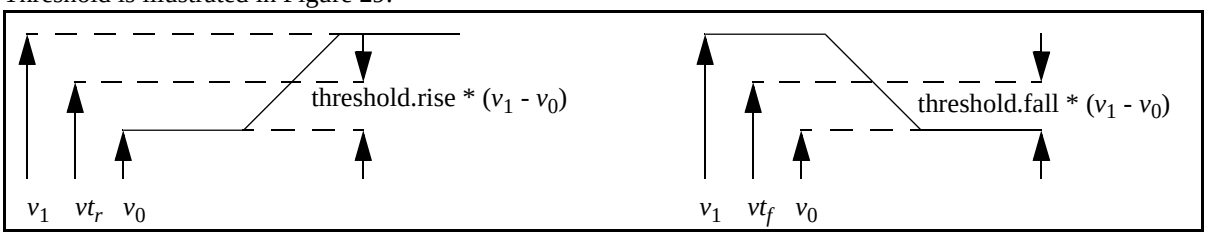


Figure 29—THRESHOLD measurement definition

The arithmetic model *threshold* can contain the arithmetic submodels *rise* and *fall* (see Section 11.21). If a timing-related arithmetic model refering to a *single event* (see Section 10.13.1) in the context of a declared *vector* (see Section 9.13) inherits a definition for threshold, the matching arithmetic submodel *rise* or *fall* shall apply according to the *single event*.

Note: The arithmetic submodel *rise* or *fall* is not necessary, if $vt_r = vt_f$.

Threshold can be specified in the context of a *from-to* statement (see Section 11.12) or in the context of a declared *pin* (see Section 9.5). As a child of a *from-to* statement, *threshold* shall apply to the parent arithmetic model of the *from-to* statement. As a child of a declared *pin*, *threshold* shall apply to the parent arithmetic model of a *from-to* statement, if the *from-to* statement acontains a *pin reference* annotation (see Section 11.13.2), refering to the declared pin.

Note: Threshold in the context of a declared pin does not apply to *slewrate* (see Section 11.11.5) or *pulsewidth* (see Section 11.11.9), since a from-to statement in the context of slewrate or pulsewidth can not contain a pin reference annotation.

11.11.14 NOISE and NOISE_MARGIN

The arithmetic models *noise* and *noise margin* shall be defined as shown in Semantics 124.

The purpose of the arithmetic model *noise* is to specify a noise measurement. The purpose of the arithmetic model *noise margin* is to specify a tolerance against noise.

Noise shall be a normalized quantity, according to the following mathematical definition.

```
KEYWORD NOISE = arithmetic_model { VALUETYPE = number ; }
SEMANTICS NOISE {
  CONTEXT {
    LIBRARY.LIMIT SUBLIBRARY.LIMIT CELL.LIMIT
    PIN PIN.LIMIT VECTOR VECTOR.LIMIT HEADER
  }
}
KEYWORD NOISE_MARGIN = arithmetic_model {
  VALUETYPE = number ;
}
SEMANTICS NOISE_MARGIN {
  CONTEXT { CLASS LIBRARY SUBLIBRARY CELL PIN VECTOR }
}
NOISE_MARGIN { MIN = 0; }
```

Semantics 124—Arithmetic models NOISE and NOISE_MARGIN

noise.low = $(vn - v_0) / (v_1 - v_0)$
noise.high = $(v_1 - vn) / (v_1 - v_0)$

where

v_0 is the nominal voltage level for the logic value logic zero,
v_1 is the nominal voltage level for the value logic one,
vn is a measured voltage level caused by noise

Note:
Noise on a signal with the logic value zero is positive or negative, respectively, if $vn > v_0$ or $vn < v_0$, respectively.
Noise on a signal with the logic value one is positive or negative, respectively, if $vn < v_1$ or $vn > v_1$, respectively.

Noise is illustrated in Figure 29.

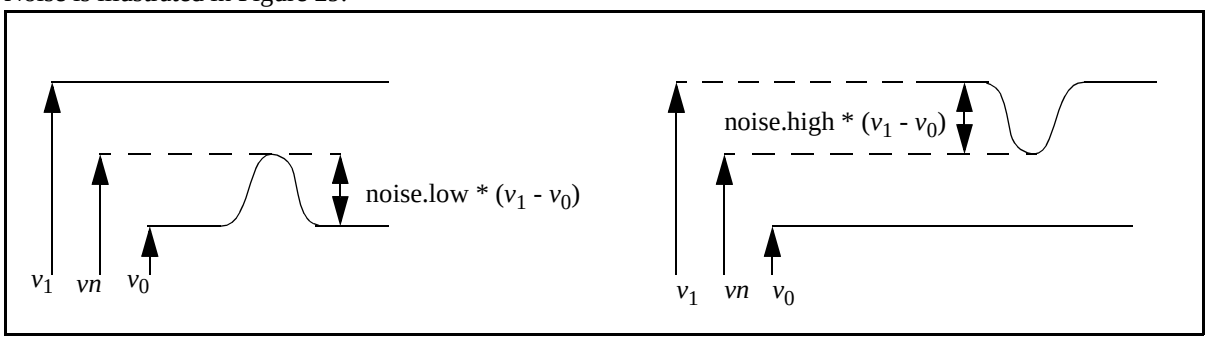


Figure 30—NOISE measurement definition

A distinction shall be made between a noise margin and a design limit for noise. A noise margin shall be defined as a value for noise that ensures that the logic value of a signal is recognizable. A design limit for noise shall be defined as a value of noise that is tolerable regardless whether the logic value is recognizable or not.

The distinction between a noise margin and a design limit for noise is illustrated in Figure 31.

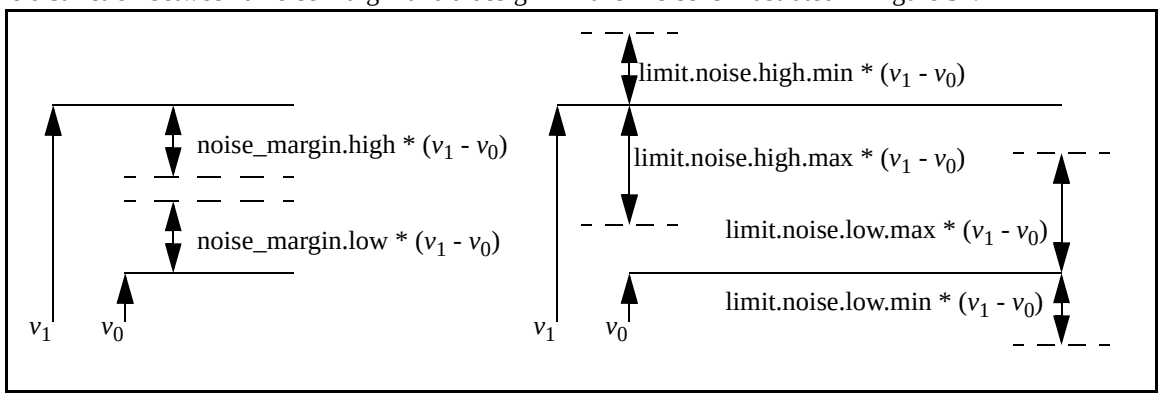


Figure 31—Definition of NOISE MARGIN and LIMIT for NOISE

Per definition, noise can be positive or negative, noise margin shall be positive, a maximum design limit for noise shall be positive, and a minimum design limit for noise shall be negative.

— NOISE in context of a declared *library* or *sublibrary* (see Section 9.1) or a declared *cell* (see Section 9.3)

The arithmetic model container *limit* (see Section 11.8.2) can be used to specify a design limit for noise. An arithmetic submodel *high*, *low* (see Section 11.21) can optionally be used.

A child shall inherit the design limit specification from its parent, unless a design limit is specified within the child. In particular, a sublibrary can inherit from a library. A cell can inherit from a sublibrary or from a library. A pin can inherit from a cell, a sublibrary or a library.

— NOISE in context of a declared *pin* (see Section 9.5)

A static noise measurement related to the pin can be described. An arithmetic submodel *high*, *low* can optionally be used.

A design limit for noise can be described in the same way as in the context of a *library*, a *sublibrary* or a *cell*.

— NOISE in context of a declared *vector* (see Section 9.13)

A noise measurement in response to a stimulus provided by the *vector* can be described. A *pin reference* annotation shall be used. A static noise measurement can be described using a *boolean expression* (see Section 10.9) as a stimulus. A transient noise measurement, i.e., either a waveform for noise or a peak value for noise, can be described using a *vector expression* (see Section 10.12) as stimulus.

A design limit for noise related to the stimulus can be specified using the arithmetic model container *limit*. A *pin reference* annotation shall be used.

— NOISE as *header arithmetic model* (see Syntax 89 in Section 11.4)

A noise that acts as a stimulus can be described. A *pin reference* annotation shall be used.

— NOISE MARGIN in context of a declared *class* (see Section 8.6)

A static noise margin can be specified. An arithmetic submodel *high*, *low* can optionally be used. A declared *pin* can inherit this specification by refering to the class.

— NOISE MARGIN in context of a declared *library* or *sublibrary* (see Section 9.1) or a declared *cell* (see Section 9.3) or a declared *pin* (see Section 9.5).

A static noise margin can be specified. The arithmetic submodels *high* or *low* can optionally be used.

A child shall inherit the noise margin specification from its parent, unless a noise margin is specified within the child. In particular, a sublibrary can inherit from a library. A cell can inherit from a sublibrary or from a library. A pin can inherit from a cell, a sublibrary or a library. Inheritance from a class by a pin shall take precedence over inheritance from a cell, a sublibrary or a library.

— NOISE MARGIN in the context of a declared *vector* (see Section 9.13)

A noise margin in the context of a stimulus given by the vector can be described. A *pin reference* annotation (see Section 11.13.6) shall be used.

A state-dependent noise margin can be described using a *boolean expression* (see Section 10.9) as stimulus.

A sensitivity window for a noise margin can be described using a *vector expression* (see Section 10.12) as stimulus. The arithmetic model time (see Section 11.11.1) shall be used as an *auxiliary arithmetic model* (see Section 11.6). A *from-to* statement (see Section 11.12) shall be associated with *time*.

A transient noise margin, i.e., a noise margin that depends on the timing characteristics of the stimulus can be described using a *vector expression* as stimulus and a timing-related arithmetic model, e.g. *pulsewidth* (see Section 11.11.9) or *slewrate* (see Section 11.11.5), as a *header arithmetic model* (see Syntax 89 in Section 11.4).

11.11.15 POWER and ENERGY

The arithmetic models *power* and *energy* shall be defined as shown in Semantics 125.

```
KEYWORD POWER = arithmetic_model { VALUETYPE = number; }
SEMANTICS POWER {
  CONTEXT {
    LIBRARY SUBLIBRARY CELL VECTOR
    CLASS.LIMIT CELL.LIMIT
  }
}
POWER { UNIT = MilliWatt; }
KEYWORD ENERGY = arithmetic_model { VALUETYPE = number; }
SEMANTICS ENERGY {
  CONTEXT { LIBRARY SUBLIBRARY CELL VECTOR }
}
ENERGY { UNIT = PicoJoule; }
```

Semantics 125—Arithmetic models POWER and ENERGY

The purpose of the arithmetic models power and energy is to specify the electrical power consumption of an electronic circuit.

— POWER in context of a declared *class* (see Section 8.6)

The arithmetic model container *limit* (see Section 11.8.2) can be used to specify a design limit for power consumption associated with a *class* with *usage* annotation value *supply-class* (see Section 9.7.16). A *measurement* annotation (see Section 11.13.7) shall be used.

— POWER in context of a declared *library* or *sublibrary* (see Section 9.1)

A *partial arithmetic model* (see Syntax 85 within Section 11.3) can be used to globally specify an *inheritable arithmetic model qualifier* (see Syntax 88 within Section 11.3) for power.

— POWER in context of a declared *cell* (see Section 9.3)

Power consumption of a cell or a design limit for power consumption of a cell can be described. A *measurement* annotation shall be used.

A *partial arithmetic model* can be used in the same way as in the context of *library* or *sublibrary.*

— POWER in context of a declared *vector* (see Section 9.13)

Power consumption related to a stimulus defined by the *vector* can be described. A *measurement* annotation shall be used.

— ENERGY in context of a declared *library* or *sublibrary* (see Section 9.1) or a declared *cell* (see Section 9.3)

A *partial arithmetic model* (see Syntax 85 within Section 11.3) can be used to globally specify an *inheritable arithmetic model qualifier* (see Syntax 88 within Section 11.3) for energy.

— ENERGY in context of a declared *vector* (see Section 9.13)

Energy consumption related to a stimulus defined by the *vector* can be described. Total energy consumption associated with different stimuli shall be additive, regardless whether the stimuli are mutually exclusive or not. Also, energy consumption shall be additive with power consumption, if the *measurement* annotation value *static* is associated with the latter.

11.12 FROM and TO statements

A *from-to* statement shall be defined as shown in Syntax 102.

```
from-to ::=
    from | to | from to
from ::=
  FROM { from-to_item { from-to_item } }
to ::=
  TO { from-to_item { from-to_item } }
from-to_item ::=
    PIN_reference_single_value_annotation
  | EDGE_NUMBER_single_value_annotation
  | THRESHOLD_arithmetic_model
```

Syntax 102—FROM and TO statements

The purpose of a *from* and a *to* statement is to define the start and end point, respectively, of a timing measurement. The timing measurement shall be applicable for digital signals.

A *from* and a *to* statement can contain a *pin reference* annotation (see Section 11.13.2), an *edge number* annotation (see Section 11.11.2) and a *threshold* arithmetic model (see Section 11.11.13).

A reference to a *single event* (see Section 10.13.1) is specified by the pin reference annotation in conjunction with the edge number annotation. The single event referenced within the *from* and *to* statement, respectively, shall be called *from-event* and *to-event*, respectively.

The from-and to-statements shall be subjected to the restriction shown in Semantics 126.

```
SEMANTICS FROM {
  CONTEXT {
    TIME DELAY RETAIN SLEWRATE PULSEWIDTH
    SETUP HOLD RECOVERY REMOVAL NOCHANGE ILLEGAL SKEW
  }
}
SEMANTICS TO {
  CONTEXT {
    TIME DELAY RETAIN SLEWRATE PULSEWIDTH
    SETUP HOLD RECOVERY REMOVAL NOCHANGE ILLEGAL SKEW
  }
}
```

Semantics 126— Restriction for FROM and TO statements

11.13 Annotations related to timing, power and signal integrity

Add lead-in text

11.13.1 EDGE_NUMBER annotation

An *edge number* annotation shall be defined as shown in .

```
KEYWORD EDGE_NUMBER = annotation {
  CONTEXT { arithmetic_model FROM TO }
  VALUETYPE = unsigned_integer ;
  DEFAULT = 0;
}
```

Semantics 127—EDGE_NUMBER annotation

The edge number annotation shall be a child of an *arithmetic model* (see Section 11.3) or a *from-to* statement (see Section 11.12).

The purpose of the edge number annotation is to specify a reference to a *single event* (see Section 10.13.1) within a vector expression. The vector expression shall be the name of a declared *vector.* The reference shall be established by using the edge number annotation in conjunction with a *pin reference* annotation (see Section 9.8.1). The pin reference annotation shall point to a *pin variable* (see Section 7.9) involved in the vector expression. The edge number annotation shall point to a single event on the pin variable. Every single event on a pin variable shall be counted in chronological order, starting with 0.

11.13.2 PIN reference and EDGE_NUMBER annotation for FROM and TO

A *pin reference* annotation shall be subjected to the restriction shown in Semantics 128.

```
SEMANTICS FROM.PIN = single_value_annotation {
  CONTEXT { TIME DELAY RETAIN SETUP HOLD
  RECOVERY REMOVAL NOCHANGE ILLEGAL }
}
SEMANTICS TO.PIN = single_value_annotation {
  CONTEXT { TIME DELAY RETAIN SETUP HOLD
  RECOVERY REMOVAL NOCHANGE ILLEGAL }
}
```

Semantics 128—Restriction for PIN reference annotation within FROM and TO

The purpose of the restriction is to define a reference to a single pin variable in the context of a *from-to* statement (see Section 11.12).

An *edge_number* annotation shall be subjected to the restriction shown in Semantics 127.

```
SEMANTICS FROM.EDGE_NUMBER = single_value_annotation {
  CONTEXT { TIME DELAY RETAIN SETUP HOLD
  RECOVERY REMOVAL NOCHANGE ILLEGAL }
}
SEMANTICS TO.EDGE_NUMBER = single_value_annotation {
  CONTEXT { TIME DELAY RETAIN SETUP HOLD
  RECOVERY REMOVAL NOCHANGE ILLEGAL }
}
```

Semantics 129—Restriction for EDGE_NUMBER annotation within FROM and TO

The purpose of the restriction is to define a reference to a *single event* (see Section 10.13.1) in the context of a *from-to* statement.

Example:

```
TIME { FROM { PIN=A; EDGE_NUMBER=1; } TO { PIN=B; EDGE_NUMBER=3; } }
```

The following Figure 32 illustrates the restriction using a timing diagram.

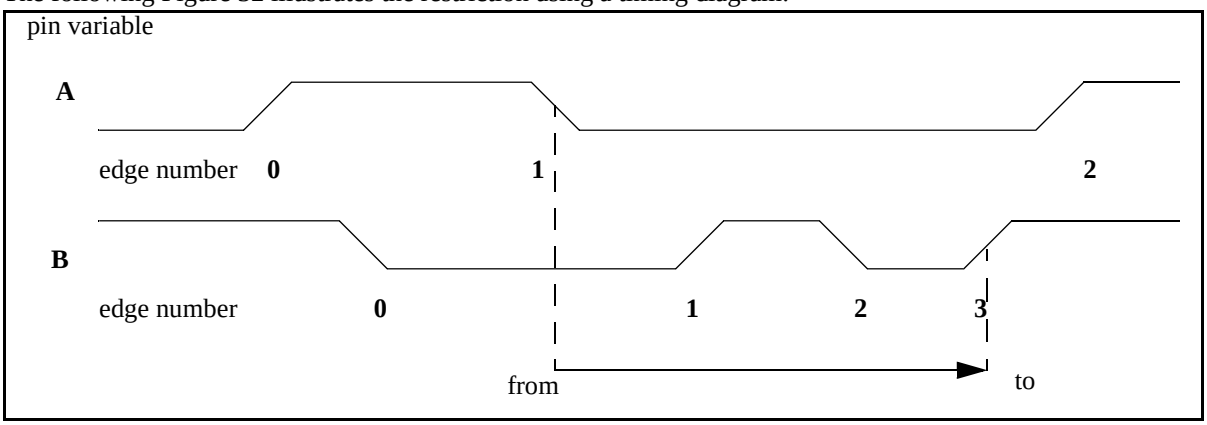


Figure 32—Illustration of PIN reference and EDGE NUMBER annotation within FROM and TO

A measurement is taken from edge number **1** at pin variable **A** to edge number **3** at pin variable **B**.

11.13.3 PIN reference and EDGE_NUMBER annotation for SLEWRATE

A *pin reference* annotation and an *edge_number* annotation shall be subjected to the restriction shown in .

```
SEMANTICS SLEWRATE.PIN = single_value_annotation ;
SEMANTICS SLEWRATE.EDGE_NUMBER = single_value_annotation ;
```

Semantics 130—Restriction for PIN reference and EDGE_NUMBER annotation within SLEWRATE

The purpose of the restriction is to define a reference to a single event for which *slewrate* (see Section 11.11.5) is measured.

11.13.4 PIN reference and EDGE_NUMBER annotation for PULSEWIDTH

A *pin reference* annotation and an *edge_number* annotation shall be subjected to the restriction shown in .

```
SEMANTICS PULSEWIDTH.PIN = single_value_annotation;
SEMANTICS PULSEWIDTH.EDGE_NUMBER = single_value_annotation;
```

Semantics 131—Restriction for PIN reference and EDGE_NUMBER annotation within PULSEWIDTH

The purpose of the restriction is to define a reference to a single event which is the leading edge of a pulse for which *pulsewidth* (see Section 11.11.9) is measured. The trailing edge shall be the following single event on the same pin.

11.13.5 PIN reference and EDGE_NUMBER annotation for SKEW

A *pin reference* annotation and an *edge number* annotation shall be subjected to the restriction shown in .

```
SEMANTICS SKEW.PIN = multi_value_annotation ;
SEMANTICS SKEW.EDGE_NUMBER = multi_value_annotation ;
```

Semantics 132—Restriction for PIN reference and EDGE_NUMBER annotation within SKEW

The purpose of the restriction is to define a reference to plural events, for which *skew* (see Section 11.11.12) is measured.

The number of annotation values within the *pin reference* and *edge number* annotation shall match. Subsequent annotation values shall correspond to each other. i.e., the first annotation value within the pin reference annotation shall correspond to the first annotation value within the edge number annotation, etc.

11.13.6 PIN reference annotation for NOISE and NOISE_MARGIN

A *pin reference* annotation shall be subjected to the restriction shown in .

```
SEMANTICS NOISE.PIN = single_value_annotation ;
SEMANTICS NOISE_MARGIN.PIN = single_value_annotation ;
```

Semantics 133—Restriction for PIN reference annotation within NOISE and NOISE MARGIN

The purpose of the restriction is to define a reference to a pin, for which *noise* or *noise margin* (see Section 11.11.14) is described.

11.13.7 MEASUREMENT annotation

A *measurement* annotation shall be defined as shown in Semantics 134.

```
KEYWORD MEASUREMENT = single_value_annotation {
  VALUETYPE = identifier ;
  VALUES {
    transient static average absolute_average rms peak
  }
  CONTEXT {
    ENERGY POWER CURRENT VOLTAGE JITTER
  }
}
```

Semantics 134—MEASUREMENT annotation

The purpose of the *measurement* annotation is to specify the mathematical definition of a temporal measurement.

The mathematical definition of the annotation values is shown in Table 97.

Table 97—MEASUREMENT annotation

Annotation value	Mathematical description
`transient`	*measurement* = $x(t)$
`static`	*measurement* = x, with x constant
`average`	$measurement = \frac{1}{T}\int_{t=0}^{t=T} x(t)dt$
`absolute_average`	$measurement = \frac{1}{T}\int_{t=0}^{t=T} \lvert x(t)\rvert dt$
`rms`	$measurement = \sqrt{\frac{1}{T}\int_{t=0}^{t=T} x^2(t)dt}$
`peak`	*measurement* = $\max(\max(x),-\min(x))$, with $x = x(t)$

The arithmetic model *time* (see Section 11.11.1) or *frequency* (see Section 11.11.2) shall be used as *auxiliary arithmetic model* (see Section 11.6), if the *measurement* annotation value is *average*, *absolute average*, or *rms*. The auxiliary arithmetic model *time* shall be interpreted as the integration time T in Table 97. The auxiliary arithmetic model frequency shall be interpreted as the repetition frequency f of the measurement, with f=1/T.

The auxiliary arithmetic model *time* can be used, if the parent arithmetic model is in the context of a declared *vector* (see Section 9.13) and the *measurement* annotation value is *peak*. Either a *from* or a *to* statement (see Section 11.12) can be used to specify the time interval between a *single event* (see Section 10.13.1) and the occurence of the measurement or vice-versa.

This is illustrated in Figure 33.

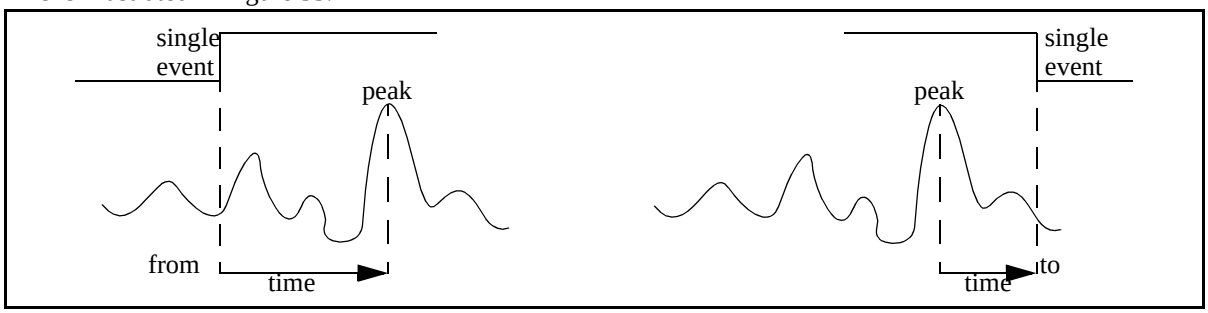


Figure 33—Illustration of peak measurement with FROM or TO statement

11.14 Arithmetic models for environmental conditions

11.14.1 PROCESS

The arithmetic model *process* shall be defined as shown in Semantics 135.

```
KEYWORD PROCESS = arithmetic_model {
  VALUETYPE = identifier ;
}
SEMANTICS PROCESS {
  CONTEXT {
    CLASS LIBRARY SUBLIBRARY CELL WIRE
    HEADER
    arithmetic_model
  }
}
PROCESS { DEFAULT = nom; TABLE { nom snsp snwp wnsp wnwp } }
```

Semantics 135—Arithmetic model PROCESS

The purpose of the arithmetic model *process* is to specify a dependency between an arithmetic model and a manufacturing process condition. A *partial arithmetic model* (see Syntax 85 within Section 11.3), a *header arithmetic model* (see Syntax 89 within Section 11.4), or an *auxiliary arithmetic model* (see Section 11.6) can be used.

The meaning of the predefined arithmetic values for *process* is explained in Table 98.

Table 98—Predefined arithmetic values for PROCESS

Value	Description
nom	NMOS and PMOS transistors with nominal strength
snsp	Strong NMOS transistor, strong PMOS transistor.

Table 98—Predefined arithmetic values for PROCESS (Continued)

Value	Description
snwp	Strong NMOS transistor, weak PMOS transistor.
wnsp	Weak NMOS transistor, strong PMOS transistor.
wnwp	Weak NMOS transistor, weak PMOS transistor.

11.14.2 DERATE_CASE

The arithmetic model *derate case* shall be defined as shown in Semantics 136.

```
KEYWORD DERATE_CASE = arithmetic_model {
  VALUETYPE = identifier ;
}
SEMANTICS DERATE_CASE {
  CONTEXT {
    CLASS LIBRARY SUBLIBRARY CELL WIRE
    HEADER
    arithmetic_model
  }
}
DERATE_CASE { DEFAULT = nom;
  TABLE { nom bccom wccom bcind wcind bcmil wcmil }}
}
```

Semantics 136—Arithmetic model DERATE_CASE

The purpose of the arithmetic model *derate case* is to specify a dependency between an arithmetic model and an environmental condition. A *partial* or a *full arithmetic model* (see Syntax 85, Syntax 86 within Section 11.3), a *header arithmetic model* (see Syntax 89 within Section 11.4), or an *auxiliary arithmetic model* (see Section 11.6) can be used.

The meaning of the predefined arithmetic values for *derate case* is explained in Table 99.

Table 99—Predefined arithmetic values for DERATE CASE

Derating case	Description
nom	Nominal environmental condition
bccom	Best case commercial condition
bcind	Best case industrial condition
bcmil	Best case military condition
wccom	Worst case commercial condition
wcind	Worst case industrial condition
wcmil	Worst case military condition

A full arithmetic model can be used to describe the dependency between the condition and its defining parameters (e.g., process, voltage, temperature).

11.14.3 TEMPERATURE

The arithmetic model *temperature* shall be defined as shown in Semantics 137.

```
KEYWORD TEMPERATURE = arithmetic_model {
  VALUETYPE = number ;
}
SEMANTICS TEMPERATURE {
  CONTEXT {
    CLASS LIBRARY SUBLIBRARY CELL WIRE
    LIMIT
    HEADER
    arithmetic_model
  }
}
TEMPERATURE { UNIT = 1DegreeCelsius; MIN = -273; }
```

Semantics 137—Arithmetic model TEMPERATUREt

The purpose of the arithmetic model *temperature* is to specify a dependency between an arithmetic model and an environmental temperature. Temperature shall be measured in degrees Celsius. A *partial* or a *full arithmetic model* (see Syntax 85, Syntax 86 within Section 11.3), a *header arithmetic model* (see Syntax 89 within Section 11.4), or an *auxiliary arithmetic model* (see Section 11.6) can be used.

11.15 Arithmetic models for electrical circuits

11.15.1 VOLTAGE

The arithmetic model *voltage* shall be defined as shown in .

```
KEYWORD VOLTAGE = arithmetic_model {
  VALUETYPE = number ;
}
SEMANTICS VOLTAGE {
  CONTEXT {
    CLASS LIBRARY SUBLIBRARY CELL PIN WIRE VECTOR HEADER
    CLASS.LIMIT CELL.LIMIT PIN.LIMIT VECTOR.LIMIT
  }
}
VOLTAGE { UNIT = 1Volt; }
```

Semantics 138—Arithmetic model VOLTAGE

The purpose of the arithmetic model *voltage* is to specify either a measurement of electrical voltage or an electrical component that can be modeled as a voltage source.

— VOLTAGE in context of a declared *class* (see Section 8.6)

An environmental voltage can be specified. An arithmetic submodel *high*, *low* (see Section 11.21) can optionally be used. A *pin* (see Section 9.5) can inherit this specification by refering to the class. In particular, a *supply class* annotation (see Section 9.7.16) or a *connect class* annotation (see Section 9.7.19) can be used for this purpose.

— VOLTAGE in context of a declared *library* or *sublibrary* (see Section 9.1)

A *partial arithmetic model* (see Syntax 85 within Section 11.3) can be used to globally specify an *inheritable arithmetic model qualifier* (see Syntax 88 within Section 11.3) or a *trivial min-max* statement (see Section 95 within Section 11.5) for voltage.

— VOLTAGE in context of a declared *cell* (see Section 9.3)

A voltage source that is part of the implementation of a cell can be specified. A *node reference* annotation (see Section 11.16.1) shall be used.

A design limit for a voltage related to the cell can be specified using the arithmetic model container *limit* (see Section 11.8.2). Either a *pin reference* annotation (see Section 11.16.3) or a *model reference* annotation (see Section 11.9.5) shall be used.

A *partial arithmetic model* can be used in the same way as in the context of *library* or *sublibrary.*

— VOLTAGE in context of a declared *pin* (see Section 9.5)

An environmental voltage related to a pin, e.g., a supply voltage, can be described. An arithmetic submodel *high*, *low* can optionally be used.

A design limit for a voltage that can be applied to the pin can be described using the arithmetic model container *limit*.

— VOLTAGE in context of a declared *wire* (see Section 9.9)

A voltage source within an electrically equivalent circuit used for interconnect analysis can be specified. A *node reference* annotation shall be used.

— VOLTAGE in context of a declared *vector* (see Section 9.13)

A voltage measurement in response to a stimulus provided by the *vector* can be described. Either a *pin reference* annotation or a *model reference* annotation shall be used.

A design limit for a voltage related to the stimulus can be specified using the arithmetic model container *limit* (see Section 11.8.2). Either a *pin reference* annotation or a *model reference* annotationshall be used.

— VOLTAGE as *header arithmetic model* (see Syntax 89 in Section 11.4)

A voltage that acts as a stimulus can be described. Either a *pin reference* annotation or a *model reference* annotation shall be used. In particular, if a *wire instantiation* (see Section 10.15) is present, a reference to a voltage source specified within the declared wire can be established.

11.15.2 CURRENT

The arithmetic model *current* shall be defined as shown in .

The purpose of the arithmetic model *current* is to specify either a measurement of electrical current or an electrical component that can be modeled as a current source.

```
KEYWORD CURRENT = arithmetic_model {
  VALUETYPE = number ;
}
SEMANTICS CURRENT {
  CONTEXT {
    LIBRARY SUBLIBRARY CELL WIRE VECTOR HEADER
    CELL.LIMIT VECTOR.LIMIT
    LAYER.LIMIT VIA.LIMIT RULE.LIMIT
  }
}
CURRENT { UNIT = MilliAmpere; }
```

Semantics 139—Arithmetic model CURRENT

— CURRENT in context of a declared *library* or *sublibrary* (see Section 9.1)

A *partial arithmetic model* (see Syntax 85 within Section 11.3) can be used to globally specify an *inheritable arithmetic model qualifier* (see Syntax 88 within Section 11.3) for current.

— CURRENT in context of a declared *cell* (see Section 9.3)

A current source that is part of the implementation of a cell can be specified. A *node reference* annotation (see Section 11.16.1) shall be used.

A design limit for a current related to the cell can be specified using the arithmetic model container *limit* (see Section 11.8.2). Either a *pin reference* annotation (see Section 11.16.3) or a *model reference* annotation (see Section 11.9.5) or a *component reference* annotation (see Section 11.16.2) shall be used.

A *partial arithmetic model* can be used in the same way as in the context of *library* or *sublibrary.*

— CURRENT in context of a declared *wire* (see Section 9.9)

A current source within an electrically equivalent circuit used for interconnect analysis can be specified. A *node reference* annotation shall be used.

— CURRENT in context of a declared *layer* (see Section 9.15), a declared *via* (see Section 9.17), or a declared *rule* (see Section 9.19)

A design limit for current can be specified using the arithmetic model container *limit*. A *measurement* annotation (see Section 11.13.7) shall be used.

In the context of a layer, the current shall flow through a general layout segment created by that layer. In the context of a via or in the context of a rule, the current shall flow through a particular layout segment in context of other layout segments described within the via or within the rule. A *pattern reference* annotation (see Section 11.20.9) shall be used.

— CURRENT in context of a declared *vector* (see Section 9.13)

A current measurement in response to a stimulus provided by the *vector* can be described. Either a *pin reference* annotation or a *model reference* annotation or a *component reference* annotation shall be used.

A design limit for a current related to the stimulus can be specified using the arithmetic model container *limit* . Either a *pin reference* annotation or a *model reference* annotation or a *component reference* annotation shall be used.

— CURRENT as *header arithmetic model* (see Syntax 89 in Section 11.4)

A current that acts as a stimulus can be described. Either a *pin reference* annotation or a *model reference* annotation or a *component reference* annotation shall be used. In particular, if a *wire instantiation* (see Section 10.15) is present, a reference to a current source or to a component specified within the declared wire can be established.

11.15.3 CAPACITANCE

The arithmetic model *capacitance* shall be defined as shown in Semantics 134.

```
KEYWORD CAPACITANCE = arithmetic_model {
  VALUETYPE = number ;
  SI_MODEL = CAPACITANCE ;
}
SEMANTICS CAPACITANCE {
  CONTEXT {
    LIBRARY SUBLIBRARY CELL CELL.LIMIT PIN PIN.LIMIT
    WIRE LAYER RULE VECTOR HEADER
  }
}
CAPACITANCE { UNIT = PicoFarad; MIN = 0; }
```

Semantics 140—Arithmetic model CAPACITANCE

The purpose of the arithmetic model *capacitance* is to describe either a measurement of electrical capacitance or an electrical component that can be modeled as a capacitor.

— CAPACITANCE in context of a declared *library* or *sublibrary* (see Section 9.1)

A *partial arithmetic model* (see Syntax 85 within Section 11.3) can be used to globally specify an *inheritable arithmetic model qualifier* (see Syntax 88 within Section 11.3) for capacitance.

— CAPACITANCE in context of a declared *cell* (see Section 9.3)

A capacitor that is part of the implementation of a cell can be described. A *node reference* annotation (see Section 11.16.1) shall be used.

A design limit for a capacitor related to the cell can be specified using the arithmetic model container *limit* (see Section 11.8.2). Either a *pin reference* annotation (see Section 11.16.3) or a *model reference* annotation (see Section 11.9.5) shall be used.

A *partial arithmetic model* can be used in the same way as in the context of *library* or *sublibrary.*

— CAPACITANCE in context of a declared *pin* (see Section 9.5)

The self-capacitance of a pin can be described as a child of a *pin*. An arithmetic submodel *rise*, *fall*, *high*, *low* (see Section 11.21) can optionally be used.

A design limit for a capacitance that can be connected to the pin can be specified using the arithmetic model container *limit* as a child of a pin.

— CAPACITANCE in context of a declared *wire* (see Section 9.9)

A capacitance with or without *node reference* annotation can be described.

A capacitance with node reference annotation shall represent a capacitor within an electrically equivalent circuit used for interconnect analysis. If the wire is a child of the cell and a permanent connectivity between pins and nodes of the cell and the nodes of the wire exists, the capacitance shall represent a parasitic capacitor within the cell. Interconnect analysis shall either use a (lumped) self-capacitance of a pin or a (distributed) parasitic capacitor connected to a pin.

A capacitance without node reference annotation shall represent an estimation model for interconnect capacitance.

— CAPACITANCE in context of a declared *layer* (see Section 9.15)

An estimation model for capacitance of a general layout segment can be described. An arithmetic submodel *horizontal*, *vertical*, *acute*, *obtuse* (see Section 11.22) can optionally be used.

— CAPACITANCE in context of a declared *rule* (see Section 9.19)

An estimation model for capacitance created by a particular layout pattern can be described.

— CAPACITANCE in context of a declared *vector* (see Section 9.13)

An *effective capacitance* can be described. Either a *pin reference* annotation or a *model reference* annotation shall be used. The effective capacitance shall be interpreted as a virtual capacitor, which, under the specific stimulus provided by the vector, behaves in a similar way as the actual load circuit.

— CAPACITANCE as *header arithmetic model* (see Syntax 89 in Section 11.4)

A capacitance as a dimension of an arithmetic model can be described. Either a *pin reference* annotation or a *model reference* annotation shall be used.

The *pin reference* annotation shall be used to specify a lumped load capacitance. The self-capacitance of the pin shall not be included in the load capacitance.

The *model reference* annotation shall be used to refer to another capacitor. In particular, if a *wire instantiation* (see Section 10.15) is present, a reference to a capacitor described within the declared *wire* can be established.

11.15.4 RESISTANCE

The arithmetic model *resistance* shall be defined as shown in .

The purpose of the arithmetic model *resistance* is to describe either a measurement of electrical resistance or an electrical component that can be modeled as a resistor.

— RESISTANCE in context of a declared *library* or *sublibrary* (see Section 9.1)

A *partial arithmetic model* (see Syntax 85 within Section 11.3) can be used to globally specify an *inheritable arithmetic model qualifier* (see Syntax 88 within Section 11.3) for resistance.

```
KEYWORD RESISTANCE = arithmetic_model {
  VALUETYPE = number ;
  SI_MODEL = RESISTANCE ;
}
SEMANTICS RESISTANCE {
  CONTEXT {
    LIBRARY SUBLIBRARY CELL WIRE LAYER RULE
    CELL.LIMIT VECTOR HEADER
  }
}
RESISTANCE { UNIT = KiloOhm; MIN = 0; }
```

Semantics 141—Arithmetic model RESISTANCE

— RESISTANCE in context of a declared *cell* (see Section 9.3)

A resistor that is part of the implementation of a cell can be described. A *node reference* annotation (see Section 11.16.1) shall be used.

A design limit for a resistor related to the cell can be specified using the arithmetic model container *limit* (see Section 11.8.2). A *model reference* annotation (see Section 11.9.5) shall be used.

A *partial arithmetic model* can be used in the same way as in the context of *library* or *sublibrary.*

— RESISTANCE in context of a declared *wire* (see Section 9.9)

A resistance with or without *node reference* annotation can be described.

A resistance with node reference annotation shall represent a resistor within an electrically equivalent circuit used for interconnect analysis. If the wire is a child of the cell and a permanent connectivity between pins and nodes of the cell and the nodes of the wire exists, the resistance shall represent a parasitic resistor within the cell.

A resistance without node reference annotation shall represent an estimation model for interconnect resistance.

— RESISTANCE in context of a declared *layer* (see Section 9.15)

An estimation model for resistance of a general layout segment can be described. An arithmetic submodel *horizontal*, *vertical*, *acute*, *obtuse* (see Section 11.22) can optionally be used.

— RESISTANCE in context of a declared *rule* (see Section 9.19)

An estimation model for resistance created by a particular layout pattern can be described.

— RESISTANCE in context of a declared *vector* (see Section 9.13)

A *driver resistance* can be described. Either a *pin reference* annotation or a *model reference* annotation shall be used. The driver resistance shall be interpreted as part of an electrically equivalent circuit, which, under the specific stimulus provided by the vector, behaves in a similar way as the actual driver circuit.

— RESISTANCE as *header arithmetic model* (see Syntax 89 in Section 11.4)

A resistance as a dimension of an arithmetic model can be described. A *model reference* annotation shall be used. In particular, if a *wire instantiation* (see Section 10.15) is present, a reference to a resistor described within the declared *wire* can be established.

11.15.5 INDUCTANCE

The arithmetic model *inductance* shall be defined as shown in .

```
KEYWORD INDUCTANCE = arithmetic_model {
  VALUETYPE = number ;
  SI_MODEL = INDUCTANCE ;
}
SEMANTICS INDUCTANCE {
  CONTEXT {
    LIBRARY SUBLIBRARY CELL WIRE RULE
    CELL.LIMIT HEADER
  }
}
INDUCTANCE { UNIT = MicroHenry; MIN = 0; }
```

Semantics 142—Arithmetic model INDUCTANCE

The purpose of the arithmetic model *inductance* is to describe either a measurement of electro-magnetic inductance or an electro-magnetic component that can be modeled as an inductor (i.e., a component with self-inductance) or a transformator (i.e., a component with mutual inductance).

— INDUCTANCE in context of a declared *library* or *sublibrary* (see Section 9.1)

A *partial arithmetic model* (see Syntax 85 within Section 11.3) can be used to globally specify an *inheritable arithmetic model qualifier* (see Syntax 88 within Section 11.3) for inductance.

— INDUCTANCE in context of a declared *cell* (see Section 9.3)

An inductor or a transformator that is part of the implementation of a cell can be described. A *node reference* annotation (see Section 11.16.1) shall be used.

A design limit for an inductor or for a transformator related to the cell can be specified using the arithmetic model container *limit* (see Section 11.8.2). A *pin reference* annotation (see Section 11.16.3) or a *model reference* annotation (see Section 11.9.5) shall be used.

A *partial arithmetic model* can be used in the same way as in the context of *library* or *sublibrary.*

— INDUCTANCE in context of a declared *wire* (see Section 9.9)

An inductance with or without *node reference* annotation can be described.

An inductance with node reference annotation shall represent a self-inductance or a mutual inductance within an electrically equivalent circuit used for interconnect analysis. If the wire is a child of the cell and a permanent connectivity between pins and nodes of the cell and the nodes of the wire exists, the inductance shall represent a parasitic self-inducatnace or mutual inductance within the cell.

An inductance without node reference annotation shall represent an estimation model for interconnect self-inductance.

— INDUCTANCE in context of a declared *rule* (see Section 9.19)

An estimation model for inductance created by a particular layout pattern can be described.

— INDUCTANCE as *header arithmetic model* (see Syntax 89 in Section 11.4)

An inductance as a dimension of an arithmetic model can be described. A *model reference* annotation shall be used. In particular, if a *wire instantiation* (see Section 10.15) is present, a reference to a self-inductance or to a mutual inductance described within the declared *wire* can be established.

11.16 Annotations for electrical circuits

11.16.1 NODE reference annotation for electrical circuits

The *node reference* annotation (see Section 9.12.1) shall be subjected to restrictions defined in the following.

```
SEMANTICS VOLTAGE.NODE = multi_value_annotation {
  CONTEXT { CELL WIRE } }
SEMANTICS CURRENT.NODE = multi_value_annotation {
  CONTEXT { CELL WIRE } }
SEMANTICS CAPACITANCE.NODE = multi_value_annotation {
  CONTEXT { CELL WIRE } }
SEMANTICS RESISTANCE.NODE = multi_value_annotation {
  CONTEXT { CELL WIRE } }
SEMANTICS INDUCTANCE.NODE = multi_value_annotation {
  CONTEXT { CELL WIRE } }
```

Semantics 143—Restrictions for NODE reference annotation

The purpose of a node reference annotation with these restrictions is to specify the connectivity of an electrical component within an electrical circuit.

The following restrictions shall further apply:

a) An arithmetic model with a node reference annotation shall always have an ALF name.
b) A node annotation associated with the arithmetic model *voltage* shall have two values, representing the terminal nodes of a voltage source. The defined polarity of the first and the second terminal shall be positive and negative, respectively.
c) A node annotation associated with the arithmetic model *current* shall have two values, representing the terminal nodes of a current source. The defined flow of the current shall be from the first to the second terminal.
d) A node annotation associated with the arithmetic model *capacitance* shall have two values, representing the terminal nodes of a capacitor.
e) A node annotation associated with the arithmetic model *resistance* shall have two values, representing the terminal nodes of a resistor.
f) A node annotation associated with the arithmetic model *inductance* shall have either two values or four values. Two values shall represent the terminal nodes of an inductor. Four values shall represent the terminal nodes of two coupled inductors. The first two values shall represent the terminals accross which an induced voltage is observed. The last two values shall represent the terminals accross which a controlling current flows.

The electrical components and their terminals are illustrated in the following Figure 34.

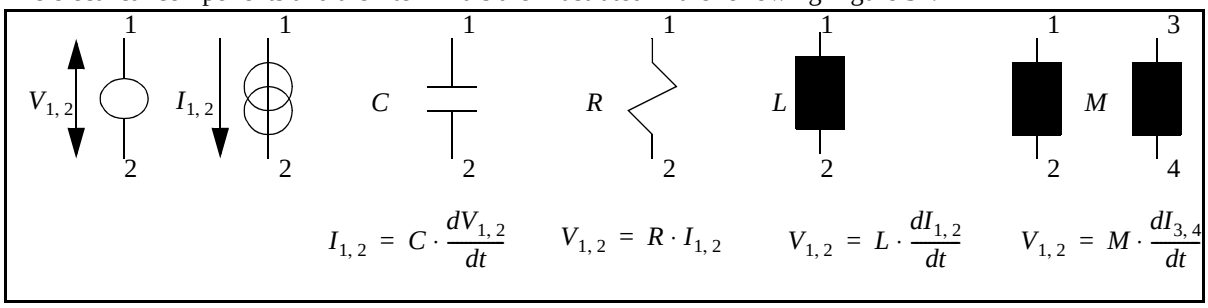


Figure 34—Electrical components and their terminals

The numbers in Figure 34 indicate the first, second, third and fourth node annotation values. However, the node annotation values shall be the ALF names of declared nodes.

11.16.2 COMPONENT reference annotation

A *component* reference annotation shall be defined as shown in Semantics 144.

```
KEYWORD COMPONENT = single_value_annotation {
  CONTEXT { CURRENT POWER ENERGY }
  VALUETYPE = identifier ;
  REFERENCETYPE {
    CURRENT VOLTAGE CAPACITANCE RESISTANCE INDUCTANCE
  }
}
```

Semantics 144—COMPONENT annotation

The purpose of the component reference annotation is to relate the arithmetic model *current* (see Section 11.15.2), *power* or *energy* (see Section 11.11.15) to an electrical component.

Electrical current shall flow through an electrical component with two terminals, i.e., a voltage source, a current source, a capacitor, a resistor, or an inductor. The defined flow of the current shall be from the first terminal to the second terminal.

Electrical power or energy shall be supplied by a voltage source or by a current source, stored in a capacitor or in an inductor and dissipated in a resistor. A negative value shall mean that a voltage source or a current source is a sink of power or energy rather than a source, that a capacitor or an inductor releases energy or power, or that a resistor virtually supplies power.

Note: A resistor that supplies power is physically impossible. However, certain active electronic circuits, for example a NIC (Negative Impedance Convertor), can be modeled using a “negative” resistor. The electrical energy “supplied” by the “negative” resistor is dissipated in other parts of the electronic circuit.

11.16.3 PIN reference annotation for electrical circuits

The *pin reference* annotation (see Section 9.7.1) shall be subjected to restrictions defined in the following.

The purpose of a *pin reference* annotation for electrical circuits is to specify an assocation between an electrical component with two terminals and a *pin variable*, i.e., a declared *pin*, *port* or *node* (see Section 7.9).

```
SEMANTICS VOLTAGE.PIN = single_value_annotation {
  CONTEXT { VECTOR VECTOR.LIMIT HEADER } }
SEMANTICS CURRENT.PIN = single_value_annotation {
  CONTEXT { VECTOR VECTOR.LIMIT HEADER } }
SEMANTICS CAPACITANCE.PIN = single_value_annotation {
  CONTEXT { VECTOR HEADER } }
SEMANTICS RESISTANCE.PIN = single_value_annotation {
  CONTEXT { VECTOR } }
```

Semantics 145—PIN reference annotation

a) A pin reference annotation associated with the arithmetic model *voltage* shall specify a connection between a pin, port or node and a voltage meter. The terminal with defined positive polarity shall be connected to the pin, port or node. The terminal with defined negative polarity shall be connected to ground.

b) A pin reference annotation associated with the arithmetic model *current* shall specify a connection between a pin, port or node and a current meter. The flow of the current shall be defined by the *flow* annotation (see Section 11.16.4).

c) A pin reference annotation associated with the arithmetic model *capacitance* shall specify a connection between a pin, port or node and one terminal of a capacitor. The other terminal of the capacitor shall be connected to ground. The capacitor shall represent either a *load capacitance* or an *effective capacitance*.

d) A pin reference annotation associated with the arithmetic model *resistance* specify a connection between a pin and one terminal of a resistor. The other terminal of the resistor shall be connected to a virtual voltage source. The resistor shall represent a *driver resistance*.

An electrical component can be associated with an *input pin* or with an *output pin*.

A node with *nodetype* annotation value *receiver* (see Section 9.12.2), a pin with *direction* annotation value *input* (see Section 9.7.5), a port or a node connected to such a pin shall be consider an *input pin*.

The association between electrical components and an input pin involves a model of a *stimulus* and a model of a *receiver circuit*, as illustrated in Figure 35.

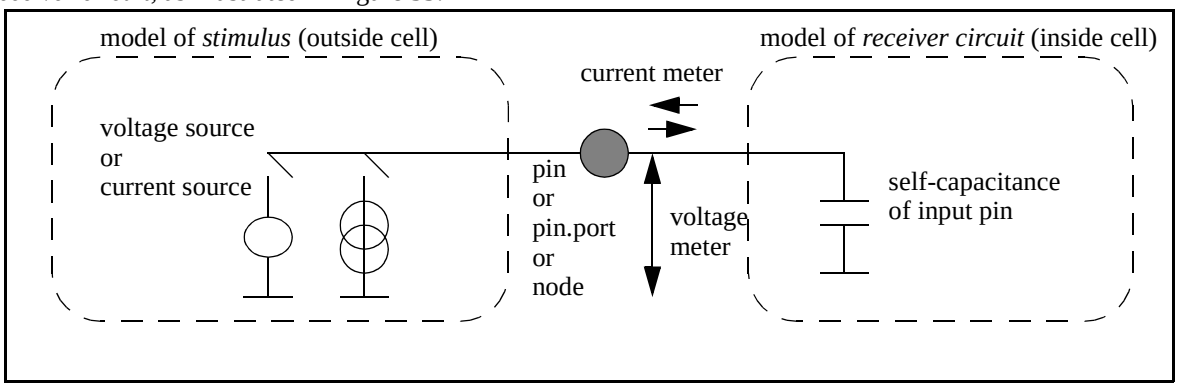


Figure 35—Association between electrical components and an input pin

A node with *nodetype* annotation value *driver* (see Section 9.12.2), a pin with direction annotation value *output* (see Section 9.7.5), a port or a node connected to such a pin shall be consider an output pin.

The association between electrical components and an output pin involves a model of a *driver circuit* and a model of a *load circuit*, as illustrated in Figure 35.

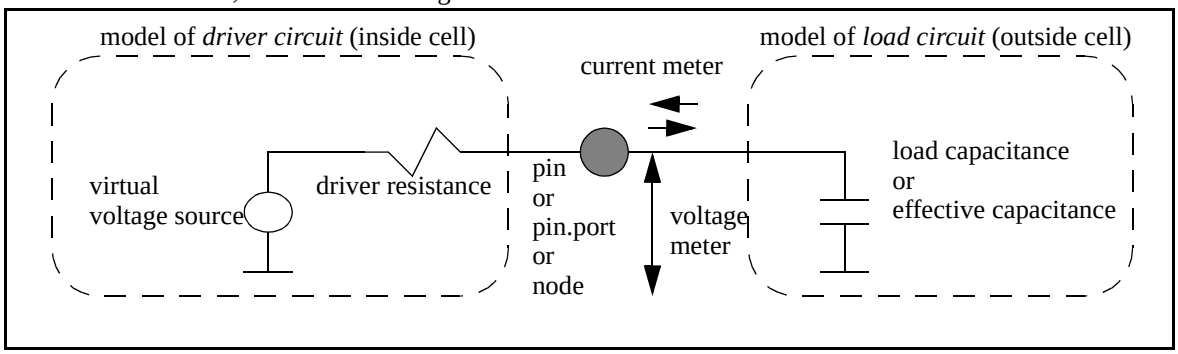


Figure 36—Association between electrical components and an output pin

Note: In order to describe a more complex model for a stimulus, a load circuit, a driver circuit or a receiver circuit, an electrical component in context of a declared wire can be used, as described in Section 11.15.

11.16.4 FLOW annotation

A *flow* annotation shall be defined as shown in Semantics 146.

```
KEYWORD FLOW = single_value_annotation {
  CONTEXT = CURRENT ;
  VALUETYPE = identifier ;
  VALUES { in out }
  DEFAULT = in;
}
```

Semantics 146—FLOW annotation

The purpose of the flow annotation is to specify the defined measurement direction of a current in conjunction with a *pin reference* annotation (see Section 11.16.3).

The meaning of the annotation values is shown in .

Table 100—FLOW annotation

Annotation value	Description
`in`	The defined flow of the current is from outside the cell to inside the cell.
`out`	The defined flow of the current is from inside the cell to outside the cell.

Note: The flow annotation is not applicable in conjunction with a *node reference* annotation (see Section 11.16.1) or a *component reference* annotation (see Section 11.16.2), since the direction of current measurement is already defined by the order of terminals of the electrical component.

11.17 Miscellaneous arithmetic models

11.17.1 DRIVE STRENGTH

The arithmetic model *drive strength* shall be defined as shown in Semantics 147.

```
KEYWORD DRIVE_STRENGTH = arithmetic_model {
  VALUETYPE = unsigned_number ;
}
SEMANTICS DRIVE_STRENGTH {
  CONTEXT { CLASS LIBRARY SUBLIBRARY CELL PIN PINGROUP }
}
```

Semantics 147—Arithmetic model DRIVE_STRENGTH

The purpose of the arithmetic model *drive strength* is to specify an abstract, unit-less measure for drivability associated with a *primitive circuit* or a *compound circuit*.

A *cell* (see Section 9.3) shall be considered either a *primitive circuit* or a compound circuit, depending on its *celltype* annotation (see Section 9.4.2). In case of a primitive circuit, drive strength can be a child of a cell. In case of a compound circuit, drive strength can be a child of a *pin* (see Section 9.5) or a *pingroup* (see Section 9.6).

A cell with *celltype* annotation value *buffer*, *combinational*, *multiplexor*, *flipflop*, or *latch* shall be considered a primitive circuit. A cell with *celltype* annotation value *memory*, *block*, or *core* shall be considered a compound circuit.

A *partial arithmetic model* (see Syntax 85 within Section 11.3) in the context of a *class* (see Section 8.6), a *library* or a *sublibrary* (see Section 9.1) can be used to globally specify a set of discrete values or a range of values for drive strength, using a *table* statement (see Syntax 91 within Section 11.4) or a trivial *min-max* statement (see Syntax 95 within Section 11.5), respectively.

11.17.2 SWITCHING_BITS with PIN reference annotation

Thearithmetic model *switching bits* shall be defined as shown in Semantics 148.

```
KEYWORD SWITCHING_BITS = arithmetic_model {
  VALUETYPE = unsigned_integer ;
}
SEMANTICS SWITCHING_BITS {
  CONTEXT { VECTOR.POWER.HEADER VECTOR.ENERGY.HEADER }
}
SEMANTICS SWITCHING_BITS.PIN = single_value_annotation;
```

Semantics 148—Arithmetic model SWITCHING_BITS

The purpose of the arithmetic model *switching bits* is to specify the number of binary value changes during a *single event* (see Section 10.13.1) on a vectorized *pin* (see Section 9.5) or a *pingroup* (see Section 9.6) .

Drive strength can be used as *header arithmetic model* (see Syntax 89 in Section 11.4) for calculation of *power* or *energy* (see Section 11.11.15) in context of a *vector* (see Section 9.13).

The *pin reference* annotation (see Section 9.7.1) shall be used.

11.18 Arithmetic models related to structural implementation

11.18.1 CONNECTIVITY

The arithmetic model *connectivity* shall be defined as shown in Semantics 149.

```
KEYWORD CONNECTIVITY = arithmetic_model {
  VALUETYPE = boolean ;
  VALUES { 1 0 ? }
}
SEMANTICS CONNECTIVITY {
  CONTEXT { LIBRARY SUBLIBRARY CELL RULE ANTENNA HEADER }
}
```

Semantics 149—Arithmetic model CONNECTIVITY

The purpose of the arithmetic model *connectivity* is to specify an actual connection or a requirement for a connection between physical objects.

Either a *table* statement (see Syntax 91 in Section 11.4) or a *between* annotation (see Section 11.20.2) shall be used to establish a relation between physical objects and the arithmetic model *connectivity.*

The interpretation of *connectivity* as an actual connection or as a requirement for a connection shall be specified by the connect-rule annotation (see Section 11.20.1).

The interpretation of the boolean values is specified in the following Table 101.

Table 101—Boolean values for CONNECTIVITY

Boolean value	Interpretation as actual connection	Interpretation as requirement for a connection
1	Connection exists.	Requirement is true.
0	Connection does not exist.	Requirement is false.
?	Connection is not relevant.	Requirement is not relevant.

Note: The boolean value "?" is *non-assignable* (see Section 10.10.3) and can therefore only be used, if the connectivity is modeled as a *table* (see Syntax 91 in Section 11.4).

11.18.2 DRIVER and RECEIVER

The arithmetic models *driver* and *receiver* shall be defined as shown in Semantics 150.

The purpose of the *header arithmetic model* (see Syntax 89 within Section 11.4) *driver* or *receiver* is to specify a dependency between *connectivity* (see Section 11.18.1) and a declared *class* (see Section 8.6) with *usage* annotation value *connect-class* (see Section 8.7.2, Section 9.7.19).

The header arithmetic model driver or receiver shall contain a *table* statement (see Syntax 91 in Section 11.4). The parent arithmetic model *connectivity* shall contain either a one-dimensional lookup table involving either dimen-

```
KEYWORD DRIVER = arithmetic_model {
  VALUETYPE = identifier ;
  REFERENCETYPE = CLASS ;
}
SEMANTICS DRIVER { CONTEXT = CONNECTIVITY.HEADER; }
KEYWORD RECEIVER = arithmetic_model {
  VALUETYPE = identifier ;
  REFERENCETYPE = CLASS ;
}
SEMANTICS RECEIVER { CONTEXT = CONNECTIVITY.HEADER; }
```

Semantics 150— Arithmetic models DRIVER and RECEIVER

sion driver or receiver, or alternatively a two-dimensional lookup table involving both dimensions driver and receiver.

A declared *pin* (see Section 9.5) shall be subjected to a connection with another pin, if a connect-class annotation exists for both pins, and the respective connect-class annotation values are found in a table statement within the header arithmetic model driver or receiver.

The association of a pin with the dimension driver or receiver shall depend on the *direction* annotation value (see Section 9.7.5). A pin with direction annotation value *input* shall be associated with the dimension *receiver.* A pin with direction annotation value *output* shall be associated with the dimension driver. A pin with direction annotation value *both* shall be associated with both dimensions driver and receiver.

Example:

```
CLASS Normal { USAGE = CONNECT_CLASS; }
CLASS Special { USAGE = CONNECT_CLASS; }
CONNECTIVITY Example1 {
    HEADER { DRIVER { Normal Special } }
    TABLE { 0 1 }
}
CONNECTIVITY Example2 {
    HEADER {
         DRIVER { Normal Special } }
         RECEIVER { Special Normal } }
    }
    TABLE { 0 1 1 0 }
}
```

Example1 specifies the following:

A connection between an output pin and another output pin associated with *Normal* is false.
A connection between an output pin and another output pin associated with *Special* is true.

Example2 specified the following:

A connection between an output pin associated with *Normal* and an input pin associated with *Special* is false.
A connection between an output pin associated with *Special* and an input pin associated with *Special* is true.
A connection between an output pin associated with *Normal* and an input pin associated with *Normal* is true.
A connection between an output pin associated with *Special* and an input pin associated with *Normal* is false.

11.18.3 FANOUT, FANIN and CONNECTIONS

The arithmetic model *fanout* shall be defined as shown in Semantics 151.

```
KEYWORD FANOUT = arithmetic_model {
  VALUETYPE = unsigned_number ;
}
SEMANTICS FANOUT {
  CONTEXT {
    PIN.LIMIT WIRE.SIZE.HEADER WIRE.CAPACITANCE.HEADER
    WIRE.RESISTANCE.HEADER WIRE.INDUCTANCE.HEADER
  }
}
```

Semantics 151— Arithmetic model FANOUT

The purpose of the arithmetic model *fanout* is to specify the total number of input pins connected to a net.

The arithmetic model *fanin* shall be defined as shown in .

```
KEYWORD FANIN = arithmetic_model {
  VALUETYPE = unsigned_number ;
}
SEMANTICS FANIN {
  CONTEXT {
    PIN.LIMI WIRE.SIZE.HEADER WIRE.CAPACITANCE.HEADER
    WIRE.RESISTANCE.HEADER WIRE.INDUCTANCE.HEADER
  }
}
```

Semantics 152— Arithmetic model FANIN

The purpose of the arithmetic model *fanin* is to specify the total number of output pins connected to a net.

The arithmetic model *connections* shall be defined as shown in .

```
KEYWORD CONNECTIONS = arithmetic_model {
  VALUETYPE = unsigned_number ;
}
SEMANTICS CONNECTIONS {
  CONTEXT {
    PIN.LIMIT WIRE.SIZE.HEADER WIRE.CAPACITANCE.HEADER
    WIRE.RESISTANCE.HEADER WIRE.INDUCTANCE.HEADER
  }
}
```

Semantics 153— Arithmetic model CONNECTIONS

The purpose of the arithmetic model *connections* is to specify the total number of pins connected to a net. The arithmetic value for *connections* shall equal the sum of arithmetic values for *fanout* and *fanin.*

The accounting of a pin shall depend on its *direction* annotation value (see).

A pin with direction annotation value *input* shall count for *fanout* and for *connections*. A pin with direction annotation value *output* shall count for *fanin* and for *connections*. A pin with direction value *both* shall count for *fanin* and for *fanout* and twice for *connections*. A pin without direction annotation or with direction annotation value *none* shall not count.

— FANOUT, FANIN, or CONNECTIONS as *limit arithmetic model* (see) in the context of a *pin* (see)

A design limit for the number of pins or nodes connected to a net can be described. The declared *pin* wherein the design limit is described shall count, according on its *direction* annotation value.

— FANOUT, FANIN, or CONNECTIONS as *header arithmetic model* (see) in the context of a *wire* (see)

The arithmetic value of *size* (see), *capacitance* (see), *resistance* (see), or *inductance* (see) can be calculated.

11.19 Arithmetic models related to layout implementation

11.19.1 SIZE

The arithmetic model *size* shall be defined as shown in Semantics 154.

```
KEYWORD SIZE = arithmetic_model {
  VALUETYPE = unsigned_number ;
}
SEMANTICS SIZE {
  CONTEXT {
    CELL WIRE
    WIRE.CAPACITANCE.HEADER
    WIRE.RESISTANCE.HEADER
    WIRE.INDUCTANCE.HEADER
    ANTENNA ANTENNA.LIMIT PIN
  }
}
```

Semantics 154—Arithmetic model SIZE

The purpose of the arithmetic model *size* is to define an abstract, unit-less measure for the space occupied by a physical object or the magnitude of a physical effect.

— SIZE as arithmetic model in the context of a *cell* (see) or a *wire* (see)

Size shall represent a measure for the space occupied by a placed *cell* or by a routed *wire*. The space occupied by a design or a subdesign shall be calculated as the sum of the space occupied by each cell instance and each routed wire. The space allocated for a design or a subdesign can be greater or equal to the space occupied by the design or subdesign.

— SIZE as *header arithmetic model* (see) in context of a *wire* (see)

The arithmetic value of *capacitance* (see), *resistance* (see), or *inductance* (see) in the context of a *wire* can be calculated. The dimension *size* shall represent a measure for space allocated for a design or subdesign wherein the wire is routed.

— SIZE as arithmetic model in the context of an *antenna* (see)

Size shall represent a measure for the magnitude of the antenna effect. A design limit for the magnitude of the antenna effect can be given using the arithmetic model container *limit* (see). The calculated size shall be compared against the design limit for size given in the context of the same antenna.

— SIZE as arithmetic model in the context of a *pin* (see)

Size shall represent a measure for the additive magnitude of an *antenna* (see), when the layout created by the connection between a pin and a routed wire is subjected to an antenna effect. An *antenna reference* annotation (see) and a *target* annotation (see) shall be used.

11.19.2 AREA

The arithmetic model *area* shall be defined as shown in Semantics 155.

```
KEYWORD AREA = arithmetic_model {
  VALUETYPE = unsigned_number ;
  SI_MODEL = AREA ;
}
SEMANTICS AREA {
  CONTEXT { CELL WIRE HEADER }
}
```

Semantics 155—Arithmetic model AREA

The purpose of the arithmetic model *area* is to define a physical area, according to the International System of Measurements and Units [reference needed].

— AREA as arithmetic model in the context of a *cell* (see) or a *wire* (see)

Area shall represent the physical area occupied by a placed *cell* or a routed *wire*, respectively. The area shall take into account the required space between neighboring objects.

The physical area occupied by a design or a subdesign shall be calculated as the sum of the physical area occupied by each cell instance and each routed wire. The physical area allocated for a design or a subdesign can be greater or equal to the physical area occupied by the design or subdesign.

— AREA as *header arithmetic model* (see) in context of a *wire* (see)

The arithmetic value of *capacitance* (see), *resistance* (see), or *inductance* (see) can be calculated. The dimension *area* shall represent the physical area allocated for a design or subdesign wherein the wire is routed.

— AREA as *header arithmetic model* (see) in context of a *layer* (see)

The arithmetic value of *capacitance* (see), *resistance* (see) can be calculated. A design limit for *current* (see) can be calculated. The dimension *area* shall represent the physical area occupied by a layout segment residing on the layer.

— AREA as *header arithmetic model* (see) in context of a *rule* (see)

The arithmetic value of *capacitance* (see), *resistance* (see) or *inductance* (see) can be calculated. A design limit for *current* (see), *distance* (see), *overhang* (see), *width* (see), *length* (see) or *extension* (see) can be calculated.

The dimension *area* shall represent the physical area occupied by a *pattern* or by a *region*. A *pattern reference* annotation (see) or a *region reference* annotation (see) shall be used.

— AREA as *header arithmetic model* (see) in context of an *antenna* (see)

The arithmetic value of *size* (see) in the context of an *antenna* can be calculated. The dimension *area* shall represent the physical area occupied by a layout segment residing on a *layer* (see). A *layer reference* annotation (see) shall be used.

11.19.3 PERIMETER

The arithmetic model *perimeter* shall be defined as shown in Semantics 156.

```
KEYWORD PERIMETER = arithmetic_model {
  VALUETYPE = unsigned_number ;
  SI_MODEL = DISTANCE ;
}
SEMANTICS PERIMETER {
  CONTEXT { CELL WIRE HEADER }
}
```

Semantics 156—Arithmetic model PERIMETER

The purpose of the arithmetic model *perimeter* is to define the *distance* (see) measured when surrounding the boundaries of a physical object.

— PERIMETER as arithmetic model in the context of a *cell* (see) or a *wire* (see)

Perimeter shall represent the perimeter surrounding a placed *cell* or a routed *wire*. The perimeter shall take into account the required space between neighboring objects.

— PERIMETER as *header arithmetic model* (see) in context of a *wire* (see)

The arithmetic value of *capacitance* (see), *resistance* (see), or *inductance* (see) can be calculated. The dimension *perimeter* shall represent the perimeter surrounding a space allocated for a design or subdesign wherein the wire is routed.

— PERIMETER as *header arithmetic model* (see) in context of a *layer* (see)

The arithmetic value of *capacitance* (see), *resistance* (see) can be calculated. A design limit for *current* (see) can be calculated. The dimension *perimeter* shall represent the perimeter surrounding a layout segment residing on the layer.

— PERIMETER as *header arithmetic model* (see) in context of a *rule* (see)

The arithmetic value of *capacitance* (see), *resistance* (see) or *inductance* (see) can be calculated. A design limit for *current* (see), *distance* (see), *overhang* (see), *width* (see), *length* (see) or *extension* (see) can be calculated. The dimension *perimeter* shall represent the perimeter surrounding a *pattern* or for a *region*. A *pattern reference* annotation (see) or a *region reference* annotation (see) shall be used.

— PERIMETER as *header arithmetic model* (see) in context of an *antenna* (see)

The arithmetic value of *size* (see) in the context of an *antenna* can be calculated. The dimension *perimeter* shall represent the perimeter surrounding a layout segment residing on a *layer* (see). A *layer reference* annotation (see) shall be used.

11.19.4 EXTENSION

The arithmetic model *extension* shall be defined as shown in Semantics 157.

```
KEYWORD EXTENSION = arithmetic_model {
  VALUETYPE = unsigned_number ;
  SI_MODEL = DISTANCE ;
}
SEMANTICS EXTENSION {
  CONTEXT { LAYER PATTERN RULE.LIMIT HEADER }
}
```

Semantics 157—Arithmetic model EXTENSION

The purpose of the arithmetic model *extension* is to specify the size of a polygon created by expanding a point within a *geometric model* (see Table 90 in Section 10.16). In the case of two allowed routing directions in an interval of 90 degrees, the expansion shall result in a rectangle. In the case of four allowed routing directions in an interval of 45 degrees, the expansion shall result in a hexagon.

This is illustrated in the following Figure 37.

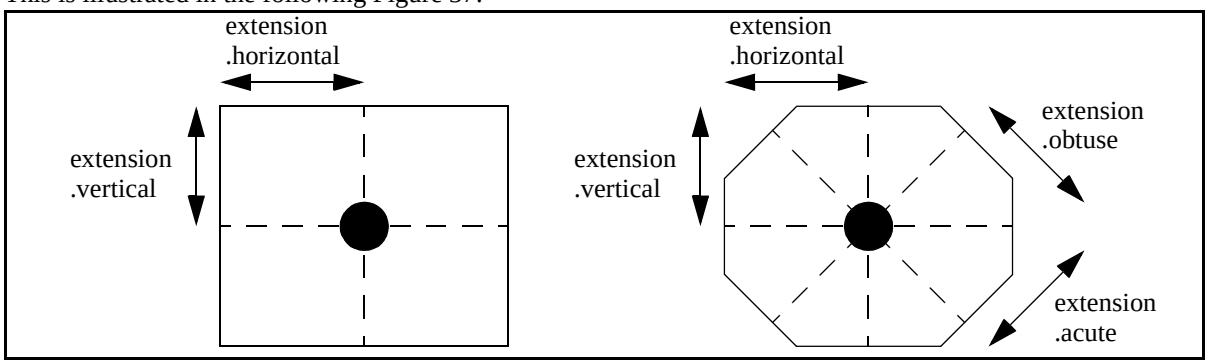


Figure 37—Illustration of EXTENSION

The arithmetic submodels *horizontal*, *vertical*, *acute* and *obtuse* (see) can be used to specify anisotrop expansion.

— EXTENSION as arithmetic model in the context of a *layer* (see)

Extension shall represent the expansion of an endpoint of a routing segment residing on a *layer* (see) with *layer-type* annotation value *routing* (see).

— EXTENSION as arithmetic model in the context of a *pattern* (see)

Extension shall represent the expansion of a *pattern* (see) with an associated *shape* annotation or with an associated *geometric model* (see). Each reference point shall be subject to expansion.

— EXTENSION as *limit arithmetic model* (see) in the context of a *rule* (see)

Extension shall represent a design limit for expansion of a *pattern*. Each reference point shall be subject to expansion. A *pattern reference* annotation (see) shall be used.

— EXTENSION as *header arithmetic model* (see) in the context of a *rule* (see)

An arithmetic value of *capacitance* (see), *resistance* (see) or *inductance* (see) can be calculated. A design limit for *current* (see), *distance* (see), *overhang* (see), *width* (see), *length* (see) or *extension* (see) can be calculated. The dimension *extension* shall represent the expansion of a *pattern* with *shape* annotation value *tee*, *cross*, *corner* or *end* (see). A *pattern reference* annotation (see) or a *model reference* annotation (see) shall be used. The *model reference* annotation shall refer to an arithmetic model *extension* as a child of a *pattern* or to an *arithmetic submodel* as a child of *extension* and a grandchild of *pattern*.

11.19.5 THICKNESS

The arithmetic model *thickness* shall be defined as shown in Semantics 158.

```
KEYWORD THICKNESS = arithmetic_model {
  VALUETYPE = unsigned_number ;
  SI_MODEL = DISTANCE ;
}
SEMANTICS EXTENSION {
  CONTEXT { LAYER HEADER }
}
```

Semantics 158—Arithmetic model THICKNESS

The purpose of the arithmetic model *thickness* is to specify the distance between the bottom and the top of a manufactured *layer* (see).

Thickness as *header arithmetic model* (see) can be used to calculate an arithmetic value of *capacitance* (see), *resistance* (see) or *inductance* (see) in the context of a *rule* (see).

11.19.6 HEIGHT

The arithmetic model *height* shall be defined as shown in Semantics 159.

```
KEYWORD HEIGHT = arithmetic_model {
  VALUETYPE = unsigned_number ;
  SI_MODEL = DISTANCE ;
}
SEMANTICS HEIGHT {
  CONTEXT { CELL SITE REGION LAYER HEADER }
}
```

Semantics 159—Arithmetic model HEIGHT

The purpose of the arithmetic model *height* is to specify a vertical distance, i.e., a distance measured in *y* direction or in *z* direction.

— HEIGHT as arithmetic model in the context of a *layer* (see)

Height shall represent a distance in *z* direction measured between the manufacturing substrate and the bottom of a manufactured layer.

— HEIGHT as arithmetic model in the context of a *cell* (see), *site* (see) or *region* (see)

Height shall represent a distance in *y* direction measured between the bottom and the top of a rectangular *cell* , *site*, *pattern* or *region*.

— HEIGHT as *header arithmetic model* (see) in the context of a *wire* (see)

Height shall represent the distance in *y* direction measured between the bottom and the top of an allocated rectangular space for a design or a subdesign wherein the *wire* is routed.

11.19.7 WIDTH

The arithmetic model *width* shall be defined as shown in Semantics 160.

```
KEYWORD WIDTH = arithmetic_model {
  VALUETYPE = unsigned_number ;
  SI_MODEL = DISTANCE ;
}
SEMANTICS WIDTH {
  CONTEXT {
    CELL SITE REGION LAYER LAYER.LIMIT
    PATTERN RULE.LIMIT HEADER
  }
}
```

Semantics 160—Arithmetic model WIDTH

The purpose of the arithmetic model *width* is to specify a distance within an *x*-*y* plane.

— WIDTH as arithmetic model in the context of a *cell* (see), a *site* (see) or a *region* (see)

Width shall represent a distance in *x* direction measured between the left and the right of a rectangular *cell* , *site* or *region*.

— WIDTH as *header arithmetic model* (see) in the context of a *wire* (see)

Width shall represent the distance in *x* direction measured between the left and the right of an allocated rectangular space for a design or a subdesign wherein the *wire* is routed.

— WIDTH as arithmetic model or *limit arithmetic model* (see) in the context of a *layer* (see)

Width shall represent a distance or a design limit for a distance between the borders of a routing segment residing on a layer with layertype annotation value routing (see). Width shall be measured orthogonal to the routing direction, i.e., in *y* (i.e., 90 degree) direction if the routing is in *x* (i.e., 0 degree) direction and vice-versa, in 135 degree direction if the routing is in 45 degree direction and vice versa.

— WIDTH as arithmetic model in the context of a *pattern* (see)

Width shall represent the distance between the borders of a *pattern* (see) with an associated *shape* annotation value *line* or *jog* (see) or with an associated a *geometric model* of type *polyline* or *ring* (see). Width shall be

measured orthogonal to the lines of the shape. A line shall be expanded by half the arithmetic value of width to each side of the line.

— WIDTH as *limit arithmetic model* (see) in the context of a *rule* (see)

Width shall represent a design limit for the distance between the borders of a *pattern* with an associated *shape* annotation value *line* or *jog* or with an associated a *geometric model* of type *polyline* or *ring*. A *pattern reference* annotation (see) shall be used.

— WIDTH as *header arithmetic model* (see) in the context of a *rule* (see)

An arithmetic value of *capacitance* (see), *resistance* (see) or *inductance* (see) can be calculated. A design limit for *current* (see), *distance* (see), *overhang* (see), *width* (see), *length* (see) or *extension* (see) can be calculated. The dimension *width* shall represent the distance between the borders of a *pattern* with *shape* annotation value *line* or *end* (see). A *pattern reference* annotation (see) or a *model reference* annotation (see) shall be used. The *model reference* annotation shall refer to an arithmetic model *width* as a child of a *pattern* or to an *arithmetic sub-model* as a child of *width* and a grandchild of *pattern*.

11.19.8 LENGTH

The arithmetic model *length* shall be defined as shown in Semantics 161.

```
KEYWORD LENGTH = arithmetic_model {
  VALUETYPE = unsigned_number ;
  SI_MODEL = DISTANCE ;
}
SEMANTICS LENGTH {
  CONTEXT { LAYER LAYER.LIMIT PATTERN RULE.LIMIT HEADER }
}
```

Semantics 161—Arithmetic model LENGTH

— LENGTH as arithmetic model or *limit arithmetic model* (see) in the context of a *layer* (see)

Length shall represent a distance or a design limit for a distance between the end points of a routing segment residing on a layer with layertype annotation value routing (see). Length shall be measured parallel to the routing direction.

— LENGTH as arithmetic model in the context of a *pattern* (see)

Length shall represent the distance between the end points of a *pattern* (see) with an associated *shape* annotation value *line* or *jog* (see).

— LENGTH as *limit arithmetic model* (see) in the context of a *rule* (see)

Length shall represent a design limit for the distance between the end points of a *pattern* with an associated *shape* annotation value *line* or *jog*. A *pattern reference* annotation (see) shall be used.

— LENGTH as *header arithmetic model* (see) in the context of a *rule* (see)

An arithmetic value of *capacitance* (see), *resistance* (see) or *inductance* (see) can be calculated. A design limit for *current* (see), *distance* (see), *overhang* (see), *width* (see), *length* (see) or *extension* (see) can be calculated. The dimension *length* shall represent the distance between the end points of a *pattern* with *shape* annotation

value *line* or *jog* (see). A *pattern reference* annotation (see), a *model reference* annotation (see) or a *between* annotation (see) shall be used. The *model reference* annotation shall refer to an arithmetic model *length* as a child of a *pattern* or to an *arithmetic submodel* as a child of *length* and a grandchild of *pattern*. A *between* annotation shall refer to two patterns representing two parallel routing segments.

11.19.9 DISTANCE

The arithmetic model *distance* shall be defined as shown in Semantics 162.

```
KEYWORD DISTANCE = arithmetic_model {
  VALUETYPE = unsigned_number ;
  SI_MODEL = DISTANCE ;
}
SEMANTICS DISTANCE {
  CONTEXT { RULE RULE.LIMIT HEADER }
}
```

Semantics 162—Arithmetic model DISTANCE

The purpose of the arithmetic model *distance* is to define a space in-between two objects, according to the International System of Measurements and Units [reference needed].

— DISTANCE as arithmetic model or as *limit arithmetic model* (see) in the context of a *rule* (see)

Distance shall represent a measured distance or a design limit for a distance between two *patterns* in the context of the rule. A *between* annotation (see) shall be used.

The arithmetic submodels *horizontal*, *vertical*, *acute* and *obtuse* (see) can be used.

— DISTANCE as *header arithmetic model* (see) in the context of a *rule* (see)

An arithmetic value of *capacitance* (see), *resistance* (see) or *inductance* (see) can be calculated. A design limit for *current* (see), *distance* (see), *overhang* (see), *width* (see), *length* (see) or *extension* (see) can be calculated. The dimension *distance* shall represent the measured distance between two patterns. A *between reference* annotation (see) or a *model reference* annotation shall be used. The *model reference* annotation shall refer to an arithmetic model *distance* as a child of a rule or to a *limit arithmetic model* distance as a grandchild of a rule.

11.19.10 OVERHANG

The arithmetic model *overhang* shall be defined as shown in Semantics 163.

```
KEYWORD OVERHANG = arithmetic_model {
  VALUETYPE = unsigned_number ;
  SI_MODEL = DISTANCE ;
}
SEMANTICS OVERHANG {
  CONTEXT { RULE RULE.LIMIT HEADER }
}
```

Semantics 163—Arithmetic model OVERHANG

The purpose of the arithmetic model *overhang* is to define an overlapping space between two objects.

Overhang can be used as arithmetic model or as *limit arithmetic model* (see) or as *header arithmetic model* (see) in the context of a *rule* (see), with similar semantic restrictions as *distance* (see).

Overhang can be interpreted as “negative” distance between the nearest parallel edges of two objects.

This is illustrated in the following Figure 38.

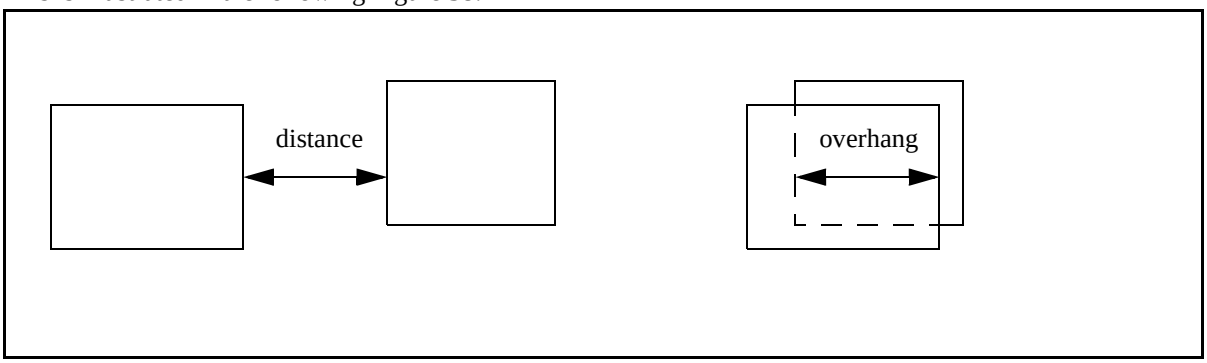


Figure 38—Illustration of DISTANCE versus OVERHANG

11.19.11 DENSITY

The arithmetic model *density* shall be defined as shown in Semantics 164.

```
KEYWORD DENSITY = arithmetic_model {
  VALUETYPE = unsigned_number ;
  MIN = 0;
  MAX = 1;
}
SEMANTICS DENSITY {
  CONTEXT { LAYER.LIMIT RULE RULE.LIMIT }
}
```

Semantics 164—Arithmetic model DENSITY

The purpose of the arithmetic model *density* is to specify a design limit or a calculation model for metal density. Metal density shall be defined as the area occupied by all metal segments residing on a *layer* (see) with *layertype* annotation value *routing* (see), divided by an allocated area wherein the metal segments are found.

— DENSITY as *limit arithmetic model* (see) in the context of a *layer* (see)

A constant design limit for metal density can be specified.

— DENSITY as arithmetic model or as *limit arithmetic model* (see) in the context of a *rule* (see)

A design limit or a calculation model for metal density can be specified. A *region reference* annotation (see) can be used to relate the design limit or the calculation model for metal density to a *region* (see) declared in the context of the same *rule*. A *model reference* annotation (see) can be used to relate a design limit to a related calculation model.

11.20 Annotations related to arithmetic models for layout implementation

Add lead-in text

11.20.1 CONNECT_RULE annotation

A *connect-rule* annotation shall be defined as shown in Semantics 165.

```
KEYWORD CONNECT_RULE = single_value_annotation {
  VALUETYPE = identifier ;
  VALUES { must_short can_short cannot_short }
  CONTEXT = CONNECTIVITY ;
}
```

Semantics 165—CONNECT_RULE annotation

The purpose of the *connect-rule* annotation is to specify that the arithmetic model *connectivity* (see Section 11.18.1) is to be interpreted as a requirement for connection rather than an actual connection.

The meaning of the annotation values is shown in Table 102.

Table 102—CONNECT_RULE annotation

Annotation value	Description
must_short	Electrical connection required.
can_short	Electrical connection allowed.
cannot_short	Electrical connection disallowed.

If multiple requirements for connection between the same objects are specified, restrictions for the boolean values of the respective arithmetic models *connectivity* shall apply.

These restrictions are specified in the following Table 103.

Table 103—Restrictions related to multiple requirements for connection

must_short	cannot_short	can_short
0	0	1
0	1	0
1	0	1

Any combination of boolean values not shown in Table 103 shall be considered invalid.

11.20.2 BETWEEN annotation

A *between* annotation shall be defined as shown in Semantics 166.

```
KEYWORD BETWEEN = multi_value_annotation {
  VALUETYPE = identifier ;
  CONTEXT { DISTANCE LENGTH OVERHANG CONNECTIVITY }
}
```

Semantics 166—BETWEEN annotation

The purpose of the *between* annotation is to specify a reference to multiple objects related to an arithmetic model *distance* (see Section 11.19.9), *length* (see Section 11.19.8), *overhang* (see Section 11.19.10), or *connectivity* (see Section 11.18.1).

11.20.3 BETWEEN annotation for CONNECTIVITY

A *between* annotation shall be subjected to the restriction shown in .

```
SEMANTICS ANTENNA.CONNECTIVITY.BETWEEN {
  REFERENCETYPE = LAYER;
}
SEMANTICS HEADER.CONNECTIVITY.BETWEEN {
  REFERENCETYPE { PATTERN REGION LAYER }
}
SEMANTICS LIBRARY.CONNECTIVITY.BETWEEN {
  REFERENCETYPE = CLASS ;
}
SEMANTICS SUBLIBRARY.CONNECTIVITY.BETWEEN {
  REFERENCETYPE = CLASS ;
}
SEMANTICS CELL.CONNECTIVITY.BETWEEN {
  REFERENCETYPE { PIN CLASS }
}
```

Semantics 167—BETWEEN annotation for CONNECTIVITY

The purpose of the restriction is to allow only a reference to objects which are semantically valid in the context of *connectivity* (see).

11.20.4 BETWEEN annotation for DISTANCE, LENGTH, OVERHANG

A *between* annotation shall be subjected to the restriction shown in .

```
SEMANTICS DISTANCE.BETWEEN {
  REFERENCETYPE { PATTERN REGION }
}
SEMANTICS LENGTH.BETWEEN {
  REFERENCETYPE { PATTERN REGION }
}
SEMANTICS OVERHANG.BETWEEN {
  REFERENCETYPE { PATTERN REGION }
}
```

Semantics 168—BETWEEN annotation for DISTANCE, LENGTH, OVERHANG

The purpose of the restriction is to allow only a reference to objects which are semantically valid in the context of *distance* (see), *length* (see) or *overhang* (see).

Furthermore, the number of annotation values, i.e., the number of referenced objects for *distance*, *length*, *overhang* shall be restricted to exactly two objects.

A *distance* between two objects can be generally defined. An *overhang* or a *length* involving two objects can be defined only between the nearest parallel edges of two objects.

In the case of two objects with nearest parallel edges, *distance* prescribes an empty space between the objects. *Overhang* prescribes an overlapping space between the objects. *Length* is defined as the distance between the end points of the intersection formed by projecting the parallel edges onto each other.

This is illustrated in the following Figure 39.

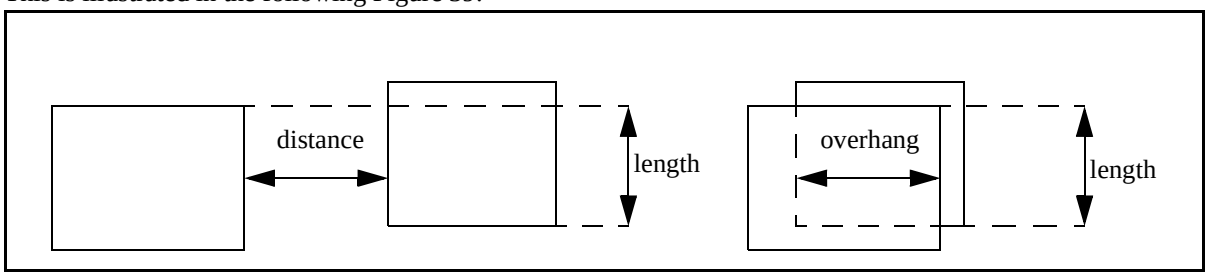


Figure 39—Illlustration of DISTANCE versus OVERHANG versus LENGTH

11.20.5 MEASURE annotation

A *measure* annotation shall be defined as shown in Semantics 169.

```
KEYWORD MEASURE = single_value_annotation {
  VALUETYPE = identifier ;
  VALUES { euclidean horizontal vertical manhattan }
  DEFAULT = euclidean ;
  CONTEXT = DISTANCE ;
}
```

Semantics 169—DISTANCE_MEASUREMENT annotation

The mathematical description of the annotation values is specified in the following .

Table 104—Annotation values for MEASURE

Annotation value	Mathematical description
`euclidean`	*measure* = $\sqrt{x^2 + y^2}$
`manhattan`	*measure* = $x + y$
`horizontal`	*measure* = x

Table 104—Annotation values for MEASURE (Continued)

Annotation value	Mathematical description
`vertical`	*measure =* *y*

Distance can be measured between two points, between a point and a line, or between two parallel lines. The *shape* annotation (see) specifies whether a pattern is represented by a point or by a line.

The specification of *x* and *y* for the mathematical definition of the measure annotation values is illustrated in Figure 40.

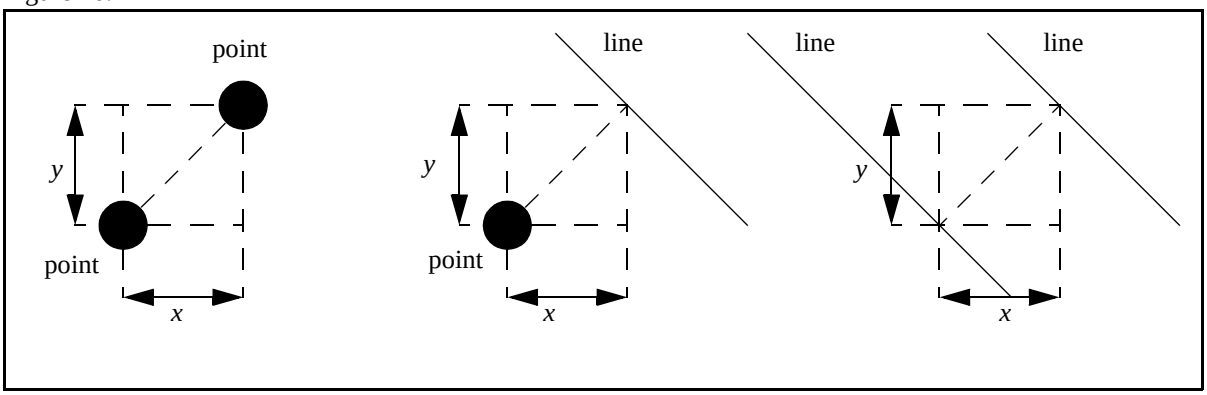


Figure 40—Illustration of MEASURE

Figure 40 shows distance between two points, a point and a line and between two parallel lines.

11.20.6 REFERENCE annotation container

A *reference* annotation container shall be defined as shown in Semantics 170.

```
KEYWORD REFERENCE = annotation_container {
  CONTEXT = DISTANCE ;
  REFERENCETYPE { PATTERN REGION }
}
SEMANTICS REFERENCE.identifier = single_value_annotation {
  VALUETYPE = identifier ;
  VALUES { center origin near_edge far_edge }
  DEFAULT = origin ;
}
```

Semantics 170—REFERENCE annotation container

The purpose of the *reference* annotation container is to specify the reference points for a measurement of *distance* (see).

An annotation within the *reference* annotation container shall associate a *pattern* (see) or a *region* (see) with a reference point specified by an annotation value.

The meaning of the annotation values is specified in the following .

Table 105—Annotation values for REFERENCE

Annotation value	Description
`origin`	The reference point is the origin of a pattern or region.
`center`	The reference point is the center of a pattern or region
`near_edge`	The reference point is the edge of a pattern or region which is nearest to a parallel edge of another pattern or region.
`far_edge`	The reference point is the edge of a pattern or region which is farest from a parallel edge of another pattern or region.

The following restrictions shall further apply:

a) The annotation value *origin* can only apply in the following cases:
   1) A *shape* annotation is associated with the pattern, and the annotation value is *tee*, *cross*, *corner* or *end*. The reference point of the shape shall be considered the origin.
   2) A *geometric model* (see) is associated with the pattern or region. A *geometric transformation* (see) can describe the location of the origin. If no geometric transformation is given, the location of the origin shall be the point x=0, y=0.

b) The annotation value *center*, *near edge* or *far edge* can only apply in the following cases:
   1) A *shape* annotation is associated with the pattern, and the annotation value is *line* or *jog*. The straight line connecting the end points shall be considered as *center*. The border of the line given by *width* (see) shall be considered either as *near edge* or as *far edge*.
   2) A predefined geometric model *rectangle* (see) is associated with the pattern or region. The point of gravity of the *rectangle* shall be considered as center.
   3) A predefined geometric model *line* (see) is associated with the pattern or region. The straight line connecting the end points shall be considered as center.

The meaning of the *reference* annotation values is further illustrated in Figure 41.

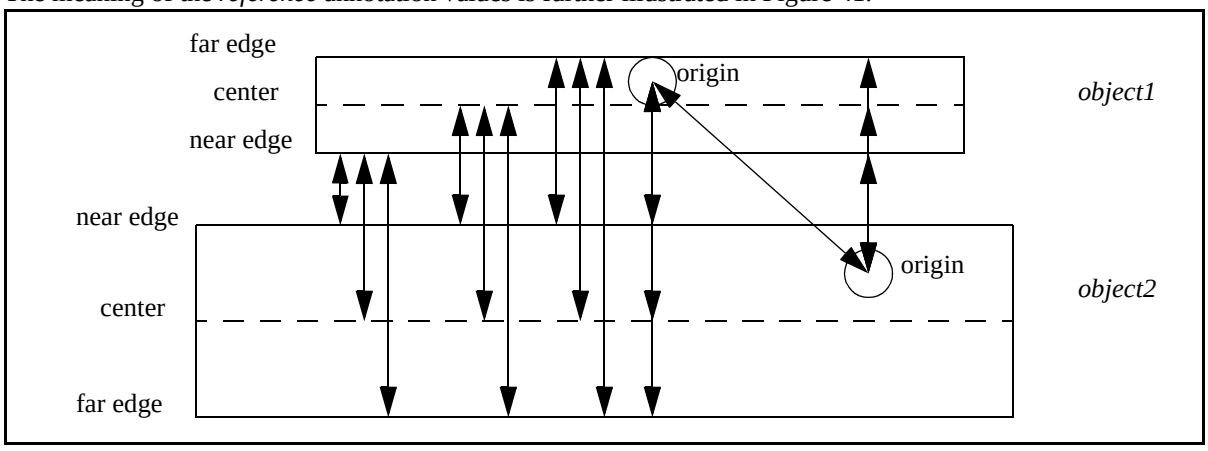


Figure 41—Illustration of REFERENCE for DISTANCE

Figure 41 shows *euclidean* distance between all possible reference points of *object1* and *object2*.

11.20.7 ANTENNA reference annotation

An *antenna* reference annotation shall be defined as shown in Semantics 171.

```
SEMANTICS ANTENNA = annotation {
  VALUETYPE = identifier ;
  CONTEXT { PIN.SIZE PIN.AREA PIN.PERIMETER }
  REFERENCETYPE = ANTENNA;
}
```

Semantics 171—ANTENNA reference annotation

An *antenna reference* annotation shall be used to relate a calculated *size* (see) or *area* (see) or *perimeter* (see) in the context of the *pin* with a calculation rule for *size* in the context of an *antenna* (see). Reference to multiple antennas can be made using a *multi-value annotation*.

11.20.8 TARGET annotation

An *target* annotation shall be defined as shown in Semantics 171.

```
SEMANTICS TARGET = annotation {
  VALUETYPE = identifier ;
  CONTEXT = PIN.SIZE;
  REFERENCETYPE = PIN.PATTERN;
}
```

Semantics 172—TARGET annotation

The *target* annotation shall be associated with the arithmetic model *size* (see) in the context of a *pin* (see).

The purpose of the *target* annotation is to specify a *pattern* (see) in the context of the same *pin* which is the victim of an antenna effect (see). The referenced pattern shall have a *layer reference* annotation (see) and a *trivial* or a *full arithmetic model* (see) for *area* (see) or *perimeter* (see).

An *antenna reference* annotation (see) shall also be associated with the arithmetic model *size*. The refered *antenna* (see) shall also contain an arithmetic model *size*, used as a calculation rule. The *size* in the context of the pin shall be considered additive to the *size* formulated by the calculation rule. The arithmetic value for *area* or *perimeter* in the referenced *pattern* shall further be used as evaluation results for the dimension *area* or *perimeter* within the calculation rule.

11.20.9 PATTERN reference annotation

A *pattern* reference annotation shall be defined as shown in .

The purpose of the *pattern reference* annotation is to relate an arithmetic model or a *header arithmetic model* (see) to a declared *pattern* (see).

```
SEMANTICS PATTERN = single_value_annotation {
  VALUETYPE = identifier ;
  CONTEXT {
    LENGTH WIDTH HEIGHT SIZE AREA THICKNESS
    PERIMETER EXTENSION
  }
}
```

Semantics 173—PATTERN annotation

11.21 Arithmetic submodels for timing and electrical data

The arithmetic submodels shown in Table 106 shall be applicable in the context of electrical modeling.

Table 106—Overview of arithmetic submodels for timing and electrical data

Keyword	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.

The arithmetic submodels *high* and *low* shall be defined as shown in .

```
KEYWORD HIGH = arithmetic_submodel {
  CONTEXT = arithmetic_model;
}
SEMANTICS HIGH { CONTEXT {
    CLASS.VOLTAGE CLASS.LIMIT.VOLTAGE
    PIN.VOLTAGE PIN.LIMIT.VOLTAGE
    LIBRARY.NOISE_MARGIN LIBRARY.LIMIT.NOISE
    PIN.NOISE PIN.NOISE_MARGIN PIN.LIMIT.NOISE
    PIN.CAPACITANCE PIN.RESISTANCE
} }
KEYWORD LOW = arithmetic_submodel {
  CONTEXT = arithmetic_model;
}
SEMANTICS LOW { CONTEXT {
    CLASS.VOLTAGE CLASS.LIMIT.VOLTAGE
    PIN.VOLTAGE PIN.LIMIT.VOLTAGE
    LIBRARY.NOISE_MARGIN LIBRARY.LIMIT.NOISE
    PIN.NOISE PIN.NOISE_MARGIN PIN.LIMIT.NOISE
    PIN.CAPACITANCE PIN.RESISTANCE
} }
```

Semantics 174—Arithmetic submodels HIGH and LOW

The arithmetic submodels *rise* and *fall* shall be defined as shown in .

```
KEYWORD RISE = arithmetic_submodel {
  CONTEXT = arithmetic_model;
}
SEMANTICS RISE { CONTEXT {
    FROM.THRESHOLD TO.THRESHOLD PIN.THRESHOLD
    PIN.CAPACITANCE PIN.RESISTANCE
    PIN.SLEWRATE PIN.LIMIT.SLEWRATE
    PIN.PULSEWIDTH PIN.LIMIT.PULSEWIDTH
} }
KEYWORD FALL = arithmetic_submodel {
  CONTEXT = arithmetic_model;
}
SEMANTICS FALL { CONTEXT {
    FROM.THRESHOLD TO.THRESHOLD PIN.THRESHOLD
    PIN.CAPACITANCE PIN.RESISTANCE
    PIN.SLEWRATE PIN.LIMIT.SLEWRATE
    PIN.PULSEWIDTH PIN.LIMIT.PULSEWIDTH
} }
```

Semantics 175—Arithmetic submodels RISE and FALL

11.22 Arithmetic submodels for physical data

The arithmetic submodels shown in Table 107 shall be applicable in the context of physical modeling.

Table 107—Overview of arithmetic submodels for physical data

Keyword	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.

The arithmetic submodels *horizontal* , *vertical*, *acute* and *obtuse* shall be defined as shown in .

```
KEYWORD HORIZONTAL = arithmetic_submodel {
  CONTEXT = arithmetic_model;
}
SEMANTICS HORIZONTAL { CONTEXT {
    WIDTH LENGTH EXTENSION DISTANCE OVERHANG
} }
KEYWORD VERTICAL = arithmetic_submodel {
  CONTEXT = arithmetic_model;
}
SEMANTICS VERTICAL { CONTEXT {
    WIDTH LENGTH EXTENSION DISTANCE OVERHANG
} }
KEYWORD ACUTE = arithmetic_submodel {
  CONTEXT = arithmetic_model;
}
SEMANTICS ACUTE { CONTEXT {
    WIDTH LENGTH EXTENSION DISTANCE OVERHANG
} }
KEYWORD OBTUSE = arithmetic_submodel {
  CONTEXT = arithmetic_model;
}
SEMANTICS OBTUSE { CONTEXT {
    WIDTH LENGTH EXTENSION DISTANCE OVERHANG
} }
```

Semantics 176—Arithmetic submodels HORIZONTAL, VERTICAL, ACUTE and OBTUSE

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

multiplier_prefix_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 }

vector_expression_macro ::= (see 6.13)
**.** non_escaped_identifier

A.3 Auxiliary definitions

all_purpose_value ::= (see 7.1)
  number
| identifier
| quoted_string
| bit_literal
| based_literal
| edge_value
| pin_variable
| control_expression

multiplier_prefix_value ::= (see 7.2)
  unsigned_number | multiplier_prefix_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 **;**

associate ::= (see 7.16)
ASSOCIATE quoted_string **;**
| **ASSOCIATE** quoted_string **{** *FORMAT*_single_value_annotation **}**

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 **;**
| **ALIAS** vector_expression_macro **=** **(** vector_expression **)**

constant_declaration ::= (see 8.2)
CONSTANT *constant*_identifier **=** constant_value **;**

constant_value ::=
number | based_literal

class_declaration ::= (see 8.6)
CLASS *class*_identifier **;**
| **CLASS** *class*_identifier **{** { class_item } **}**

class_item ::=
all_purpose_item
| geometric_model
| geometric_transformation

keyword_declaration ::= (see 8.3)
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
| *REFERENCETYPE*_annotation
| *SI_MODEL*_single_value_annotation

semantics_declaration ::= (see 8.4)
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
| *REFERENCETYPE*_annotation
| *SI_MODEL*_single_value_annotation

group_declaration ::= (see 8.8)
GROUP *group*_identifier **{** all_purpose_value { all_purpose_value } **}**
| **GROUP** *group*_identifier **{** *left*_index_value **:** *right*_index_value **}**

template_declaration ::= (see 8.9)
TEMPLATE *template*_identifier **{** ALF_statement { ALF_statement } **}**

template_instantiation ::= (see 8.10)
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

pin ::= (see 9.5)
  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
| pattern
| 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 9.6)
simple_pingroup | vector_pingroup

simple_pingroup ::=
PINGROUP *pingroup*_identifier
{ *MEMBERS*_multi_value_annotation { all_purpose_item } **}**
| *simple_pingroup*_template_instantiation

vector_pingroup ::=
| **PINGROUP** multi_index *pingroup*_identifier
{ *MEMBERS*_multi_value_annotation { vector_pingroup_item } **}**
| *vector_pingroup*_template_instantiation

vector_pingroup_item ::=
all_purpose_item
| range

primitive ::= (see 9.8)
PRIMITIVE *primitive*_identifier **{** { primitive_item } **}**
| **PRIMITIVE** *primitive*_identifier **;**
| *primitive*_template_instantiation

primitive_item ::=
all_purpose_item
| pin
| pingroup
| function
| test

wire ::= (see 9.9)
WIRE *wire*_identifier **{** { wire_item } **}**
| **WIRE** *wire*_identifier **;**
| *wire*_template_instantiation

wire_item ::=
all_purpose_item
| node

*wire_instance*_pin_assignment ::=
*wire_reference*_pin_variable **=** *wire_instance*_pin_value **;**

node ::= (see 9.11)
NODE *node*_identifier **;**
| **NODE** *node*_identifier **{** { node_item } **}**
| *node*_template_instantiation

node_item ::=
all_purpose_item

vector ::= (see 9.13)
VECTOR control_expression **;**
| **VECTOR** control_expression **{** { vector_item } **}**

| *vector*_template_instantiation

vector_item ::=
all_purpose_item
| wire_instantiation

layer ::= (see 9.15)
LAYER *layer*_identifier **;**
| **LAYER** *layer*_identifier **{** { layer_item } **}**
| *layer*_template_instantiation

layer_item ::=
all_purpose_item

via ::= (see 9.17)
VIA *via*_identifier **;**
| **VIA** *via*_identifier **{** { via_item } **}**
| *via*_template_instantiation

via_item ::=
all_purpose_item
| pattern
| artwork

via_instantiation ::= (see 9.20)
*via*_identifier *instance*_identifier **;**
| *via*_identifier *instance*_identifier **{** { geometric_transformation } **}**

rule ::= (see 9.19)
RULE *rule*_identifier **;**
| **RULE** *rule*_identifier **{** { rule_item } **}**
| *rule*_template_instantiation

rule_item ::=
all_purpose_item
| pattern
| region
| via_instantiation

antenna ::= (see 9.20)
ANTENNA *antenna*_identifier **;**
| **ANTENNA** *antenna*_identifier **{** { antenna_item } **}**
| *antenna*_template_instantiation

antenna_item ::=
all_purpose_item

blockage ::= (see 9.21)
BLOCKAGE *blockage*_identifier **;**
| **BLOCKAGE** *blockage*_identifier **{** { blockage_item } **}**
| *blockage*_template_instantiation

blockage_item ::=
all_purpose_item
| pattern
| region
| rule
| via_instantiation

port ::= (see 9.22)
PORT *port*_identifier **;{** { port_item } **}**
| **PORT** *port*_identifier **;**
| *port*_template_instantiation

port_item ::=
all_purpose_item
| pattern

| region
| rule
| via_instantiation

site ::= (see 9.24)
SITE *site*_identifier **;**
| **SITE** *site*_identifier **{** { site_item } **}**
| *site*_template_instantiation

site_item ::=
all_purpose_item
| *WIDTH*_arithmetic_model
| *HEIGHT*_arithmetic_model

array ::= (see 9.26)
ARRAY *array*_identifier **;**
| **ARRAY** *array*_identifier **{** { array_item } **}**
| *array*_template_instantiation

array_item ::=
all_purpose_item
| geometric_transformation

pattern ::= (see 9.28)
PATTERN *pattern*_identifier **;**

| **PATTERN** *pattern*_identifier **{** { pattern_item } **}**
| *pattern*_template_instantiation

pattern_item ::=
all_purpose_item
| geometric_model
| geometric_transformation

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 { cell_instantiation { cell_instantiation } **}**
| *structure*_template_instantiation

cell_instantiation ::=
*cell_reference*_identifier *cell_instance*_identifier **;**
| *cell_reference*_identifier *cell_instance*_identifier **{** { *cell_instance*_pin_value } **}**
| *cell_reference*_identifier *cell_instance*_identifier **{** { *cell_instance*_pin_assignment } **}**
| *cell_instantiation*_template_instantiation

*cell_instance*_pin_assignment ::=
*cell_reference*_pin_variable **=** *cell_instance*_pin_value **;**

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 = non_scan_cell_reference
| **NON_SCAN_CELL {** non_scan_cell_reference { non_scan_cell_reference } **}**
| *non_scan_cell*_template_instantiation

non_scan_cell_reference ::=
*non_scan_cell*_identifier **{** { *scan_cell_pin*_identifier } **}**
| *non_scan_cell*_identifier **{** { *non_scan_cell_pin*_identifier **=** *scan_cell_pin*_identifier **;** } **}**

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_expression_macro
vector_unary ::=
        edge_literal

vector_binary ::=
```

```
        &
      | &&
      | |
      | ||
      | ->
      | ~>
      | <->
      | <~>
      | &>
      | <&>
control_and ::=
      & | &&
control_expression ::=
        ( vector_expression )
      | ( boolean_expression )
geometric_model ::=                                                    (see 10.16)
        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
geometric_transformation ::=                                           (see 10.18)
        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 10.19)
        ARTWORK = artwork_identifier ;
      | ARTWORK = artwork_reference
      | ARTWORK { artwork_reference { artwork_reference } }
      | artwork_template_instantiation
artwork_reference ::=
        artwork_identifier { { geometric_transformation } { cell_pin_identifier } }
      | artwork__identifier { { geometric_transformation } { artwork_pin_identifier = cell_pin_identifier ; } }
```

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 11.2.1)
+
| **-**

binary_arithmetic_operator ::= (see 11.2.2)
+
| **-**
| *****
| **/**
| ******
| **%**

macro_arithmetic_operator ::= (see 11.2.3)
abs
| **exp**
| **log**
| **min**
| **max**

arithmetic_model ::= (see 11.3)
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 11.4)
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 11.6)
nonescaped_identifier **=** arithmetic_value **;**
| nonescaped_identifier [**=** arithmetic_value] **{** auxiliary_qualifier { auxiliary_qualifier } **}**

auxiliary_qualifier
annotation
| annotation_container
| event_reference
| from-to

arithmetic_submodel ::= (see 11.7)
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 11.5)
[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 11.8)
limit
| early-late
| *arithmetic_model_container*_identifier **{** arithmetic_model { arithmetic_model } **}**

limit ::= (see 11.8.2)
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 11.12)
      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 11.8.3)
      early [late]
    | [early] late
early ::=
    EARLY { early-late_item { early-late_item } **}**
late ::=
    LATE { early-late_item { early-late_item } **}**
early-late_item ::=
      *DELAY*_arithmetic_model
    | *RETAIN*_arithmetic_model
    | *SLEWRATE*_arithmetic_model
violation ::= (see 11.10)
      VIOLATION { violation_item { violation_item } **}**
    | *violation*_template_instantiation
violation_item ::=
      *MESSAGE_TYPE*_single_value_annotation
    | *MESSAGE*_single_value_annotation
    | behavior
wire_instantiation ::= (see 11.11)
      *wire_reference*_identifier *wire_instance*_identifier **;**
    | *wire_reference*_identifier *wire_instance*_identifier **{** { *wire_instance*_pin_value } **}**
    | *wire_reference*_identifier *wire_instance*_identifier **{** { *wire_instance*_pin_assignment } **}**
    | *wire_instantiation*_template_instantiation
*wire_instance*_pin_assignment ::=
    *wire_reference*_pin_variable **=** *wire_instance*_pin_value **;**

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 Auxiliary and generic definitions

```
KEYWORD FORMAT = single_value_annotation {
  CONTEXT = ASSOCIATE;
  VALUETYPE = identifier;
  VALUES { vhdl verilog c \c++ alf }
  DEFAULT = alf;
}
KEYWORD VALUETYPE = single_value_annotation {
  CONTEXT = KEYWORD;
}
KEYWORD VALUES = multi_value_annotation {
  CONTEXT { KEYWORD SEMANTICS }
}
KEYWORD DEFAULT = single_value_annotation {
  CONTEXT { KEYWORD arithmetic_model }
}
KEYWORD CONTEXT = annotation {
  VALUETYPE = identifier;
}
KEYWORD REFERENCETYPE = annotation {
  CONTEXT { KEYWORD SEMANTICS }
  VALUETYPE = identifier;
}
KEYWORD SI_MODEL = single_value_annotation {
  CONTEXT = KEYWORD;
  VALUETYPE = identifier;
  VALUES {
    TIME FREQUENCY CURRENT VOLTAGE POWER ENERGY
    RESISTANCE CAPACITANCE INDUCTANCE
    DISTANCE AREA FLUENCE FLUX
  }
}
SEMANTICS CLASS = annotation {
  CONTEXT { library_specific_object arithmetic_model }
  VALUETYPE = identifier;
```

```
  REFERENCETYPE = CLASS;
}
KEYWORD USAGE = annotation {
  CONTEXT = CLASS;
  VALUETYPE = identifier;
  VALUES {
    SWAP_CLASS RESTRICT_CLASS
    SIGNAL_CLASS SUPPLY_CLASS CONNECT_CLASS
    SELECT_CLASS NODE_CLASS
    EXISTENCE_CLASS CHARACTERIZATION_CLASS
    ORIENTATION_CLASS SYMMETRY_CLASS
  }
}
```

B.2 Library definitions

```
SEMANTICS LIBRARY = annotation {
  VALUETYPE = identifier;
  REFERENCETYPE { LIBRARY SUBLIBRARY }
}
KEYWORD INFORMATION = annotation_container {                (see 9.2.2)
  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 CELL = annotation {
  VALUETYPE = identifier;
  REFERENCETYPE = CELL;
}
KEYWORD CELLTYPE = single_value_annotation {                (see 9.4.2)
  CONTEXT = CELL;
  VALUETYPE = identifier;
  VALUES {
    buffer combinational multiplexor flipflop latch
    memory block core special
  }
}
```

```
KEYWORD SWAP_CLASS = annotation {                              (see 9.4.3)
  CONTEXT = CELL;
  VALUETYPE = identifier;
  REFERENCETYPE = CLASS;
}
KEYWORD RESTRICT_CLASS = annotation {                          (see 9.4.4)
  CONTEXT { CELL CLASS }
  VALUETYPE = identifier;
  REFERENCETYPE = CLASS;
}
KEYWORD SCAN_TYPE = single_value_annotation {                  (see 9.4.5)
  CONTEXT = CELL;
  VALUETYPE = identifier;
  VALUES { muxscan clocked lssd control_0 control_1 }
}
KEYWORD SCAN_USAGE = single_value_annotation {                 (see 9.4.6)
  CONTEXT = CELL;
  VALUETYPE = identifier;
  VALUES { input output hold }
}
KEYWORD BUFFERTYPE = single_value_annotation {                 (see 9.4.7)
  CONTEXT = CELL;
  VALUETYPE = identifier;
  VALUES { input output inout internal }
  DEFAULT = internal;
}
KEYWORD DRIVERTYPE = single_value_annotation {                 (see 9.4.8)
  CONTEXT = CELL;
  VALUETYPE = identifier;
  VALUES { predriver slotdriver both }
}
KEYWORD PARALLEL_DRIVE = single_value_annotation {             (see 9.4.9)
  CONTEXT = CELL;
  VALUETYPE = unsigned_integer;
  DEFAULT = 1;
}
KEYWORD PLACEMENT_TYPE = single_value_annotation {            (see 9.4.10)
  CONTEXT = CELL;
  VALUETYPE = identifier;
  VALUES { pad core ring block connector }
  DEFAULT = core;
}
KEYWORD MEMBERS = multi_value_annotation {                     (see 9.7.2)
  CONTEXT = PINGROUP;
  VALUETYPE = identifier;
  REFERENCETYPE = PIN;
}
KEYWORD VIEW = single_value_annotation {                       (see 9.7.3)
  CONTEXT { PIN PINGROUP }
  VALUETYPE = identifier;
  VALUES { functional physical both none }
```

```
  DEFAULT = both;
}
KEYWORD PINTYPE = single_value_annotation {                    (see 9.7.4)
  CONTEXT = PIN;
  VALUETYPE = identifier;
  VALUES { digital analog supply }
  DEFAULT = digital;
}
KEYWORD DIRECTION = single_value_annotation {                  (see 9.7.5)
  CONTEXT = PIN;
  VALUETYPE = identifier;
  VALUES { input output both none }
}
KEYWORD SIGNALTYPE = single_value_annotation {                 (see 9.7.6)
  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 9.7.7)
  CONTEXT = PIN;
  VALUETYPE = identifier;
  VALUES { asynchronous synchronous }
}
KEYWORD POLARITY = single_value_annotation {                   (see 9.7.8)
  CONTEXT = PIN;
  VALUETYPE = identifier;
  VALUES { high low rising_edge falling_edge double_edge }
}
KEYWORD DATATYPE = single_value_annotation {                  (see 9.7.10)
  CONTEXT { PIN PINGROUP }
  VALUETYPE = identifier;
  VALUES { signed unsigned }
 }
KEYWORD INITIAL_VALUE = single_value_annotation {             (see 9.7.11)
  CONTEXT = CELL;
  VALUETYPE = boolean_value;
}
KEYWORD SCAN_POSITION = single_value_annotation {             (see 9.7.12)
  CONTEXT = PIN;
  VALUETYPE = unsigned;
  DEFAULT = 0;
}
KEYWORD STUCK = single_value_annotation {                     (see 9.7.13)
  CONTEXT = PIN;
  VALUETYPE = identifier;
  VALUES { stuck_at_0 stuck_at_1 both none }
```

```
  DEFAULT = both;
}
KEYWORD SUPPLYTYPE = annotation {                          (see 9.7.14)
  CONTEXT = PIN;
  VALUETYPE = identifier;
  VALUES { power ground reference }
}
KEYWORD SIGNAL_CLASS = annotation {                        (see 9.7.15)
  CONTEXT { PIN PINGROUP }
  VALUETYPE = identifier;
  REFERENCETYPE = CLASS;
}
KEYWORD SUPPLY_CLASS = annotation {                        (see 9.7.16)
  CONTEXT { PIN PINGROUP CLASS }
  VALUETYPE = identifier;
  REFERENCETYPE = CLASS;
}
KEYWORD DRIVETYPE = single_value_annotation {              (see 9.7.17)
  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 9.7.18)
  CONTEXT = PIN;
  VALUETYPE = identifier;
  VALUES { behavior measure both none }
  DEFAULT = both;
}
KEYWORD CONNECT_CLASS = single_value_annotation {          (see 9.7.19)
  CONTEXT = PIN;
  VALUETYPE = identifier;
  REFERENCETYPE = CLASS;
}
KEYWORD SIDE = single_value_annotation {                   (see 9.7.20)
  CONTEXT { PIN PINGROUP }
  VALUETYPE = identifier;
  VALUES { left right top bottom inside }
}
KEYWORD ROW = annotation {                                 (see 9.7.21)
  CONTEXT { PIN PINGROUP }
  VALUETYPE = unsigned_integer;
}
KEYWORD COLUMN = annotation {
  CONTEXT { PIN PINGROUP }
  VALUETYPE = unsigned_integer;
}
KEYWORD ROUTING_TYPE = single_value_annotation {           (see 9.7.22)
  CONTEXT { PIN PORT }
```

```
  VALUETYPE = identifier;
  VALUES { regular abutment ring feedthrough }
  DEFAULT = regular;
}
KEYWORD PULL = single_value_annotation {                    (see 9.7.23)
  CONTEXT = PIN;
  VALUETYPE = identifier;
  VALUES { up down both none }
  DEFAULT = none;
}
KEYWORD WIRETYPE = single_value_annotation {                (see 9.10.2)
  CONTEXT = WIRE;
  VALUETYPE = identifier;
  VALUES { estimated extracted analytical load }
}
KEYWORD SELECT_CLASS = annotation {                         (see 9.10.3)
  CONTEXT = WIRE;
  VALUETYPE = identifier;
  REFERENCETYPE = CLASS;
}
KEYWORD NODETYPE = single_value_annotation {                (see 9.12.2)
  CONTEXT = NODE;
  VALUETYPE = identifier;
  VALUES { power ground source sink
    driver receiver interconnect }
  DEFAULT = interconnect;
}
KEYWORD NODE_CLASS = annotation {                           (see 9.12.3)
  CONTEXT = NODE;
  VALUETYPE = identifier;
  REFERENCETYPE = CLASS;
}
KEYWORD PURPOSE = annotation {                              (see 9.14.2)
  CONTEXT { VECTOR CLASS }
  VALUETYPE = identifier ;
  VALUES { bist test timing power noise reliability }
}
KEYWORD OPERATION = single_value_annotation {               (see 9.14.3)
  CONTEXT = VECTOR;
  VALUETYPE = identifier;
  VALUES {
    read write read_modify_write refresh load
    start end iddq
  }
}
KEYWORD LABEL = single_value_annotation {                   (see 9.14.4)
  CONTEXT = VECTOR;
  VALUETYPE = string_value;
}
KEYWORD EXISTENCE_CONDITION = single_value_annotation {     (see 9.14.5)
  CONTEXT { VECTOR CLASS }
  VALUETYPE = boolean_expression;
```

```
  DEFAULT = 1;
}
KEYWORD EXISTENCE_CLASS = annotation {                    (see 9.14.6)
  CONTEXT { VECTOR CLASS }
  VALUETYPE = identifier;
  REFERENCETYPE = CLASS;
}
KEYWORD
CHARACTERIZATION_CONDITION = single_value_annotation {    (see 9.14.7)
  CONTEXT { VECTOR CLASS }
  VALUETYPE = boolean_expression;
}
KEYWORD CHARACTERIZATION_VECTOR = single_value_annotation { (see 9.14.8)
  CONTEXT { VECTOR CLASS }
  VALUETYPE = control_expression;
}
KEYWORD CHARACTERIZATION_CLASS = annotation {
  CONTEXT { VECTOR CLASS }                                (see 9.14.9)
  VALUETYPE = identifier;
  REFERENCETYPE = CLASS;
}
KEYWORD LAYERTYPE = single_value_annotation {             (see 9.16.2)
  CONTEXT = LAYER;
  VALUETYPE = identifier;
  VALUES {
    routing cut substrate dielectric reserved abstract
  }
}
KEYWORD PITCH = single_value_annotation {                 (see 9.16.3)
  CONTEXT = LAYER;
  VALUETYPE = unsigned_number;
}
KEYWORD PREFERENCE = single_value_annotation {            (see 9.16.4)
  CONTEXT = LAYER;
  VALUETYPE = identifier;
  VALUES { horizontal vertical acute obtuse }
}
KEYWORD VIATYPE = single_value_annotation {               (see 9.18.2)
  CONTEXT = VIA;
  VALUETYPE = identifier;
  VALUES { default non_default partial_stack full_stack }
  DEFAULT = default;
}
KEYWORD CONNECT_TYPE = single_value_annotation {          (see 9.23.1)
  CONTEXT = PORT;
  VALUETYPE = identifier;
  VALUES { external internal }
  DEFAULT = external;
}
KEYWORD ORIENTATION_CLASS = annotation {                  (see 9.25.2)
  CONTEXT { SITE CELL }
  VALUETYPE = identifier;
```

```
  REFERENCETYPE = CLASS;
}
KEYWORD SYMMETRY_CLASS = annotation {                        (see 9.25.3)
  CONTEXT = SITE;
  VALUETYPE = identifier;
  REFERENCETYPE = CLASS;
}
KEYWORD ARRAYTYPE = single_value_annotation {                (see 9.27.1)
  CONTEXT = ARRAY;
  VALUETYPE = identifier;
  VALUES { floorplan placement
    global_routing detailed_routing }
}
KEYWORD SHAPE = single_value_annotation {                    (see 9.29.2)
  CONTEXT = PATTERN;
  VALUETYPE = identifier;
  VALUES { line tee cross jog corner end }
  DEFAULT = line;
}
KEYWORD VERTEX = single_value_annotation {                   (see 9.29.3)
  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.3 Arithmetic definitions

```
SEMANTICS VIOLATION {                                         (see 11.10)
  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 = single_value_annotation {                    (see 11.9.1)
  CONTEXT = arithmetic_model ;
  VALUETYPE = multiplier_prefix_value ;
  DEFAULT = 1 ;
}
KEYWORD CALCULATION = annotation {                          (see 11.9.2)
  CONTEXT = library_specific_object.arithmetic_model ;
  VALUES { absolute incremental }
  DEFAULT = absolute ;
}

KEYWORD INTERPOLATION = single_value_annotation {           (see 11.9.3)
  CONTEXT = HEADER.arithmetic_model ;
  VALUES { linear fit ceiling floor }
  DEFAULT = fit ;
}
SEMANTICS PIN = single_value_annotation {                  (see 11.13.2)
  CONTEXT {
    FROM TO SLEWRATE PULSEWIDTH
    CAPACITANCE RESISTANCE INDUCTANCE VOLTAGE CURRENT
  }
  REFERENCETYPE { PIN PIN.PORT NODE WIRE.NODE }
}
SEMANTICS SKEW.PIN = multi_value_annotation {
  REFERENCETYPE { PIN PINGROUP PIN.PORT NODE WIRE.NODE }
}
KEYWORD EDGE_NUMBER = annotation {                         (see 11.11.2)
  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 11.13.7)
  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 11.20.1)
  VALUETYPE = identifier ;
  VALUES { must_short can_short cannot_short }
  CONTEXT = CONNECTIVITY;
}
KEYWORD BETWEEN = multi_value_annotation {                (see 11.20.2)
  VALUETYPE = identifier ;
  CONTEXT { DISTANCE LENGTH OVERHANG CONNECTIVITY }
}
KEYWORD DISTANCE_MEASUREMENT = single_value_annotation {  (see 11.20.5)
  VALUETYPE = identifier ;
  VALUES { euclidean horizontal vertical manhattan }
  DEFAULT = euclidean ;
  CONTEXT = DISTANCE ;
}
KEYWORD REFERENCE = annotation_container {                (see 11.20.6)
  CONTEXT = DISTANCE ;
}
SEMANTICS REFERENCE.identifier = single_value_annotation {
  VALUETYPE = identifier ;
  VALUES { center origin near_edge far_edge }
  DEFAULT = origin ;
}
SEMANTICS ANTENNA = annotation {                          (see 11.20.7)
  VALUETYPE = identifier ;
  CONTEXT { PIN.SIZE PIN.AREA PIN.PERIMETER }
}
SEMANTICS PATTERN = single_value_annotation {             (see 11.20.9)
  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

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W