IEEE P1603/D9
A Draft Standard for an Advanced Library Format (ALF) describing Integrated Circuit (IC) technology, cells, and blocks

IEEE Standards Activities Department
Standards Licensing and Contracts
445 Hoes Lane, P.O. Box 1331
Piscataway, NJ 08855-1331, USA

Abstract: The Advanced Library Format (ALF) is a modeling language for library elements used in integrated circuit (IC) technology. ALF enables description of electrical, functional, and physical models in a formal language suitable for Electronic Design Automation (EDA) application tools targeted for design and analysis of an IC. This document provides rules that describe ALF and how tool developers, integrators, library creators, and library users should use it.

List of keywords: Integrated circuit, modeling, library, cell, block, technology, language, format, electrical, functional, physical, behavioral, RTL, gate-level, layout, EDA, timing, derate, power, signal integrity.

Introduction

(This introduction is not part of IEEE P1603, Advanced Library Format (ALF) Reference Manual.)

The purpose of the Advanced Library Format (ALF) is to provide a modeling language and semantics for the functional, physical, and electrical performance description of technology-specific libraries for cell-based and block-based design. Without a standard, EDA tools would be left to use tool-specific and fragmented library descriptions. The semantics would be defined by tool implementations only, which are subject to change and prone to misinterpretation. Therefore, ALF is proposed to create a consistent library view suitable as a reference for library creators and users, as well as for electronic design automation (EDA) tool developers and integrators.

The IEEE P1603 standard for ALF is based on the work of Open Verilog International (OVI) and its successor organization Accellera.

The ALF standard began as the OVI Power & Synthesis Technical Steering Committee (PS-TSC) early in 1996, with the charter to define a standard library data format for synthesis, power analysis, and optimization. As the committee grew in membership, with the addition of experts in other fields, such as design for test, it became clear that such a format could be easily extended to cover other design tools. Furthermore, the benefit to both silicon and EDA vendors of having a single, flexible format that would fully describe the functional, electrical, and physical performance of a technology library in an accurate and unambiguous fashion was widely recognized.

ALF was announced at the occasion of the OVI/VI-sponsored HDL conference in March 1997, where a trial version of the standard was released. Amongst the pioneers of proving the feasibility of ALF was the European CAD Standardization Initiative, sister organization of VSIA, who demonstrated an ALF-based ASIC implementation flow in 1997. In November 1997, OVI approved and released ALF version 1.0.

In 1998, the ASIC Council, under the auspices of the Silicon Integration Initiative (SI2), selected ALF as a complementary description of library elements within the Open Library Architecture (OLA), which builds upon the IEEE 1481-1999 standard for a delay calculation system. This endorsement triggered the initial adoption of ALF libraries by major ASIC vendors and the development of ALF version 1.1, which was approved and released by OVI in April 1999.

In June 1999, the ASIC council encouraged the ALF workgroup to include layout modeling. Consequently, deep submicron (DSM) issues, such as on-chip interconnect modeling, signal integrity, and reliability became a major focus for ALF. The work culminated in the release of ALF version 2.0 in December 2000, under the auspices of the OVI/VI successor organization Accellera.

ALF version 2.0 became the foundation for this IEEE standard. An IEEE study group was formed in February 2001. The study group became the IEEE P1603 workgroup in June 2001. The name ALF has been retained due to already existing name recognition. By that time, the ALF had already set a standard for the industry, which can be measured by direct adoption and the influence on existing vendor-proprietary library formats. Major EDA vendors also made the specification of their existing proprietary library formats available to the industry and allowed the user community to extend those formats and strive for compatibility with ALF.

Although IEEE is now the legal owner of ALF, Accellera continues to foster and promote ALF. As a result, ALF has gained attention of other national and international standardization bodies, such as JEITA in May 2002 and the IEC in October 2003.

From its inception, the goal for ALF has been to provide a solid foundation for library modeling within a continuously evolving application space. ALF has been designed to be more general in scope and purpose than a particular tool-oriented format. At the same time, care has been taken to make ALF easily adoptable and to make the migration path from legacy formats as smooth as possible. Therefore, an ALF library can be very similar in appearance to a library in a conventional format, but ALF has also the expression power of a modeling language.

The construction principles for ALF can be summarized as follows:

Simplicity

ALF has relatively few basic syntax construction principles. Once they are understood, reading and writing an ALF library or translating other library formats into ALF is very easy. Also, attention has been paid to the fact all ALF keywords are taken from natural language, i.e., spoken and written English, and their semantic meaning is as close to the natural language as possible. The use of artificial words or acronyms is limited to constructs, which have already become part of technical language in the industry.

Completeness

Conventional library formats would support data without self-evident meaning such as coefficients, scaling factors, etc. The interpretation of the data would be left to the application tool. On the other hand, an ALF library specifies a complete and self-contained description by providing the complete model, i.e., a calculation rule using an *arithmetic expression*. Furthermore, ALF contains information for characterization of particular measurement data, for example delay, power, or noise. ALF introduces the original concept of a *vector expression* to describe the event pattern associated with the measurement. This concept has a far-reaching potential for creating abstract, yet accurate, modeling views for cells and larger blocks. Any timing, power, or signal integrity measurement on a digital circuit or a mixed-signal circuit can be associated with a vector expression.

Orthogonality

Orthogonality allows for modeling features to be combined most efficiently with each other to yield a maximum expression capability. In ALF, orthogonality is closely related to context-sensitivity. A particular semantic meaning is created by describing a particular model in a particular context. For example, a model for capacitance can be described in the context of a wire, pin, or rule. A model for delay can be described in the context of a cell or wire. In a non-orthogonal approach, different keywords might be used for cell delay, wire delay, etc., and the fundamental semantics of delay would not be inherited by each construct.

Re-usability and self-extensibility

ALF supports the language constructs *template* and *group*, which allow for efficient representation of replicated statements with parameterized values. As these constructs follow the principle of orthogonality, a template can be used for parameterizing any ALF statement and not just a particular type of statements, such as a lookup table. ALF also supports language constructs for the definition of new keywords, their usage for construction of statements, and the context where they can be used.

In summary, ALF is a well-structured language that supports a true superset of virtually all existing library formats. Its conciseness and unique description features make it well-suited for innovative EDA applications.

Participants

At the time this standard was completed, the Advanced Library Format Working Group had the following membership:

Wolfgang Roethig, *Chair*
Joe Daniels, *Technical Editor*

Jay Abraham	Vassilios Gerousis	Wolfgang Roethig
Tim Baldwin	Kevin Grotjohn	Sergei Sokolov
Dennis Brophy	Jake Karrfalt	Vinay Srinivas
Peter Christensen	Srinivas Madaboosi	Balchandra Thatte
Joe Daniels	Nancy Nettleton	Tak Young
Gregory DuFour	Kim Nguyen	Alex Zamfirescu
Timothy Ehrler	Cynthia Parrish	
Uma Ekabaram	Steffen Rochel	

The following members of the balloting group voted on this standard. Balloters may have voted for approval, disapproval, or abstention.

(to be supplied by IEEE)

Table of Contents

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

1. Overview

This clause explains the scope and purpose of this standard, gives an overview of applications of this standard, explains the conventions used in this standard and summarizes the contents of this standard.

1.1 Scope and purpose of this standard

The scope of this standard is to serve as the data specification language of library elements for design applications used to implement an integrated circuit (IC). The range of abstraction shall include from the register-transfer level (RTL) to the physical level. The language shall model behavior, timing, power, signal integrity, physical abstraction and physical implementation rules of library elements.

Library elements for implementation of an IC include sets of predefined components, composed of transistors and interconnect, and sets of predefined rules for the assembly of such components. The design of application-specific ICs (ASICs) in particular relies on the availability of predefined components, called cells. An IC that uses large predefined compound library elements with a standardized functionality, for example, microprocessors as building blocks, is called a system on a chip (SOC).

The design of an ASIC or of an SOC involves electronic design automation (EDA) tools. These tools assist the designer in the choice and assembly of library elements for creating and implementing the IC and verifying the functionality and performance specification of the IC. In order to create an IC involving several million instances of library elements within a manageable time period counted in weeks or months, the usage of EDA tools is mandatory.

A suitable description of library elements for design applications involving EDA tools is required. A key feature is to represent a library element at a level of abstraction that does not reveal the implementation of the library element itself. This is important for the following reasons:

- The complexity of the design data itself mandates data reduction.
- The complexity of the verification process, i.e., the verification for functional, physical, and electrical correctness, mandates a library elements is already characterized and verified by itself. Only the data necessary for creation and verification of the assembled IC is represented in the library.
- A library element is considered an intellectual property (IP) of the library provider.

Therefore, the purpose of this standard is to provide a modeling language and semantics for the functional, physical, and electrical performance description of technology-specific libraries for cell-based and block-based design. Without a standard, EDA tools would use multiple proprietary and tool-specific library descriptions. The semantics would be defined by tool implementations only, which are subject to change and prone to misinterpretation. Also there would be redundancy using multiple descriptions for similar library aspects. Therefore, this standard is proposed to create a consistent library view suitable as a reference for IC designers as well as for electronic design automation (EDA) tool developers and integrators.

1.2 Application of this standard

The ALF standard can be used in many different places throughout the design flow. The major use include creation and characterization of library elements, basic implementation and performance analysis of an IC, and hierarchical implementation and virtual prototyping of an IC.

An application, as described in 1.2.1 through 1.2.3, shall be called *compliant to ALF*, if and only if it satisfies the following criteria:

a) An application tool that uses ALF as input is capable of parsing any ALF file according to the rules specified in Clause 5 through Clause 10, even if not all data in that file is used by the application. In this way, one ALF library can be used for multiple applications with different scope.
b) A tool, as referred to in (a), uses a well-defined set of data from the ALF file within the scope of its application and interprets this data according to the rules specified in Clause 5 through Clause 10. In this way, any two applications using the same set of ALF data interpret the ALF data in the same consistent way.
c) An application tool that uses ALF as output is capable of generating an ALF file according to the rules specified in Clause 5 through Clause 10, and the generated file contains a well-defined set of data for an application as referred to in (a).

The following conventions are used in the flow diagrams depicted in Figure 1 through Figure 4:

- *Rectangle*: data file, format optionally indicated in parentheses
- *Oval*: application
- *Solid arrow*: existing, established function in the design flow
- *Dotted arrow*: possible design flow

1.2.1 Creation and characterization of library elements

ALF can be used to specify the desired functionality and characterization space of a library element, i.e., a cell.

The application for creation of a cell is shown in Figure 1.

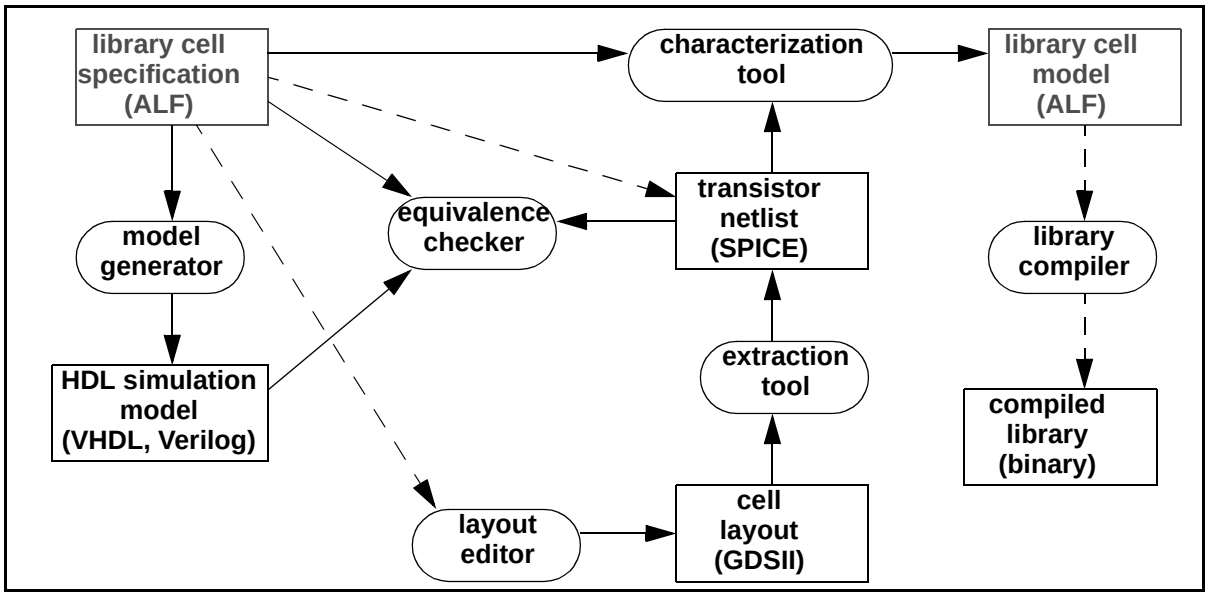

Figure 1—Cell library creation flow

A specification of a library element, i.e., a cell (see 8.2, 8.4), can be described in ALF. This specification includes the name of the cell and its terminals, i.e., pins (see 8.6) and a formal description of the function (see 9.1) performed by the cell. This formal description is sufficient for the purpose of generating hardware description language (HDL) simulation models in various languages, for example, VHDL (see IEEE Std 1076-2002)[1] or Verilog (see IEEE Std 1364-2001).

Multiple HDL models can be generated for different purposes, where the difference is defined by the user's preference for modeling style rather than by the functionality of the cell. For example, one model can handle unknown logic states in a crude way, resulting in fast simulation, while another model can handle unknown logic states in a case-by-case way, resulting is slow but more accurate simulation. The ALF model can serve as a common reference for all those HDL models.

A physical layout of a cell can be represented in the GDSII format. A transistor-level netlist of a cell in SPICE format [B4][2] can be extracted from the physical layout. Such a transistor netlist includes parasitic electrical components. Alternatively, a designer can create a transistor netlist by hand or by using an EDA tool that maps a functional specification described in ALF into a transistor level netlist. Such a transistor netlist is less accurate than one extracted from layout, but can still be useful for prototyping a library.

Both the transistor netlist and the various HDL models can be compared against the functional specification described in ALF. More importantly, the transistor netlist can be used to characterize the performance of the cell, i.e., measure timing, power, noise (see 10.11), and other electrical characteristics (see 10.15) by running a SPICE simulation. The set of necessary SPICE simulations is determined and controlled by a characterization tool. The characterization tool can infer pertinent information from the specification represented in ALF, as far as this information relates to the functionality of the cell itself. For example, the timing arcs that need to be characterized can be represented in or inferred from ALF. The output of the characterization tool is a library cell model, populated with characterization data, also represented in ALF.

[1]For information on references, see Clause 2.

[2]The numbers in brackets correspond to those in the bibliography in Annex D.

Optionally, a library compiler can be used to combine all the library cell models into a binary file, as a data preparation step for an EDA application tool.

1.2.2 Basic implementation and performance analysis of an IC

The ALF library can be used in an IC implementation flow which uses cells as building blocks, in particular, an ASIC implementation flow.

A basic flow for an IC implementation using cells as building blocks is shown in Figure 2.

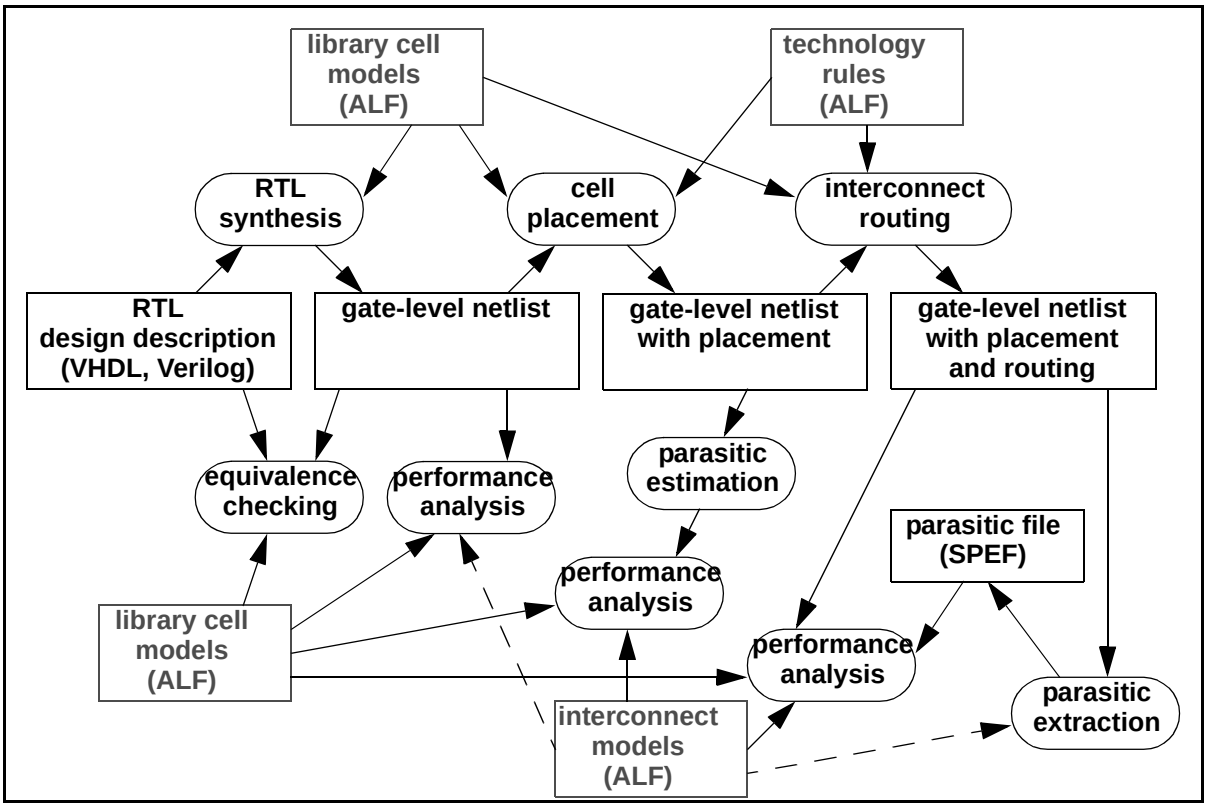

Figure 2—Basic IC implementation flow

In this flow, an RTL design description is transformed into a netlist by an RTL synthesis tool. The netlist contains instances of cells, also called gates, rather than transistors. This application can use the ALF library to find the library elements needed to map the RTL description into a netlist containing instances of cells. The transistors inside the cells are not described in the ALF cell models.

An equivalence checking tool can be used to decide whether the RTL-to-netlist transformation has been done correctly, by comparing the RTL design description with the netlist. This application can use the same ALF library as the RTL synthesis tool. Also, an HDL simulation tool (not shown in Figure 2) can be used to decide whether both the RTL design description and the netlist behave as expected in response to a given stimulus. The simulation tool can use an ALF model or an HDL model derived from the ALF model (see 9.4, 9.6).

The flow in Figure 2 is simplified. Special netlist transformations, such as the creation of data path structures, creation structures related to design for test (DFT), and especially scan insertion, are not shown here. However, the ALF cell models also contain information pertaining to these applications (see 8.5.3, 8.5.5, 8.5.6, 8.8.12, 9.2, 9.7).

The process of cell placement and interconnect routing is summarily referred to as layout. Special layout operations, such as the layout of a power supply structure or layout of a clock network structure, are not explicitly shown in Figure 2. The ALF cell models contain abstract physical information, such as the size and shape of the cell, and the location, size, and shape of the cell pins and routing blockages, which are pertinent for layout (see 8.22, 8.23). Also, abstract information concerning the artwork within the cell can be represented in ALF, for example, the area, perimeter, and connectivity of artwork on specific layers (see 10.18, 10.19). This information is pertinent for manufacturability, such as antenna rule (see 8.21, 10.19.1, 10.19.2, 10.19.3) and metal density checks (see 8.31, 10.19.11).

In addition to cell models, technology rules for routing can also be represented in ALF, such as constraints for the width and length of routing segments, the distance between routing segments, the distance between vias, etc. (see 8.16, 8.18, 8.20, 10.19.7, 10.19.8, 10.19.9).

The implemented IC needs not only be correct in terms of functionality and layout, it also has to meet electrical performance constraints, predominantly timing constraints. Other aspects of electrical performance, such as power consumption, signal integrity, and reliability have become increasingly important. Signal integrity aspects include the cleanliness of signal waveform shapes and the immunity against noise induced by crosstalk and voltage drop (see 10.11.1, 10.11.14, 10.15.1, 10.15.2). Reliability aspects include dependable long-term operation in the presence of electromigration stress, hot electron effect, and thermal instability. The cell models in ALF support characterization data for timing, power, signal integrity, and reliability. For example, reliability data can be described as a limit for voltage, current, or operation frequency (see 10.11.2). A particular feature in ALF is the representation of these data in the context of a stimulus, described by a *vector expression* (see 8.14, 9.12, 9.13). With this feature, the data can be related to particular environmental operation conditions, and a more accurate performance analysis can be performed.

Performance analysis happens within each step of the IC implementation process. RTL synthesis, cell placement and interconnect routing applications have embedded static timing analysis (STA) and other performance analysis capabilities. Also, after completion of each step, a standalone performance analysis can be applied to measure the achieved performance more accurately.

Electrical performance depends not only on the interaction between instances of cells, but also on the parasitics introduced by the interconnect wires. After netlist creation, parasitics can be statistically estimated using a wire load model (WLM). After placement, parasitics can be more accurately predicted by estimating the length of particular routing wires between pins of placed cells. After routing, actual parasitics can be extracted and represented in a file using the standard parasitic exchange format (SPEF) [B5]. An interconnect model in ALF can describe a statistical WLM, a rule for parasitic estimation based on estimated routes, or an interconnect analysis model (see 8.10, 8.12). The interconnect analysis model specifies the desired level of granularity for the parasitics (see 10.15.3, 10.15.4, 10.15.5, 10.16) and the calculation of timing, noise, voltage, or current based on instances of parasitics and on an electrical model of a driver cell. The data for the electrical model of a particular driver cell can be represented in ALF as a part of the cell characterization data.

1.2.3 Hierarchical implementation and virtual prototyping of an IC

An IC implementation flow with cells as building blocks has its limits imposed by the number of objects, i.e., the instances of cells and nets that can be reasonably handled by designers and by application flows.

For ICs exceeding the limits of objects that can be reasonably handled, the following approaches are used, possibly in combination with each other:

— *Bottom-up design*: Create larger building blocks from cells first, then use these blocks for IC implementation.
— *Top-down design*: Divide a design into subdesigns first, implement each subdesign as a block, then assemble the blocks.

— *Virtual prototyping*: Do a simplified so-called virtual implementation of the entire design first, then partition the virtually implemented design into blocks, use the results of the virtual implementation as constraints for actual implementation of each block, and implement and assemble the blocks.

The common denominator for all these methods is creation of blocks, in order to reduce the number of objects seen by the application.

The application for creation of a block is shown in Figure 3.

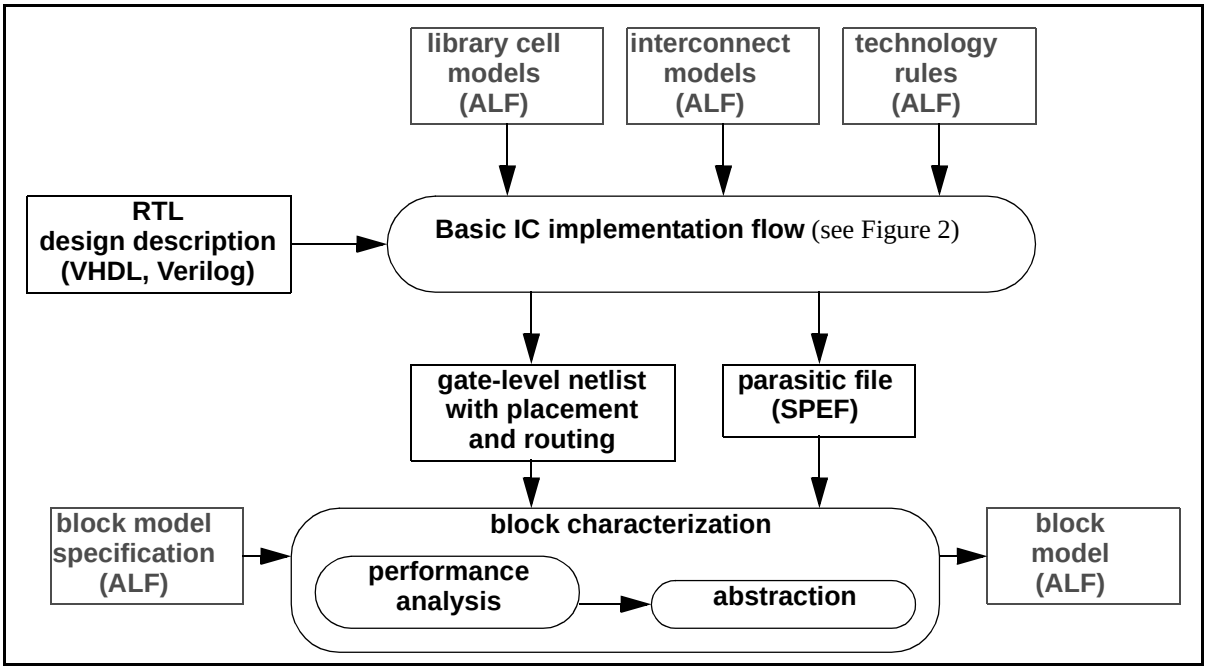

Figure 3—Block creation flow

A block can be created by using the basic IC implementation flow (see Figure 2). A block with a functionality that can be used and re-used is commonly referred to as intellectual property (IP) of the designer. In case of a "hard" block, the primary output of the implementation flow, i.e., a gate-level netlist with placement and routing, is preserved and eventually transformed into a physical artwork. In case of a "soft" block, only the primary input of the implementation flow, i.e., the RTL design description, is preserved. The output of the implementation flow serves only for the purpose of block characterization, i.e., creation of an abstract model for the block. The block characterization consists of a repeated application of performance analysis within the range of desired characterization followed by abstraction. Abstraction includes reduction of the physical implementation data and association of the performance analysis data with a specified model. Both the specification of the model and the model itself can be represented in ALF.

Variants to this flow include partial IC implementation, for example, only RTL synthesis and placement without routing, especially in the case of a soft block, where the implementation data is not preserved. The rationale for not preserving the implementation data of a block is the possibility of achieving a better overall IC implementation result by implementing the block later in context of other blocks, instead of implementing the block standalone up front.

Depending on whether a block is used as a hard block or a soft block, the ALF model can represent a different level of abstraction. An ALF model for a hard block can have similar features as an ALF model for a cell (see 1.2.1 and 1.2.2). In addition, the netlist and the parasitics representing the output of the implementation flow can

be partially preserved in the ALF model, especially at the boundary of the block (see 9.5). This enables accurate analysis of the electrical interaction of a block with adjacent blocks in the context of an IC implementation. On the other hand, an ALF model for a soft block can represent a statistical range or upper and lower bounds (see 10.5) for characterization data rather than “hard” characterization data, since there is a degree of variability in the implementation of actual instances of the block. Also, a statistical WLM can be encapsulated within the model of the block.

ALF supports specific modeling features for parameterizeable blocks, i.e., blocks which can be implemented in various physical shapes or sizes and with variable bitwidth and performance characteristics. The ALF constructs *group* (see 7.14), *template* (see 7.15), *static* and *dynamic template instantiation* (see 7.16) can be used for this purpose.

Independent of whether a block is a hard block or a soft block, the application for creating the IC can now use the abstract model of the block as a library element rather than using a cell. In a similar way, as an ALF model of a cell does not reveal transistor-level implementation details, an ALF model of a block does not reveal gate-level implementation details. However, the ALF model of a block still provides enough information for an application to implement or explore the implementation of an IC and analyze the performance and the compliance to logical and physical design constraints.

An IC is designed in the context of a specific environment with specific constraints. Environmental constraints include for the characteristics of the package, the printed board, the range of process, voltage, and temperature (PVT) conditions (see 10.14). Other constraints are given by globally applicable physical design rules, for example, the available routing layers, the amount of routing resources reserved for the power distribution, and the available locations for IO pins at the boundary and in the center of a chip. The virtual prototyping approach can be used to evaluate whether a design can be implemented within these constraints. The electrical characterization data in ALF, i.e., timing, power, noise, physical and electrical rules, estimation models for parasitics, etc., can be represented as mathematical functions of environmental conditions and constraints (see 10.3, 10.4).

A conceptual flow for the virtual prototyping and hierarchical implementation of an IC involving ALF models at different levels of abstraction is shown in Figure 4.

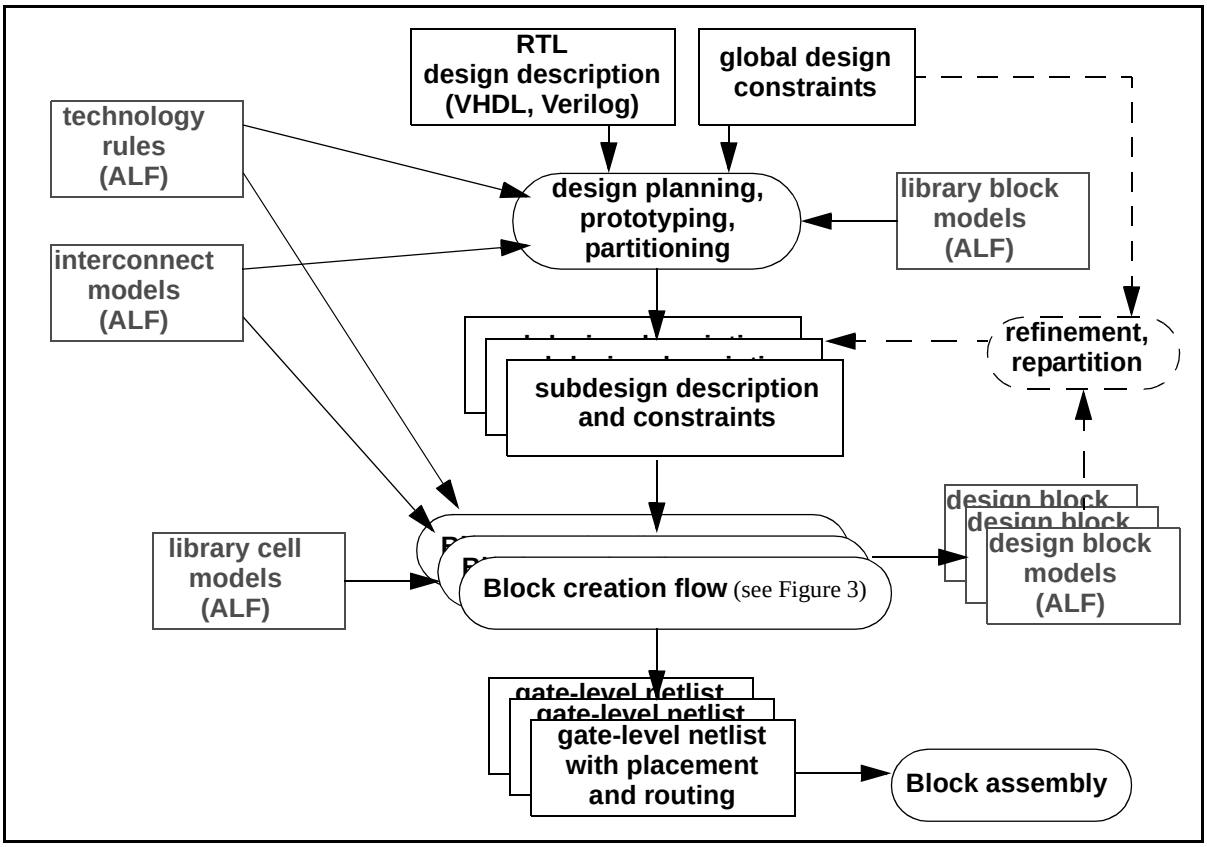

Figure 4—IC prototyping and hierarchical implementation flow

The design planning and prototyping application uses predefined models of blocks as library elements, referred to as “library block models”. The design is partitioned into subdesigns. The block creation flow (see Figure 3), i.e., a combination of block implementation and block characterization, is applied to each subdesign. The applicable library elements for each block are cells. The outputs of the block creation flow are the characterized models of the subdesigns, referred to as “design block models”. The design block models can be used to iterate on the design planning application, resulting in a possible refinement and repartitioning of the design. Once the evaluation of each block against the subdesign constraints and the evaluation of the virtually assembled blocks against the global design constraints are satisfactory, the block implementation results, i.e., the netlist with placement and routing for each block, can actually be assembled to form the IC.

The design of an IC can use a combination of cells, hard blocks and soft blocks, blocks with fixed specification, and parameterizeable blocks as library elements. Some of the library elements are available independent of the design, others are created during and only for the purpose of that particular design. An abstract model for a soft block can be used in conjunction with a more detailed model for a hard block. The abstract model can be replaced with a more detailed model during implementation of the block. Technology rules and interconnect models are used throughout the flow.

In summary, the ALF standard provides a common modeling language for library elements, technology rules, and interconnect models. ALF models at different levels of abstraction can be used concurrently by EDA applications for planning, prototyping, implementation, analysis, optimization and verification of complex ICs.

1.3 Conventions used in this standard

The syntax for description of lexical and syntax rules uses the following conventions.

```
::=     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    boldface specifies verbatim usage of a string of characters.
ITEM    uppercase boldface specifies verbatim usage of a keyword.
prefix_item
        prefix in italic is for explanation purpose only
PREFIX_item
        prefix in uppercase italic indicates that a keyword is used
```

NOTE: These conventions do not prescribe usage of uppercase or lowercase characters, as ALF is case-insensitive.

1.4 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 this standard.
- Clause 3 (Definitions) defines terms used throughout the different specifications contained in this standard.
- Clause 4 (Acronyms) defines the acronyms used in this standard.
- Clause 5 (ALF language construction principles) defines the language construction principles used in this standard.
- Clause 6 (Lexical rules) specifies the lexical rules.
- Clause 7 (Generic objects and related statements) defines syntax and semantics of generic objects used in this standard.
- Clause 8 (Library-specific objects and related statements) defines syntax and semantics of library-specific objects used in this standard.
- Clause 9 (Description of functional and physical implementation) defines syntax and semantics of statements related to functional and physical implementation of library elements used in this standard
- Clause 10 (Description of electrical and physical measurements) defines syntax and semantics of statements describing electrical and physical measurements related to library elements used in this standard.
- Annexes. Following Clause 10 are a series of informative annexes.

2. References

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.

IEEE Std 1076-2002, IEEE Standard VHDL Language Reference Manual.

IEEE Std 1364-2001, IEEE Standard for Verilog Hardware Description Language.

IEEE Std 1497-2001, IEEE Standard for Standard Delay Format (SDF) for the Electronic Design Process.

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

ANSI/ISO/IEC 14882, C++ Standard.

ISO/IEC 8859-1: 1987(E), ASCII character set.[3]

"American National Standard for Use of the International System of Units (SI): The Modern Metric System", IEEE/ASTM SI 10-2002.

[3]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* [B1] should be consulted for terms not defined in this standard.

3.1 **ALF:** *See:* **advanced library format**.

3.2 **ALF name:** The name of an ALF object.

3.3 **ALF object:** An element described in ALF.

3.4 **ALF type:** The type of an ALF object.

3.5 **ALF value:** A value associated with an ALF object.

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

3.7 **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**.

3.8 **arc**: *See:* **timing arc**.

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

3.10 **arithmetic model:** A description of a mathematical model for an electrical or physical measurement in ALF.

3.11 **cell, library cell:** An electronic circuit that is a component of a library described in ALF.

3.12 **geometric model:** A description of a layout geometry in ALF.

3.13 **register transfer level:** A technology-independent description of a digital electronic design allowing inference of sequential and combinatorial logic components.

3.14 **timing arc:** An abstract representation of a measurement of an interval between two points in time during operation of a library cell.

4. Acronyms

This clause lists the acronyms used in this standard.

ALF	advanced library format, title of the herein proposed standard
ASIC	application specific integrated circuit
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
CPU	central processing unit
DFT	design for test
DSP	digital signal processor
EDA	electronic design automation
EDIF	electronic design interchange format
GPU	graphical processing unit
HDL	hardware description language
IC	integrated circuit
IP	intellectual property
LSB	least significant bit
LSSD	level-sensitive scan design
MPU	micro processor unit
MSB	most significant bit
PLL	phase-locked loop
PVT	process/voltage/temperature (denoting a set of environmental conditions)
RAM	random access memory
RC	resistance (times) capacitance
ROM	read-only memory
RTL	register transfer level
SDF	standard delay format (see IEEE Std 1497-2001)
SOC	system on a chip
SPEF	standard parasitic exchange format (see IEEE Std 1481-1999)
SPICE	simulation program with integrated circuit emphasis [B4]
STA	static timing analysis
VHDL	VHSIC hardware description language (see IEEE Std 1076-2002)
VHSIC	very high-speed integrated circuit
VLSI	very large-scale integration
WLM	wire load model

5. ALF language construction principles

This section presents the ALF language construction principles and gives an overview of the language features. The ALF statements and the rules for relationships between ALF statements are presented summarily. Keywords are involved in the declaration of ALF statements. The keywords in ALF shall be case-insensitive. However, uppercase is used for keywords throughout this section for clarity.

5.1 ALF metalanguage

Syntax 1 establishes an *ALF metalanguage*.

```
ALF_statement ::=
      ALF_type [ [ index ] ALF_name [ index ] ] [ = ALF_value ] ;
    | ALF_type [ [ index ] ALF_name [ index ] ] [ = ALF_value ] { { ALF_value | : | ; } }
    | ALF_type [ [ index ] ALF_name [ index ] ] [ = ALF_value ] { { ALF_statement } }
ALF_type ::=
      identifier
    | @
    | :
ALF_name ::=
      identifier
    | control_expression
ALF_value ::=
      number
    | multiplier_prefix_symbol
    | identifier
    | quoted_string
    | bit_literal
    | based_literal
    | edge_value
    | arithmetic_expression
    | boolean_expression
    | control_expression
```

Syntax 1—Syntax construction for ALF metalanguage

The *ALF type* is defined by an *identifier* (see 6.13) or by the *operator* "**@**" (see 6.4) or by the *delimiter* "**:**" (see 6.3). The usage of an identifier, an operator, or a delimiter as ALF type is defined by ALF language rules concerning the particular ALF type. The identifier can be a predefined *keyword* (see 6.13.7).

The *ALF name* is defined by an *identifier* (see 6.13) or by a *control expression* (see 9.4). Depending on the ALF type, the ALF name is mandatory or optional or not applicable. The usage of an identifier or a control expression as ALF name is defined by ALF language rules concerning the particular ALF type. The ALF name is optionally preceded by an *index* (see 6.6) to specify a *vectorized object*. Another index can optionally succeed the ALF name to specify a two-dimensional vectorized object. A two-dimensional vectorized object shall be called *matrix object*. An object without index shall be called *scalar object*. The usage of an index in conjunction with an ALF name is defined by ALF language rules concerning the particular ALF type.

The *ALF value* is defined by a *number* (see 6.5), a *multiplier prefix symbol* (see 6.7), an *identifier* (see 6.13), a *quoted string* (see 6.14), a *bit literal* (see 6.8), a *based literal* (see 6.9), an *edge value* (see 6.12), an *arithmetic expression* (see 10.1), a *boolean expression* (see 9.9), or a *control expression* (see 9.4). Depending on the type of the ALF statement, the ALF value is mandatory or optional or not applicable. The usage of a particular kind of 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 a child is called a *compound* ALF statement. An ALF statement that is related to another ALF statement by ancestry in the parent/child relationship is called an *ancestor* of the other ALF statement. Conversely, the latter is called a *descendant* of the former.

An ALF statement containing one or more ALF values, possibly 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.

Example

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
}
```

End of example

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 1.

Table 1—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.
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 *library-specific object*.
Arithmetic model	Describe an abstract mathematical quantity that can be calculated and possibly 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 *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 *arithmetic model*.
Geometric model	Describe an abstract geometry 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 or semi compound. Name does not apply. Value is mandatory, if atomic. Value does not apply, if semi compound.
Annotation container	Provide a context for an annotation.	Statement is compound. Name and value do not apply. Category of child is *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 5 illustrates the parent/child relationship between categories of statements.

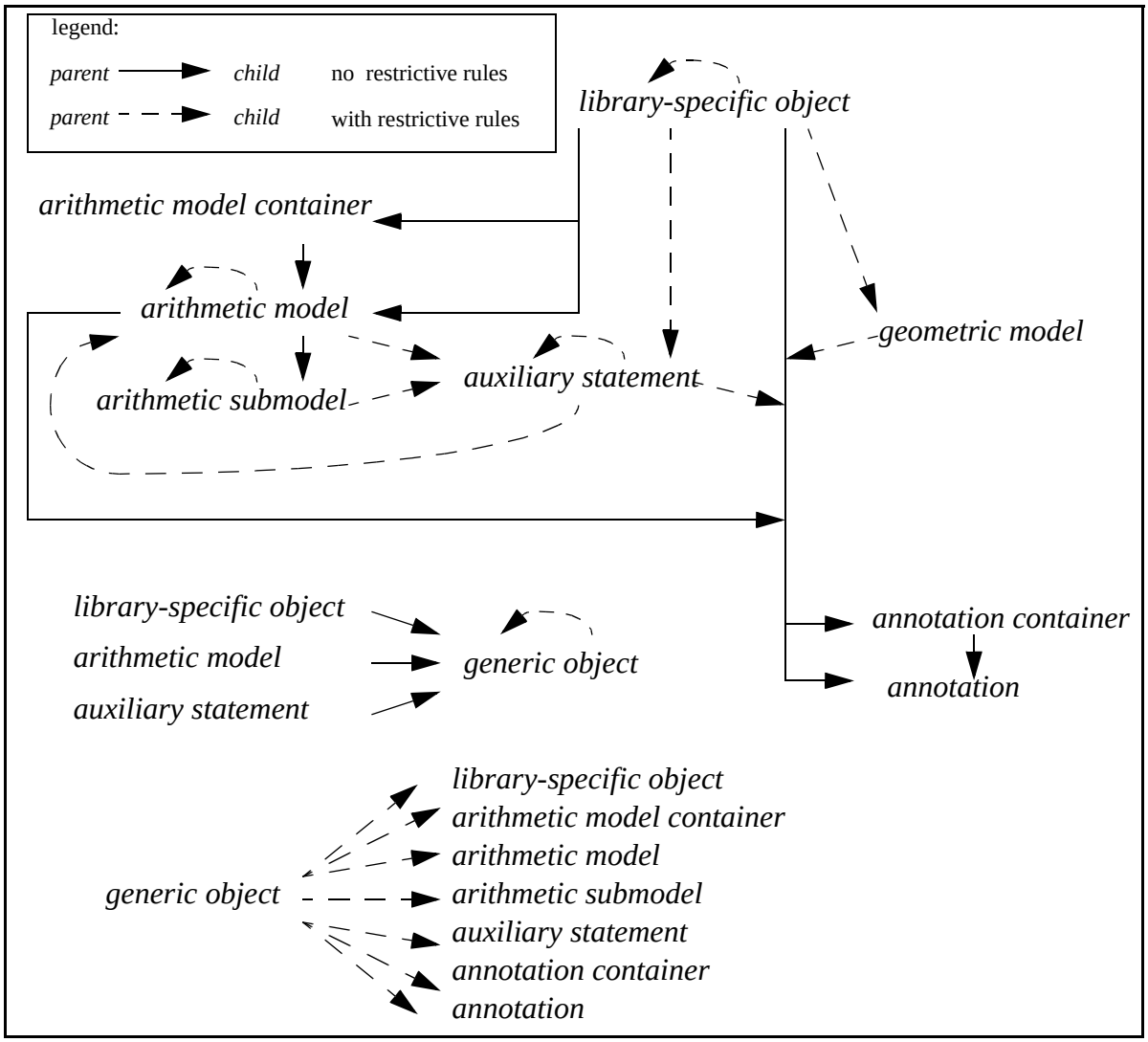

Figure 5—Parent/child relationship between ALF statements

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

Table 2—Generic objects

Keyword	Item	Section
ALIAS	Alias declaration	See 7.7.
CONSTANT	Constant declaration	See 7.8.

Table 2—Generic objects (Continued)

Keyword	Item	Section
CLASS	Class declaration	See 7.12.
GROUP	Group declaration	See 7.14.
KEYWORD	Keyword declaration	See 7.9.
SEMANTICS	Semantics declaration	See 7.10.
TEMPLATE	Template declaration	See 7.15.

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

Table 3—Library-specific objects

Keyword	Item	Section
ANTENNA	Antenna declaration	See 8.21.
ARRAY	Array declaration	See 8.27.
BLOCKAGE	Blockage declaration	See 8.22.
CELL	Cell declaration	See 8.4.
LAYER	Layer declaration	See 8.16.
LIBRARY	Library declaration	See 8.2.
NODE	Node declaration	See 8.12.
PATTERN	Pattern declaration	See 8.29.
PIN	Pin declaration	See 8.6.
PINGROUP	Pin group declaration	See 8.7.
PORT	Port declaration	See 8.23.
PRIMITIVE	Primitive declaration	See 8.9.
REGION	Region declaration	See 8.31.
RULE	Rule declaration	See 8.20.
SITE	Site declaration	See 8.25.
SUBLIBRARY	Sublibrary declaration	See 8.2.
VECTOR	Vector declaration	See 8.14.
VIA	Via declaration	See 8.18.
WIRE	Wire declaration	See 8.10.

Figure 6 illustrates the parent/child relationship between statements within the category *library-specific object.*

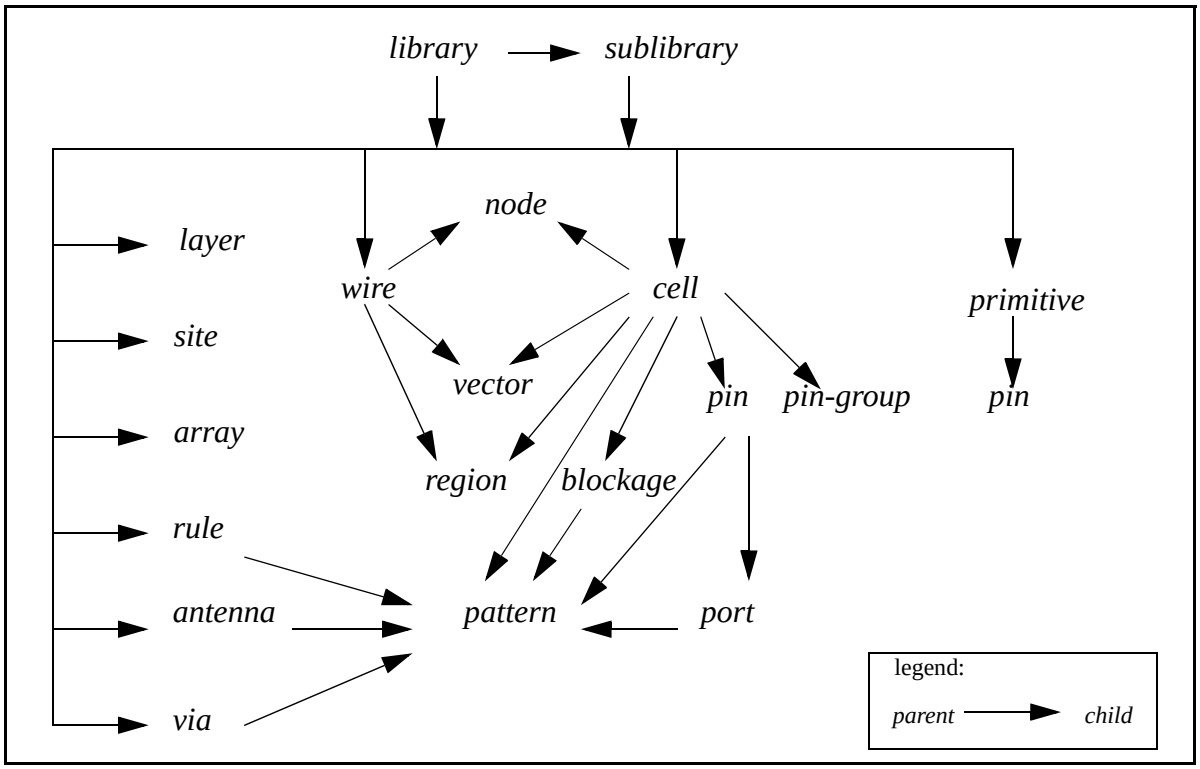

Figure 6—Parent/child relationship amongst library-specific objects

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 4 lists the singular statements.

Table 4—Singular statements

Keyword	Item	Value	Complexity	Section
FUNCTION	Function statement	N/A	Compound	See 9.1.
TEST	Test statement	N/A	Compound	See 9.2.
RANGE	Range statement	N/A	Semi compound	See 9.8.
FROM	From statement	N/A	Compound	See 10.12.
TO	To statement	N/A	Compound	See 10.12.

Table 4—Singular statements (Continued)

Keyword	Item	Value	Complexity	Section
VIOLATION	Violation statement	N/A	Compound	See 10.10.
HEADER	Header statement	N/A	Compound	See 10.4.
TABLE	Table statement	N/A	Semi compound	See 10.4.
EQUATION	Equation statement	N/A	Semi compound	See 10.4.
BEHAVIOR	Behavior statement	N/A	Compound	See 9.4.
STRUCTURE	Structure statement	N/A	Compound	See 9.5.
NON_SCAN_CELL	Non-scan cell statement	Optional	Compound or semi compound	See 9.7.
ARTWORK	Artwork statement	Mandatory	Compound or atomic	See 9.19.

Table 5 lists the plural statements.

Table 5—Plural statements

Keyword	Item	Name	Complexity	Section
STATETABLE	State table statement	Optional	Semi compound	See 9.6.
@	Control statement	Mandatory	Compound	See 9.4.
:	Alternative control statement	Mandatory	Compound	See 9.4.

Figure 7 illustrates the parent/child relationship for singular statements and plural statements.

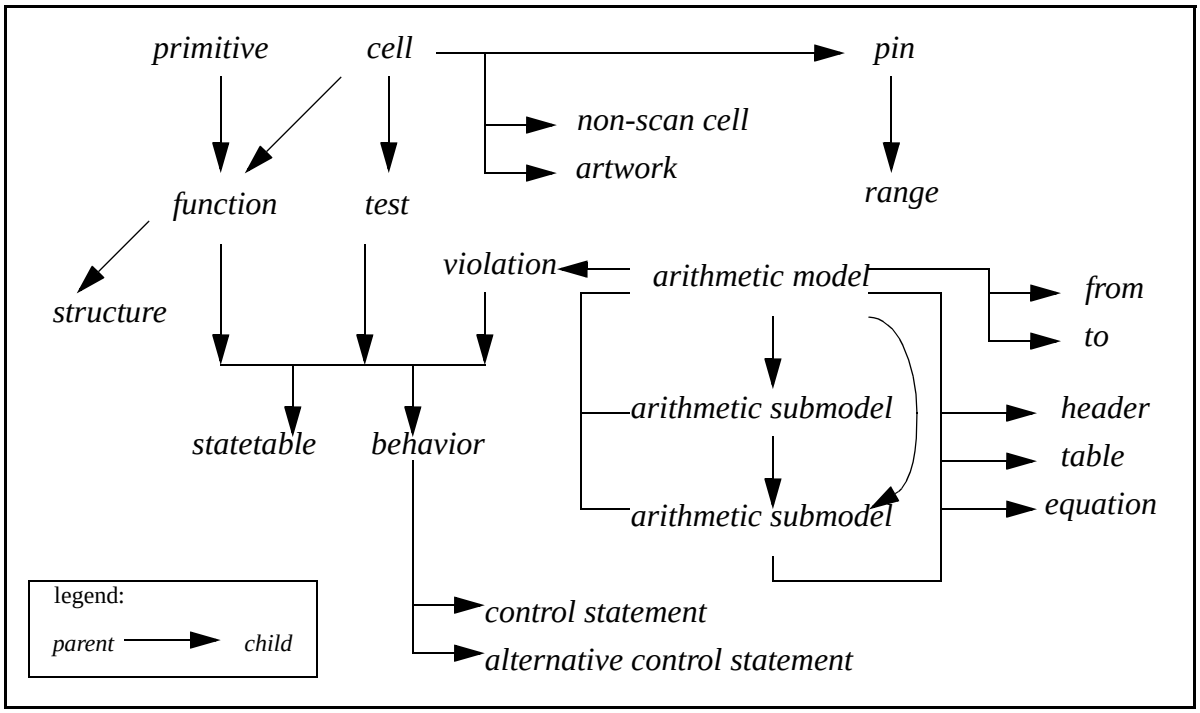

Figure 7—Parent/child relationship involving singular statements and plural statements

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 6 lists the instantiation statements.

Table 6—Instantiation statements

Item	Section
Cell instantiation	See 9.5.
Primitive instantiation	See 9.4.
Template instantiation	See 7.16.
Via instantiation	See 9.20.
Wire instantiation	See 9.15

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 7 lists the assignment statements.

Table 7—Assignment statements

Item	Section
Pin assignment	See 9.3.2, Syntax 68.
Arithmetic assignment	See 7.16, Syntax 42.
Boolean assignment	See 9.4, Syntax 69.

Figure 8 illustrates the parent/child relationship involving instantiation and assignment statements.

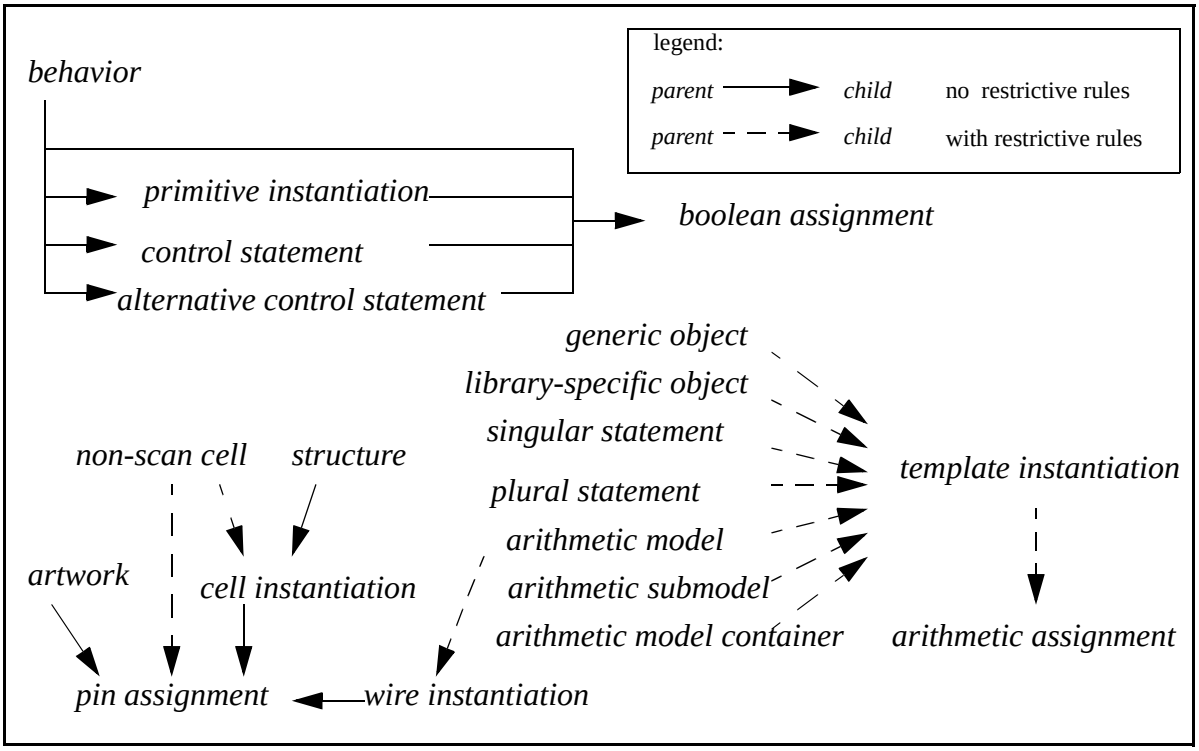

Figure 8—Parent/child relationship involving instantiation and assignment statements

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

Table 8—Other categories of ALF statements

Item	Section
Arithmetic model	See 10.3, Syntax 82.
Arithmetic submodel	See 10.7, Syntax 96.
Arithmetic model container	See 10.8, Syntax 97.
Annotation	See 7.3, Syntax 31.
Annotation container	See 7.4, Syntax 32.
Geometric model	See 9.16, Syntax 77.

There exist predefined keywords with generic semantics in the category *annotation* and *annotation container.* They are called *generic keywords*, comparable to keywords for *generic objects*. Table 9 lists the generic keywords in the category annotation and annotation container.

Table 9—Annotations and annotation containers with generic keyword

Keyword	Item / subcategory	Section
`PROPERTY`	Annotation container.	See 7.6.
`ATTRIBUTE`	Multi-value annotation.	See 7.5.
`INFORMATION`	Annotation container.	See 8.3.2.

Table 10 lists predefined keywords in categories related to arithmetic model.

Table 10—Keywords related to arithmetic model

Keyword	Item / category	Section
`LIMIT`	Arithmetic model container.	See 10.8.2.
`MIN`	Arithmetic submodel, also operator within *arithmetic expression.*	See 10.5, 10.2.3.
`MAX`	Arithmetic submodel, also operator within *arithmetic expression.*	See 10.5, 10.2.3.
`TYP`	Arithmetic submodel.	See 10.5.
`DEFAULT`	Annotation.	See 10.9.4.
`ABS`	Operator within *arithmetic expression*.	See 10.2.3.
`EXP`	Operator within *arithmetic expression*.	See 10.2.3.
`LOG`	Operator within *arithmetic expression*.	See 10.2.3.

The definitions of other predefined keywords, especially in the category arithmetic model, can be self-described in ALF using the *keyword declaration* statement (see 7.9).

5.7 Statements for parser control

Table 11 provides a reference to statements used for ALF parser control.

Table 11—Statements for ALF parser control

Keyword	Statement	Section
`INCLUDE`	Include statement	See 7.17.
`ASSOCIATE`	Associate statement	See 7.18.
`ALF_REVISION`	Revision statement	See 7.19.

The statements for parser control do not necessarily follow the ALF metalanguage 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 (see ISO/IEC 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 | horizontal_tab | new_line | vertical_tab | form_feed | carriage_return
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 divided into categories

Table 12 shows the list of *whitespace* characters and their ASCII code.

Table 12—List of whitespace characters

Name	ASCII code (octal)
Space	040
Horizontal tab	011
New line	012
Vertical tab	013

Table 12—List of whitespace characters (Continued)

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

Table 13 shows the list of *special* characters and their names used in this standard.

Table 13—List of special characters

Symbol	Name
&	Ampersand
|	Vertical bar
^	Caret
~	Tilde
+	Plus
-	Dash
*	Asterix
/	Slash
%	Percent
?	Question mark
!	Exclamation mark
:	Colon
;	Semicolon
,	Comma
"	Double quote
'	Single quote
@	At sign
=	Equal sign
\	Backslash
.	Dot
$	Dollar
_	Underscore

Table 13—List of special characters (Continued)

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

The start of a block comment shall be determined by the occurrence of a *slash* character followed by an *asterix* without whitespace in-between. The end of a block comment shall be determined by the occurrence 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 operator*, and *meta operator*, as shown in Syntax 5.

```
operator ::=
      arithmetic_operator
    | boolean_operator
    | relational_operator
    | shift_operator
    | event_operator
    | meta_operator
arithmetic_operator ::=
    + | - | * | / | % | **
boolean_operator ::=
    && | || | ~& | ~| | ^ | ~^ | ~ | ! | & | |
relational_operator ::=
    == | != | >= | <= | > | <
shift_operator ::=
    << | >>
event_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 14 shows the list of arithmetic operators and their names used in this standard.

Table 14—List arithmetic operators

Symbol	Operator name	Unary / binary	Section
+	Plus	Binary	See 9.11.4.
-	Minus	Both	See 9.11.4.
*	Multiply	Binary	See 9.11.4.
/	Divide	Binary	See 9.11.4.
%	Modulus	Binary	See 9.11.4.
**	Power	Binary	See 10.2.2.

Arithmetic operators shall be used to specify arithmetic operations.

6.4.2 Boolean operator

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

Table 15—List of boolean operators

Symbol	Operator name	Unary / binary	Section
!	Logical inversion	Unary	See 9.11.1.
&&	Logical and	Binary	See 9.11.1.
||	Logical or	Binary	See 9.11.1.
~	bit-wise inversion	Unary	See 9.11.2.
&	bit-wise and	Both	See 9.11.2.
~&	bit-wise nand	Both	See 9.11.2.
|	bit-wise or	Both	See 9.11.2.
~|	bit-wise nor	Both	See 9.11.2.
^	Exclusive or	Both	See 9.11.2.
~^	Exclusive nor	Both	See 9.11.2.

Boolean operators shall be used to specify boolean operations.

6.4.3 Relational operator

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

Table 16—List of relational operators

Symbol	Operator name	Unary / binary	Section
==	Equal	Binary	See 9.11.6.
!=	Not equal	Binary	See 9.11.6.
>	Greater	Binary	See 9.11.6.
<	Lesser	Binary	See 9.11.6.
>=	Greater or equal	Binary	See 9.11.6.
<=	Lesser or equal	Binary	See 9.11.6.

Relational operators shall be used to specify mathematical relationships between numbers.

6.4.4 Shift operator

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

Table 17—List of shift operators

Symbol	Operator name	Unary / binary	Section
<<	Shift left	Binary	See 9.11.5.
>>	Shift right	Binary	See 9.11.5.

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

6.4.5 Event operator

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

Table 18—List of event operators

Symbol	Operator name	Unary / binary	Section
->	Immediately followed by	Binary	See 9.13.2.
~>	Eventually followed by	Binary	See 9.13.2.
<->	Immediately following each other	Binary	See 9.13.3.
<~>	Eventually following each other	Binary	See 9.13.3.
&>	Simultaneous or immediately followed by	Binary	See 9.13.3.
<&>	Simultaneous or immediately following each other	Binary	See 9.13.3.

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

6.4.6 Meta operator

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

Table 19—List of meta operators

Symbol	Operator name	Unary / binary	Section
=	Assignment	Binary	See 9.3.2, 7.16, 9.4.
?	Condition	Binary	See 9.13.5.
@	Control	Unary	See 9.4.

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 ::=
     mantissa [ exponent ]
    | unsigned_integer exponent
sign ::=
    + | -
mantissa ::=
     . unsigned_integer
    | unsigned_integer . [ unsigned_integer ]
exponent ::=
     E [ sign ] unsigned_integer
    | e [ sign ] unsigned_integer
```

Syntax 6—Number

A number shall be used to represent a numerical quantity.

6.6 Index value and Index

An *index value* shall be defined as shown in Syntax 7.

```
index_value ::=
    unsigned_integer | atomic_identifier
```

Syntax 7—Index value

The purpose of an *index value* is to represent a position within a range of discrete, countable values. A discrete, countable value shall be represented by an *unsigned integer* (see 6.5). The usage of *atomic identifier* (see 6.13) as index value shall only be allowed, if the semantic interpretation of the atomic identifier resolves to a value of the category *unsigned integer.*

An index value can represent a particular position within a *pin* of the category *vector pin*, a *matrix pin* (see 8.6) or a *pingroup* (see 8.7).

An index value can also be used in the context of a *group* declaration (see 7.14) and in the context of a *range* statement (see 9.8).

An *index* shall be defined as shown in Syntax 8.

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

Syntax 8—Index

An *index* shall be used in conjunction with the name of a *pingroup*, a *vector pin* or a *matrix pin*. 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 position of the most significant bit (MSB) is specified by the left index value and the position of the least significant bit (LSB) is specified by the right index value.

6.7 Multiplier prefix symbol and multiplier prefix value

A *multiplier prefix symbol* shall be defined as shown in Syntax 9.

```
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 9—Multiplier prefix symbol

The purpose of a multiplier prefix symbol is the specification of a multiplier for the base unit associated with an *arithmetic model* (see 10.3). 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.

A multiplier prefix symbol shall relate to the *International System of Units* (see U.S. National Bureau of Standards, Spec. Pub. 330) as shown in Table 20.

Table 20—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	1e0
K	k	kilo	1e+3
MEG	M	mega	1e+6
G	G	giga	1e+9

A *multiplier prefix value* shall be defined as shown in Syntax 10.

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

Syntax 10—Multiplier prefix value

The *multiplier prefix value* shall be represented either as an *unsigned number* (see 6.5) or a *multiplier prefix symbol* (see 6.7). An application shall interpret a multiplier prefix value semantically as unsigned number.

6.8 Bit literal

Bit literals shall be divided into the subcategories *alphanumeric bit literal* and *symbolic bit literal*, as shown in Syntax 11.

```
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 11—Bit literal

Bit literals shall be used to specify scalar values within a boolean value system (see 9.10).

6.9 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 12.

```
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 12—Based literal

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

6.10 Boolean value

A *boolean value* shall be defined as shown in Syntax 13.

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

Syntax 13—Boolean value

The semantics of a boolean value are explained in section 9.10.

6.11 Arithmetic value

An *arithmetic value* shall be defined as shown in Syntax 14.

An arithmetic value shall represent data for an *arithmetic model* (see 10.3) or for an *arithmetic assignment* (see 7.16). Semantic restrictions apply, depending on the particular type of arithmetic model.

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

Syntax 14—Arithmetic value

6.12 Edge literal and edge value

Edge literals shall be divided into subcategories *bit edge literal*, *based edge literal*, and *symbolic edge literal*, as shown in Syntax 15.

```
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 15—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.

An *edge value* shall be defined as shown in Syntax 16.

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

Syntax 16—Edge value

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

6.13 Identifier

Identifiers shall be divided into the subcategories *atomic identifier*, *indexed identifier*, *hierarchical identifier* and *escaped identifier*, as shown in Syntax 17. The subcategory *atomic identifier* shall be further divided into *non-escaped identifier* and *placeholder identifier*. The subcategory *hierarchical identifier* shall be further divided into *full hierarchical identifier* and *partial hierarchical identifier*.

```
identifier ::=
    atomic_identifier | indexed_identifier | hierarchical_identifier | escaped_identifier
atomic_identifier ::=
    non_escaped_identifier | placeholder_identifier
hierarchical_identifier ::=
    full_hierarchical_identifier | partial_hierarchical_identifier
```

Syntax 17—Identifier

An identifier shall be used to specify an *ALF name* or an *ALF value*. An identifier can also appear as a *variable* in an *arithmetic expression* (see 10.1), in a *boolean expression* (see 9.9) or in a *vector expression* (see 9.12).

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

6.13.1 Non-escaped identifier

A *non-escaped identifier* shall be defined as shown in Syntax 18.

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

Syntax 18—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.

6.13.2 Placeholder identifier

A *placeholder identifier* shall be defined as a *non-escaped identifier* enclosed by angular brackets without whitespace, as shown in Syntax 19.

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

Syntax 19—Placeholder identifier

A placeholder identifier shall be used to represent a formal parameter in a *template* statement (see 7.15), which is to be replaced by an actual parameter in a *template instantiation* statement (see 7.16).

6.13.3 Indexed identifier

An *indexed identifier* shall be defined as an *atomic identifier* followed by an *index* (see 6.6) without whitespace, as shown in Syntax 20.

```
indexed_identifier ::=
    atomic_identifier index
```

Syntax 20—Indexed identifier

The *atomic identifier* shall be interpreted as the *ALF name* of a one-or a two-dimensional object, i.e., a *vector pin* or a *matrix pin* (see 8.6). The *index* shall be interpreted as the position of a scalar element within a one-dimensional object or a one-dimensional slice within a two-dimensional object.

6.13.4 Full hierarchical identifier

A *full hierarchical identifier* shall be defined as shown in Syntax 21.

```
full_hierarchical_identifier ::=
    atomic_identifier [ index ] . atomic_identifier [ index ] { . atomic_identifier [index ] }
```

Syntax 21—Hierarchical identifier

A *full hierarchical identifier* shall be used to specify a hierarchical name, 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.

6.13.5 Partial hierarchical identifier

A *partial hierarchical identifier* shall be defined as shown in Syntax 22.

```
partial_hierarchical_identifier ::=
    atomic_identifier [ index ] { . atomic_identifier [ index ] } . .
    { atomic_identifier [ index ] { . atomic_identifier [ index ] } . . }
    [ atomic_identifier [ index ] { . atomic_identifier [ index ] } ]
```

Syntax 22—Partial hierarchical identifier

A *partial hierarchical identifier* shall be used to specify an incomplete hierarchical name. The two dots shall indicate that the preceding atomic identifier is an *ancestor* of the subsequent atomic identifier. A partial hierarchical identifier terminated by two dots shall be interpreted as a reference to any possible *descendant* of the preceding ancestor.

NOTE — A restriction as to which descendant is applicable, can be given by a particular syntax or semantic rule.

6.13.6 Escaped identifier

An *escaped identifier* shall be defined as shown in Syntax 23.

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

Syntax 23—Escaped identifier

An *escaped identifier* shall be used to legalize the usage of a special character or the usage of an identifier otherwise reserved as a keyword.

A *dot* within an escaped identifier shall be semantically interpreted in the same way as a dot within a *full hierarchical identifier* (see 6.13.4), unless the dot is immediately preceded by a *backslash*.

A lexical sequence of characters according to Syntax 8 at the end of the escaped identifier or preceding a dot within the escaped identifier shall be interpreted as an *index* (see 6.6) in the same way as within a *full hierarchical identifier* or within an *indexed identifier* (see 6.13.3), unless the lexical sequence of characters is immediately preceded by a *backslash*.

A backslash within an escaped identifier shall semantically be considered part of an ALF name or of an ALF value designated by the escaped identifier, with exception of the leading backslash and a backslash immediately preceding a dot or an index.

Example

\id1[0].id2\[1].\id3\.id4 represents 3 levels of hierarchy.

The ancestor is the element at position **0** of the one-dimensional object "**id1**". The child of "**id1[0]**" is the scalar object "**id2[1]**". The child of "**id2[1]**" is the scalar object "**id3.id4**".

NOTE — The scalar object "**id2[1]**" by itself has to be declared as "**\id2\[1]**". The scalar object "**id3.id4**" by itself has to be declared as "**\id3\.id4**".

End of example

6.13.7 Keyword identifier

Keywords shall be lexically equivalent to non-escaped identifiers. Predefined keywords are listed in Table 2 — Table 5 and Table 9 — Table 11. Additional keywords are predefined in 7.9.

The predefined keywords in this standard shall follow a more restrictive lexical rule than general non-escaped identifiers, as shown in Syntax 24.

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

Syntax 24—Keyword identifier

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.

6.14 Quoted string

A *quoted string* shall be a sequence of zero or more characters enclosed between two double quote characters, as shown in Syntax 25.

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

Syntax 25—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 21.

Table 21—Character symbols within a quoted string

Symbol	Character	ASCII code (octal)
\g	Alert or bell.	007
\h	Backspace.	010

Table 21—Character symbols within a quoted string (Continued)

\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.

6.15 String value

A *string value* shall be defined as shown in Syntax 26.

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

Syntax 26—String value

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

6.16 Generic value

An *generic value* shall be defined as shown in Syntax 27.

```
generic_value ::=
      number
    | multiplier_prefix_symbol
    | identifier
    | quoted_string
    | bit_literal
    | based_literal
    | edge_value
```

Syntax 27—Generic value

A *generic value* shall be used as an *ALF value* for an *annotation* (see 7.3), for a *group* declaration (see 7.14) or for a *template* instantiation (see 7.16). Restrictions for applicable values in a particular context shall be defined by semantic rules.

6.17 Vector expression macro

A *vector expression macro* shall be defined as shown in Syntax 28.

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

Syntax 28—Vector expression macro

A *vector expression macro* shall be used as a substitution for a predefined vector expression (see 9.12). The *alias* declaration (see 7.7) shall be used to establish the substitution mechanism.

6.18 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.

6.19 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. Generic objects and related statements

7.1 Generic object

A *generic object* shall be defined as shown in Syntax 29.

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

Syntax 29—Generic object

The purpose of a *generic object* is to specify a re-usable statement in ALF. A generic object shall be either a declared *alias* (see 7.7), a declared *constant* (see 7.8), a declared *class* (see 7.12), a declared *keyword* (see 7.9), a declared *semantics* (see 7.10), a declared *group* (see 7.14) or a declared *template* (see 7.15).

A generic object shall have an *ALF name*. Plural generic objects of the same *ALF type* can be declared within the same context. They shall be distinguished by their *ALF name*.

7.2 All purpose item

An *all-purpose item* shall be defined as shown in Syntax 30.

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

Syntax 30—All purpose item

The purpose of an *all-purpose item* is to specify a category of statements that are supported in the syntax rules of a *library-specific object* (see 8.1), without semantic restrictions. The semantic restrictions for an *all-purpose item* shall be defined by a *keyword* declaration (see 7.9) or by a *semantics* declaration (see 7.10).

An all-purpose item shall be either a *generic object* (see 7.1), an *include* statement (see 7.17), an *associate* statement (see 7.18), an *annotation* (see 7.3), an *annotation container* (see 7.4), an *arithmetic model* (see 10.3), or an *arithmetic model container* (see 10.8).

7.3 Annotation

An *annotation* shall be divided into the subcategories *single value annotation* and *multi value annotation*, as shown in Syntax 31.

```
annotation ::=
      single_value_annotation
    | multi_value_annotation
single_value_annotation ::=
    annotation_identifier = annotation_value ;
multi_value_annotation ::=
    annotation_identifier { annotation_value { annotation_value } }
annotation_value ::=
      generic_value
    | control_expression
    | boolean_expression
    | arithmetic_expression
```

Syntax 31—Annotation

The purpose of an *annotation* is to describe a particular semantic aspect of a statement in ALF.

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.

The *annotation identifier* shall be either a declared *keyword* (see 7.9) or the ALF type of an object, i.e., a *generic object* (see 7.1) or a *library-specific object* (see 8.1). In the latter case, the annotation shall be called *reference annotation*. A *semantics* declaration (see 7.10) shall be used to legalize a reference annotation. The annotation value of a reference annotation shall be the ALF name of an object of the specified ALF type.

7.4 Annotation container

An *annotation container* shall be defined as shown in Syntax 32.

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

Syntax 32—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.

7.5 ATTRIBUTE statement

An *attribute* statement shall be defined as shown in Syntax 33.

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

Syntax 33—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 7.3). 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 }
}
```

7.6 PROPERTY statement

A *property* statement shall be defined as shown in Syntax 34.

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

Syntax 34—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 7.4). 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 }
}
```

7.7 ALIAS declaration

An *alias* shall be declared as shown in Syntax 35.

alias_declaration ::=
ALIAS *alias*_identifier **=** *original*_identifier **;**
| **ALIAS** vector_expression_macro **= (** vector_expression **);**

Syntax 35—ALIAS declaration

The alias declaration shall specify an alias *identifier* (see 6.13) or a *vector expression macro* (see 6.17).

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 );
```

7.8 CONSTANT declaration

A *constant* shall be declared as shown in Syntax 36.

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

Syntax 36—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;
```

7.9 KEYWORD declaration

A *keyword* shall be declared as shown in Syntax 37.

```
keyword_declaration ::=
    KEYWORD keyword_identifier = syntax_item_identifier ;
  | KEYWORD keyword_identifier = syntax_item_identifier { { CONTEXT_annotation } }
```

Syntax 37—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.

A *keyword item* can be used to qualify the contents of the keyword declaration. One or more annotations (see 7.11) can be used as a keyword item.

A legal *syntax item* identifier shall be defined as shown in Table 22.

Table 22—Syntax item identifier

Syntax item identifier	Semantic meaning
`annotation`	The keyword shall specify an *annotation* (see 7.3).
`single_value_annotation`	The keyword shall specify a *single value annotation* (see 7.3).
`multi_value_annotation`	The keyword shall specify a *multi-value annotation* (see 7.3).

Table 22—Syntax item identifier (Continued)

Syntax item identifier	Semantic meaning
`annotation_container`	The keyword shall specify an *annotation container* (see 7.4).
`arithmetic_model`	The keyword shall specify an *arithmetic model* (see 10.3).
`arithmetic_submodel`	The keyword shall specify an *arithmetic submodel* (see 10.7).
`arithmetic_model_container`	The keyword shall specify an *arithmetic model container* (see 10.8).
`geometric_model`	The keyword shall specify a *geometric model* (see 9.16).

A keyword declaration shall be equivalent to an extension of the ALF syntax. A keyword declaration shall not be overwritten or duplicated.

Example

Declaration of a keyword:

```
KEYWORD MySingleValueAnnotation = single_value_annotation ;
```

The equivalent syntax rule in BNF looks as follows:

MySingleValueAnnotation ::=
MySingleValueAnnotation = annotation_value **;**

End of example

7.10 SEMANTICS declaration

Semantics shall be declared as shown in Syntax 38.

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

Syntax 38—SEMANTICS declaration

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

The *semantics identifier* shall be a *keyword identifier* (see 6.13.7) or a *syntax item identifier* (see 7.9, Table 22) or a *full hierarchical identifier* (see 6.13.4), composed of one or more keyword identifiers and/or syntax item identifiers.

A *syntax item* identifier can be used as ALF value of a semantics declaration under the following restriction:

a) The syntax item identifier in a related keyword declaration is "*annotation*",

and

b) the syntax item identifier of the actual semantics declaration is "*single value annotation*" or "*multi-value annotation*".

A *semantic item* can be used to qualify the contents of the semantics declaration. One or more annotations (see 7.11) can be used as a semantic item.

A semantics declaration can be used to complement a keyword declaration or another semantics declaration. A semantics declaration shall not be contradictory to an existing keyword or semantics declaration.

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

This subsection defines annotations and rules related to a keyword or a semantics declaration.

7.11.1 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 related keyword declaration, as shown in Table 23.

Table 23—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`, `multiplier_prefix_value`, `identifier`, `string_value`, `quoted_string`, `boolean_value`, `edge_value`, `control_expression`, `boolean_expression`, `arithmetic_expression`.	`identifier`	See Syntax 31, definition of *annotation value.*
`annotation_container`	N/A	N/A	An *annotation container* (see Syntax 32) has no value.

Table 23—VALUETYPE annotation (Continued)

Syntax item identifier	Set of legal values for VALUETYPE	Default value for VALUETYPE	Comment
`arithmetic_model`	`number`, `signed_integer`, `unsigned_integer`, `identifier`, `bit_literal`, `based_literal`.	`number`	See Syntax 14, definition of *arithmetic value*.
`arithmetic_submodel`	N/A	N/A	An *arithmetic submodel* (see 10.7) shall always have the same *valuetype* as its parent arithmetic model.
`arithmetic_model_container`	N/A	N/A	An *arithmetic model container* (see 10.8) has no value.
`geometric_model`	N/A	N/A	A *geometric model* (see 9.16) 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 shall refer to the semantic interpretation of a value, not to the encountered lexical token. For example, a *non-escaped identifier* (see 6.13.1) can be the name of a *constant* (see 7.8) holding a numerical value. Therefore the *identifier* (see 6.13) would be semantically interpreted as a *number* (see 6.5).

The valuetype annotation can be partially self-described as shown in Semantics 1.

```
KEYWORD VALUETYPE = single_value_annotation {
  CONTEXT = SEMANTICS;
}
SEMANTICS VALUETYPE {
  VALUES {
    number signed_integer unsigned_integer
    multiplier_prefix_value
    identifier quoted_string string_value
    bit_literal based_literal boolean_value edge_value
    control_expression boolean_expression
    arithmetic_expression
  }
}
```

Semantics 1—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.

7.11.2 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 within the *semantics* declaration and the *valuetype* annotation within a related *keyword* declaration shall be compatible.

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

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

Semantics 2—Partial self-description of VALUES annotation

Example:

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

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

End of example

7.11.3 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 3.

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

Semantics 3—Partial self-description of DEFAULT annotation

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 ; }
```

7.11.4 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 pre-defined 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 ancestor of the statement.

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

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

Semantics 4—Partial self-description of CONTEXT annotation

A context annotation within a *keyword* declaration shall be equivalent to a syntax rule applicable to the syntax item specified by the *context* annotation value. Only a *keyword identifier* (see 6.13.7) or a *syntax item identifier* (see 7.9, Table 22) shall be a legal annotation value.

Example

Declaration of a keyword with context:

```
KEYWORD MyAnnotationContainer = annotation_container;
KEYWORD MyAnnotation = single_value_annotation {
  CONTEXT = MyAnnotationContainer;
}
```

The equivalent syntax rule in BNF looks as follows:

MyAnnotationContainer ::=
MyAnnotationContainer { [**MyAnnotation =** annotation_value **;**] **}**

End of example

A context annotation within a *semantics* declaration shall be used to specify a legal ancestor of a statement. Only a *keyword identifier* (see 6.13.7) or a *syntax item identifier* (see 7.9, Table 22) or a *full hierarchical identifier* (see 6.13.4) or a *partial hierarchical identifier* (see 6.13.5) involving one or more keyword identifiers and/or one or more syntax item identifiers shall be a legal annotation value.

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, illegal context
    }
}
```

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 = PIN ; }
SEMANTICS PrimitivePinQualifier { CONTEXT = PRIMITIVE.PIN; }
```

End of example

7.11.5 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 related keyword declaration is *annotation*, *single value annotation* or *multi value annotation*.

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

```
KEYWORD REFERENCETYPE = annotation {
  CONTEXT = SEMANTICS;
}
SEMANTICS REFERENCETYPE {
  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 5—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 possibly by a *full hierarchical identifier* (see 6.13.4) involving the ALF name of the parent of the object and the ALF name of the object itself.

Example:

The following example shows the definition of an annotation “myReference”, which refers to an object of the ALF type “CLASS” with the ALF name “myClass”.

```
CLASS myClass;
KEYWORD myReference = single_value_annotation;
SEMANTICS myReference { REFERENCETYPE = CLASS; }
myReference = myClass;
```

In this example, a full hierarchical identifier is used to refer to a CLASS with the ALF name “myOtherClass”, declared as a child of a CELL with ALF name “myCell”.

```
CELL myCell {
    CLASS myOtherClass;
}
myReference = myCell.myOtherClass;
```

End of example

7.11.6 SI_MODEL annotation

The *SI-model* annotation shall be a *single value annotation*. It shall be only applicable for a keyword declaring an *arithmetic model* (see 10.3). It shall specify a relation of a declared keyword with the *International System of Units* (see U.S. National Bureau of Standards, Spec. Pub. 330). In particular, it shall specify the *base unit* of an arithmetic model.

A self-description of the SI-model annotation is given in Semantics 6.

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

Semantics 6—SI model annotation

The set of legal annotation values is shown in Table 24.

Table 24—SI_MODEL annotation

Annotation value	Mathematical symbol	Base unit	Relationship with other quantity	Reference to arithmetic model declaration
TIME	*t*	Second		See 10.11.1
FREQUENCY	*f*	Hertz	1 / *t*	See 10.11.2
CURRENT	*I*	Ampere		See 10.15.2
VOLTAGE	*V*	Volt		See 10.15.1
RESISTANCE	*R*	Ohm	*V* / *I*	See 10.15.4
CAPACITANCE	*C*	Farad	*I* / (d*V* / d*t*)	See 10.15.3
INDUCTANCE	*L*	Henry	*V* / (d*I* / d*t*)	See 10.15.5
ENERGY	*E*	Joule		See 10.11.15
POWER	*P*	Watt	*I V*, d*E* / d*t*	See 10.11.15
DISTANCE	*d*	Meter		See 10.19.9
AREA	*A*	Square meter	d^2	See 10.19.2

7.11.7 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 {
    CONTEXT { CELL PIN }
}
SEMANTICS myAnnotation {
    VALUES { value1 value2 value3 value4 value5 }
}
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; }
}
```

7.12 CLASS declaration

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

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

Syntax 39—CLASS declaration

A class declaration shall be used to establish a semantic association between ALF statements, including, but not restricted to, other class declarations. ALF statements shall be associated with each other, if they contain a reference to the same class. Such a reference is made by a *class reference* annotation (see 7.13).

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 7.2), a *geometric model* (see 9.16) or a *geometric transformation* (see 9.18).

7.13 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.

7.13.1 General CLASS reference annotation

A general *class reference* annotation shall be defined as shown in Semantics 7.

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

Semantics 7—CLASS reference annotation

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"
```

NOTES

1 — A class declaration itself can not contain a general class reference annotation. This avoids circular reference.

2 — 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.

7.13.2 USAGE annotation

The *usage* annotation shall be defined as shown in Semantics 8.

```
KEYWORD USAGE = annotation { CONTEXT = CLASS; }
SEMANTICS USAGE {
  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 8—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 Table 25.

Table 25—USAGE annotation

Annotation value	Definition of specific class reference annotation
SWAP_CLASS	See 8.5.4
RESTRICT_CLASS	See 8.5.3
SIGNAL_CLASS	See 8.8.15
SUPPLY_CLASS	See 8.8.16
CONNECT_CLASS	See 8.8.19
SELECT_CLASS	See 8.11.3
NODE_CLASS	See 8.13.3
EXISTENCE_CLASS	See 8.15.6
CHARACTERIZATION_CLASS	See 8.15.9
ORIENTATION_CLASS	See 8.26.2
SYMMETRY_CLASS	See 8.26.3

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 parser can evaluate the contents of the class declaration for semantic correctness only when encountering a reference to the class.

7.14 GROUP declaration

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

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

Syntax 40—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 generic 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 {
        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 {
        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];} }
    }
```

7.15 TEMPLATE declaration

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

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

Syntax 41—TEMPLATE declaration

A template declaration shall be used to specify one or more ALF statements with variable contents. A template instantiation (see 7.16) shall specify the usage of such an ALF statement. Within the template declaration, the variable contents shall be specified by a *placeholder identifier* (see 6.13.2).

An ALF statement within a template declaration shall be partially exempt from the semantics rule check defined by *valuetype*, *values*, *context*, and *referencetype*, as follows:

a) A declared template shall be presumed a legal ancestor within an applicable *context*.
b) A placeholder identifier shall be presumed a value within an applicable set of *values*.
c) A placeholder identifier shall be presumed a value of applicable *valuetype*.
d) A placeholder identifier shall be presumed a legal reference within an applicable *referencetype*.

The semantic rule check that can not be performed during parsing of the *template declaration* shall be deferred until parsing of the *template instantiation*.

7.16 TEMPLATE instantiation

A *template* shall be instantiated in form of a *static template instantiation* or a *dynamic template instantiation*, as shown in Syntax 42.

```
template_instantiation ::=
      static_template_instantiation
    | dynamic_template_instantiation
static_template_instantiation ::=
      template_identifier [ = static ] ;
    | template_identifier [ = static ] { { generic_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 42—TEMPLATE instantiation

A template instantiation shall be semantically equivalent to the ALF statement or the ALF statements found within the template declaration, after replacing the placeholder identifiers with replacement values. A static template instantiation shall support replacement by order, using an generic value, or alternatively, replacement by reference, using an annotation (see 7.3). A dynamic template instantiation shall support replacement by reference only, using an annotation and/or an arithmetic model (see 7.3 and 10.3) and/or an arithmetic assignment.

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 generic 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 generic value. The first amongst the remaining placeholder identifiers shall then be considered the next placeholder to be replaced by the next generic 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 d).

```
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 e).

```
// 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 = everything ; }
// Both everything (without brackets) and <nothing> (with brackets)
// are legal values for annotation5.
```

7.17 INCLUDE statement

An *include* statement shall be defined as shown in Syntax 43.

include ::=
INCLUDE quoted_string **;**

Syntax 43—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.

7.18 ASSOCIATE statement and FORMAT annotation

An *associate* statement shall be defined as shown in Syntax 44.

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

Syntax 44—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 7.17), 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 7.9) in conjunction with a *context* annotation (see 7.11.4), a *valuetype* annotation (see 7.11.1), a *values* annotation (see 7.11.2), and a *default* annotation (see 7.11.3), the *format* annotation shall be defined as shown in Semantics 9.

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

Semantics 9—FORMAT annotation

The meaning of the annotation values is specified in Table 26.

Table 26—FORMAT annotation values

Annotation value	Description
`vhdl`	The associated file is in a format specified by the IEEE Std 1076-2002.
`verilog`	The associated file is in a format specified by the IEEE Std 1364-2001.
`c`	The associated file is in a format specified by the ISO/IEC 9899:1990.
`\c++`	The associated file is in a format specified by the ANSI/ISO/IEC 14882.
`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.

7.19 REVISION statement

A *revision statement* shall be defined as shown in Syntax 45

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

Syntax 45—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.

A set of recognized string values within the revision statement shall be defined as shown in Table 27

Table 27—Recognized string values within the REVISION statement

String value	Revision or version
`"1.1"`	Advanced Library Format, Version 1.1 [B2]
`"2.0"`	Advanced Library Format, Version 2.0 [B3]

Table 27—Recognized string values within the REVISION statement (Continued)

String value	Revision or version
"P1603.2003-07-18"	Advanced Library Format specified by this draft IEEE P1603/D9 ** please delete this row after ballot approval **
"IEEE 1603-2003"	Advanced Library Format specified by this standard

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.

NOTE — This document suggests that this standard is largely backward compatible with the previous versions of the Advanced Library Format mentioned in Table 27.

8. Library-specific objects and related statements

8.1 Library-specific object

A *library-specific object* shall be defined as shown in Syntax 46.

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

Syntax 46—Library-specific object

A library-specific object shall be defined as a *library* (see 8.2), a *sublibrary* (see 8.2), a *cell* (see 8.4), a *primitive* (see 8.9), a *wire* (see 8.10), a *pin* (see 8.6), a *pingroup* (see 8.7), a *vector* (see 8.14), a *node* (see 8.12), a *layer* (see 8.16), a *via* (see 8.18), a *rule* (see 8.20), an *antenna* (see 8.21), a *site* (see 8.25), an *array* (see 8.27), a *blockage* (see 8.22), a *port* (see 8.23), a *pattern* (see 8.29) or a *region* (see 8.31).

The purpose of a library-specific object is to specify a model for a technology item, distinguished by an ALF name.

8.2 LIBRARY and SUBLIBRARY declaration

A *library* and a *sublibrary* shall be declared as shown in Syntax 47.

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.

Any two objects of the same ALF type and the same ALF name can not appear in one library or in one sublibrary. However, they can appear in two libraries, or in two sublibraries with the same library as parents. For example, two *cells* (see 8.4) with the same name can appear in two different libraries. It shall be the responsibility of the application tool to properly handle such cases, as the selection of a library or a sublibrary is controlled by the user of the application tool.

```
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

8.3 Annotations related to a LIBRARY or a SUBLIBRARY declaration

8.3.1 LIBRARY reference annotation

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

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

Semantics 10—LIBRARY reference annotation

The purpose of a library reference annotation is to establish an association between a library or a sublibrary and an *arithmetic model* (see 10.3).

A *full hierarchical identifier* (see 6.13.4) can be used to specify a reference to a sublibrary as a child of a library.

8.3.2 INFORMATION annotation container

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

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.

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

Semantics 11—INFORMATION statement

The semantics of the *information* contents are specified in Table 28.

Table 28—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";
    }
}
```

8.4 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.

8.5 Annotations related to a CELL declaration

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

8.5.1 CELL reference annotation

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

```
KEYWORD CELL = annotation { CONTEXT = arithmetic_model; }
SEMANTICS CELL { 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 10.3).

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

8.5.2 CELLTYPE annotation

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

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

Semantics 13—CELLTYPE annotation

The meaning of the celltype annotation values is specified in Table 29.

Table 29—CELLTYPE annotation values

Annotation value	Description
`buffer`	`CELL` is a *buffer*, i.e., an element for transmission of a digital signal without performing a logic operation, except for possible logic inversion.
`combinational`	`CELL` is a combinatorial logic element, i.e., an element performing a logic operation on two or more digital input signals.
`multiplexor`	`CELL` is a *multiplexor*, i.e., an element for selective transmission of digital signals.
`flipflop`	`CELL` is a *flip-flop*, i.e., a one-bit storage element with edge-sensitive clock
`latch`	`CELL` is a *latch*, i.e., a one-bit storage element without edge-sensitive clock
`memory`	`CELL` is a *memory*, i.e., a multi-bit storage element with selectable addresses.
`block`	`CELL` is a hierarchical *block*, i.e., a complex element which has an associated netlist for implementation purpose. All instances of the netlist are library elements, i.e., there is a `CELL` model for each of them in the library.
`core`	`CELL` is a *core*, i.e., a complex element which has no associated netlist for implementation purpose. However, a netlist representation can exist for modeling purpose.
`special`	`CELL` is a special element, which does not fall into any other category of cells. Examples: bus holder, protection diode, filler cell.

Example

```
CELL myNandGate {
    CELLTYPE = combinational;
    // put detailed description here
}
CELL myFlipflop {
    CELLTYPE = flipflop;
    // put detailed description here
}
```

8.5.3 RESTRICT_CLASS annotation

A *restrict-class* annotation shall be defined as shown in Semantics 14.

```
KEYWORD RESTRICT_CLASS = annotation {
  CONTEXT { CELL CLASS }
}
SEMANTICS RESTRICT_CLASS {
  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 14—RESTRICT_CLASS annotation

The *annotation value* shall be the name of a declared *class* (see 7.12).

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 established by Semantics 14 is specified in Table 30.

Table 30—Predefined RESTRICT_CLASS annotation values

Annotation value	Description
`synthesis`	Cell is suitable for creation or modification of a structural 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* (see 8.2), using a *class* declaration (see 7.12) and a *semantics* declaration (see 7.10) in a similar way as shown in Semantics 14.

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:

```
CLASS A { USAGE = RESTRICT_CLASS; }
CLASS B { USAGE = RESTRICT_CLASS; }
CLASS C { USAGE = RESTRICT_CLASS; }
CLASS D { USAGE = RESTRICT_CLASS; }
CLASS E { USAGE = RESTRICT_CLASS; }
CLASS F { USAGE = RESTRICT_CLASS; }
CLASS G { USAGE = RESTRICT_CLASS; }
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".

8.5.4 SWAP_CLASS annotation

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

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

Semantics 15—SWAP_CLASS annotation

The *annotation value* shall be the name of a declared *class* (see 7.12). *Single-value* or *multi-value annotation* can be used.

Cells referring to the same class can be swapped for certain applications. Cell-swapping shall be only allowed under the following conditions:

a) The *restrict-class* annotation (see 8.5.3) authorizes usage of the cell.
b) The cells are compatible from an application standpoint.

Example:

```
CLASS U { USAGE = SWAP_CLASS; }
CLASS V { USAGE = SWAP_CLASS; }
CELL X1 { SWAP_CLASS { U V } }
CELL X2 { SWAP_CLASS { U } }
CELL Y1 { SWAP_CLASS { U V } }
CELL Y2 { SWAP_CLASS { V } }
```

Cell X1 can be swapped with cell X2, provided the application authorizes the usage of both X1 and X2.
Cell X1 can be swapped with cell Y1, provided the application authorizes the usage of both X1 and Y1.
Cell Y1 can be swapped with cell Y2, provided the application authorizes the usage of both Y1 and Y2.
Cell X2 can not be swapped with cell Y2, even if the application authorizes the usage of both X2 and Y2.

End of example

8.5.5 SCAN_TYPE annotation

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

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

Semantics 16—SCAN_TYPE annotation

The meaning of the *scan type* annotation values is specified in Table 31.

Table 31—SCAN_TYPE annotation values

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.

8.5.6 SCAN_USAGE annotation

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

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

Semantics 17—SCAN_USAGE annotation

The meaning of the *scan usage* annotation values is specified in Table 32.

Table 32—SCAN_USAGE annotation values

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

The scan usage annotation is applicable for a cell which is designed to be the primary input, output or intermediate stage of a scan chain.

8.5.7 BUFFERTYPE annotation

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

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

Semantics 18—BUFFERTYPE annotation

The meaning of the *buffertype* annotation values is specified in Table 33.

Table 33—BUFFERTYPE annotation values

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.

8.5.8 DRIVERTYPE annotation

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

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

Semantics 19—DRIVERTYPE annotation

The meaning of the *drivertype* annotation values is specified in Table 34.

Table 34—DRIVERTYPE annotation values

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.

The *drivertype* annotation applies only for a cell with *buffertype* value *input* or *output* or *inout*.

8.5.9 PARALLEL_DRIVE annotation

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

The annotation value shall specify the number of cells connected in parallel.

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

Semantics 20—PARALLEL_DRIVE annotation

8.5.10 PLACEMENT_TYPE annotation

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

```
KEYWORD PLACEMENT_TYPE = single_value_annotation {
  CONTEXT = CELL;
}
SEMANTICS PLACEMENT_TYPE {
  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.

The meaning of the *placement type* annotation values is specified in Table 35.

Table 35—PLACEMENT_TYPE annotation values

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.

8.5.11 SITE reference annotation for a CELL

A *site* reference annotation (see 8.26.1) in the context of a cell shall be defined as shown in Semantics 22.

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

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

Semantics 22—SITE reference annotation

8.5.12 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 36 can be used within cell with *celltype* annotation value *memory.*

Table 36—Attribute values for a CELL with CELLTYPE memory

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

The attributes shown in Table 37 can be used within a cell with *celltype* annotation value *block*.

Table 37—Attribute values for 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.

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

Attribute item	Description
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 38 can be used within a cell with *celltype* annotation value *core*.

Table 38—Attribute values for 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 39 can be used within a cell with *celltype* annotation value *special*.

Table 39—Attribute values for 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 Table 74 in 9.10).
clamp	CELL connects a net to a constant *logic value* (see 9.10).
diode	CELL is a *diode*.
capacitor	CELL is a *capacitor.*
resistor	CELL is a *resistor.*
inductor	CELL is an *inductor.*
fillcell	CELL is used to fill unused space in layout.

A cell with attribute value *busholder* shall have one or more *pin* declarations (see 8.6). The *direction* annotation value shall be *both* (see 8.8.5). A cell with attribute value *clamp* shall have one or more *pin* declarations. The *direction* annotation value shall be *output*. The logical value and drive strength shall be defined within a *function* statement (see 9.1). A cell with attribute value *diode*, *capacitor*, *resistor*, or *inductor* shall have two *pin* declarations and no *function* statement. A cell with attribute value *fillcell* shall have no pin declaration and no function statement.

8.6 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 8.8.3) shall further specify which pin is visible in a netlist.

A scalar pin can be associated with a general electrical signal. However, a vector pin or a matrix pin can only be associated with 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 6.6). A reference to a scalar within the vector-pin shall be established by the pin identifier followed by a *single index* (see 6.6). 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]`.

8.7 PINGROUP declaration

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

```
pingroup ::=
    simple_pingroup | vector_pingroup
simple_pingroup ::=
     PINGROUP pingroup_identifier
      { MEMBERS_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 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.

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.

8.8 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.

8.8.1 PIN reference annotation

A *pin reference* annotation shall be defined as shown in Semantics 23.

```
KEYWORD PIN = annotation {
  CONTEXT { arithmetic_model FROM TO }
}
SEMANTICS PIN {
  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 8.23) or a *node* (see 8.12) and an *arithmetic model* (see 10.3) or a *from-to* statement (see 10.12). 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.

8.8.2 MEMBERS annotation

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

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

Semantics 24—MEMBERS annotation

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

8.8.3 VIEW annotation

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

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

Semantics 25—VIEW annotation

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

It can take the values shown in Table 40.

Table 40—VIEW annotation values

Annotation value	Description
`functional`	`pin` appears in functional netlist.
`physical`	`pin` appears in physical netlist.
`both`	`pin` appears in both functional and physical netlist.
`none`	`pin` does not appear in netlist.

8.8.4 PINTYPE annotation

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

```
KEYWORD PINTYPE = single_value_annotation {
  CONTEXT = PIN;
}
SEMANTICS PINTYPE {
  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 41.

Table 41—PINTYPE annotation values

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

8.8.5 DIRECTION annotation

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

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.

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

Semantics 27—DIRECTION annotation

The direction annotation can take the values shown in Table 42.

Table 42—DIRECTION annotation values

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 (see 8.8.4), i.e., all combinations of annotation values are possible.

Examples

- The power and ground pins of a regular cell have the *direction* value *input*.
- A level converter cell has a power supply pin with *direction* value *input* and another power supply pin with *direction* value *output*.
- A level converter can have a common ground pin with *direction* value *both* or separate ground pins related to its power supply pins, i.e., one ground pin with *direction* value *input* and another ground pin with *direction* value *output*.
- The power and ground pins of a feed through cell have the *direction* value *none*.

8.8.6 SIGNALTYPE annotation

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

The purpose of the *signaltype* annotation is to classify the functionality of a pin. The set of defined values apply for pins with *pintype* value *digital*. Conceptually, a pin with *pintype* value *analog* can also have a *signaltype* annotation. However, no values are currently defined.

```
KEYWORD SIGNALTYPE = single_value_annotation {
  CONTEXT = PIN;
}
SEMANTICS SIGNALTYPE {
  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

The fundamental *signaltype* values are defined in Table 43

Table 43—Fundamental SIGNALTYPE annotation values

Annotation value	Description
`data`	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`, possibly 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
`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`, possibly 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 9 shows how to construct composite signaltype values.

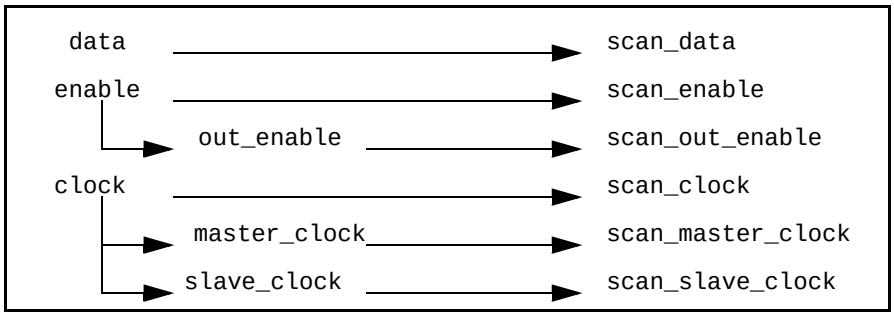

Figure 9—Scheme for constructing composite signaltype values

The composite *signaltype* values are defined in Table 44

Table 44—Composite SIGNALTYPE annotation values

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.
`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 43 and Table 44, the elements *flipflop*, *latch*, *multiplexor*, or *memory* can be standalone cells or embedded in larger cells. In the former case, the *celltype* value (see 8.5.2) is *flipflop*, *latch*, *multiplexor*, or *memory*, respectively. In the latter case, the *celltype* value can be *block* or *core*.

8.8.7 ACTION annotation

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

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

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

Semantics 29—ACTION annotation

The *action* annotation can take the values shown in Table 45.

Table 45—ACTION annotation values

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 in conjunction with specific *signaltype* values (see 8.8.6), as shown in Table 46.

Table 46—ACTION in conjunction with SIGNALTYPE

fundamental SIGNALTYPE value	composite SIGNALTYPE value	ACTION applicable
data	scan_data	No
address		No
control		Yes
select		No
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

8.8.8 POLARITY annotation

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

```
KEYWORD POLARITY = single_value_annotation {
  CONTEXT = PIN;
}
SEMANTICS POLARITY {
  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 47.

Table 47—POLARITY annotation values

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 in conjunction with specific *signaltype* values (see 8.8.6), as shown in Table 48.

Table 48—POLARITY in conjunction with SIGNALTYPE

fundamental SIGNALTYPE value	composite SIGNALTYPE value	Applicable POLARITY value
data	scan_data	N/A
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

Table 48—POLARITY in conjunction with SIGNALTYPE (Continued)

fundamental SIGNALTYPE value	composite SIGNALTYPE value	Applicable POLARITY value
clock	scan_clock master_clock slave_clock scan_master_clock scan_slave_clock	high low rising_edge falling_edge double_edge

8.8.9 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 {
  VALUES { high low rising_edge falling_edge double_edge }
}
```

Semantics 31—Control polarity annotation container

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, wherein the name of the mode of operation is given by the annotation identifier.

The *control polarity* annotation container can be used only in conjunction with specific *signaltype* values (see 8.8.6), as shown in Table 49.

Table 49—CONTROL_POLARITY in conjunction with SIGNALTYPE

fundamental SIGNALTYPE value	composite SIGNALTYPE value	Applicable annotation value within CONTROL_POLARITY
control		high low
clock	scan_clock master_clock slave_clock scan_master_clock scan_slave_clock	high low rising_edge falling_edge double_edge
other		N/A

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; }
}
```

The control-polarity specification in this example is equivalent to the following truth table.

ModeSel1	ModeSel2	Mode of operation
0	0	hold
0	1	scan
1	don’t care	normal

8.8.10 DATATYPE annotation

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

```
KEYWORD DATATYPE = single_value_annotation {
  CONTEXT { PIN PINGROUP }
}
SEMANTICS DATATYPE {
  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 annotation values

Annotation value	Description
`signed`	Result of arithmetic operation is signed 2’s complement.
`unsigned`	Result of arithmetic operation is unsigned.

The *datatype* annotation is only relevant for a *bus*, i.e., a *vector pin* (see Syntax 49 in 8.6).

8.8.11 INITIAL_VALUE annotation

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

```
KEYWORD INITIAL_VALUE = single_value_annotation {
  CONTEXT { PIN PINGROUP }
}
SEMANTICS INITIAL_VALUE {
  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 9.10.1, Table 74).

8.8.12 SCAN_POSITION annotation

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

```
KEYWORD SCAN_POSITION = single_value_annotation {
  CONTEXT = PIN;
}
SEMANTICS SCAN_POSITION {
  VALUETYPE = unsigned_integer;
  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.

8.8.13 STUCK annotation

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

```
KEYWORD STUCK = single_value_annotation {
  CONTEXT = PIN;
}
SEMANTICS STUCK {
  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 annotation values

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.

8.8.14 SUPPLYTYPE annotation

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

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

Semantics 36—SUPPLYTYPE annotation

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

Table 52—SUPPLYTYPE annotation values

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 41).

8.8.15 SIGNAL_CLASS annotation

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

```
KEYWORD SIGNAL_CLASS = annotation {
  CONTEXT { PIN PINGROUP }
}
SEMANTICS SIGNAL_CLASS {
  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 8.8.6).

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 8.23).

```
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; }
}
```

8.8.16 SUPPLY_CLASS annotation

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

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

Semantics 38—SUPPLY_CLASS annotation

The annotation value shall be the name of a declared *class* (see 7.12).

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

The supply-class annotation shall apply for a pin with any *signaltype* value (see 8.8.6) or any *supplytype* value (see 8.8.14).

The supply-class annotation shall also apply for a class with *usage* value *connect-class* (see 8.8.19). 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 10.11.15).

Example 1:

A cell supports two power supplies. Each pin is related to at least one power supply.

```
CLASS supply1 { USAGE = SUPPLY_CLASS; }
CLASS supply2 { USAGE = SUPPLY_CLASS; }
CELL myLevelShifter {
    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 provides 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; }
}
```

8.8.17 DRIVETYPE annotation

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

```
KEYWORD DRIVETYPE = single_value_annotation {
  CONTEXT { PIN CLASS }
}
SEMANTICS DRIVETYPE {
  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 annotation values

Annotation value	Description
`cmos`	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.
`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 nmos threshold voltage.
`pmos_pass`	Pmos passgate signal. Signal is not amplified by passgate stage. Logic high voltage level is preserved, logic low 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.

8.8.18 SCOPE annotation

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

```
KEYWORD SCOPE = single_value_annotation {
  CONTEXT { PIN PINGROUP }
}
SEMANTICS SCOPE {
  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* (see 9.12) within a *vector* declaration (see 8.14) or within a *behavior* statement (see 9.4).

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

Table 54—SCOPE annotation values

Annotation value	Description
`behavior`	The pin is used for modeling functional behavior. Pin can be involved in a control expression within a `BEHAVIOR` statement.
`measure`	Measurements related to the pin can be described. Pin can be involved in a control expression within a `VECTOR` declaration.
`both`	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.

8.8.19 CONNECT_CLASS annotation

A *connect-class* annotation shall be defined as shown in Semantics 41.

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

Semantics 41—CONNECT_CLASS annotation

The *annotation value* shall be the name of a declared *class* (see 7.12).

The purpose of the *connect-class* annotation is to specify a relationship between a pin and an environmental rule for *connectivity* (see 10.18.1). The *connect-class* annotation can be used in conjunction with *supply-class* (see 8.8.16) or in conjunction with *connect-rule* (see 10.20.1).

8.8.20 SIDE annotation

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

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

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

Semantics 42—SIDE annotation

The side annotation can take the values shown in Table 55.

Table 55—SIDE annotation values

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

8.8.21 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 }
}
SEMANTICS ROW {
  VALUETYPE = unsigned_integer;
}
KEYWORD COLUMN = annotation {
  CONTEXT { PIN PINGROUP }
}
SEMANTICS COLUMN {
  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 10.

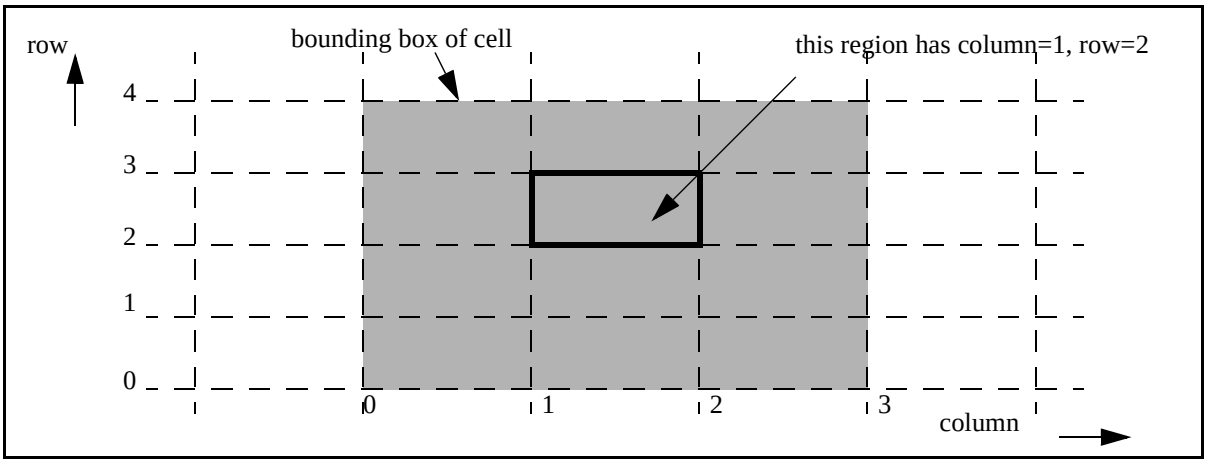

Figure 10—ROW and COLUMN relative to a bounding box of a CELL

The row annotation is applicable for a pin with *side* value *left* or *right*. The column annotation 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.

8.8.22 ROUTING_TYPE annotation

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

```
KEYWORD ROUTING_TYPE = single_value_annotation {
  CONTEXT { PIN PORT }
}
SEMANTICS ROUTING_TYPE {
  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 annotation values

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

Table 56—ROUTING-TYPE annotation values (Continued)

Annotation value	Description
`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

8.8.23 PULL annotation

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

```
KEYWORD PULL = single_value_annotation {
  CONTEXT = PIN;
}
SEMANTICS PULL {
  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 annotation values

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.

8.8.24 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—Attribute values for a PIN

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—Attribute values for a PIN of a CELL with CELLTYPE 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.
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—Attribute values for a PIN within a pair 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.

Table 60—Attribute values for a PIN within a pair of signals (Continued)

Attribute item	Description
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—ATTRIBUTE values for a PIN or a PINGROUP related to memory BIST

Attribute item	Description
ROW_INDEX	Vector pin or pingroup with a contiguous range of values, indicating a physical row of a memory.
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 9.2).

8.9 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 8.4). A primitive can be instantiated within a behavior statement (see 9.4).

8.10 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.

8.11 Annotations related to a WIRE declaration

8.11.1 WIRE reference annotation

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

```
KEYWORD WIRE = annotation {
  CONTEXT = arithmetic_model;
}
SEMANTICS WIRE {
  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 10.3).

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.

8.11.2 WIRETYPE annotation

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

```
KEYWORD WIRETYPE = single_value_annotation {
  CONTEXT = WIRE;
}
SEMANTICS WIRETYPE {
  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 annotation values

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.
`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 9.15) shall be used to describe such a connection.

An R, L, C component within the context of the wire declaration shall be described as an *arithmetic model* (see 10.3). A related electrical measurement, e.g., voltage, current, noise, shall also be described as arithmetic model.

8.11.3 SELECT_CLASS annotation

A *select-class* annotation shall be defined as shown in Semantics 48.

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

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

Semantics 48—SELECT_CLASS annotation

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
}
```

8.12 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 8.10) or in the context of a *cell* declaration (see 8.4).

8.13 Annotations related to a NODE declaration

8.13.1 NODE reference annotation

A *node reference* annotation shall be defined as shown in Semantics 49.

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

Semantics 49—NODE reference annotation

The purpose of a node reference annotation is to establish an association between a pin, a pingroup, a *port* (see 8.23) or a *node* (see 8.12) and an *arithmetic model* (see 10.3). 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.

8.13.2 NODETYPE annotation

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

```
KEYWORD NODETYPE = single_value_annotation {
  CONTEXT = NODE;
}
SEMANTICS NODETYPE {
  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.

Table 63—NODETYPE annotation values (Continued)

Annotation value	Description
`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 Figure 11.

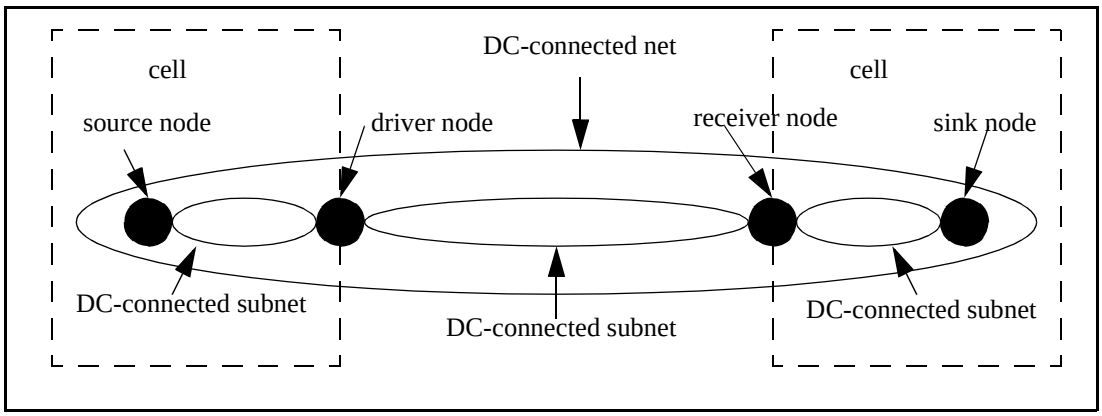

Figure 11—NODETYPE in context of a DC-connected net

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 receiver 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 inferred 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.

8.13.3 NODE_CLASS annotation

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

```
KEYWORD NODE_CLASS = annotation {
  CONTEXT = NODE;
}
SEMANTICS NODE_CLASS {
  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 inferred by the connectivity within a DC-connected net.

Example:

```
WIRE CrosstalkAcrossPowerDomains {
    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.

8.14 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 9.4) shall specify a stimulus related to characterization or test.

8.15 Annotations related to a VECTOR declaration

8.15.1 VECTOR reference annotation

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

```
KEYWORD VECTOR = single_value_annotation {
  CONTEXT = arithmetic_model;
}
SEMANTICS VECTOR {
  VALUETYPE = control_expression;
  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 10.3).

8.15.2 PURPOSE annotation

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

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

Semantics 53—PURPOSE annotation

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

The *values* shall have the semantic meaning shown in Table 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 10.11.1 to 10.11.11)

Table 64—PURPOSE annotation values (Continued)

Annotation value	Description
power	The vector contains an arithmetic model related to power calculation (see 10.11.15)
noise	The vector contains an arithmetic model related to noise calculation (see 10.11.14)
reliability	The vector contains an arithmetic model related to reliability calculation (see 10.11.1 and 10.11.2)

8.15.3 OPERATION annotation

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

```
KEYWORD OPERATION = single_value_annotation {
  CONTEXT = VECTOR;
}
SEMANTICS OPERATION {
  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 association 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.

Table 65—OPERATION annotation values (Continued)

Annotation value	Description
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.

8.15.4 LABEL annotation

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

```
KEYWORD LABEL = single_value_annotation {
  CONTEXT = VECTOR;
}
SEMANTICS LABEL {
  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 10.11.3) and an annotated delay within an SDF file (see IEEE Std 1497-2001) can be established, since the SDF standard also supports a LABEL statement.

8.15.5 EXISTENCE_CONDITION annotation

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

```
KEYWORD EXISTENCE_CONDITION = single_value_annotation {
  CONTEXT { VECTOR CLASS }
}
SEMANTICS EXISTENCE_CONDITION {
  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.

8.15.6 EXISTENCE_CLASS annotation

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

```
KEYWORD EXISTENCE_CLASS = annotation {
  CONTEXT { VECTOR CLASS }
}
SEMANTICS EXISTENCE_CLASS {
  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.

8.15.7 CHARACTERIZATION_CONDITION annotation

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

```
KEYWORD
CHARACTERIZATION_CONDITION = single_value_annotation {
  CONTEXT { VECTOR CLASS }
}
SEMANTICS CHARACTERIZATION_CONDITION {
  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 possibly 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 possibly with the existence-condition.

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

8.15.8 CHARACTERIZATION_VECTOR annotation

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

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.

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

Semantics 59—CHARACTERIZATION_VECTOR annotation

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

8.15.9 CHARACTERIZATION_CLASS annotation

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

```
KEYWORD CHARACTERIZATION_CLASS = annotation {
  CONTEXT { VECTOR CLASS }
}
SEMANTICS CHARACTERIZATION_CLASS {
  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.

8.15.10 MONITOR annotation

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

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

Semantics 61—MONITOR annotation

The purpose of the monitor annotation is to specify a set of *pin variables* (see 9.3) 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 9.13.2).

8.16 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.

8.17 Annotations related to a LAYER declaration

8.17.1 LAYER reference annotation

A *layer reference* annotation shall be defined as shown in Semantics 62.

```
KEYWORD LAYER = annotation {
  CONTEXT { arithmetic_model PATTERN ARRAY }
}
SEMANTICS LAYER {
  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 8.29), an *array* (see 8.27) or an *arithmetic model* (see 10.3).

8.17.2 LAYERTYPE annotation

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

```
KEYWORD LAYERTYPE = single_value_annotation {
  CONTEXT = LAYER;
}
SEMANTICS LAYERTYPE
  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.

8.17.3 PITCH annotation

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

```
KEYWORD PITCH = single_value_annotation {
  CONTEXT = LAYER;
}
SEMANTICS PITCH {
  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.

8.17.4 PREFERENCE annotation

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

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

Semantics 65—PREFERENCE annotation

The purpose of the preference annotation is to specify the preferred routing direction for a routing segment on a layer with *layertype* value *routing* (see 8.17.2).

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

Table 67—PREFERENCE annotation values

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

8.18 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.

8.19 Annotations related to a VIA declaration

8.19.1 VIA reference annotation

A *via reference* annotation shall be defined as shown in Semantics 66.

```
KEYWORD VIA = annotation {
  CONTEXT = arithmetic_model;
}
SEMANTICS VIA {
  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 10.3).

8.19.2 VIATYPE annotation

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

```
KEYWORD VIATYPE = single_value_annotation {
  CONTEXT = VIA;
}
SEMANTICS VIATYPE {
  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.

8.20 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 8.29), a *region* (see 8.31), or a *via instantiation* (see 9.20). The electrical or physical constraint shall be described as arithmetic model (see 10.3).

8.21 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 8.31), wherein the regions are created by physical objects. The physical objects shall be associated with a *layer* (see 8.16). Within the context of an antenna declaration, arithmetic models for *size* (see 10.19.1), *area* (see 10.19.2), *perimeter* (see 10.19.3) associated with a layer or with a region can be described. The arithmetic models can be combined, based on electrical *connectivity* (see 10.18.1) 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 Figure 12.

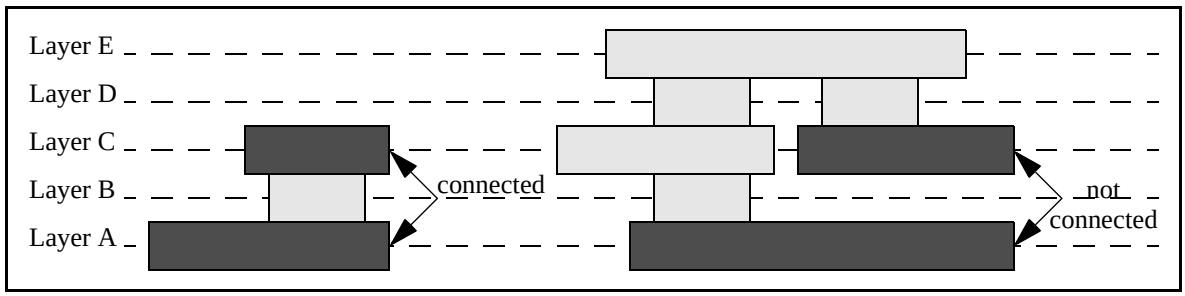

Figure 12—Connection between layers during manufacturing

The dark objects on layer A and layer C on the left side of Figure 12 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 12 are not considered connected, because the connection involves layer D and E which do not yet exist when layer C is manufactured.

8.22 BLOCKAGE declaration

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

```
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

A blockage declaration shall be used in context of a *cell* (see 8.4) 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 8.20). A rule within the context of a blockage shall only be applicable for a physical 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.

8.23 PORT declaration

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

```
port ::=
    PORT port_identifier ;
  | PORT port_identifier { { port_item } }
  | 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 8.6) to describe a part of the physical artwork of a cell (see 8.4) 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 8.20). A rule within the context of a port shall only be applicable for a physical object within the port in relation to its environment. A physical object within the port can also be subjected to a more general rule, i.e. a rule that is declared outside the context of the port.

8.24 Annotations related to a PORT declaration

8.24.1 Reference to a PORT using PIN reference annotation

The pin reference annotation (see 8.8.1) can be used to refer to the hierarchical name of a port.

8.24.2 PORTTYPE annotation

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

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

Semantics 68—PORTTYPE annotation

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

Table 69—PORTTYPE annotation values

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

8.25 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 8.4).

8.26 Annotations related to a SITE declaration

8.26.1 SITE reference annotation

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

The purpose of a site reference annotation is to establish an association between a site and a *cell* (see 8.4) or an *array* (see 8.27). A cell or an array can inherit a site reference annotation from a *class* (see 7.12).

```
KEYWORD SITE = annotation {
  CONTEXT { CELL ARRAY CLASS }
}
SEMANTICS SITE {
  REFERENCETYPE = SITE;
}
```

Semantics 69—SITE reference annotation

8.26.2 ORIENTATION_CLASS annotation

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

```
KEYWORD ORIENTATION_CLASS = annotation {
  CONTEXT { SITE CELL }
}
SEMANTICS ORIENTATION_CLASS {
  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 8.4) on a site. The annotation value shall be the name of a declared *class* (see 7.12). The declared class can contain a *geometric transformation* statement (see 9.18). 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.

8.26.3 SYMMETRY_CLASS annotation

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

```
KEYWORD SYMMETRY_CLASS = multi_value_annotation {
  CONTEXT = SITE;
}
SEMANTICS SYMMETRY_CLASS {
  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 8.4) on a site.

A legal orientation is specified by the *orientation class* annotation (see 8.26.2). 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.

8.27 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 declaration

An array declaration shall be used for the purpose to describe a grid for creating physical objects within design. A *geometric transformation* (see 9.18) 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.

8.28 Annotations related to an ARRAY declaration

8.28.1 ARRAYTYPE annotation

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

```
KEYWORD ARRAYTYPE = single_value_annotation {
  CONTEXT = ARRAY;
}
SEMANTICS ARRAYTYPE {
  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.

8.28.2 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 8.28.1). It shall specify a *layer* (see 8.16) with *layertype* value *routing* (see 8.17.2).

8.28.3 SITE reference annotation for ARRAY

A *site* reference annotation in the context of an *array* shall be defined as shown in Semantics 74.

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

Semantics 74—SITE reference annotation for ARRAY

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 8.28.1).

8.29 PATTERN declaration

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

```
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

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

8.30 Annotations related to a PATTERN declaration

8.30.1 PATTERN reference annotation

A *pattern* reference annotation shall be defined as shown in Semantics 75.

```
KEYWORD PATTERN = annotation {
  CONTEXT = arithmetic_model ;
}
SEMANTICS PATTERN {
  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 10.3).

8.30.2 SHAPE annotation

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

```
KEYWORD SHAPE = single_value_annotation {
  CONTEXT = PATTERN;
}
SEMANTICS SHAPE {
  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 8.17.2).

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

Table 71—SHAPE annotation values

Annotation value	Description
`line`	A routing segment in preferred routing direction. Each end is connected with a via or with another routing segment.
`jog`	A routing segment in non-preferred routing direction. Each end is connected with a routing segment in preferred 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 13.

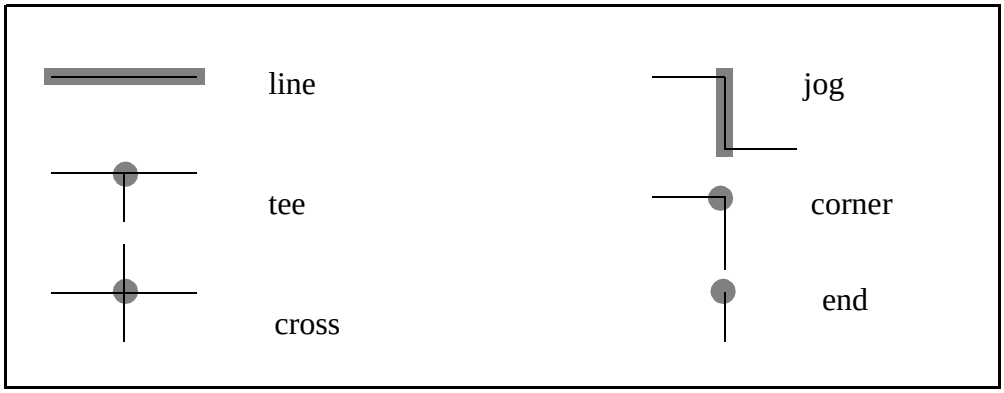

Figure 13—SHAPE annotation illustration

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.

8.30.3 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 8.30.2).

```
KEYWORD VERTEX = single_value_annotation {
  CONTEXT = PATTERN;
}
SEMANTICS VERTEX {
  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 14.

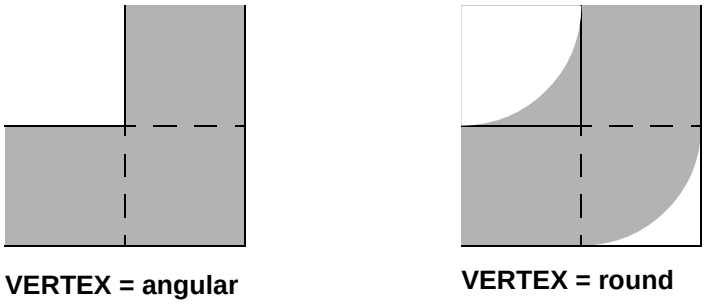

Figure 14—VERTEX annotation illustration

8.30.4 ROUTE annotation

A *route* annotation shall be defined as shown in Semantics 78.

```
KEYWORD ROUTE = single_value_annotation {
  CONTEXT = PATTERN;
}
SEMANTICS ROUTE {
  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 8.30.2).

The purpose of a route annotation is to specify the actual routing direction for the pattern. This is illustrated in Figure 15.

.

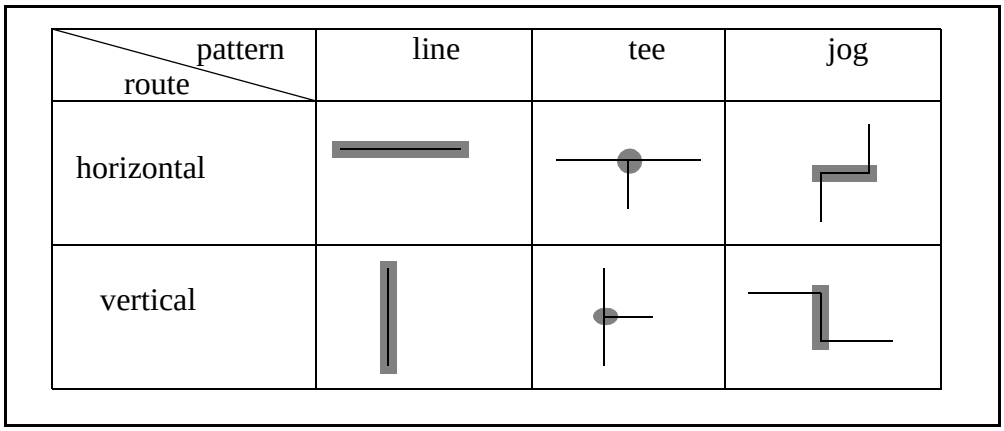

Figure 15—ROUTE annotation illustration

If the route annotation does not appear and a layer reference annotation (see 8.30.5) appears, the preferred routing direction specified by the *preference* annotation (see 8.17.4) 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.

8.30.5 LAYER reference annotation for PATTERN

A *layer* reference annotation in the context of a *pattern* shall be defined as shown in Semantics 79.

```
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 8.16). The physical object represented by the pattern shall reside on a layer. A pattern declaration without layer reference annotation shall be considered incomplete.

8.31 REGION declaration

A *region* object shall be declared as shown in Syntax 64.

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 9.16) and/or by a *boolean* annotation (see 8.32.2) shall be additive, i.e., the region shall be considered the union of the specified geometries. If a *geometric transformation* (see 9.18) is present, it shall apply to all specified geometries within the region.

```
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

8.32 Annotations related to a REGION declaration

8.32.1 REGION reference annotation

A *region* reference annotation shall be defined as shown in Semantics 80.

```
KEYWORD REGION = annotation {
  CONTEXT = arithmetic_model ;
}
SEMANTICS REGION
  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 10.3).

8.32.2 BOOLEAN annotation

A *boolean* annotation shall be defined as shown in Semantics 81.

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

Semantics 81—BOOLEAN annotation

The purpose of the boolean annotation is to specify a region by a boolean operation (see 9.11). The name of a *pattern* (see 8.29) or the name of another region shall be considered a legal operand. The operators specified in Table 76 and Table 81 shall be considered legal operators.

9. Description of functional and physical implementation

9.1 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 provide a compact 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 9.4) or a set of one or more *statetable* statements (see 9.6). The purpose of the behavior and statetable statements is to formally specify the logic state space of the circuit and the change in logic state as a response to a given stimulus.

The function statement can also contain a specification for implementation using the *structure* statement (see 9.5).

9.2 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 provide a compact specification of a test environment for a digital electronic circuit implemented by a cell. A cell can contain at most one test statement.

The test statement can contain a *behavior* statement (see 9.4) or a set of one or more *statetable* statements (see 9.6). The purpose of the behavior and statetable statements is to formally specify the logic state space of the test environment and the change in logic state as a response to a given stimulus.

9.3 Definition and usage of a pin variable

9.3.1 Pin variable and pin value

A *pin variable* and a *pin value* shall be defined as shown in Syntax 67.

```
pin_variable ::=
    pin_variable_identifier
pin_value ::=
    pin_variable | boolean_value
```

Syntax 67—Pin variable and pin value

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

the name of a declared *pin* (see 8.6) in conjunction with an optional *index* (see 6.6),
the name of a declared *pingroup* (see 8.7) in conjunction with an optional *index*,
the name of a declared *node* (see 8.12), or
the hierarchical name of a declared *port* (see 8.23) as a child of a declared *scalar pin*.

A *pin value* shall be either an identifier referring to a *pin variable* or a *boolean value* (see 6.10).

A declared pin can be used as a pin variable involved in a *test* statement (see 9.2) or in a *function* statement (see 9.1), according to its *direction* and *view* annotation value (see 9.3.3, Table 73).

9.3.2 Pin assignment

A *pin assignment* shall be defined as shown in Syntax 68.

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

Syntax 68—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.

9.3.3 Usage of a pin variable in the context of a FUNCTION or a TEST statement

A declared *pin* (see 8.6) with *pintype* annotation value *digital* (see 8.8.4) or a declared *pingroup* (see 8.7) can be used as a *pin variable*.

A pin variable can be involved in a *function* statement (see 9.1) or in a *test* statement (see 9.2), depending on the annotation values for *direction* (see 8.8.5) and *view* (see 8.8.3), according to Table 73.

Table 73—Annotation values for PINs involved in FUNCTION and TEST

Category	DIRECTION	VIEW
Input for function	*input*	*functional* or *both*
Output for function	*output*	*functional* or *both*
Bidirectional for function	*both*	*functional* or *both*

Table 73—Annotation values for PINs involved in FUNCTION and TEST (Continued)

Category	DIRECTION	VIEW
Internal for function	*none*	*none*
Input for test	*input*	*none*
Output for test	*output*	*none*
Bidirectional for test	*both*	*none*
Internal for test	*none*	*none*

An *attribute* statement (see 7.5) can be used to specify a relationship between a pin variable and a particular test method. See section 8.8.24, Table 61 for attribute values related to memory BIST.

The relationship between pin variables involved in the *test* statement and in the *function* statement and the applicable *direction* annotation values are illustrated in Figure 16.

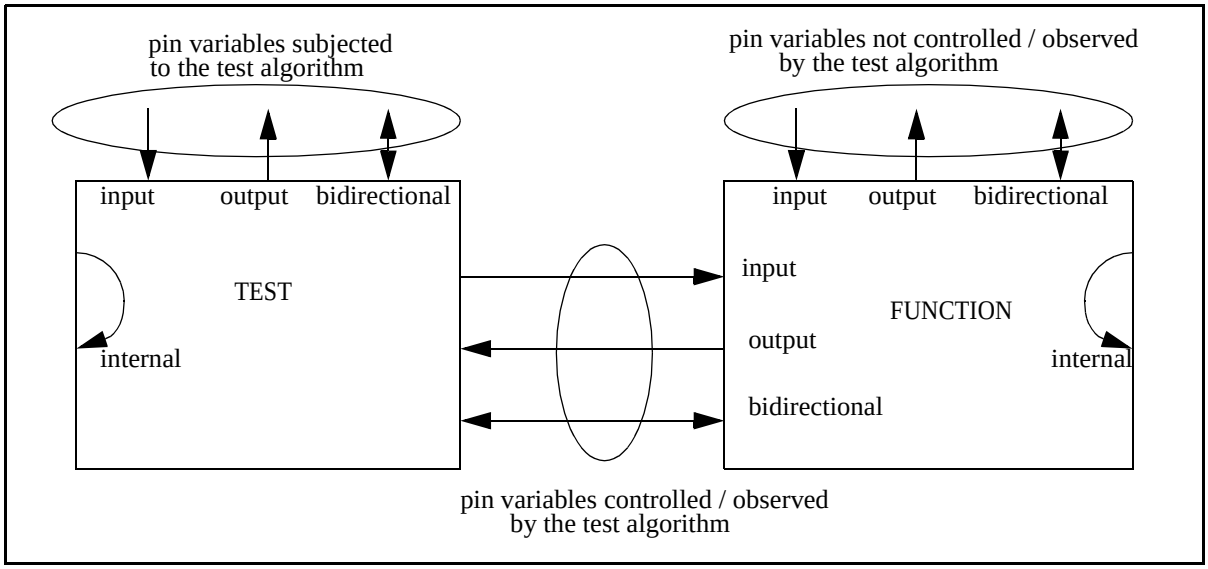

Figure 16—Relationship between FUNCTION and TEST

The digital electronic circuit symbolized by the *function* box communicates with its environment. Part of its environment is the test environment symbolized by the *test* box. A test algorithm, i.e., an algorithmically specified stimulus can be applied to the test environment. The test algorithm controls input variables and observes output variables of the electronic circuit. In addition, the electronic circuit can have other input and output variables which are not controlled or observed by the test algorithm. The electronic circuit and the test environment can also have their internal variables which do not communicate with their environment.

NOTE—The *direction* and *view* annotations are defined from a circuit-centric perspective from which the test environment is viewed as a virtual extension of the circuit.

9.4 BEHAVIOR statement

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

```
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 } }
primitive_instantiation ::=
    primitive_identifier [ identifier ] { pin_value { pin_value } }
  | primitive_identifier [ identifier ] { boolean_assignment { boolean_assignment } }
```

Syntax 69—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 9.9) or a *vector expression* (see 9.12). The order of alternative 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 9.3.1). 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 8.9). 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 primitive declarations is specified in 9.14.

9.5 STRUCTURE statement and CELL instantiation

A *structure* statement shall be defined as shown in Syntax 70.

```
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 70—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.

NOTE—A *structure* statement is intended to be complementary to a *behavior* or a *statetable* statement. An application that requires knowledge of the functional behavior of a cell, for example a synthesis application, is expected to comprehend the behavior statement rather than to infer the functional behavior from the structure 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.

Mapping by order shall be established using a *pin value* (see 9.3.1) associated with the cell instance. A corresponding pin variable associated with the cell reference shall be inferred by the order of pin declarations within the cell reference.

Mapping by name shall be established using a *pin assignment* (see 9.3.2). 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.

9.6 STATETABLE statement

A *statetable* statement shall be defined as shown in Syntax 71.

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.

```
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 71—STATETABLE statement

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 be 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.

It shall be the responsibility of the ALF parser to check for a consistent format of the statetable. It shall be the responsibility of the application to check for complete and consistent contents of the statetable.

9.7 NON_SCAN_CELL statement

A *non-scan cell* statement shall be defined as shown in Syntax 72.

```
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 72—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; } }
```

9.8 RANGE statement

A *range* statement shall be defined as shown in Syntax 73.

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

Syntax 73—RANGE statement

The range statement shall be used to specify a valid address space for elements of a *vector pin* or a *matrix pin* (see 8.6) or a *vector pingroup* (see 8.7). In case of a matrix pin, the range shall pertain to the *second multi-index* (see 8.6, Syntax 49).

If no range statement is specified, the valid address space A is given by the following mathematical relationship:

$$0 \le A \le 2^B - 1 \qquad B = \begin{cases} 1 + i_L - i_R & \text{if}(i_L > i_R) \\ 1 + i_R - i_L & \text{if}(i_L \le i_R) \end{cases}$$

where

A is an unsigned integer representing the address space within a vector-pin or a matrix-pin,
B is the *bitwidth* of the vector-pin or the matrix-pin,
i_L is the left index within the vector-pin or the matrix-pin,
i_R is the right index bit within the vector-pin or the matrix-pin,

in accordance with 6.6.

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:	$3 \le A \le 13$

End of example

9.9 Boolean expression

A *boolean expression* shall be defined as shown in Syntax 74.

```
boolean_expression ::=
      ( boolean_expression )
    | boolean_value
    | identifier
    | boolean_unary_operator boolean_expression
    | boolean_expression boolean_binary_operator boolean_expression
    | boolean_expression ? boolean_expression : boolean_expression
boolean_unary_operator ::=
    ! | ~ | & | ~& | | | ~| | ^ | ~^
boolean_binary_operator ::=
      & | && | ~& | | | || | ~| | ^ | ~^
    | relational_operator
    | arithmetic_operator
    | shift_operator
```

Syntax 74—Boolean expression

The purpose of a boolean expression is to specify a *boolean operation* (see 9.11). The evaluation result of a boolean expression shall be a *boolean value* (see 6.10, 9.10).

A legal operand in a boolean expression shall be a *boolean value* (see 6.10) or an *identifier* (see 6.13) representing a boolean value. In case of a *comparison operation* (see 9.11.6), a legal operand can also be a *number* (see 6.5) or a *string value* (see 6.15).

A legal operator in a boolean expression shall be a *boolean unary operator*, a *boolean binary operator*, an *arithmetic operator* for *integer arithmetic operation* (see 6.4.1, 9.11.4), a *relational operator* for *comparison opera-*

tion (see 6.4.3, 9.11.6), a *shift operator* for *shift operation* (see 6.4.4, 9.11.5), or a combination of a *question mark* and a *colon* defining a *conditional operation* (see 9.11.3).

The precedence of operators in a boolean expression shall be from the strongest to the weakest in the following order:

a) boolean operation enclosed by *parentheses*, i.e., **(** , **)**
b) *bitwise operation* using a *boolean unary operator*, i.e., **~**, **&**, **~&**, **|**, **~|**, **^**, **~^** (see 9.11.2)
c) *logical inversion*, i.e., **!** (see 9.11.1)
d) *shift*, i.e., **<<**, **>>** (see 9.11.5)
e) *comparison*, i.e., **==**, **!=**, **>**, **<**, **>=**, **<=** (see 9.11.6)
f) *bitwise xor, xnor* using a *boolean binary operator*, i.e., **^**, **~^** (see 9.11.2)
g) *multiply, divide, modulus*, i.e., *****, **/**, **%** (see 9.11.4)
h) *bitwise and, nand* using a *boolean binary operator*, i.e., **&**, **~&** (see 9.11.2)
i) *logical and*, i.e., **&&** (see 9.11.1)
j) *add, subtract*, i.e., **+**, **-** (see 9.11.4)
k) *bitwise or, nor* using a *boolean binary operator*, i.e., **|**, **~|** (see 9.11.2)
l) *logical or*, i.e., **||** (see 9.11.1)
m) delimiter for *conditional operation*, i.e., **?**, **:** (see 9.11.3)

When operators of the same precedence are subsequently encountered in a boolean expression, the evaluation shall proceed from the left to the right.

9.10 Boolean value system

9.10.1 Scalar boolean value

A *scalar boolean value* shall be described by an *alphanumerical bit literal* (see 6.8). 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 value in 3-value system	Comment
0	false	strong	0	Use when logical value is defined and drive strength is strong or not defined.
1	true	strong	1	
X or x	unknown	strong	X or x	
L or l	false	weak	0	Use for modeling a bus holder, a pull up or a pull down device.
H or h	true	weak	1	
W or w	unknown	weak	X or x	
Z or z	not defined	zero	X or x	Use for high impedance.
U or u	not defined	not defined	X or x	Use for uninitialized signal in simulation.

A *boolean expression* (see 9.9) 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 explicitly specified as evaluation result in the boolean expression.

9.10.2 Vectorized boolean value

A *vectorized boolean value* shall be described either by a *based literal* (see 6.9) or by an *integer* (see 6.5). A vectorized boolean value can be mapped into a vector of *alphanumeric bit literals* (see 6.8). The number of bit literals shall be called *bitwidth.*

An *octal digit* (see 6.9) can be mapped into a three bit vector of bit literals, by numerically converting a number in octal base to a number in binary base.

A *hexadecimal digit* (see 6.9) can be mapped into a four bit vector of bit literals, by numerically converting a number in hexadecimal base to a number in binary base. The uppercase letters *A* through *F* or the corresponding lowercase letters *a* through *f* shall be used to represent the decimal numbers 10 through 15.

An *alphabetic bit literal* (see 6.8) 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`**

End of example

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 $$N = \sum_{p=0}^{B-2} s(p) \cdot 2^p - s(B-1) \cdot 2^{B-1}$$

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(*B*-1) is the scalar value (zero or one) of 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 8.6) 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:

$$p = \begin{cases} i - i_R & \text{if}(i_L > i_R) \\ i_R - i & \text{if}(i_L \le i_R) \end{cases}$$

where

i is the index within a vector pin.
i_R is the right-most index within a vector pin. The corresponding position is 0.
i_L is the left-most index within a vector pin. The corresponding position is B-1.

Example:

```
PIN [5:8] pin1;
PIN [7:4] pin2;
```

bit[index]	bit[index]	position
pin1[5]	pin2[7]	3
pin1[6]	pin2[6]	2
pin1[7]	pin2[5]	1
pin1[8]	pin2[4]	0

End of example

9.10.3 Non-assignable boolean value

A *non-assignable boolean value* shall be described by a *symbolic bit literal* (see 6.8), as shown in Table 75.

Table 75—Symbolic boolean values

Symbol	Logical value	Drive strength	Comment
?	arbitrary, yet constant	arbitrary	use for “don’t care”
*	subject to random change	arbitrary	variable is not monitored

A *symbolic bit literal* or a *based literal* (see 6.9) containing a symbolic bit literal can not be assigned to a pin variable as a boolean value. A symbolic bit literal can be used within a *statetable* (see 9.6) as a *statetable control value*, but not as a *statetable data value*.

When being part 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.

9.11 Boolean operations and operators

9.11.1 Logical operation

The operators for a *logical operation* shall be defined as shown in Table 76.

Table 76—Logical operations

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

A logical *inversion* shall be evaluated within the 3-value system according to Table 77.

Table 77—Evaluation of logical inversion

A	! A
false	true
true	false
unknown	unknown

A logical *and* or a logical *or* shall be evaluated within the 3-value system according to Table 78.

Table 78—Evaluation of logical AND and logical OR

A	B	A && B	A || B
false	false	false	false
true	false	false	true
unknown	false	false	unknown
false	true	false	true
true	true	true	true
unknown	true	unknown	true
false	unknown	false	unknown
true	unknown	unknown	true
unknown	unknown	unknown	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*.

9.11.2 Bitwise operation

The operators for a *bitwise operation* shall be defined as shown in Table 79.

Table 79—Bitwise operations

Operator	Description
~	bit-wise *inversion*
&	bit-wise *and*
|	bit-wise *or*
^	bit-wise exclusive *or* (*xor*)
~&	bit-wise *and* with inversion (*nand*)
~|	bit-wise *or* with inversion (*nor*)
~^	bit-wise exclusive *or* with inversion (*xnor*)

A bit-wise operation is defined as a repeated single-bit operation to all bits of the operand. The operators for bit-wise operations, except bit-wise inversion, can be used as *boolean unary* or as *boolean binary* operators.

A *bit-wise inversion* operator shall apply a *logical inversion* (see Table 77) to each bit of a vectorized boolean value. The result shall be a vectorized boolean value containing the inverted bits.

A bit-wise *boolean binary* operator for one of the operations *and*, *or*, *nand*, *nor*, *xor*, *xnor* shall apply a single-bit operation to each corresponding bit of two vectorized boolean values. The operands shall be aligned to the right-most bit. If the operands have different bitwidths, the missing bits of the operand with smaller bitwidth shall be *not defined*, i.e., represented by the symbol ‘U’. If at least one operand is a vectorized boolean value, the result shall be a vectorized boolean value. If both operands are scalar boolean values, the result shall be a scalar boolean value.

The single-bit operation *or* and the single-bit operation *and*, respectively, shall be defined in the same way as the logical operation *or* and the logical operation *and*, respectively (see Table 78).

```
A & B  is equivalent to  A && B  for single bit operands
A | B  is equivalent to  A || B  for single bit operands
```

The single-bit operation *nor* and the single-bit operation *nand*, respectively, shall be defined by applying a logical inversion to the result of the logical operation *or* and the logical operation *and*, respectively.

```
A ~& B  is equivalent to  ! (A && B)  for single bit operands
A ~| B  is equivalent to  ! (A || B)  for single bit operands
```

The single-bit operations *xor* and *xnor* shall be defined according to Table 80.

Table 80—Evaluation of single-bit XOR and XNOR

A	B	A ^ B	A ~^B
false	false	false	true
true	false	true	false
unknown	false	unknown	unknown
false	true	true	false
true	true	false	true
unknown	true	unknown	unknown
false	unknown	unknown	unknown
true	unknown	unknown	unknown
unknown	unknown	unknown	unknown

A *boolean unary* operator for the operation *and*, *or*, *xor*, respectively, shall reduce a vectorized boolean value to a scalar boolean value by applying a single-bit operation *and*, *or*, *xor*, respectively, to all bits of the operand combined.

```
& V[3:1] is equivalent to V[3] && V[2] && V[1]
| V[3:1] is equivalent to V[3] || V[2] || V[1]
^ V[3:1] is equivalent to V[3] ^  V[2] ^  V[1]
```

A *boolean unary* operator for the operation *nand*, *nor*, *xnor*, respectively, shall apply a logical inversion to the result of the operation *and*, *or*, *xor*, respectively.

```
~& V  is equivalent to  ! (& V)
~| V  is equivalent to  ! (| V)
~^ V  is equivalent to  ! (^ V)
```

A vectorized boolean value can be used as operand for a logical operation. For this purpose, the vectorized boolean value shall be reduced to a scalar boolean value by applying the bit-wise *boolean unary* operation *or*.

```
  ! (V)  is equivalent to      ! (| V)
A && V   is equivalent to A && (| V)
V || B   is equivalent to (| V) || B
```

NOTE—A and B stand for scalar boolean values, V stands for a vectorized boolean value.

9.11.3 Conditional operation

The evaluation of a *boolean expression* (see 9.9), a *vector expression* (see 9.12), or an *arithmetic expression* (see 10.1) involving the symbols shown in Table 81 shall be called a *conditional operation*.

Table 81—Conditional operation

Symbol	Description
?	delimiter between *if-clause* and *then-clause*
:	delimiter between *then-clause* and *else-clause*

The boolean expression to the left of the question mark shall be called *if-clause*. The expression, i.e., a boolean expression or a vector expression or an arithmetic expression, to the right of the question mark shall be called *then-clause*. The expression to the right of the colon shall be called *else-clause*.

If the *if-clause* evaluates true, the *then-clause* shall be evaluated. Otherwise, the *else-clause* shall be evaluated.

NOTE—The else-clause within a conditional operation can represent a conditional operation in itself. Thus nested conditional operations can be described, wherein the evaluation of clauses proceeds from the left to the right.

9.11.4 Integer arithmetic operation

The operators for an *integer arithmetic operation* shall be defined as shown in Table 82.

Table 82—Integer arithmetic operation

Operator	Description
+	add
-	subtract
*****	multiply
/	divide
%	modulus

All operations involving the operators in Table 82 shall be *integer* operations. A legal operand shall be either an *integer* or a *boolean value* that is converted into an integer.

A *scalar boolean value* (see 9.10.1) represented as a *bit literal* (see 6.8) shall be converted into an *unsigned integer.*

A *vectorized boolean value* (see 9.10.2) represented as a *based literal* (see 6.9) shall be converted into an *unsigned integer* or into a *signed integer.* The conversion shall depend on the *datatype* annotation value (see 8.8.10) of the pin variable associated with the operand.

The application shall be responsible for handling exceptions. Exceptions include the following cases:

— integer conversion of a boolean value involving the logical value *unknown*,
— the operation *division* and *modulus* involving a second operand with value zero,
— any evaluation results that do not fit the bitwidth of the pin variable which the result is assigned to, i.e., overflow or underflow.

9.11.5 Shift operation

The operators for a *shift operation* shall be defined as shown in Table 83

Table 83—Shift operation

Operator	Description
<<	shift left
>>	shift right

A shift operation shall involve two operands. The first operand shall be a *vectorized boolean value* (see 9.10.2), represented by an *integer* (see 6.5), by a *based literal* (see 6.9), or, as a trivial case, by a *bit literal* (see 6.8). The second operand shall be an *unsigned integer* (see 6.5), specifying the number of positions *N* by which the bits of the first operand are to be shifted.

For *shift left*, N bits of the first operand, starting from the right, shall be replaced with the logical value *unknown*. For *shift right*, N bits of the first operand, starting from the left, shall be replaced with the logical value *unknown*.

9.11.6 Comparison operation

A comparison operation shall be defined as a *numerical comparison*, a *logical comparison* or a *string comparison*. The evaluation result shall be *true*, *false* or *unknown*.

The operators for a *numerical comparison* shall be defined as shown in Table 84.

Table 84—Numerical comparison

Operator	Description
==	equal
!=	not equal
>	greater
<	lesser
>=	greater or equal
<=	lesser or equal

A legal operand for a numerical comparison shall be a *number* (see 6.5) or a boolean value that can be interpreted as an *integer* according to 9.10.2.

The operators for a *logical comparison* shall be defined as shown in Table 85.

Table 85—Logical comparison

Operator	Description	comment
~^	equal in logical value, also called *xnor*	symbols from Table 76 are overloaded
^	not equal in logical value, also called *xor*	
==	equal in logical value and drive strength	symbols from Table 84 are overloaded
!=	not equal in logical value and drive strength	

A legal operand for a logical comparison shall be a *scalar boolean value* (see 9.10.1, Table 74).

The operations *equal in logical value* and *not equal in logical value* shall be evaluated as specified for the single-bit operations *xnor* and *xor* in Table 80.

The operations *equal in logical value and drive strength* and *not equal in logical value and drive strength* shall be evaluated according to Table 86.

Table 86—Evaluation of logical comparison involving drive strength

Logical value of operands A and B (true, false, unknown, or not defined)	Drive strength of operands A and B (strong, weak, zero, or not defined)	Result for A == B	Result for A != B
Same for both operands.	Same for both operands.	true	false
Same for both operands.	Different for each operand.	false	true
Different for each operand.	Any.	false	true

Example

```
'b0 ~^ 'bL evaluates true
'b0 == 'bL evaluates false
'b1 ~^ 'bH evaluates true
'b1 == 'bH evaluates false
'bX ~^ 'bW evaluates unknown
'bX == 'bW evaluates false
'bZ ~^ 'bZ evaluates unknown
'bZ == 'bZ evaluates true
```

End of example

The operators for a *string comparison* shall be defined as shown in Table 87.

Table 87—String comparison

Operator	Description	comment
==	string values are equal	symbols from Table 84 are overloaded
!=	string values are different	

A legal operand for a string comparison shall be a *string value* (see 6.15). If at least one operand is a *quoted string* (see 6.14), the comparison shall be case-sensitive. Otherwise, the comparison shall be case-insensitive. If an operand is an *identifier* (see 6.13) representing a constant or a variable holding a string value, the comparison shall apply to the string value rather than to the identifier.

9.12 Vector expression and control expression

A *vector expression* and a *control expression* shall be defined as shown in Syntax 75.

```
vector_expression ::=
      ( vector_expression )
    | single_event
    | vector_expression vector_operator vector_expression
    | boolean_expression ? vector_expression : vector_expression
    | boolean_expression control_and vector_expression
    | vector_expression control_and boolean_expression
    | vector_expression_macro
single_event ::=
    edge_literal boolean_expression
vector_operator ::=
    event_operator | event_and | event_or
event_and ::=
    & | &&
event_or ::=
    | | ||
control_and ::=
    & | &&
control_expression ::=
      ( vector_expression )
    | ( boolean_expression )
```

Syntax 75—Vector expression and control expression

The purpose of a *control expression* is to specify the ALF name of a declared *vector* (see 8.14), a *control statement* within a *behavior* statement (see 9.4), or an annotation with *valuetype* control expression (see 7.11.1).

The purpose of a *vector expression* is to specify a pattern of events. A vector expression shall be satisfied when the pattern of events specified within the vector expression matches an actually realized pattern of events within an application context.

A legal operand for a vector expression shall be a *single event* (see 9.13.1) or a *vector expression macro* (see 6.17).

A legal operator for a vector expression shall be an *event operator* (see 6.4.5), i.e., an *event-sequence* operator (see 9.13.2, 9.13.3) or an *event-permutation* operator (see 9.13.3), an *event-and* (see 9.13.2), an *event-or* (see 9.13.3), a *control-and* (see 9.13.5), or a combination of a *question mark* and a *colon* defining a *conditional operation* (see 9.11.3).

The precedence of operators involved in a vector expression shall be from the strongest to the weakest in the following order:

a) boolean operation enclosed by *parentheses*, i.e., **(** , **)**
b) *edge literal* (see 6.12, 9.13.1)
c) *event permutation* operators, i.e., **<~>**, **<->**, **<&>** (see 9.13.3)
d) *event-and* operator and *control-and* operator, i.e., **&**, **&&** (see 9.13.2, 9.13.5)
e) *event sequence* operators, i.e., **~>**, **->**, **&>** (see 9.13.2, 9.13.3)
f) *event-or* operator, i.e., **|**, **||** (see 9.13.3)
g) delimiter for *conditional operation*, i.e., **?**, **:** (see 9.11.3, 9.13.5)

When operators of the same precedence are subsequently encountered in a vector expression, the evaluation shall proceed from the left to the right.

9.13 Specification of a pattern of events

9.13.1 Specification of a single event

In order to evaluate a *vector expression* (see 9.12) against an actually realized pattern of events, a set of variables shall be observed for a temporal change of their value (see 9.13.4). A change of value within one observed variable shall be called a *single event*. An *edge literal* (see 6.12) shall be used as unary operator to specify the pattern of a single event. The operand, i.e., the variable subjected to the change of value, shall be a *boolean expression* (see 9.9).

A single event shall be interpreted according to Table 88.

Table 88—Specification of a single event

Row	Edge literal	Event on operand
1	*first*_bit_literal *second*_bit_literal	value changes from *first*_bit_literal to *second*_bit_literal
2	*first*_based_literal *second*_based_literal	value changes from *first*_based_literal to *second*_based_literal
3	**??**	value before and after the change is *arbitrary*
4	**?***	value is *random* after the change
5	***?**	value is *random* before the change
6	**?!**	value changes from any value to a different value
7	**?~**	every binary digit changes from any value to a different value
8	**?-**	value does not change

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 with 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.

The edge literal in row 8 specifies the same value before and after the event. Such a specification shall be interpreted as event by exclusion, i.e., a change of value does not happen on the operand but on another observed variable.

An *arbitrary* value in row 3, 6, and 7 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 in row 4 and 5 shall be interpreted as a value subjected to random change. The random change is not monitored.

The usage of an edge literal for specification of a single event is illustrated by the timing diagram in Figure 17.

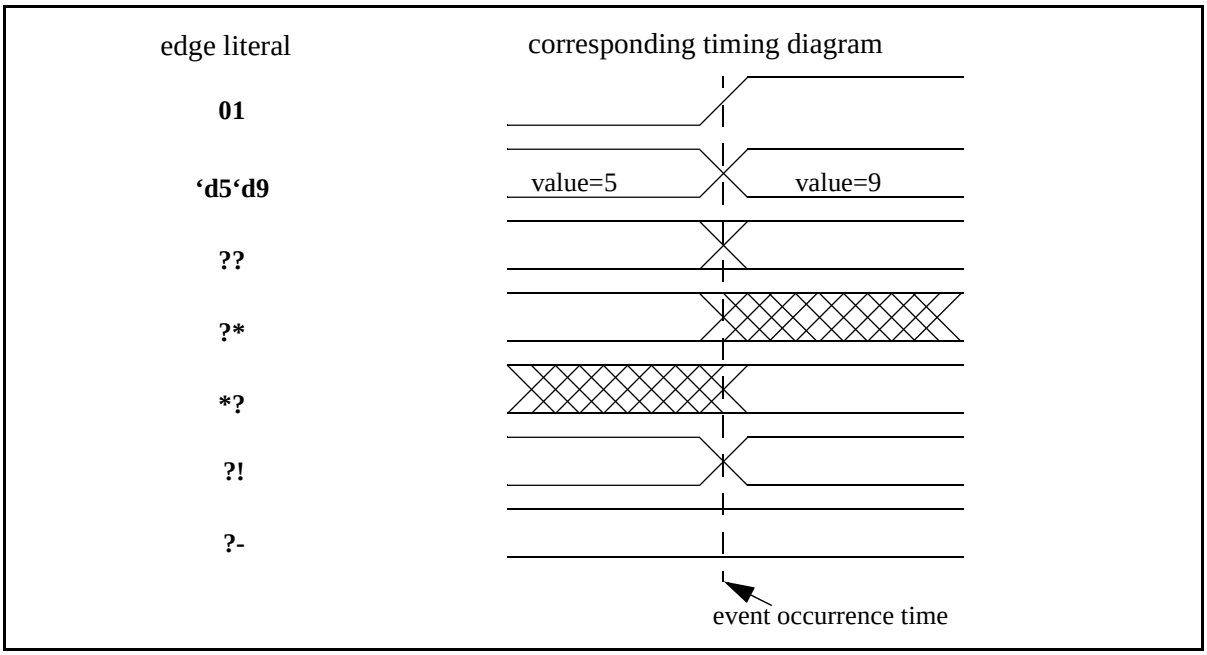

Figure 17—Timing diagram for single events

NOTE—The specification of a single event does not imply any transition time. The transition time in Figure 17 is only for the purpose of illustrating the difference between **??** and **?!**.

NOTE—The operator **??** can be called a *neutral operator*, since a specified single event involving **??** on an arbitrary operand always matches a single event on any operand. A single event involving the neutral operator can be called a *neutral single event*.

9.13.2 Specification of a compound event

A pattern of events involving one or more single events shall be called a *compound event*. A pattern of events involving more than one single event shall be called a *truly compound event*. A pattern of events involving only one single event shall be called a *degenerate compound event*.

The operators in Table 89 shall be used for specification of a truly compound event.

Table 89—Operators for specification of a compound event

Operator	Description
~>	The event to the left is *eventually followed by* the event to the right
->	The event to the left is *immediately followed by* the event to the right
&& or **&**	The event to the left and the event to the right occur *at the same time*

The purpose of said operators is to specify a temporal relation between two single events *A* and *B* within a truly compound event *C*.

— (*A*~>*B*) means that *A* occurs before *B*.
— (*A*->*B*) means that (*A*~>*B*) is satisfied and there exists no single event *O* that could satisfy both (*A*~>*O*) and (*O*~>*B*).
— (*A*&*B*) means that both *A* and *B* occur, but neither (*A*~>*B*) nor (*B*~>*A*) is satisfied.

In order to extend the applicability of said operators to compound events, the *earliest* and *latest* events are defined as follows:

— A single event *A* within *C* shall be called *earliest event* within *C*, if there exists no single event *O* within *C* that could satisfy (*O*~>*A*).
— A single event *B* within *C* shall be called *latest event* within *C*, if there exists no single event *O* within *C* that could satisfy (*B*~>*O*).
— Within a degenerate compound event, the single event shall be called both earliest and latest event.

NOTE—A truly compound event can have more than one earliest or latest event, since events can occur at the same time.

Using these definitions, said operators shall specify a temporal relation between two compound events *C* and *D* as follows:

— (*C*~>*D*) means that the latest event within *C* occurs before the earliest event within *D*.
— (*C*->*D*) means that (*C*~>*D*) is satisfied and there exists no single event *O* that could satisfy both (*C*~>*O*) and (*O*~>*D*).
— (*C*&*D*) means that both *C* and *D* are satisfied and the latest events within *C* and *D* occur at the same time.

9.13.3 Specification of a compound event with alternatives

A vector expression that satisfies more than one pattern of events shall be called a *compound event with alternatives*.

The operators in Table 90 shall be used for specification of a compound event with alternatives.

Table 90—Operators for specification of a compound event with alternatives

Operator	Description
|| or |	The vector expression is satisfied if the compound event to the left or the compound event to the right occurs.

Table 90—Operators for specification of a compound event with alternatives

Operator	Description
&>	The vector expression (*C***&>***D*) is equivalent to (*C***&***D* | *C*->*D*), wherein *C* and *D* are compound events.

A particular case of a compound event with alternatives is a *permutation of compound events*, i.e., a vector expression that is satisfied when the compound events occur in permutable order.

An operator that specifies occurrence of compound events in permutable order shall be called *event permutation* operator. In contrast, an operator that specifies occurrence of compound events in a particular order shall be called *event sequence* operator.

The operators in Table 91 shall be used for specification of a permutation of compound events.

Table 91—Operators for specification of permutations of compound events

Event permutation operator	Description	Corresponding event sequence operator	
<~>	(*C*<~>*D*) is equivalent to (*C*~>*D* | *D*~>*C*)	~>	(see Table 89)
<->	(*C*<->*D*) is equivalent to (*C*->*D* | *D*->*C*)	->	(see Table 89)
<**&**>	(*C*<**&**>*D*) is equivalent to (*C***&**>*D* | *D***&**>*C*)	**&>**	(see Table 90)

Permutation of more than two compound events shall be defined as follows:

A vector expression wherein

a) all operands are related to each other by the same event permutation operator, and,
b) each operand is bound by higher precedence than said event permutation operator,

shall be satisfied, if any permutation of the operands, related to each other by the corresponding event sequence operator, is satisfied.

Example:

(*A*<&>*B*<&>*C*) is equivalent to (*A*&>*B*&>*C* | *A*&>*C*&>*B* | *C*&>*A*&>*B* | *B*&>*A*&>*C* | *B*&>*C*&>*A* | *C*&>*B*&>*A*)

wherein *A*, *B*, *C* denote compound events, and *A*, *B*, *C* do not contain operators of the same or lower precedence than &>, unless such operators are bound within parentheses.

End of example

9.13.4 Evaluation of a specified pattern of events against a realized pattern of events

A vector expression, i.e., a specified pattern of events, shall be evaluated against an actually realized pattern of events in an application context. The realized pattern of events shall be established according to the following rules a) and b):

a) A primary pattern of events on a set of *pin variables* (see 9.3) shall be observed. The set of pin variables shall be specified by the *monitor* annotation (see 8.15.10) within a *vector* declaration (see 8.14) or by the *scope* annotation (see 8.8.18) within a *pin* or a *pingroup* declaration (see 8.6, 8.7). A monitor annotation shall take precedence over a scope annotation.

b) The primary pattern of events shall be reduced by replacing the events on the pin variables involved in the vector expression with events on boolean expressions involved in the vector expression. The events on any pin variables not involved in the vector expression shall be not be replaced.

Example:

The 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 -> 01 (C & ! D))**.

Therefore, the single events on **A**, **B**, **C** and **D** are observed.
For evaluation of v_1, the events on **(A&B)**, **(B|C)** and **D** are observed.
For evaluation of v_2, the events on **A**, **B** and **(C & ! D)** are observed.

Figure 18 shows a realized pattern of events. The grey circles and bold edges indicate where the realized pattern of events satisfies the respective vector expression v_1 and v_2.

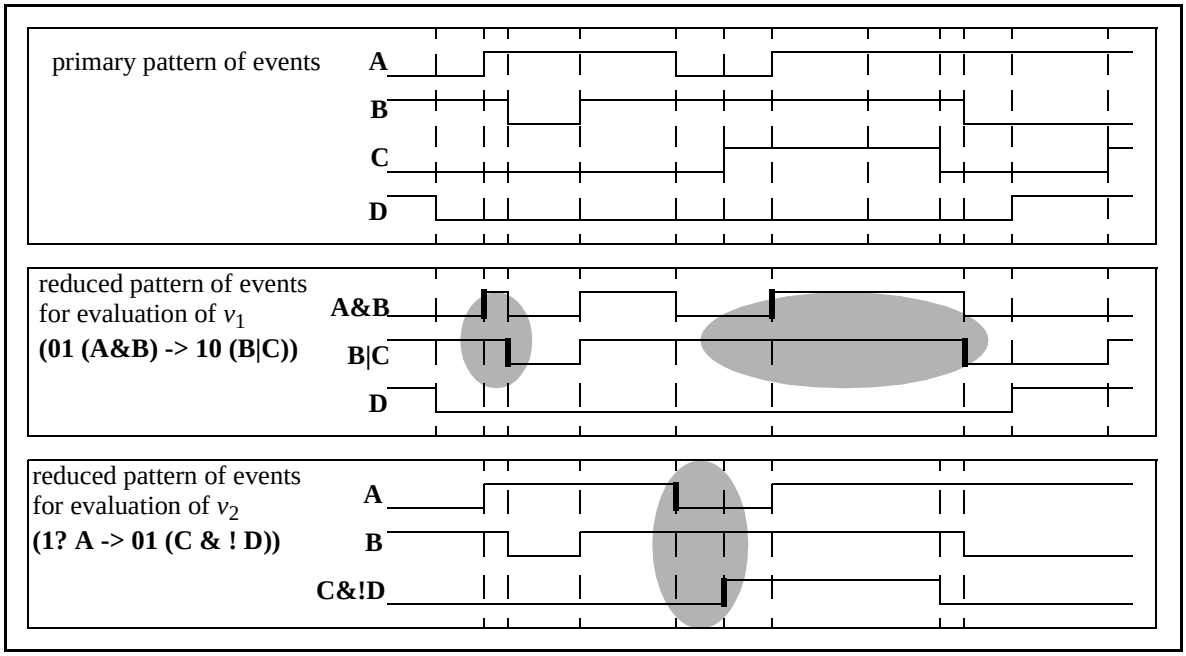

Figure 18—Realized pattern of events

End of example

The occurrence time of each single event within a realized pattern of events can be interpreted as a totally ordered set of real numbers, using the mathematical relation “lesser or equal”. It can be shown that the properties of a totally ordered set are satisfied. The following notations are used:

A, *B* denote single events within a realized event pattern
t(*A*), *t*(*B*) denote the occurrence time of respective single events *A*, *B* within a realized event pattern

For reference, the following properties are required for a totally ordered set:

1) Reflexivity: $t(A) \leq t(A)$
2) Weak anti symmetry: $t(A) \leq t(B)$ and $t(B) \leq t(A)$ implies $t(A) = t(B)$
3) Transitivity: $t(A) \leq t(B)$ and $t(B) \leq t(C)$ implies $t(A) \leq t(C)$
4) Comparability: For any element within the set, either $t(A) \leq t(B)$ or $t(B) \leq t(A)$

A specified pattern of events shall be satisfied, if each relation between single events therein is satisfied by the realized pattern of events, according to Table 92.

Table 92—Satisfaction of a specified relation within a realized pattern of events

Specified relation		Condition for satisfaction by realized pattern of events
A&>B	(see Table 90)	$t(A) \leq t(B)$
A~>B	(see Table 89)	$t(A) \leq t(B)$, but not $t(B) \leq t(A)$, i.e., $t(A) < t(B)$
A->B	(see Table 89)	$t(A) < t(B)$, and no event *O* exists with $t(A) < t(O) < t(B)$
A&&B	(seeTable 89)	$t(A) \leq t(B)$ and $t(B) \leq t(A)$, i.e., $t(A) = t(B)$

A realized pattern of events can be completely described using the relations *A&&B*, i.e., the single events *A* and *B* occur *at the same time*, and *A->B*, i.e., the single event *A* is *immediately followed by* the single event *B*. In the case of single events occurring *at the same time*, a distinction shall be made between *at the same time by implication* and *at the same time by coincidence*.

NOTE—In order to evaluate the vector expression against the realized pattern of events, it is not necessary to record the actual occurrence time of the single events. It suffices to record the relations pertinent to the ordered set.

The following rules shall apply concerning the relations between single events within a realized pattern of events:

a) A value change of a boolean expression and a single event on a pin variable causing this value change shall be interpreted to occur *at the same time 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 interpreted to occur *at the same time by implication*.
c) If a value change of a pin variable occurs as a consequence of a value change of another pin variable within the context of a *behavior* statement (see 9.4), the consequence shall be interpreted to occur *immediately followed by* the cause.
d) If the elapsed time between single events on mutually independent pin variables is measured zero, said events can be interpreted to occur *at the same time by coincidence*.
e) In the context of a declared *vector* (see 8.14), all pin variables shall be considered mutually independent, even though a causal dependency between some pin variables can exist in the context of a *behavior* statement. Therefore events can not occur *at the same time by implication* within the context of a vector.

NOTE—It is possible that an application can not determine the temporal relation between events occurring *at the same time by coincidence*. Instead, the events could be represented in random order with the temporal relation *immediately followed by* each other. Therefore it is recommended to use the operator <&> to specify *at the same time by coincidence* and to use the operator && to specify *at the same time by implication*.

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 interpreted to occur at the same time by implication.
Within the context of the behavior statement, the single event **(01 Z)** is interpreted 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** can be greater than zero, lesser than zero, or zero. Therefore, the single events **(01 A)** and **(01 Z)** can occur at the same time by coincidence.

End of example

9.13.5 Specification of a conditional pattern of events

A pattern of events specified within a vector expression shall be called a *conditional pattern of events*, if the evaluation against the realized pattern of events is made dependent on a condition described as a boolean expression. A conditional pattern of events shall be evaluated against the realized pattern of events only if the boolean expression evaluates true in the realized pattern of events.

A conditional pattern of events shall be described using the *control-and* operator or the *if-then-else* construct, as specified in Table 93.

Table 93—Specification a conditional pattern of events

Operator	Description	Comment
&& or **&**	pattern of events shall be evaluated while boolean expression is true	*control-and* uses overloaded symbol, which is also used for *logical and* (see Table 76) and *bitwise and* (see Table 79).
? and **:**	if-then-else construct, see 9.11.3	If-then-else construct exists for *boolean expression* (see Syntax 74), for *vector expression* (see Syntax 75) and for *arithmetic expression* (see Syntax 81).

The order of operands within a vector expression involving the *control-and* operator shall be free, i.e.:

(*v* **&** *b*) shall be equivalent to (*b* **&** *v*)

wherein *v* denotes a vector expression, and *b* denotes a boolean expression.

A vector expression involving the *if-then-else* construct can be transformed into a vector expression involving the *control-and* operator, according to the following rule:

(*b* **?** v_1 **:** v_2) shall be equivalent to (v_1 **&** *b* | v_2 **&** **!** *b*)

wherein *b* denotes a boolean expression representing the *if-clause*, v_1 denotes a vector expression representing the *then-clause*, and v_2 denotes a vector expression representing the *else-clause*.

9.14 Predefined PRIMITIVE

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

9.14.1 Predefined PRIMITIVE ALF_BUF

The primitive *ALF_BUF* shall be defined as shown in Semantics 82.

```
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

9.14.2 Predefined PRIMITIVE ALF_NOT

The primitive *ALF_NOT* shall be defined as shown in Semantics 83.

```
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

9.14.3 Predefined PRIMITIVE ALF_AND

The primitive *ALF_AND* shall be defined as shown in Semantics 84.

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

Semantics 84—Predefined PRIMITIVE ALF_AND

9.14.4 Predefined PRIMITIVE ALF_NAND

The primitive *ALF_NAND* shall be defined as shown in Semantics 85.

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

Semantics 85—Predefined PRIMITIVE ALF_NAND

9.14.5 Predefined PRIMITIVE ALF_OR

The primitive *ALF_OR* shall be defined as shown in Semantics 86.

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

Semantics 86—Predefined PRIMITIVE ALF_OR

9.14.6 Predefined PRIMITIVE ALF_NOR

The primitive *ALF_NOR* shall be defined as shown in Semantics 87.

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

Semantics 87—Predefined PRIMITIVE ALF_NOR

9.14.7 Predefined PRIMITIVE ALF_XOR

The primitive *ALF_XOR* shall be defined as shown in Semantics 88.

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

Semantics 88—Predefined PRIMITIVE ALF_XOR

9.14.8 Predefined PRIMITIVE ALF_XNOR

The primitive *ALF_XNOR* shall be defined as shown in Semantics 89.

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

Semantics 89—Predefined PRIMITIVE ALF_XNOR

9.14.9 Predefined PRIMITIVE ALF_BUFIF1

The primitive *ALF_BUFIF1* shall be defined as shown in Semantics 90.

```
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

9.14.10 Predefined PRIMITIVE ALF_BUFIF0

The primitive *ALF_BUFIF0* shall be defined as shown in Semantics 91.

```
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

9.14.11 Predefined PRIMITIVE ALF_NOTIF1

The primitive *ALF_NOTIF1* shall be defined as shown in Semantics 92.

```
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

9.14.12 Predefined PRIMITIVE ALF_NOTFIF0

The primitive *ALF_NOTIF0* shall be defined as shown in Semantics 93.

```
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

9.14.13 Predefined PRIMITIVE ALF_MUX

The primitive *ALF_MUX* shall be defined as shown in Semantics 94.

```
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

9.14.14 Predefined PRIMITIVE ALF_LATCH

The primitive *ALF_LATCH* shall be defined as shown in Semantics 95.

9.14.15 Predefined PRIMITIVE ALF_FLIPFLOP

The primitive *ALF_FLIPFLOP* shall be defined as shown in Semantics 96.

```
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

```
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

9.15 WIRE instantiation

A *wire instantiation* shall be defined as shown in Syntax 76.

```
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 76—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 8.10). 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 9.3.1) shall be associated with the wire instance. A corresponding pin variable associated with the wire reference shall be inferred 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 9.3.2). 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.

9.16 Geometric model

A *geometric model* shall be defined as shown in Syntax 77.

```
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 77—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 8.29) or a *region* (see 8.31).

The numbers in the *point* statement shall be measured in units of *distance* (see 10.19.9).

The parent object of the geometric model can contain a *geometric transformation* (see 9.18) applicable to the geometric model.

The keywords for geometric models shown in Semantics 97 shall be predefined.

```
KEYWORD DOT = geometric_model;
KEYWORD POLYLINE = geometric_model;
KEYWORD RING = geometric_model;
KEYWORD POLYGON = geometric_model;
```

Semantics 97—Predefined geometric models

Table 94 specifies the meaning of predefined geometric model identifiers.

Table 94—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>2 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>2 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 19.

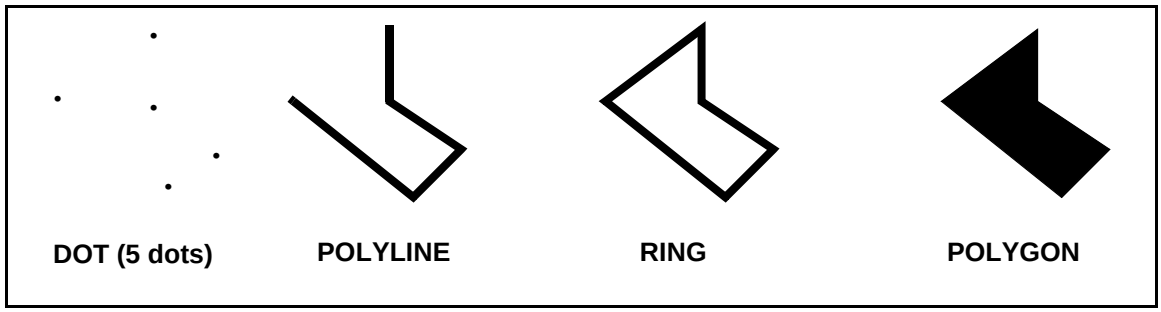

Figure 19—Illustration of geometric models

A *point_to_point* annotation shall be defined as shown in Semantics 98.

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 20. It specifies the shortest possible connection between points.

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

Semantics 98—POINT_TO_POINT annotation

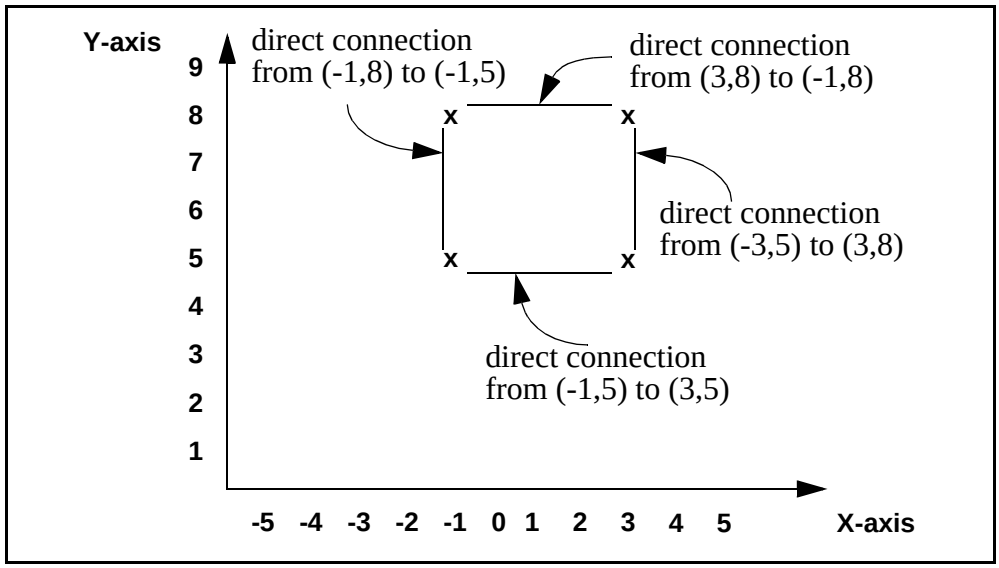

Figure 20—Illustration of direct point-to-point connection

The meaning of the annotation value *manhattan* is illustrated in Figure 21. 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.

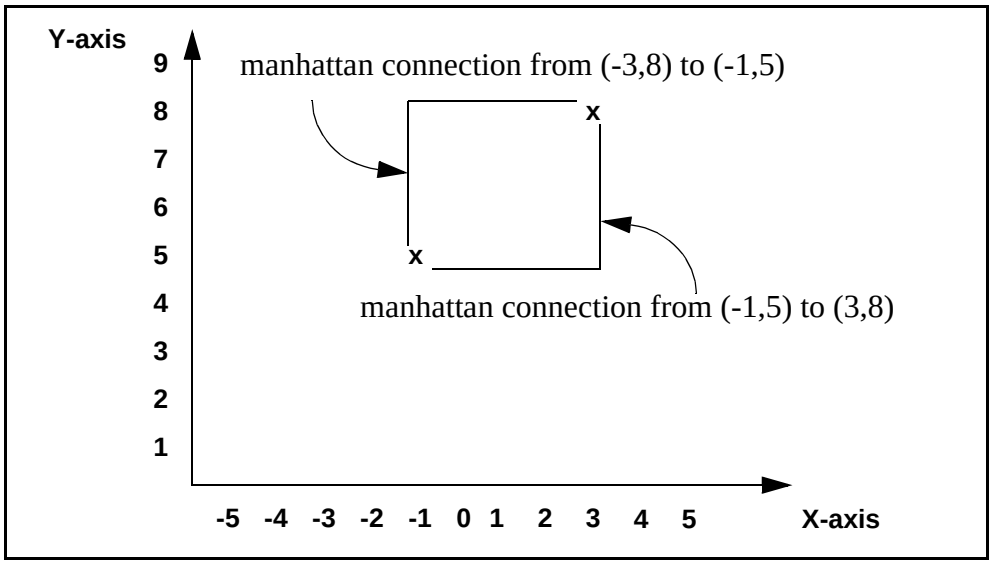

Figure 21—Illustration of manhattan point-to-point connection

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.

9.17 Predefined geometric models using TEMPLATE

A *template* declaration (see 7.15) can be used to describe particular geometric models. This section describes predefined geometric models.

9.17.1 Predefined TEMPLATE RECTANGLE

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

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

Semantics 99—Predefined TEMPLATE RECTANGLE

9.17.2 Predefined TEMPLATE LINE

The template *line* shall be predefined as shown in Semantics 100.

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

Semantics 100—Predefined TEMPLATE LINE

9.18 Geometric transformation

A *geometric transformation* shall be defined as shown in Syntax 78.

```
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 78—Geometric transformation

A *geometric model* (see 9.16) 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 10.19.9).
— 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 22.

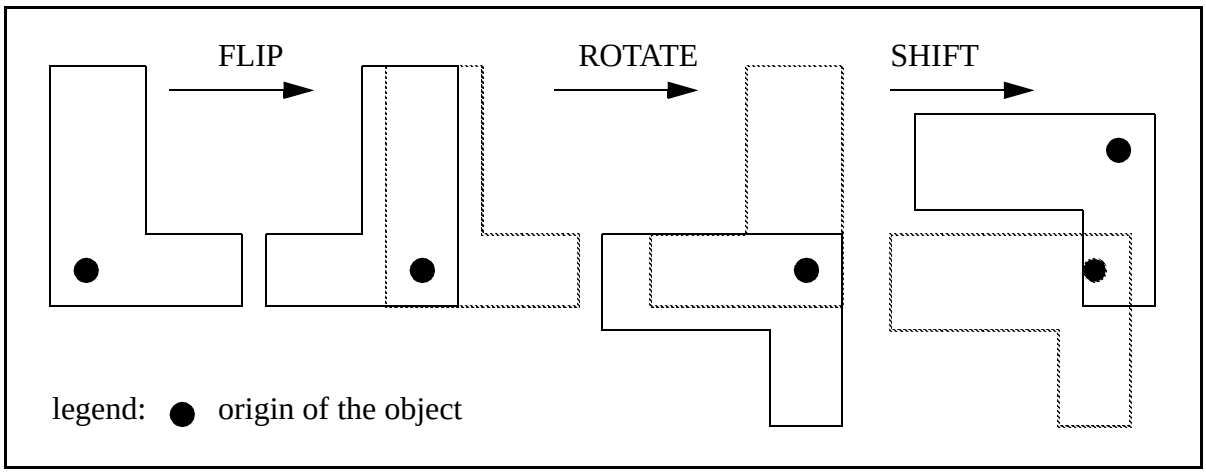

Figure 22—Illustration of FLIP, ROTATE, and SHIFT

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 horizontal and a vertical distance of 4 units each.

```
REPEAT = 5 {
    SHIFT { 4 4 }
}
```

The following example replicates an object twice 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 }
    }
}
```

9.19 ARTWORK statement

An *artwork* statement shall be defined as shown in Syntax 79.

```
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 79—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, and the cell described in the ALF.

A *geometric transformation* (see 9.18) 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 establish 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;
```

```
        }
    }
```

9.20 VIA instantiation

A *via instantiation* shall be defined as shown in Syntax 80.

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

Syntax 80—VIA instantiation

The purpose of a via instantiation is to enable the definition of a design *rule* (see 8.20), a *blockage* (see 8.22) or a *port* (see 8.23) involving a declared *via* (see 8.18). A geometric transformation (see 9.18) 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.

10. Description of electrical and physical measurements

10.1 Arithmetic expression

An *arithmetic expression* shall be defined as shown in Syntax 81.

```
arithmetic_expression ::=
      ( arithmetic_expression )
    | arithmetic_value
    | identifier
    | boolean_expression ? arithmetic_expression : arithmetic_expression
    | sign arithmetic_expression
    | arithmetic_expression arithmetic_operator arithmetic_expression
    | macro_arithmetic_operator ( arithmetic_expression { , arithmetic_expression } )
macro_arithmetic_operator ::=
      abs | exp | log | min | max
```

Syntax 81—Arithmetic expression

The purpose of an arithmetic expression is the construction of an *arithmetic model* (see 10.3) or an *arithmetic assignment* (see 7.16).

A legal operand in an arithmetic expression shall be an arithmetic value or an *identifier* (see 6.13) representing an arithmetic value.

A legal operator in an arithmetic expression shall be a *sign* (see 6.5, 10.2.1), an *arithmetic operator* for *floating point arithmetic operation* (see 6.4.1, 10.2.2), a *macro arithmetic operator* (see 10.2.3), or a combination of a *question mark* and a *colon* defining a *conditional operation* (see 9.11.3).

The precedence of operators in arithmetic expressions shall be from strongest to weakest in the following order:

a) arithmetic operation enclosed by *parentheses*, i.e., **(** , **)**
b) *sign*, i.e., **+**, **-** (see 10.2.1)
c) *power*, i.e., ****** (see 10.2.2)
d) *multiplication*, *division*, *modulus*, i.e.,*****, /, **%** (see 10.2.2)
e) *addition*, *subtraction*, i.e., **+**, **-** (see 10.2.2)
f) delimiter for *conditional operation*, i.e., **?**, **:** (see 9.11.3)

When operators of the same precedence are subsequently encountered in an arithmetic expression, the evaluation shall proceed from the left to the right.

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
```

End of example

10.2 Arithmetic operations and operators

10.2.1 Sign inversion

A sign can be used as unary operator in an arithmetic expression.

Table 95 defines the semantics of the sign used as unary operator.

Table 95—Sign used as unary arithmetic operator

Operator	Description
+	no sign inversion.
-	sign inversion.

NOTE: The positive sign can be considered as neutral operator.

10.2.2 Floating point arithmetic operation

Table 96 defines the semantics of binary arithmetic operators.

Table 96—Binary arithmetic operators

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus
**	Power

All operations involving the operators in Table 96, including *division* and *modulus*, shall be *floating point* operations.

The following mathematical restrictions apply:

— The second operand of *division* can not be zero.
— The second operand of *modulus* can not be zero.
— The second operand of *power* shall be a positive value if the first operand is zero.
— The second operand of *power* shall be an integer value if the first operand is negative.

The application shall be responsible for handling the mathematical restrictions.

10.2.3 Macro arithmetic operator

Table 97 defines the semantics of macro arithmetic operators.

Table 97—Macro arithmetic operators

Operator	Description	number of operands
`log`	Natural logarithm.	1 operand
`exp`	Natural exponential.	1 operand
`abs`	Absolute value.	1 operand
`min`	Minimum.	`N` operands, N $\geq$ 1
`max`	Maximum.	N operands, N $\geq$ 1

The following mathematical restrictions shall apply:

— The operand of the *natural logarithm* shall be a positive value.

The application shall be responsible for handling the mathematical restrictions.

10.3 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 82.

```
arithmetic_model ::=
      trivial_arithmetic_model
    | partial_arithmetic_model
    | full_arithmetic_model
    | arithmetic_model_template_instantiation
```

Syntax 82—Arithmetic model

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

A *trivial arithmetic model* shall be defined as shown in Syntax 83.

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

Syntax 83—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 87).

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

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

Syntax 84—Partial arithmetic model

The purpose of a partial arithmetic model is to specify a singular or a plural *model qualifier* (see Syntax 87), or a *table* (see Syntax 91) or a *trivial min-max* statement (see Syntax 94). The specification contained within a partial arithmetic model can be inherited by another arithmetic model of the same type, according to the following rules.

a) If the partial arithmetic model has no name, the specification shall be inherited by all arithmetic models of the same type appearing 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 only be inherited by an arithmetic model containing a reference to the name, using the *model reference annotation* (see 10.9.5).
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 85.

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

Syntax 85—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 86). A full arithmetic model can also contain a singular or a plural *arithmetic model qualifier* (see Syntax 87).

The *arithmetic model identifier* in Syntax 83, Syntax 84 and Syntax 85 shall be declared as a *keyword* (see 7.9) and provide specific semantics for the arithmetic model.

An *arithmetic model body* shall be defined as shown in Syntax 86.

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

Syntax 86—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 10.4), or by a *min-typ-max* statement (see 10.5), or by a singular or a plural *arithmetic submodel* (see 10.7).

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

```
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 87—Arithmetic model qualifier

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 7.3), an *annotation container* (see 7.4) or a *from-to* statement (see 10.12) can be inherited by another arithmetic model using a *model reference annotation* (see 10.9.5).

A *non-inheritable arithmetic model qualifier*, i.e., an *auxiliary arithmetic model* (see 10.6), a *violation* (see 10.10) or a *wire instantiation* (see 9.15) shall apply only for the arithmetic model under evaluation.

10.4 HEADER, TABLE, and EQUATION statements

A *header-table-equation* statement shall be defined as shown in Syntax 88.

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

Syntax 88—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 85).

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

```
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 89—HEADER statement

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

Any *arithmetic model* (see 10.3) with a *header* as a parent shall be interpreted as a *header arithmetic model*. A declared *keyword* (see 7.9) for *arithmetic model* shall apply as identifier.

NOTE — The syntax for *header arithmetic model* is a true subset of the syntax for *arithmetic model*.

An *equation* statement shall be defined as shown in Syntax 90.

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

Syntax 90—EQUATION statement

The arithmetic expression within the equation statement shall represent the mathematical operation for evaluation of the arithmetic model.

Each dimension shall be involved in the arithmetic expression. The arithmetic expression shall refer to a dimension by name, if a name identifier exists or by type otherwise. Consequently, the type or the name of a dimension shall be unique.

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

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

Syntax 91—TABLE statement

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 contains 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:

$$S = \prod_{i=1}^{N} S(i) \qquad N \geq 1$$

$$S \geq 1$$

$$0 \leq P(p_1, ..p_i.., p_N) \leq S - 1$$

$$P(p_1, ..p_i.., p_N) = \sum_{i=1}^{N} p_i \prod_{k=1}^{i-1} S(k) \qquad S(i) \geq 1$$

$$0 \leq p_i \leq S(i) - 1$$

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
$P(p_1, ..p_i.., p_N)$ 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

Figure 23 shows an example of a three-dimensional table.

dimension 1: (a_0 a_1 a_2 a_3) $S(1) = 4$
dimension 2: (b_0 b_1) $S(2) = 2$
dimension 3: (c_0 c_1 c_2) $S(3) = 3$

$P(p_1, p_2, p_3) = p_1 + 4\,p_2 + 8\,p_3$

table: $S = 24$

$x_0(a_0, b_0, c_0)$ $x_1(a_1, b_0, c_0)$ $x_2(a_2, b_0, c_0)$ $x_3(a_3, b_0, c_0)$
$x_4(a_0, b_1, c_0)$ $x_5(a_1, b_1, c_0)$ $x_6(a_2, b_1, c_0)$ $x_7(a_3, b_1, c_0)$
$x_8(a_0, b_0, c_1)$ $x_9(a_1, b_0, c_1)$ $x_{10}(a_2, b_0, c_1)$ $x_{11}(a_3, b_0, c_1)$
$x_{12}(a_0, b_1, c_1)$ $x_{13}(a_1, b_1, c_1)$ $x_{14}(a_2, b_1, c_1)$ $x_{15}(a_3, b_1, c_1)$
$x_{16}(a_0, b_0, c_2)$ $x_{17}(a_1, b_0, c_2)$ $x_{18}(a_2, b_0, c_2)$ $x_{19}(a_3, b_0, c_2)$
$x_{20}(a_0, b_1, c_2)$ $x_{21}(a_1, b_1, c_2)$ $x_{22}(a_2, b_1, c_2)$ $x_{23}(a_3, b_1, c_2)$

Figure 23—Example of a three-dimensional table

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 monotonic ascending order.

A *full arithmetic model* or any of its dimensions can inherit a set of legal values from a *partial arithmetic model* (see Syntax 84), 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.

Rules and restrictions for the mathematical evaluation of an arithmetic model can only be defined within the *header-table-equation* statement. A legal set or a legal range of values defined within an arithmetic model shall not interfere with the mathematical evaluation of the arithmetic model itself. In particular, an *arithmetic expression* shall be evaluated within the domain of its mathematical validity. A lookup table shall be evaluated according to the *interpolation* annotation (see 10.9.3).

10.5 MIN, MAX, and TYP statements

A *min-typ-max* statement shall be defined as shown in Syntax 92.

```
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 92—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 93.

```
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 93—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 88) or a *violation* statement (see 10.10). 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 87).

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

```
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 94—Trivial MIN, MAX and TYP statements

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 84) 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.

A trivial min-max statement within a *header arithmetic model* (see Syntax 89) shall define the range of validity of a particular dimension. An application tool can evaluate the *header-table-equation* statement (see Syntax 88) 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 84), 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 82) that can be interpreted as either a *partial arithmetic model* (see Syntax 84) or a *full arithmetic model* (see Syntax 85), the interpretation of a trivial min-max statement as a *min-typ-max statement* (see Syntax 94) shall take precedence. As a consequence, the interpretation of an arithmetic model as a full arithmetic model takes precedence.

Semantics 101 defines the interpretation of *min*, *max*, *typ* as a particular *arithmetic submodel* (see 10.7).

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

Semantics 101—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 92, Syntax 93, and Syntax 94, respectively) is a true subset of the syntax rule for *arithmetic submodel* (see Syntax 96).

10.6 Auxiliary arithmetic model

An *auxiliary arithmetic model* shall be defined as shown in Syntax 95.

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

Syntax 95—Auxiliary arithmetic model

An *arithmetic model* (see 10.3) with another arithmetic model as a parent shall be called *auxiliary arithmetic model*. A declared *keyword* (see 7.9) for *arithmetic model* shall apply as identifier. The parent of the *auxiliary arithmetic model* shall be called *principal arithmetic model*.

The purpose of an auxiliary arithmetic model is to serve as a *non-inheritable arithmetic model qualifier* (see Syntax 87) for the principal arithmetic model. The auxiliary arithmetic model can be associated with a constant *arithmetic value* and with an *inheritable arithmetic model qualifier* (see Syntax 87).

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.

10.7 Arithmetic submodel

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

```
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 96—Arithmetic submodel

The purpose of an arithmetic submodel is to serve as *arithmetic model body* (see Syntax 86), wherein the data associated with the *full arithmetic model* (see Syntax 82) 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 7.9) and provide specific semantics.

10.8 Arithmetic model container

10.8.1 General arithmetic model container

A general *arithmetic model container* shall be defined as shown in Syntax 97.

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

Syntax 97—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 7.9) and provide specific semantics.

10.8.2 Arithmetic model container LIMIT

The arithmetic model container *limit* shall be defined as shown in Syntax 98.

```
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 98—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 10.5) in the context of a *limit arithmetic model* or a *limit arithmetic submodel*.

Any *arithmetic model* (see 10.3) with a *limit* as a parent shall be interpreted as a *limit arithmetic model*. A declared *keyword* (see 7.9) for *arithmetic model* shall apply as identifier. Any *arithmetic submodel* (see 10.7)

with a *limit arithmetic model* as a parent shall be interpreted as a *limit arithmetic submodel*. A declared *keyword* (see 7.9) 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*.

Semantics 102 defines the interpretation of *limit* as *arithmetic model container.*

```
KEYWORD LIMIT = arithmetic_model_container;
```

Semantics 102—Arithmetic model container LIMIT

10.8.3 Arithmetic model container EARLY and LATE

The arithmetic model containers *early* and *late* shall be defined as shown in Syntax 99.

```
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 99—Arithmetic model container EARLY and LATE

The purpose of the arithmetic model containers *early* and *late* is to specify an envelope of a timing waveform. The arithmetic model *delay* (see 10.11.3), *retain* (see 10.11.4) or *slewrate* (see 10.11.5) 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.

Semantics 103 defines the interpretation of *early* and *late* as arithmetic model container.

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

Semantics 103—Arithmetic model container EARLY and LATE

The arithmetic model containers *early* and *late* shall be children of a declared *vector* (see 8.14).

10.9 Generally applicable annotations for arithmetic models

10.9.1 UNIT annotation

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

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

Semantics 104—UNIT annotation

The purpose of the unit annotation is to specify a *multiplier prefix value* (see 6.7) 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 7.11.6).

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 7.9) of an arithmetic model. Therefore it can not be changed.

Example:

The arithmetic model *delay* (see 10.11.3) has the SI-model annotation value *time*. Therefore *delay* can inherit the unit annotation value of the arithmetic model *time* (see 10.11.1).

10.9.2 CALCULATION annotation

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

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

Semantics 105—CALCULATION annotation

The meaning of the annotation values is shown in Table 98.

Table 98—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.

A specific application of rule c) is described in section 10.11.3 for the arithmetic model *delay.*

10.9.3 INTERPOLATION annotation

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

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

Semantics 106—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 99.

Table 99—Interpolation annotation

Annotation value	Evaluation method	Handling data out of range
`linear`	Linear interpolation	Linear extrapolation
`ceiling`	Select the next greater value in the table	Select the largest value in the table
`floor`	Select the next lesser value in the table	Select the smallest value in the table
`fit`	Linear or higher-order interpolation	Linear extrapolation

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 $x^- < x$. If no such value exists, x^- denotes the smallest value in the table.
x^+ denotes the value to the right of x, such that $x < x^+$. If no such value exists, 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 $N>0$, i.e., 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 two straight lines in the region between x^- and x^+.
One line is constructed by linear extrapolation based on x^- and its left neighbor.
The other line is constructed by linear extrapolation based on x^+ and its right neighbor.
In case of a monotonic derivative, $y(x)$ is also bound by another straight line.
This line is constructed by linear interpolation based on x^- and x^+.

These conditions are illustrated in Figure 24.

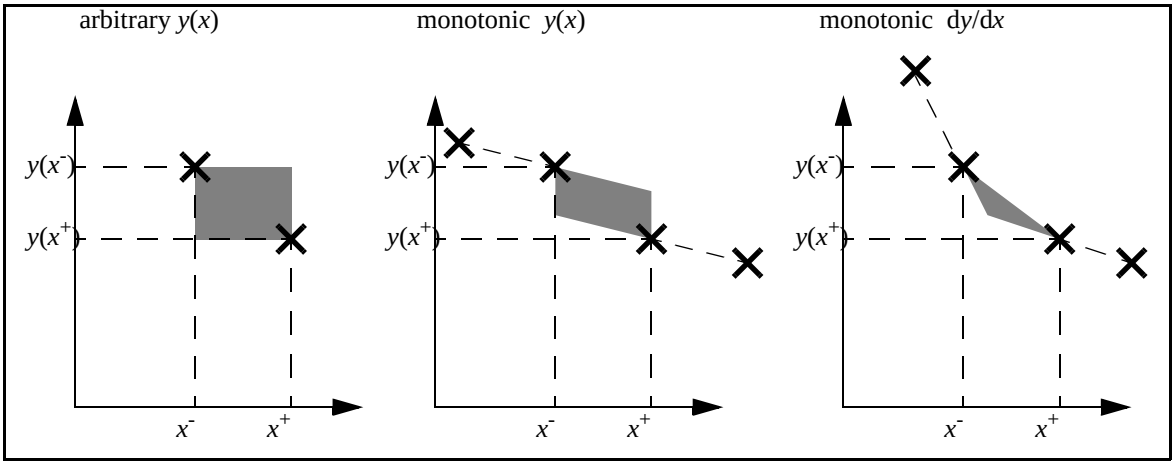

Figure 24—Bounding regions for y(x) with INTERPOLATION=fit

The application shall use a higher-order interpolation only if it provides a tighter bound than linear interpolation.

10.9.4 DEFAULT annotation

A *default* annotation (see 7.11.3) shall be applicable for an arithmetic model, unless the *keyword* declaration (see 7.9) 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 Syntax 85) or a *header arithmetic model* (see Syntax 89) in case the arithmetic model can not be evaluated otherwise. A default annotation shall not apply for a *trivial arithmetic model* (see Syntax 83). A default annotation for a *partial arithmetic model* (see Syntax 84) shall serve as *inheritable arithmetic model qualifier* (see Syntax 87), 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 6.11) compatible with the arithmetic model's *valuetype* (see 7.11.1). A default annotation value associated with a *full arithmetic model* shall be either an arithmetic value compatible with its valuetype, or, alternatively, an *identifier* referring to another arithmetic model or to an *arithmetic submodel* (see 10.7).

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.

10.9.5 MODEL reference annotation

A *model* reference annotation shall be defined as shown in Semantics 107.

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

Semantics 107—MODEL reference annotation

The purpose of a model reference annotation is to acquire an *inheritable arithmetic model qualifier* (see Syntax 87), an evaluation result (see Syntax 91 and Syntax 90) 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 10.7). In this case, the model reference annotation value shall be a *hierarchical identifier* (see 6.13.4) 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 Syntax 89). 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 7.11.6), 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 10.11.1) can refer to the arithmetic model of ALF type *time*.

Rule b): The arithmetic model *delay* (see 10.11.3) 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 10.9.1), the annotation values shall be the same.

10.10 VIOLATION statement, MESSAGE TYPE and MESSAGE annotation

A *violation* statement shall be defined as shown in Syntax 100.

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

Syntax 100—VIOLATION statement

The purpose of a violation statement is to specify the consequence of an evaluation of an *arithmetic model* (see 10.3) that results in a violation of a design constraint or a design limit.

A violation statement shall be subjected to the restriction shown in Semantics 108.

```
SEMANTICS VIOLATION {
  CONTEXT {
    SETUP HOLD RECOVERY REMOVAL NOCHANGE ILLEGAL
    NOISE_MARGIN LIMIT..
  }
}
```

Semantics 108—Semantic restriction for VIOLATION statement

The purpose of the restriction is to specify a legal ancestor 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 ancestor of a violation statement.

A violation statement can contain a *message-type* annotation, a *message* annotation, and a *behavior* statement (see 9.4). A *behavior* statement as a child of a *violation* statement shall only be legal, if its ancestor is a *vector* (see 8.14). This rule is formulated in Semantics 109.

```
SEMANTICS VIOLATION.BEHAVIOR { CONTEXT { VECTOR.. } }
```

Semantics 109—BEHAVIOR statement within VIOLATION

In a simulation application, the *control expression* (see 9.12) associated with the vector shall trigger the behavior as a consequence of the violation.

Example:

Consider a flip-flop with the following functional behavior:

```
FUNCTION {
    BEHAVIOR {
        @ ( 01 clock ) { Q = data; Qbar = ! data; }
    }
}
```

The behavior will change if a setup violation is encountered.

```
VECTOR ( ?! data -> 01 clock ) {
    SETUP = 0.1 { FROM { PIN = data; } TO { PIN = clock; }
        VIOLATION {
            BEHAVIOR { Q = ‘bX; Qbar = ‘bX; }
        }
    }
}
```

End of example

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

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

Semantics 110—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 100.

Table 100—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 111.

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.

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

Semantics 111—MESSAGE annotation

10.11 Arithmetic models for timing, power and signal integrity

10.11.1 TIME

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

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

Semantics 112—Arithmetic model TIME

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 8.2), a declared *cell* (see 8.4), or a declared *wire* (see 8.10)

A *partial arithmetic model* (see Syntax 84) can be used to globally specify an *inheritable arithmetic model qualifier* (see Syntax 87).

— TIME in context of a declared *vector* (see 8.14)

If the *control expression* associated with the vector is a *vector expression* (see 9.12), a *from-to* statement (see 10.12) shall be used as *model qualifier.* The arithmetic model shall represent a measured time interval between two *single events* (see 9.13.1).

Otherwise, if the *control expression* associated with the vector is a *boolean expression* (see 9.9), 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 10.8.2), 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)

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 ancestor of the header arithmetic model is a *vector* with an associated *vector expression*, a *from* statement can be used as *model qualifier* to define a temporal relationship between a *single event* and the dimension time.

If the ancestor 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 95)

The auxiliary arithmetic model *time* shall be used in conjunction with a *measurement* annotation (see 10.13.7). The auxiliary arithmetic model shall specify the time interval during which the measurement is taken.

If the ancestor of the auxiliary arithmetic model is a *vector* with an associated *vector expression*, 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.

10.11.2 FREQUENCY

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

```
KEYWORD FREQUENCY = arithmetic_model ;
SEMANTICS FREQUENCY {
  CONTEXT {
    LIBRARY SUBLIBRARY CELL WIRE VECTOR arithmetic_model
    VECTOR.arithmetic_model_container
    VECTOR..HEADER LIMIT..HEADER
  }
  VALUETYPE = number ;
  SI_MODEL = FREQUENCY ;
}
FREQUENCY { UNIT = GigaHertz; MIN = 0; }
```

Semantics 113—Arithmetic model FREQUENCY

The purpose of the arithmetic model *frequency* is to specify a temporal frequency, i.e., a frequency measured in units of 1/time.

NOTE: If someone desires to specify a spatial frequency, i.e., a frequency measured in units of 1/distance, a different keyword can be declared (see 7.9).

The arithmetic model *frequency* can be a child or a grandchild of a declared *library* or *sublibrary* (see 8.2), a declared *cell* (see 8.4), *wire* (see 8.10) or *vector* (see 8.14).

— FREQUENCY in context of a declared *vector* (see 8.14)

As a descendant of a declared vector with an associated *vector expression* (see 9.12), the arithmetic model shall specify a statistical occurrence frequency of the vector.

As a child of the arithmetic model container *limit* (see 10.8.2), the arithmetic model shall specify a design limit for an occurrence frequency. Otherwise, the arithmetic model shall specify a measured occurrence frequency.

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

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

If the ancestor of the header arithmetic model is a *vector* with an associated *vector expression*, the dimension frequency shall represent the occurrence frequency of the vector.

If the ancestor 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 95)

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 10.13.7). The auxiliary arithmetic model shall specify the repetition frequency of the measurement.

The auxiliary arithmetic models *frequency* and *time* (see 10.11.1) can be used interchangeably, 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 $f = 1 / t$.

10.11.3 DELAY

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

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

Semantics 114—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 10.12) shall be used as *model qualifier*.

— DELAY in context of a declared *vector* (see 8.14)

As a child or a grandchild of a declared vector with an associated *vector expression* (see 9.12), the arithmetic model *delay* shall specify a measured time interval between two *single events* (see 9.13.1), which are referred to as *from-event* and *to-event* (see 10.12). It shall be implied 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 10.9.2) shall be used in conjunction with 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 8.2), a declared *cell* (see 8.4), or a declared *wire* (see 8.10)

As a *partial arithmetic model* (see Syntax 84), *delay* can be used for global specification of a *model qualifier*. In particular, the arithmetic model *threshold* (see 10.11.13) 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 10.11.4), *setup* and *hold* (see 10.11.6), *recovery* and *removal* (see 10.11.7) and *skew* (see 10.11.12) in the context of a *vector*.

10.11.4 RETAIN

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

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

Semantics 115—Arithmetic model RETAIN

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 10.12) shall be used as *model qualifier*.

As a child or a grandchild of a declared vector with an associated *vector expression* (see 9.12), the arithmetic model *retain* shall specify a measured time interval between two *single events* (see 9.13.1), which are referred to as *from-event* and *to-event* (see 10.12). It shall be implied 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 25.

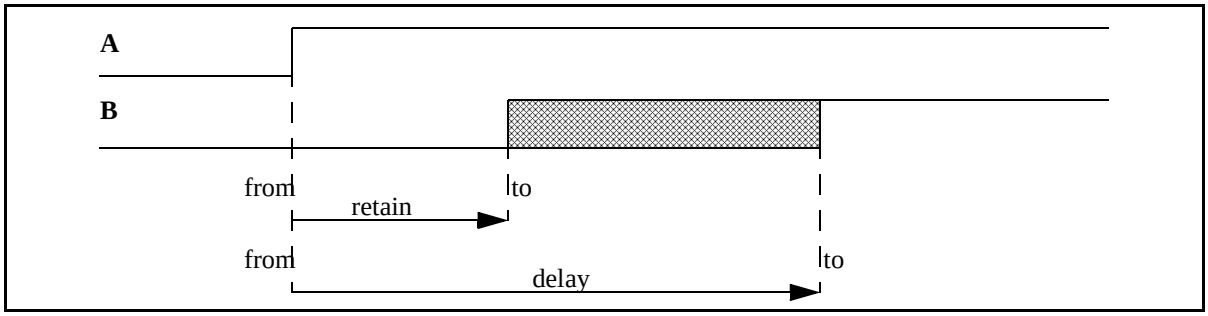

Figure 25—Illustration of RETAIN and DELAY

10.11.5 SLEWRATE

The arithmetic model *slewrate* statement shall be defined as shown in Semantics 116.

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

Semantics 116—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 10.11.13) within a *from-to* statement (see 10.12). No particular waveform shape shall be implied for the transient event.

— SLEWRATE in context of a declared *vector* (see 8.14)

If *slewrate* is a descendant of a declared *vector* with an associated *vector expression* (see 9.12), a *pin reference* annotation, possibly in conjunction with an *edge number* annotation, shall be used (see 10.13.2) to refer to a *single event* (see 9.13.1).

— SLEWRATE in context of a declared *pin* (see 8.6)

If *slewrate* is a child or a grandchild of a declared *pin*, the arithmetic submodel *rise* or *fall* (see 10.21) can be used as a substitute for a reference to a single event.

— SLEWRATE in context of a declared *library* or *sublibrary* (see 8.2), a declared *cell* (see 8.4), or a declared *wire* (see 8.10)

As a *partial arithmetic model* (see Syntax 84), *slewrate* can be used for global specification of a *model qualifier.* In particular, the arithmetic model *threshold* (see 10.11.13) 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)

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 Figure 26.

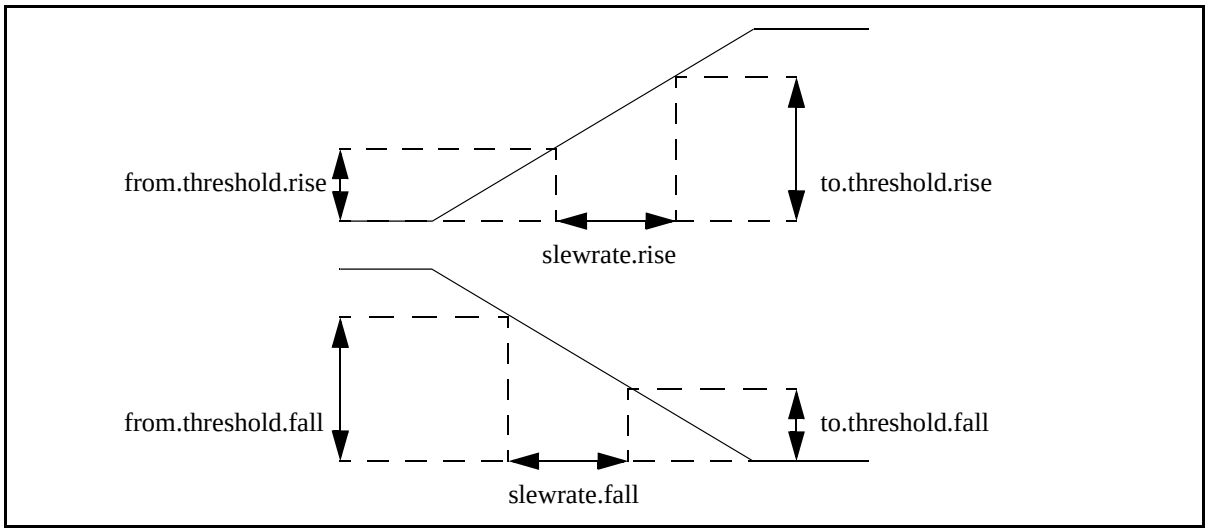

Figure 26—Illustration of SLEWRATE

10.11.6 SETUP and HOLD

The arithmetic models *setup* and *hold* shall be defined as shown in Semantics 117.

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

Semantics 117—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 8.14) with an associated *vector expression* (see 9.12). A *from-to* statement (see 10.12) 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 referred to within a *from* statement. The clock signal shall be referred 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 referred to within a *from* statement. The data signal shall be referred 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 27.

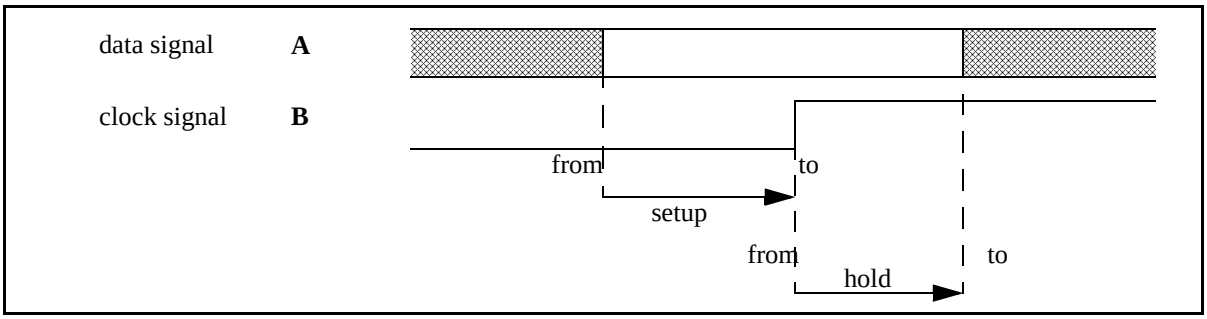

Figure 27—Illustration of SETUP and HOLD

10.11.7 RECOVERY and REMOVAL

The arithmetic models *recovery* and *removal* shall be defined as shown in Semantics 118.

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

Semantics 118—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 8.14) with an associated *vector expression* (see 9.12). A *from-to* statement (see 10.12) 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 shall be referred to within a *from* statement. The clock signal shall be referred 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 referred to within a *from* statement. The asynchronous control signal shall be referred 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 28.

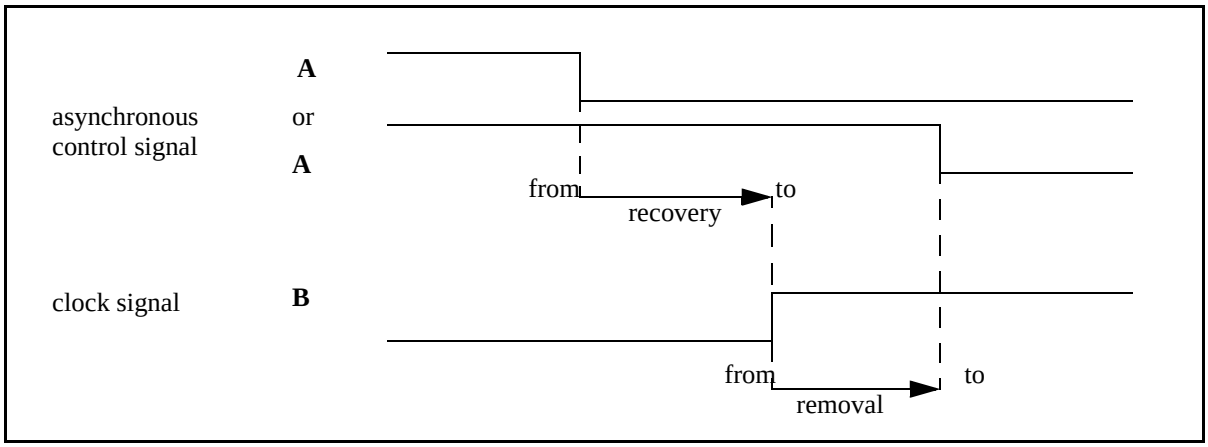

Figure 28—RECOVERY and REMOVAL

10.11.8 NOCHANGE and ILLEGAL

The arithmetic models *nochange* and *illegal* shall be defined as shown in Semantics 119.

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

Semantics 119—Arithmetic models NOCHANGE and ILLEGAL

The purpose of the arithmetic models *nochange* and *illegal* is to specify requirements for the observation or duration of an event pattern in the context of a declared *vector* (see 8.14).

If the *control expression* associated with the vector is a *vector expression* (see 9.12), a *from-event* and a *to-event* can be specified, using a *from-to* statement (see 10.12) as *model qualifier.*

— NOCHANGE in the context of a declared *vector*

If the *control expression* associated with the vector is a *boolean expression* (see 9.9), the arithmetic model *nochange* shall specify a requirement for a minimum time interval during which the boolean expression is true. A partial arithmetic model *nochange* shall specify a requirement for the boolean expression to be forever true.

If the *control expression* associated with the vector is a *vector expression* (see 9.12), the arithmetic model *nochange* shall specify a requirement for a minimum time interval during which the event pattern specified by the vector expression is observed. If a *from-to* statement is specified, this requirement shall pertain only to the event pattern bound by the *from-event* and the *to-event*. A partial arithmetic model *nochange* shall specify a requirement for the event pattern specified by the vector expression or the event pattern bound by the *from-event* and the *to-event* to be observed without change.

— ILLEGAL in the context of a declared *vector*

If the *control expression* associated with the vector is a *boolean expression* (see 9.9), the arithmetic model *illegal* shall specify a requirement for a maximum time interval during which the boolean expression is true. A partial arithmetic model *illegal* shall specify a requirement for the boolean expression to be never true.

If the *control expression* associated with the vector is a *vector expression* (see 9.12), the arithmetic model *illegal* shall specify a requirement for a maximum time interval during which the event pattern specified by the vector expression is observed. If a *from-to* statement is specified, this requirement shall pertain only to the event pattern bound by the *from-event* and the *to-event*. A partial arithmetic model *illegal* shall specify a requirement for the event pattern specified by the vector expression or the event pattern bound by the *from-event* and the *to-event* not to be observed as specified.

Nochange and *illegal* in the context of a *vector expression* are illustrated in Figure 29.

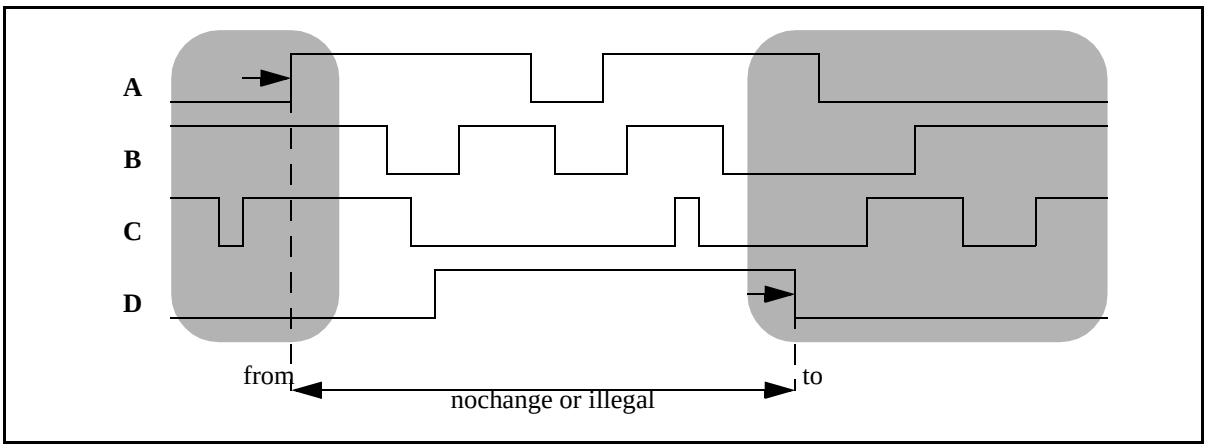

Figure 29—Illustration of NOCHANGE and ILLEGAL

A *vector expression* corresponding to the whole timing diagram (both grey and white parts) is required to trigger the evaluation of the arithmetic model *nochange* or *illegal*.

If a realized sequence of events 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 small arrows), the actual event sequence in-between the *from*-and *to*-events shall be examined.

In the case of *nochange*, the realized sequence of events is required to match the middle of the timing diagram, and possibly a minimal time interval between *from* and *to* is required.

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

10.11.9 PULSEWIDTH

The arithmetic model *pulsewidth* shall be defined as shown in Semantics 120.

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 10.11.13) within a *from-to* statement (see 10.12). No particular waveform shape shall be implied for the sequence of transient events.

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

Semantics 120—Arithmetic model PULSEWIDTH

For a *noise* waveform (see 10.11.14), 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 8.14)

If *pulsewidth* is a child or a grandchild of a declared *vector* with an associated *vector expression* (see 9.12), a *pin reference* annotation, possibly in conjunction with an *edge number* annotation, shall be used (see 10.13.2) to refer to a *single event* (see 9.13.1), representing the leading edge of the pulse.

— PULSEWIDTH in context of a declared *pin* (see 8.6)

If *pulsewidth* is a child or a grandchild of a declared *pin*, the arithmetic submodel *rise* or *fall* (see 10.21) can be used as a substitute for a reference to a single event.

— PULSEWIDTH in context of a declared *library* or *sublibrary* (see 8.2), a declared *cell* (see 8.4), or a declared *wire* (see 8.10)

As a *partial arithmetic model* (see Syntax 84), *pulsewidth* can be used for global specification of a *model qualifier*. In particular, the arithmetic model *threshold* (see 10.11.13) 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)

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 Figure 30.

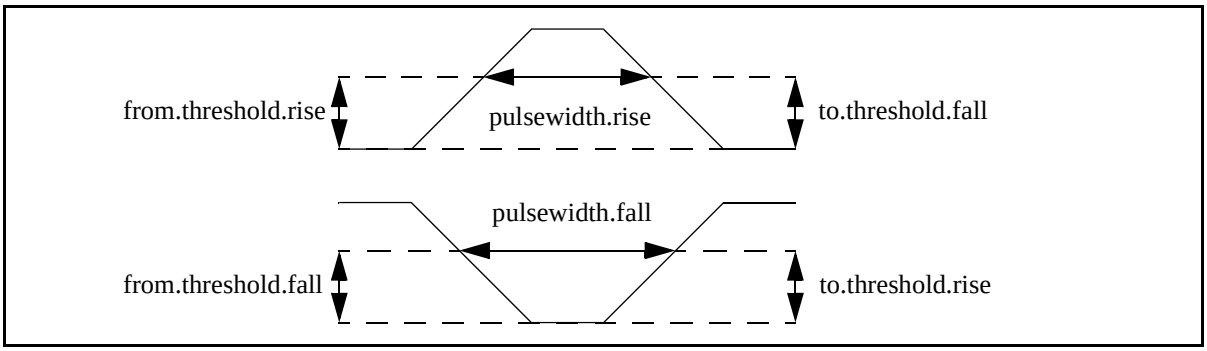

Figure 30—Illustration of PULSEWIDTH

10.11.10 PERIOD

The arithmetic model *period* shall be defined as shown in Semantics 121.

```
KEYWORD PERIOD = arithmetic_model ;
SEMANTICS PERIOD {
  CONTEXT { VECTOR VECTOR.LIMIT VECTOR..HEADER }
  SI_MODEL = TIME ;
}
PERIOD { MIN = 0; }
```

Semantics 121—Arithmetic model PERIOD

The purpose of the arithmetic model *period* is to specify a primitive time interval between periodical repetitions of events.

The arithmetic model *period* shall be in the context of a declared *vector* (see 8.14) with an associated *vector expression* (see 9.12). The *vector expression* shall specify an event pattern within the primitive time interval (see Figure 31).

The *header arithmetic model* (see Syntax 89) *period* shall represent a *dimension* of another arithmetic model, which shall be in the context of a *vector.* Period is illustrated in Figure 31.

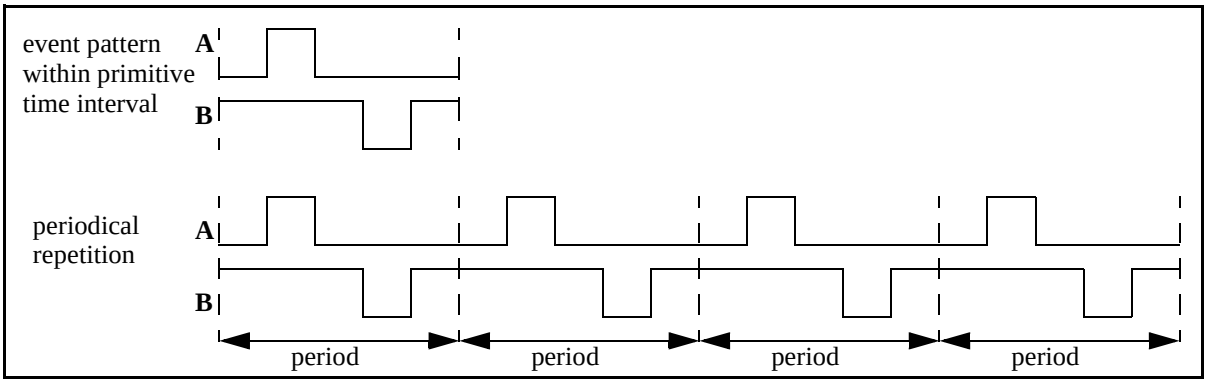

Figure 31—Illustration of PERIOD

An event pattern involving two signals **A** and **B** is repeated periodically.

10.11.11 JITTER

The arithmetic model *jitter* shall be defined as shown in Semantics 122.

```
KEYWORD JITTER = arithmetic_model ;
SEMANTICS JITTER {
  CONTEXT { VECTOR VECTOR.LIMIT VECTOR..HEADER }
  SI_MODEL = TIME ;
}
JITTER { MIN = 0; }
```

Semantics 122—Arithmetic model JITTER

The purpose of the arithmetic model *jitter* is to specify the variability of a primitive time interval between periodical repetitions of an event pattern. The *measurement* annotation (see 10.13.7) shall be applicable as *model qualifier.*

The arithmetic model *jitter* shall be in the context of a declared *vector* (see 8.14) with an associated *vector expression* (see 9.12). The *vector expression* shall specify an event pattern within the primitive time interval (see Figure 32).

A *header arithmetic model* (see Syntax 89) *jitter* shall represent a *dimension* of another arithmetic model, which shall be in the context of a *vector.*

Jitter is illustrated in Figure 32.

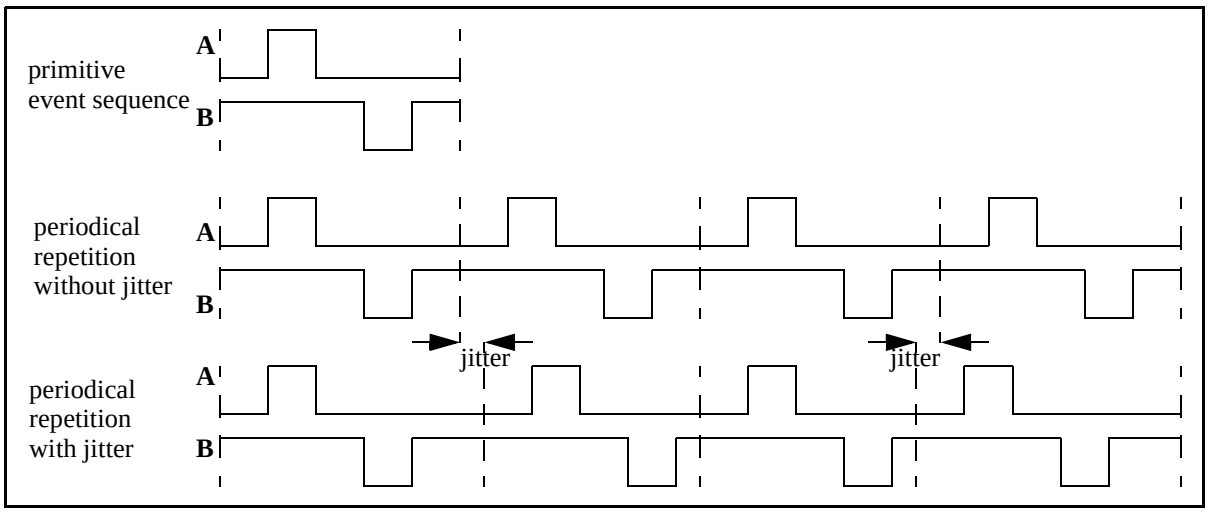

Figure 32—Illustration of JITTER

An event pattern involving two signals **A** and **B** is repeated periodically. A timing diagram with and without jitter is shown.

10.11.12 SKEW

The arithmetic model *skew* shall be defined as shown in Semantics 123.

```
KEYWORD SKEW = arithmetic_model ;
SEMANTICS SKEW {
  CONTEXT { VECTOR VECTOR.LIMIT VECTOR..HEADER }
  SI_MODEL = TIME ;
}
SKEW { MIN = 0; }
```

Semantics 123—Arithmetic model SKEW

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 8.14) with an associated *vector expression* (see 9.12), a *pin reference* annotation, possibly in conjunction with a matching *edge number* annotation, shall be used (see 10.13.5) to refer to multiple *single events* (see 9.13.1). 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 occurring events* (see 9.13.2), 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 occurrence.

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 Figure 33.

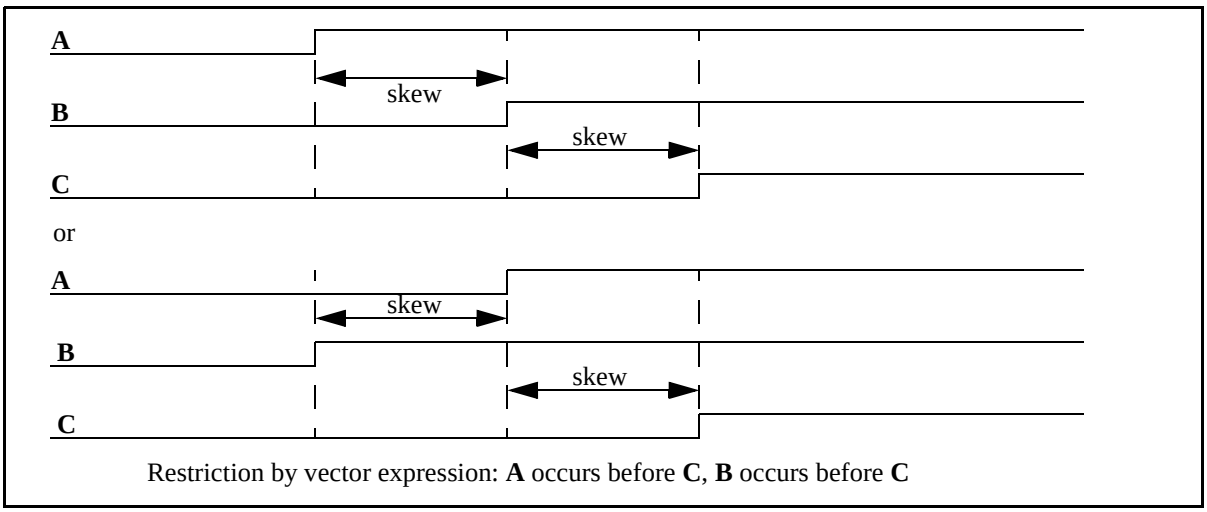

Figure 33—Illustration of SKEW

The arithmetic model skew involves three signals **A**, **B** and **C**, and the vector expression restricts **A** and **B** to occur before **C**.

10.11.13 THRESHOLD

The arithmetic model *threshold* shall be defined as shown in Semantics 124.

```
KEYWORD THRESHOLD = arithmetic_model ;
SEMANTICS THRESHOLD {
  CONTEXT { PIN FROM TO }
  VALUETYPE = number ;
}
THRESHOLD { MIN = 0; MAX = 1; }
```

Semantics 124—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:

threshold.rise = $(vt_r - v_0) / (v_1 - v_0)$
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 34.

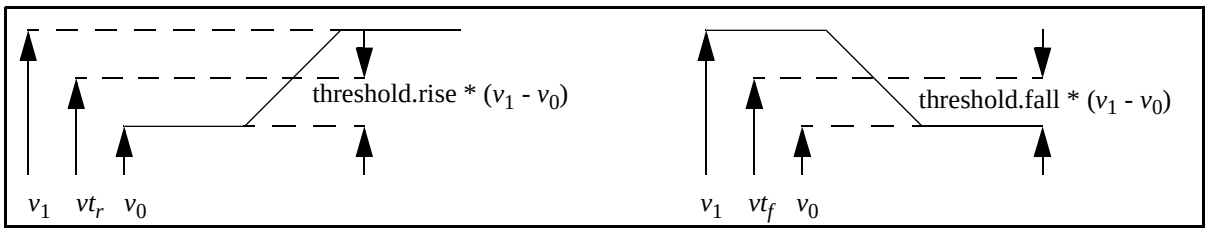

Figure 34—THRESHOLD measurement definition

The arithmetic model *threshold* can contain the arithmetic submodels *rise* and *fall* (see 10.21). If a timing-related arithmetic model referring to a *single event* (see 9.13.1) in the context of a declared *vector* (see 8.14) 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 10.12) or in the context of a declared *pin* (see 8.6). 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 contains a *pin reference* annotation (see 10.13.2) referring to the declared pin.

NOTE — Threshold in the context of a declared pin does not apply to *slewrate* (see 10.11.5) or *pulsewidth* (see 10.11.9), since a from-to statement in the context of slewrate or pulsewidth can not contain a pin reference annotation.

10.11.14 NOISE and NOISE_MARGIN

The arithmetic models *noise* and *noise margin* shall be defined as shown in Semantics 125.

```
KEYWORD NOISE = arithmetic_model ;
SEMANTICS NOISE {
  CONTEXT {
    LIBRARY.LIMIT SUBLIBRARY.LIMIT CELL.LIMIT
    PIN PIN.LIMIT VECTOR VECTOR.LIMIT VECTOR..HEADER
  }
  VALUETYPE = number ;
}
KEYWORD NOISE_MARGIN = arithmetic_model ;
SEMANTICS NOISE_MARGIN {
  CONTEXT { CLASS LIBRARY SUBLIBRARY CELL PIN VECTOR }
  VALUETYPE = number ;
}
NOISE_MARGIN { MIN = 0; }
```

Semantics 125—Arithmetic models NOISE and NOISE_MARGIN

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:

$$\text{noise.low} = (vn - v_0) / (v_1 - v_0)$$
$$\text{noise.high} = (v_1 - vn) / (v_1 - v_0)$$

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,
vn is a measured voltage level due to noise.

NOTE — Noise on a signal with the logic value zero is positive if $vn > v_0$, and negative if $vn < v_0$.
Noise on a signal with the logic value one is positive if $vn < v_1$, and negative if $vn > v_1$.

Noise is illustrated in Figure 34.

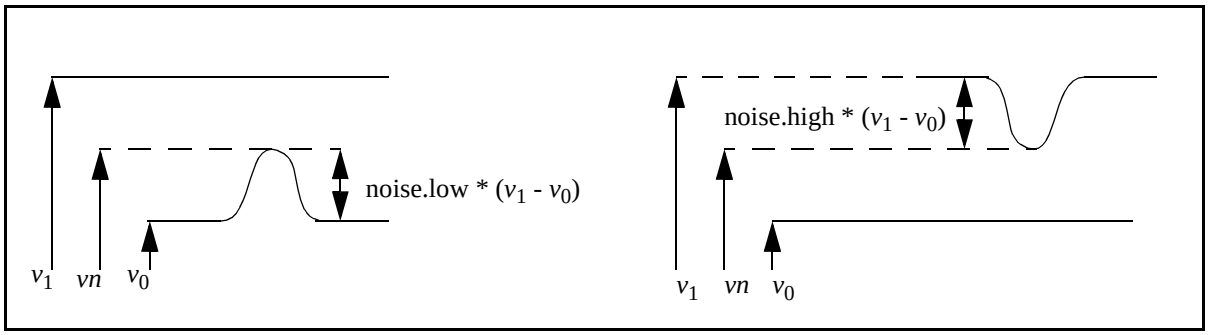

Figure 35—NOISE measurement definition

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 36.

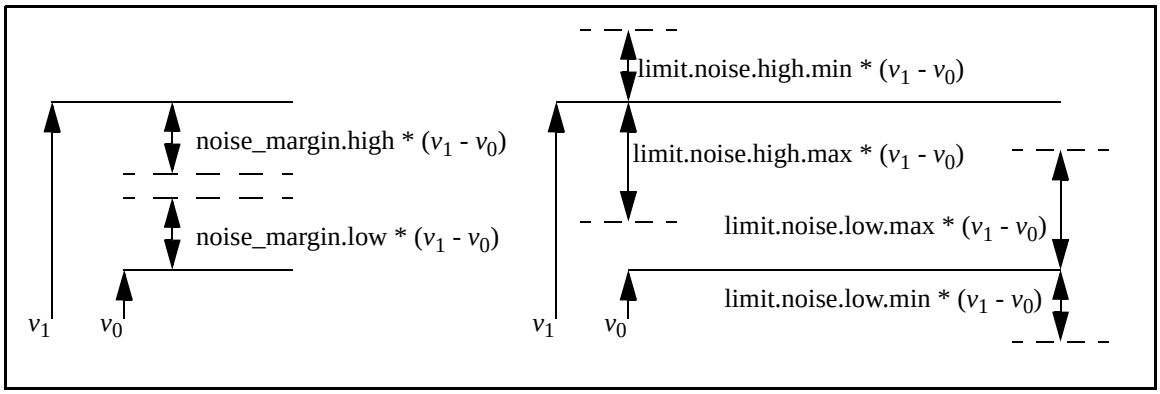

Figure 36—Definition of NOISE MARGIN and LIMIT for NOISE

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 8.2) or a declared *cell* (see 8.4)

The arithmetic model container *limit* (see 10.8.2) can be used to specify a design limit for noise. An arithmetic submodel *high*, *low* (see 10.21) 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 8.6)

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 8.14)

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 9.9) 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 9.12) 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)

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 7.12)

A static noise margin can be specified. An arithmetic submodel *high*, *low* can optionally be used. A declared *pin* can inherit this specification by referring to the class.

— NOISE MARGIN in context of a declared *library* or *sublibrary* (see 8.2) or a declared *cell* (see 8.4) or a declared *pin* (see 8.6).

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 8.14)

A noise margin in the context of a stimulus given by the vector can be described. A *pin reference* annotation (see 10.13.6) shall be used.

A state-dependent noise margin can be described using a *boolean expression* (see 9.9) as stimulus.

A sensitivity window for a noise margin can be described using a *vector expression* (see 9.12) as stimulus. The arithmetic model time (see 10.11.1) shall be used as an *auxiliary arithmetic model* (see 10.6). A *from-to* statement (see 10.12) 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 10.11.9) or *slewrate* (see 10.11.5), as a *header arithmetic model* (see Syntax 89).

10.11.15 POWER and ENERGY

The arithmetic models *power* and *energy* shall be defined as shown in Semantics 126.

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 7.12)

```
KEYWORD POWER = arithmetic_model ;
SEMANTICS POWER {
  CONTEXT {
    LIBRARY SUBLIBRARY CELL VECTOR
    CLASS.LIMIT CELL.LIMIT
  }
  VALUETYPE = number;
}
POWER { UNIT = MilliWatt; }
KEYWORD ENERGY = arithmetic_model ;
SEMANTICS ENERGY {
  CONTEXT { LIBRARY SUBLIBRARY CELL VECTOR }
  VALUETYPE = number;
}
ENERGY { UNIT = PicoJoule; }
```

Semantics 126—Arithmetic models POWER and ENERGY

The arithmetic model container *limit* (see 10.8.2) can be used to specify a design limit for power consumption associated with a *class* with *usage* annotation value *supply-class* (see 8.8.16). A *measurement* annotation (see 10.13.7) shall be used.

— POWER in context of a declared *library* or *sublibrary* (see 8.2)

A *partial arithmetic model* (see Syntax 84) can be used to globally specify an *inheritable arithmetic model qualifier* (see Syntax 87) for power.

— POWER in context of a declared *cell* (see 8.4)

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 8.14)

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 8.2) or a declared *cell* (see 8.4)

A *partial arithmetic model* (see Syntax 84) can be used to globally specify an *inheritable arithmetic model qualifier* (see Syntax 87) for energy.

— ENERGY in context of a declared *vector* (see 8.14)

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.

10.12 FROM and TO statements

A *from-to* statement shall be defined as shown in Syntax 101.

```
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 101—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 10.13.2), an *edge number* annotation (see 10.13.1) and a *threshold* arithmetic model (see 10.11.13).

A reference to a *single event* (see 9.13.1) 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 127.

```
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 127—Restriction for FROM and TO statements

10.13 Annotations related to timing, power and signal integrity

10.13.1 EDGE_NUMBER annotation

An *edge number* annotation shall be defined as shown in Semantics 128.

The edge number annotation shall be a child of an *arithmetic model* (see 10.3) or a *from-to* statement (see 10.12).

```
KEYWORD EDGE_NUMBER = annotation {
  CONTEXT { arithmetic_model FROM TO }
}
SEMANTICS EDGE_NUMBER
  CONTEXT { VECTOR.. }
  VALUETYPE = unsigned_integer ;
  DEFAULT = 0;
}
```

Semantics 128—EDGE_NUMBER annotation

The purpose of the edge number annotation is to specify a reference to a *single event* (see 9.13.1) 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 8.8.1). The pin reference annotation shall point to a *pin variable* (see 9.3) 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.

10.13.2 PIN reference and EDGE_NUMBER annotation for FROM and TO

A *pin reference* annotation shall be subjected to the restriction shown in Semantics 129.

```
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 129—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 10.12).

An *edge_number* annotation shall be subjected to the restriction shown in Semantics 130.

```
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 130—Restriction for EDGE_NUMBER annotation within FROM and TO

The purpose of the restriction is to define a reference to a *single event* (see 9.13.1) in the context of a *from-to* statement.

Example:

```
TIME { FROM { PIN=A; EDGE_NUMBER=1; } TO { PIN=B; EDGE_NUMBER=3; } }
```

Figure 37 illustrates the restriction using a timing diagram.

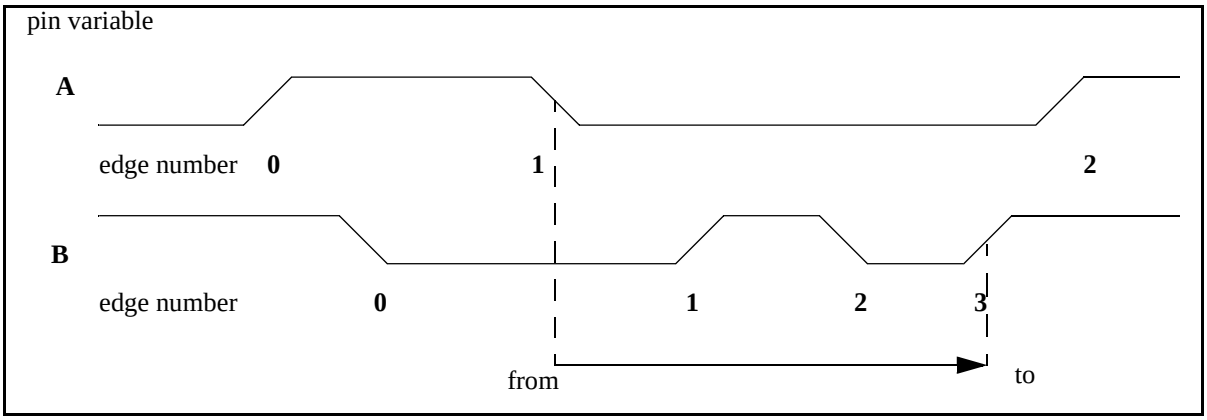

Figure 37—Illustration of PIN reference and EDGE NUMBER annotation within FROM and TO

A measurement is taken from edge number **1** at pin variable **A** to edge number **3** at pin variable **B**.

10.13.3 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 131.

```
SEMANTICS SLEWRATE.PIN = single_value_annotation ;
SEMANTICS SLEWRATE.EDGE_NUMBER = single_value_annotation ;
```

Semantics 131—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 10.11.5) is measured.

10.13.4 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 132.

```
SEMANTICS PULSEWIDTH.PIN = single_value_annotation;
SEMANTICS PULSEWIDTH.EDGE_NUMBER = single_value_annotation;
```

Semantics 132—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 10.11.9) is measured. The trailing edge shall be the following single event on the same pin.

10.13.5 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 133.

```
SEMANTICS SKEW.PIN = multi_value_annotation ;
SEMANTICS SKEW.EDGE_NUMBER = multi_value_annotation ;
```

Semantics 133—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 10.11.12) 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.

10.13.6 PIN reference annotation for NOISE and NOISE_MARGIN

A *pin reference* annotation shall be subjected to the restriction shown in Semantics 134.

```
SEMANTICS NOISE.PIN = single_value_annotation ;
SEMANTICS NOISE_MARGIN.PIN = single_value_annotation ;
```

Semantics 134—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 10.11.14) is described.

10.13.7 MEASUREMENT annotation

A *measurement* annotation shall be defined as shown in Semantics 135.

```
KEYWORD MEASUREMENT = single_value_annotation {
  CONTEXT = arithmetic_model ;
}
SEMANTICS MEASUREMENT {
  CONTEXT {  ENERGY POWER CURRENT VOLTAGE JITTER }
  VALUETYPE = identifier ;
  VALUES {
    transient static average absolute_average rms peak
  }
}
```

Semantics 135—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 101.

Table 101—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 10.11.1) or *frequency* (see 10.11.2) shall be used as *auxiliary arithmetic model* (see 10.6), 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 101. 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 8.14) and the *measurement* annotation value is *peak*. Either a *from* or a *to* statement (see 10.12) can be used to specify the time interval between a *single event* (see 9.13.1) and the occurrence of the measurement or vice-versa.

This is illustrated in Figure 38.

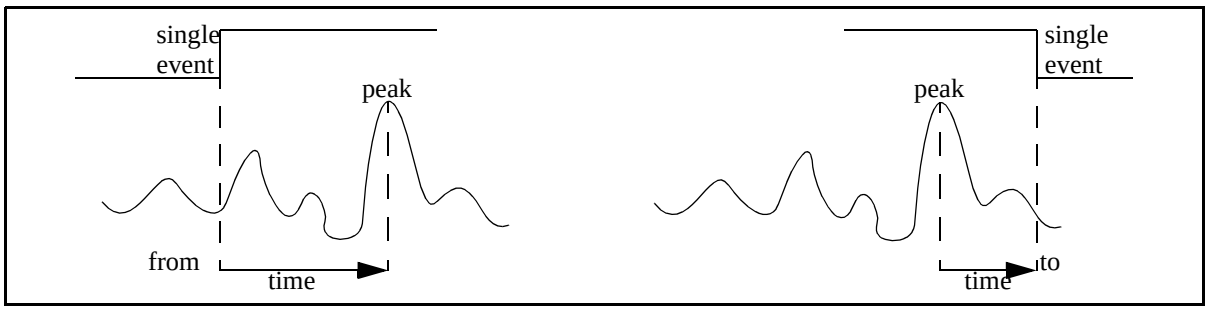

Figure 38—Illustration of peak measurement with FROM or TO statement

10.14 Arithmetic models for environmental conditions

10.14.1 PROCESS

The arithmetic model *process* shall be defined as shown in Semantics 136.

```
KEYWORD PROCESS = arithmetic_model ;
SEMANTICS PROCESS {
  CONTEXT {
    CLASS LIBRARY SUBLIBRARY CELL WIRE HEADER
    arithmetic_model
  }
  VALUETYPE = identifier ;
}
PROCESS { DEFAULT = nom; TABLE { nom snsp snwp wnsp wnwp } }
```

Semantics 136—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 84), a *header arithmetic model* (see Syntax 89), or an *auxiliary arithmetic model* (see 10.6) can be used.

The meaning of the predefined arithmetic values for *process* is explained in Table 102.

Table 102—Predefined arithmetic values for PROCESS

Value	Description
nom	NMOS and PMOS transistors with nominal strength
snsp	Strong NMOS transistor, strong PMOS transistor.
snwp	Strong NMOS transistor, weak PMOS transistor.
wnsp	Weak NMOS transistor, strong PMOS transistor.
wnwp	Weak NMOS transistor, weak PMOS transistor.

10.14.2 DERATE_CASE

The arithmetic model *derate case* shall be defined as shown in Semantics 137.

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 84 and Syntax 85), a *header arithmetic model* (see Syntax 89), or an *auxiliary arithmetic model* (see 10.6) can be used.

```
KEYWORD DERATE_CASE = arithmetic_model ;
SEMANTICS DERATE_CASE {
  CONTEXT {
    CLASS LIBRARY SUBLIBRARY CELL WIRE HEADER
    arithmetic_model
  }
  VALUETYPE = identifier ;
}
DERATE_CASE { DEFAULT = nom;
  TABLE { nom bccom wccom bcind wcind bcmil wcmil }
}
```

Semantics 137—Arithmetic model DERATE_CASE

The meaning of the predefined arithmetic values for *derate case* is explained in Table 103.

Table 103—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).

10.14.3 TEMPERATURE

The arithmetic model *temperature* shall be defined as shown in Semantics 138.

```
KEYWORD TEMPERATURE = arithmetic_model ;
SEMANTICS TEMPERATURE {
  CONTEXT {
    CLASS LIBRARY SUBLIBRARY CELL WIRE
    LIMIT HEADER arithmetic_model
  }
  VALUETYPE = number ;
}
TEMPERATURE { UNIT = 1DegreeCelsius; MIN = -273; }
```

Semantics 138—Arithmetic model TEMPERATURE

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 84 and Syntax 85), a *header arithmetic model* (see Syntax 89), or an *auxiliary arithmetic model* (see 10.6) can be used.

10.15 Arithmetic models for electrical circuits

10.15.1 VOLTAGE

The arithmetic model *voltage* shall be defined as shown in Semantics 139.

```
KEYWORD VOLTAGE = arithmetic_model ;
SEMANTICS VOLTAGE {
  CONTEXT {
    CLASS LIBRARY SUBLIBRARY CELL PIN WIRE VECTOR HEADER
    CLASS.LIMIT CELL.LIMIT PIN.LIMIT VECTOR.LIMIT
  }
  VALUETYPE = number ;
}
VOLTAGE { UNIT = 1Volt; }
```

Semantics 139—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 7.12)

An environmental voltage can be specified. An arithmetic submodel *high*, *low* (see 10.21) can optionally be used. A *pin* (see 8.6) can inherit this specification by referring to the class. In particular, a *supply class* annotation (see 8.8.16) or a *connect class* annotation (see 8.8.19) can be used for this purpose.

— VOLTAGE in context of a declared *library* or *sublibrary* (see 8.2)

A *partial arithmetic model* (see Syntax 84) can be used to globally specify an *inheritable arithmetic model qualifier* (see Syntax 87) or a *trivial min-max* statement (see Syntax 94) for voltage.

— VOLTAGE in context of a declared *cell* (see 8.4)

A voltage source that is part of the implementation of a cell can be specified. A *node reference* annotation (see 10.16.1) shall be used.

A design limit for a voltage related to the cell can be specified using the arithmetic model container *limit* (see 10.8.2). Either a *pin reference* annotation (see 10.16.3) or a *model reference* annotation (see 10.9.5) 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 8.6)

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 8.10)

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 8.14)

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 10.8.2). Either a *pin reference* annotation or a *model reference* annotation shall be used.

— VOLTAGE as *header arithmetic model* (see Syntax 89)

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 9.15) is present, a reference to a voltage source specified within the declared wire can be established.

10.15.2 CURRENT

The arithmetic model *current* shall be defined as shown in Semantics 140.

```
KEYWORD CURRENT = arithmetic_model ;
SEMANTICS CURRENT {
  CONTEXT {
    LIBRARY SUBLIBRARY CELL WIRE VECTOR HEADER
    CELL.LIMIT VECTOR.LIMIT
    LAYER.LIMIT VIA.LIMIT RULE.LIMIT
  }
  VALUETYPE = number ;
}
CURRENT { UNIT = MilliAmpere; }
```

Semantics 140—Arithmetic model CURRENT

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.

— CURRENT in context of a declared *library* or *sublibrary* (see 8.2)

A *partial arithmetic model* (see Syntax 84) can be used to globally specify an *inheritable arithmetic model qualifier* (see Syntax 87) for current.

— CURRENT in context of a declared *cell* (see 8.4)

A current source that is part of the implementation of a cell can be specified. A *node reference* annotation (see 10.16.1) shall be used.

A design limit for a current related to the cell can be specified using the arithmetic model container *limit* (see 10.8.2). Either a *pin reference* annotation (see 10.16.3) or a *model reference* annotation (see 10.9.5) or a *component reference* annotation (see 10.16.2) 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 8.10)

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 8.16), a declared *via* (see 8.18), or a declared *rule* (see 8.20)

A design limit for current can be specified using the arithmetic model container *limit*. A *measurement* annotation (see 10.13.7) 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 10.20.9) shall be used.

— CURRENT in context of a declared *vector* (see 8.14)

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)

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 9.15) is present, a reference to a current source or to a component specified within the declared wire can be established.

10.15.3 CAPACITANCE

The arithmetic model *capacitance* shall be defined as shown in Semantics 141.

```
KEYWORD CAPACITANCE = arithmetic_model ;
SEMANTICS CAPACITANCE {
  CONTEXT {
    LIBRARY SUBLIBRARY CELL CELL.LIMIT PIN PIN.LIMIT
    WIRE LAYER RULE VECTOR HEADER
  }
  VALUETYPE = number ;
  SI_MODEL = CAPACITANCE ;
}
CAPACITANCE { UNIT = PicoFarad; MIN = 0; }
```

Semantics 141—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 8.2)

A *partial arithmetic model* (see Syntax 84) can be used to globally specify an *inheritable arithmetic model qualifier* (see Syntax 87) for capacitance.

— CAPACITANCE in context of a declared *cell* (see 8.4)

A capacitor that is part of the implementation of a cell can be described. A *node reference* annotation (see 10.16.1) shall be used.

A design limit for a capacitor related to the cell can be specified using the arithmetic model container *limit* (see 10.8.2). Either a *pin reference* annotation (see 10.16.3) or a *model reference* annotation (see 10.9.5) 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 8.6)

The self-capacitance of a pin can be described as a child of a *pin*. An arithmetic submodel *rise, fall, high, low* (see 10.21) 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 8.10)

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 8.16)

An estimation model for capacitance of a general layout segment can be described. An arithmetic submodel *horizontal, vertical, acute, obtuse* (see 10.22) can optionally be used.

— CAPACITANCE in context of a declared *rule* (see 8.20)

An estimation model for capacitance created by a particular layout pattern can be described.

— CAPACITANCE in context of a declared *vector* (see 8.14)

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)

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 9.15) is present, a reference to a capacitor described within the declared *wire* can be established.

10.15.4 RESISTANCE

The arithmetic model *resistance* shall be defined as shown in Semantics 142.

```
KEYWORD RESISTANCE = arithmetic_model ;
SEMANTICS RESISTANCE {
  CONTEXT {
    LIBRARY SUBLIBRARY CELL WIRE LAYER RULE
    CELL.LIMIT VECTOR HEADER
  }
  VALUETYPE = number ;
  SI_MODEL = RESISTANCE ;
}
RESISTANCE { UNIT = KiloOhm; MIN = 0; }
```

Semantics 142—Arithmetic model RESISTANCE

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 8.2)

A *partial arithmetic model* (see Syntax 84) can be used to globally specify an *inheritable arithmetic model qualifier* (see Syntax 87) for resistance.

— RESISTANCE in context of a declared *cell* (see 8.4)

A resistor that is part of the implementation of a cell can be described. A *node reference* annotation (see 10.16.1) shall be used.

A design limit for a resistor related to the cell can be specified using the arithmetic model container *limit* (see 10.8.2). A *model reference* annotation (see 10.9.5) 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 8.10)

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 8.16)

An estimation model for resistance of a general layout segment can be described. An arithmetic submodel *horizontal*, *vertical*, *acute*, *obtuse* (see 10.22) can optionally be used.

— RESISTANCE in context of a declared *rule* (see 8.20)

An estimation model for resistance created by a particular layout pattern can be described.

— RESISTANCE in context of a declared *vector* (see 8.14)

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)

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 9.15) is present, a reference to a resistor described within the declared *wire* can be established.

10.15.5 INDUCTANCE

The arithmetic model *inductance* shall be defined as shown in Semantics 143.

```
KEYWORD INDUCTANCE = arithmetic_model ;
SEMANTICS INDUCTANCE {
  CONTEXT {
    LIBRARY SUBLIBRARY CELL WIRE LAYER RULE
    CELL.LIMIT VECTOR HEADER
  }
  VALUETYPE = number ;
  SI_MODEL = INDUCTANCE ;
}
INDUCTANCE { UNIT = 1e-6; MIN = 0; }
```

Semantics 143—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 transformer (i.e., a component with mutual inductance).

— INDUCTANCE in context of a declared *library* or *sublibrary* (see 8.2)

A *partial arithmetic model* (see Syntax 84) can be used to globally specify an *inheritable arithmetic model qualifier* (see Syntax 87) for inductance.

— INDUCTANCE in context of a declared *cell* (see 8.4)

An inductor or a transformer that is part of the implementation of a cell can be described. A *node reference* annotation (see 10.16.1) shall be used.

A design limit for an inductor or for a transformer related to the cell can be specified using the arithmetic model container *limit* (see 10.8.2). A *pin reference* annotation (see 10.16.3) or a *model reference* annotation (see 10.9.5) 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 8.10)

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-inductance 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 *layer* (see 8.16)

An estimation model for self-inductance of a general layout segment can be described. An arithmetic submodel *horizontal*, *vertical*, *acute*, *obtuse* (see 10.22) can optionally be used.

— INDUCTANCE in context of a declared *rule* (see 8.20)

An estimation model for inductance created by a particular layout pattern can be described.

— INDUCTANCE in context of a declared *vector* (see 8.14)

An *equivalent inductance* can be described. A *model reference* annotation shall be used. The equivalent inductance 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 circuit.

— INDUCTANCE as *header arithmetic model* (see Syntax 89)

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 9.15) is present, a reference to a self-inductance or to a mutual inductance described within the declared *wire* can be established.

10.16 Annotations for electrical circuits

10.16.1 NODE reference annotation for electrical circuits

The *node reference* annotation (see 8.13.1) shall be subjected to restrictions defined in Semantics 144.

The purpose of a node reference annotation with these restrictions is to specify the connectivity of an electrical component within an electrical circuit.

```
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 144—Restrictions for NODE reference annotation

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 across which an induced voltage is observed. The last two values shall represent the terminals across which a controlling current flows.

The electrical components and their terminals are illustrated in Figure 39.

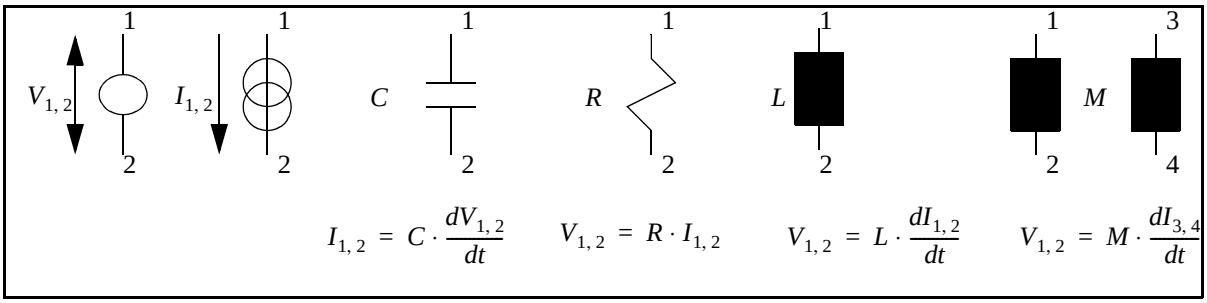

Figure 39—Electrical components and their terminals

The numbers in Figure 39 indicate the first, second, third and fourth node annotation values. However, the node annotation values shall be the ALF names of declared nodes.

10.16.2 COMPONENT reference annotation

A *component* reference annotation shall be defined as shown in Semantics 145.

```
KEYWORD COMPONENT = single_value_annotation {
  CONTEXT = arithmetic_model ;
}
SEMANTICS COMPONENT {
  CONTEXT { CURRENT POWER ENERGY }
  REFERENCETYPE {
    CURRENT VOLTAGE CAPACITANCE RESISTANCE INDUCTANCE
  }
}
```

Semantics 145—COMPONENT annotation

The purpose of the component reference annotation is to relate the arithmetic model *current* (see 10.15.2), *power* or *energy* (see 10.11.15) 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 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.

10.16.3 PIN reference annotation for electrical circuits

The *pin reference* annotation (see 8.8.1) shall be subjected to restrictions defined in Semantics 146.

```
SEMANTICS VOLTAGE.PIN = single_value_annotation {
  CONTEXT { VECTOR VECTOR.LIMIT VECTOR..HEADER } }
SEMANTICS CURRENT.PIN = single_value_annotation {
  CONTEXT { VECTOR VECTOR.LIMIT VECTOR..HEADER } }
SEMANTICS CAPACITANCE.PIN = single_value_annotation {
  CONTEXT { VECTOR VECTOR..HEADER } }
SEMANTICS RESISTANCE.PIN = single_value_annotation {
  CONTEXT { VECTOR } }
```

Semantics 146—PIN reference annotation

The purpose of a *pin reference* annotation for electrical circuits is to specify an association between an electrical component with two terminals and a *pin variable*, i.e., a declared *pin*, *port* or *node* (see 9.3).

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 10.16.4).

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* shall 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 8.13.2), a pin with *direction* annotation value *input* (see 8.8.5), a port, or a node connected to such a pin shall be considered 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 40.

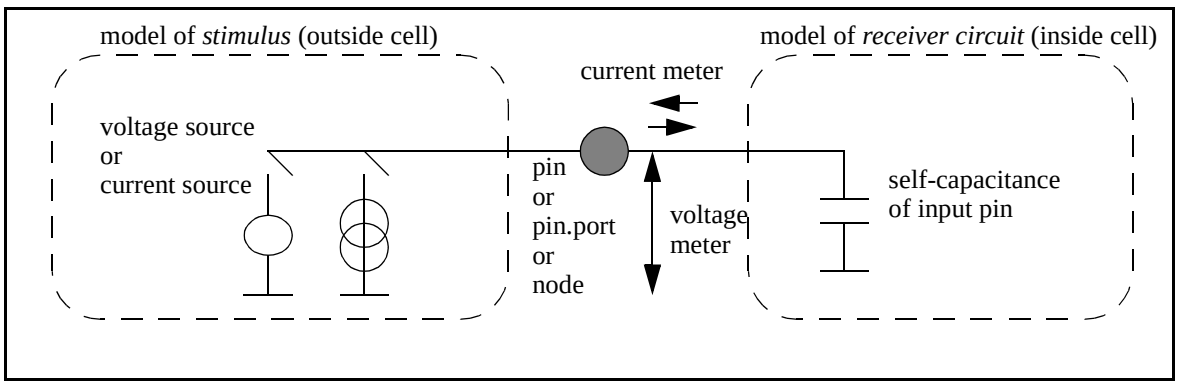

Figure 40—Association between electrical components and an input pin

A node with *nodetype* annotation value *driver* (see 8.13.2), a pin with direction annotation value *output* (see 8.8.5), a port, or a node connected to such a pin shall be considered 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 41.

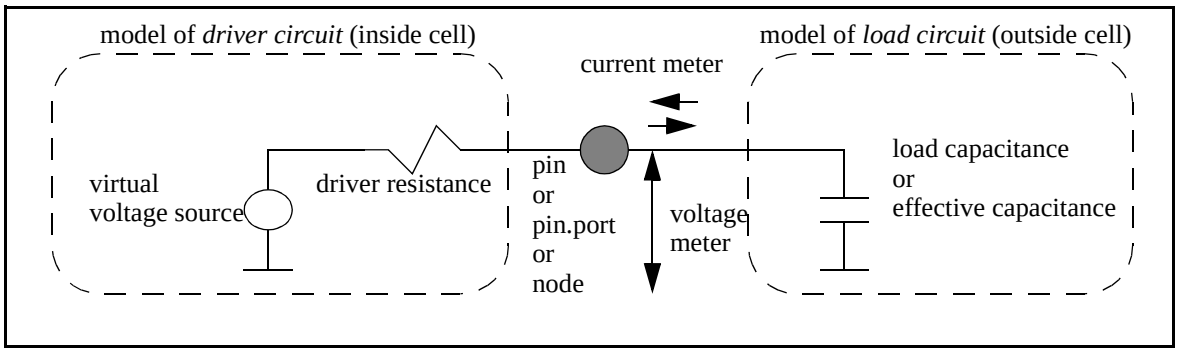

Figure 41—Association between electrical components and an output pin

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 10.15.

10.16.4 FLOW annotation

A *flow* annotation shall be defined as shown in Semantics 147.

```
KEYWORD FLOW = single_value_annotation {
  CONTEXT = arithmetic_model ;
}
SEMANTICS FLOW {
  CONTEXT = CURRENT ;
  VALUES { in out }
  DEFAULT = in;
}
```

Semantics 147—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 10.16.3).

The meaning of the annotation values is shown in Table 104.

Table 104—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 10.16.1) or a *component reference* annotation (see 10.16.2), since the direction of current measurement is already defined by the order of terminals of the electrical component.

10.17 Miscellaneous arithmetic models

10.17.1 DRIVE STRENGTH

The arithmetic model *drive strength* shall be defined as shown in Semantics 148.

```
KEYWORD DRIVE_STRENGTH = arithmetic_model ;
SEMANTICS DRIVE_STRENGTH {
  CONTEXT { CLASS LIBRARY SUBLIBRARY CELL PIN PINGROUP }
  VALUETYPE = unsigned_number ;
}
DRIVE_STRENGTH { MIN = 0; }
```

Semantics 148—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 8.4) shall be considered either a *primitive circuit* or a *compound circuit*, depending on its *celltype* annotation (see 8.5.2). 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 8.6) or a *pingroup* (see 8.7).

A cell with *celltype* annotation value *buffer, combinational, multiplexor, flip-flop*, 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 84) in the context of a *class* (see 7.12), a *library* or a *sublibrary* (see 8.2) 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) or a trivial *min-max* statement (see Syntax 94), respectively.

10.17.2 SWITCHING_BITS with PIN reference annotation

The arithmetic model *switching bits* shall be defined as shown in Semantics 149.

```
KEYWORD SWITCHING_BITS = arithmetic_model ;
SEMANTICS SWITCHING_BITS {
  CONTEXT { VECTOR.POWER.HEADER VECTOR.ENERGY.HEADER }
  VALUETYPE = unsigned_integer ;
}
SEMANTICS SWITCHING_BITS.PIN = single_value_annotation;
```

Semantics 149—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 9.13.1) on a vectorized *pin* (see 8.6) or a *pingroup* (see 8.7).

Drive strength can be used as *header arithmetic model* (see Syntax 89) for calculation of *power* or *energy* (see 10.11.15) in context of a *vector* (see 8.14).

The *pin reference* annotation (see 8.8.1) shall be used.

10.18 Arithmetic models related to structural implementation

10.18.1 CONNECTIVITY

The arithmetic model *connectivity* shall be defined as shown in Semantics 150.

```
KEYWORD CONNECTIVITY = arithmetic_model ;
SEMANTICS CONNECTIVITY {
  CONTEXT { LIBRARY SUBLIBRARY CELL RULE ANTENNA HEADER }
  VALUES { 1 0 ? }
}
```

Semantics 150—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) or a *between* annotation (see 10.20.2) shall be used to establish a relation between physical objects and the arithmetic model *connectivity*. The interpretation of *connectivity* as a requirement for a connection shall be specified by the *connect-rule* annotation (see 10.20.1).

The arithmetic model connectivity shall evaluate to a *bit literal* (see 6.8). The interpretation of the bit literal is specified in Table 105.

Table 105—Interpretation of bit literals for CONNECTIVITY

Bit literal	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 specified.	Requirement is not specified.

NOTE — The bit literal "?" is defined as a *non-assignable* boolean value (see 9.10.3) and can therefore only be used, if the connectivity is modeled as a *table* (see Syntax 91).

10.18.2 DRIVER and RECEIVER

The arithmetic models *driver* and *receiver* shall be defined as shown in Semantics 151.

```
KEYWORD DRIVER = arithmetic_model ;
SEMANTICS DRIVER {
  CONTEXT = CONNECTIVITY.HEADER;
  REFERENCETYPE = CLASS ;
}
KEYWORD RECEIVER = arithmetic_model ;
SEMANTICS RECEIVER {
  CONTEXT = CONNECTIVITY.HEADER;
  REFERENCETYPE = CLASS ;
}
```

Semantics 151—Arithmetic models DRIVER and RECEIVER

The purpose of the *header arithmetic model* (see Syntax 89) *driver* or *receiver* is to specify a dependency between *connectivity* (see 10.18.1) and a declared *class* (see 7.12) with *usage* annotation value *connect-class* (see 7.13.2 and 8.8.19).

The header arithmetic model driver or receiver shall contain a *table* statement (see Syntax 91). The parent arithmetic model *connectivity* shall contain either a one-dimensional lookup table involving either dimension driver or receiver, or alternatively a two-dimensional lookup table involving both dimensions driver and receiver.

A declared *pin* (see 8.6) 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 8.8.5). 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.

10.18.3 FANOUT, FANIN and CONNECTIONS

The arithmetic model *fanout* shall be defined as shown in Semantics 152.

```
KEYWORD FANOUT = arithmetic_model ;
SEMANTICS FANOUT {
  CONTEXT {
    PIN.LIMIT WIRE.SIZE.HEADER WIRE.CAPACITANCE.HEADER
    WIRE.RESISTANCE.HEADER WIRE.INDUCTANCE.HEADER
  }
  VALUETYPE = unsigned_integer ;
}
```

Semantics 152—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 Semantics 153.

The purpose of the arithmetic model *fanin* is to specify the total number of output pins connected to a net.

```
KEYWORD FANIN = arithmetic_model ;
SEMANTICS FANIN {
  CONTEXT {
    PIN.LIMIT WIRE.SIZE.HEADER WIRE.CAPACITANCE.HEADER
    WIRE.RESISTANCE.HEADER WIRE.INDUCTANCE.HEADER
  }
  VALUETYPE = unsigned_integer ;
}
```

Semantics 153—Arithmetic model FANIN

The arithmetic model *connections* shall be defined as shown in Semantics 154.

```
KEYWORD CONNECTIONS = arithmetic_model ;
SEMANTICS CONNECTIONS {
  CONTEXT {
    PIN.LIMIT WIRE.SIZE.HEADER WIRE.CAPACITANCE.HEADER
    WIRE.RESISTANCE.HEADER WIRE.INDUCTANCE.HEADER
  }
  VALUETYPE = unsigned_integer ;
}
```

Semantics 154—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 8.8.5).

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 10.8.2) in the context of a *pin* (see 8.6)

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 to its *direction* annotation value.

— FANOUT, FANIN, or CONNECTIONS as *header arithmetic model* (see Syntax 89) in the context of a *wire* (see 8.10)

The arithmetic value of *size* (see 10.19.1), *capacitance* (see 10.15.3), *resistance* (see 10.15.4), or *inductance* (see 10.15.5) can be calculated.

10.19 Arithmetic models related to layout implementation

10.19.1 SIZE

The arithmetic model *size* shall be defined as shown in Semantics 155.

```
KEYWORD SIZE = arithmetic_model ;
SEMANTICS SIZE {
  CONTEXT {
    CELL ANTENNA ANTENNA.LIMIT PIN WIRE
    WIRE.CAPACITANCE.HEADER
    WIRE.RESISTANCE.HEADER
    WIRE.INDUCTANCE.HEADER
  }
  VALUETYPE = number ;
}
SIZE { MIN = 0; }
```

Semantics 155—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 8.4) or a *wire* (see 8.10)

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 Syntax 89) in context of a *wire* (see 8.10)

The arithmetic value of *capacitance* (see 10.15.3), *resistance* (see 10.15.4), or *inductance* (see 10.15.5) 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 8.21)

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 10.8.2). 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 8.6)

Size shall represent a measure for the additive magnitude of an *antenna* (see 8.21), 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 10.20.7) and a *target* annotation (see 10.20.8) shall be used.

10.19.2 AREA

The arithmetic model *area* shall be defined as shown in Semantics 156.

The purpose of the arithmetic model *area* is to define a physical area, according to the International System of Measurements and Units [reference needed].

```
KEYWORD AREA = arithmetic_model ;
SEMANTICS AREA {
  CONTEXT {
    CELL WIRE WIRE..HEADER LAYER..HEADER
    RULE..HEADER ANTENNA..HEADER
  }
  VALUETYPE = unsigned_number ;
  SI_MODEL = AREA ;
}
AREA { UNIT = 1e-12; MIN = 0; }
```

Semantics 156—Arithmetic model AREA

— AREA as arithmetic model in the context of a *cell* (see 8.4) or a *wire* (see 8.10)

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 Syntax 89) in context of a *wire* (see 8.10)

The arithmetic value of *capacitance* (see 10.15.3), *resistance* (see 10.15.4), or *inductance* (see 10.15.5) 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 Syntax 89) in context of a *layer* (see 8.16)

The arithmetic value of *capacitance* (see 10.15.3) or *resistance* (see 10.15.4) can be calculated. A design limit for *current* (see 10.15.2) 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 Syntax 89) in context of a *rule* (see 8.20)

The arithmetic value of *capacitance* (see 10.15.3), *resistance* (see 10.15.4), or *inductance* (see 10.15.5) can be calculated. A design limit for *current* (see 10.15.2), *distance* (see 10.19.9), *overhang* (see 10.19.10), *width* (see 10.19.7), *length* (see 10.19.8), or *extension* (see 10.19.4) can be calculated. The dimension *area* shall represent the physical area occupied by a *pattern* or by a *region*. A *pattern reference* annotation (see 10.20.9) or a *region reference* annotation (see 8.32.1) shall be used.

— AREA as *header arithmetic model* (see Syntax 89) in context of an *antenna* (see 8.21)

The arithmetic value of *size* (see 10.19.1) 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 8.16). A *layer reference* annotation (see 8.17.1) shall be used.

10.19.3 PERIMETER

The arithmetic model *perimeter* shall be defined as shown in Semantics 157.

```
KEYWORD PERIMETER = arithmetic_model ;
SEMANTICS PERIMETER {
  CONTEXT {
    CELL WIRE WIRE..HEADER LAYER..HEADER
    RULE..HEADER ANTENNA..HEADER
  }
  SI_MODEL = DISTANCE ;
}
```

Semantics 157—Arithmetic model PERIMETER

The purpose of the arithmetic model *perimeter* is to define the *distance* (see 10.19.9) measured when surrounding the boundaries of a physical object.

— PERIMETER as arithmetic model in the context of a *cell* (see 8.4) or a *wire* (see 8.10)

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 Syntax 89) in context of a *wire* (see 8.10)

The arithmetic value of *capacitance* (see 10.15.3), *resistance* (see 10.15.4), or *inductance* (see 10.15.5) 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 Syntax 89) in context of a *layer* (see 8.16)

The arithmetic value of *capacitance* (see 10.15.3) or *resistance* (see 10.15.4) can be calculated. A design limit for *current* (see 10.15.2) can be calculated. The dimension *perimeter* shall represent the perimeter surrounding a layout segment residing on the layer.

— PERIMETER as *header arithmetic model* (see Syntax 89) in context of a *rule* (see 8.20)

The arithmetic value of *capacitance* (see 10.15.3), *resistance* (see 10.15.4), or *inductance* (see 10.15.5) can be calculated. A design limit for *current* (see 10.15.2), *distance* (see 10.19.9), *overhang* (see 10.19.10), *width* (see 10.19.7), *length* (see 10.19.8), or *extension* (see 10.19.4) can be calculated. The dimension *perimeter* shall represent the perimeter surrounding a *pattern* or by a *region*. A *pattern reference* annotation (see 10.20.9) or a *region reference* annotation (see 8.32.1) shall be used.

— PERIMETER as *header arithmetic model* (see Syntax 89) in context of an *antenna* (see 8.21)

The arithmetic value of *size* (see 10.19.1) 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 8.16). A *layer reference* annotation (see 8.17.1) shall be used.

10.19.4 EXTENSION

The arithmetic model *extension* shall be defined as shown in Semantics 158.

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 94). 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 intervals of 45 degrees, the expansion shall result in a hexagon.

```
KEYWORD EXTENSION = arithmetic_model ;
SEMANTICS EXTENSION {
  CONTEXT { LAYER PATTERN RULE.LIMIT RULE..HEADER }
  SI_MODEL = DISTANCE ;
}
```

Semantics 158—Arithmetic model EXTENSION

This is illustrated in Figure 42.

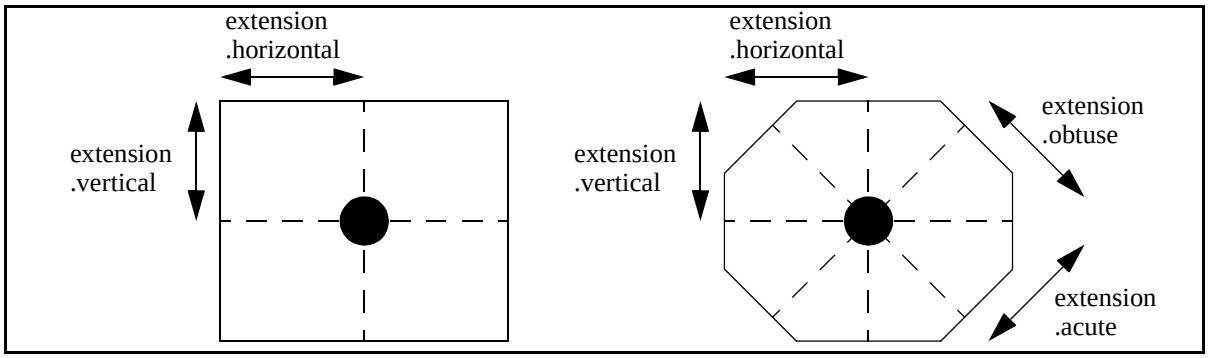

Figure 42—Illustration of EXTENSION

The arithmetic submodels *horizontal*, *vertical*, *acute* and *obtuse* (see 10.22) can be used to specify anisotropic expansion.

— EXTENSION as arithmetic model in the context of a *layer* (see 8.16)

Extension shall represent the expansion of an endpoint of a routing segment residing on a *layer* (see 8.16) with *layertype* annotation value *routing* (see 8.17.2).

— EXTENSION as arithmetic model in the context of a *pattern* (see 8.29)

Extension shall represent the expansion of a *pattern* (see 8.29) with an associated *shape* annotation or with an associated *geometric model* (see 9.16). Each reference point shall be subject to expansion.

— EXTENSION as *limit arithmetic model* (see 10.8.2) in the context of a *rule* (see 8.20)

Extension shall represent a design limit for expansion of a *pattern*. Each reference point shall be subject to expansion. A *pattern reference* annotation (see 10.20.9) shall be used.

— EXTENSION as *header arithmetic model* (see Syntax 89) in context of a *rule* (see 8.20)

The arithmetic value of *capacitance* (see 10.15.3), *resistance* (see 10.15.4), or *inductance* (see 10.15.5) can be calculated. A design limit for *current* (see 10.15.2), *distance* (see 10.19.9), *overhang* (see 10.19.10), *width* (see 10.19.7), *length* (see 10.19.8), or *extension* (see 10.19.4) can be calculated. The dimension *extension* shall represent the expansion of a *pattern* with *shape* annotation value *tee*, *cross*, *corner* or *end* (see 8.30.2). A *pattern reference* annotation (see 10.20.9) or a *model reference* annotation (see 10.9.5) 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*.

10.19.5 THICKNESS

The arithmetic model *thickness* shall be defined as shown in Semantics 159.

```
KEYWORD THICKNESS = arithmetic_model ;
SEMANTICS EXTENSION {
  CONTEXT { LAYER RULE..HEADER }
  SI_MODEL = DISTANCE ;
}
```

Semantics 159—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 8.16).

Thickness as *header arithmetic model* (see Syntax 89) can be used to calculate an arithmetic value of *capacitance* (see 10.15.3), *resistance* (see 10.15.4), or *inductance* (see 10.15.5) in the context of a *rule* (see 8.20).

10.19.6 HEIGHT

The arithmetic model *height* shall be defined as shown in Semantics 160.

```
KEYWORD HEIGHT = arithmetic_model ;
SEMANTICS HEIGHT {
  CONTEXT { CELL SITE REGION LAYER WIRE..HEADER }
  SI_MODEL = DISTANCE ;
}
```

Semantics 160—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 8.16)

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 8.4), *site* (see 8.25) or *region* (see 8.31)

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 Syntax 89) in context of a *wire* (see 8.10)

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.

10.19.7 WIDTH

The arithmetic model *width* shall be defined as shown in Semantics 161.

```
KEYWORD WIDTH = arithmetic_model ;
SEMANTICS WIDTH {
  CONTEXT {
    CELL SITE REGION LAYER LAYER.LIMIT
    PATTERN RULE.LIMIT RULE..HEADER
  }
  SI_MODEL = DISTANCE ;
}
```

Semantics 161—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 8.4), *site* (see 8.25) or *region* (see 8.31)

Width shall represent a distance in *x* direction measured between the left and the right border of a rectangular *cell*, *site* or *region.*

— WIDTH as *header arithmetic model* (see Syntax 89) in context of a *wire* (see 8.10)

Width shall represent the distance in *x* direction measured between the left and the right border 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 10.8.2) in the context of a *layer* (see 8.16)

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 8.17.2). 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 8.29)

Width shall represent the distance between the borders of a *pattern* (see 8.29) with an associated *shape* annotation value *line* or *jog* (see 8.30.2) or with an associated *geometric model* of type *polyline* or *ring* (see 9.16). 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 10.8.2) in the context of a *rule* (see 8.20)

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 10.20.9) shall be used.

— WIDTH as *header arithmetic model* (see Syntax 89) in the context of a *rule* (see 8.20)

The arithmetic value of *capacitance* (see 10.15.3), *resistance* (see 10.15.4), or *inductance* (see 10.15.5) can be calculated. A design limit for *current* (see 10.15.2), *distance* (see 10.19.9), *overhang* (see 10.19.10), *width* (see 10.19.7), *length* (see 10.19.8), or *extension* (see 10.19.4) can be calculated. The dimension *width* shall represent the distance between the borders of a *pattern* with *shape* annotation value *line* or *end* (see 8.30.2). A *pattern reference* annotation (see 10.20.9) or a *model reference* annotation (see 10.9.5) 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*.

10.19.8 LENGTH

The arithmetic model *length* shall be defined as shown in Semantics 162.

```
KEYWORD LENGTH = arithmetic_model ;
SEMANTICS LENGTH {
  CONTEXT {
    LAYER LAYER.LIMIT PATTERN RULE.LIMIT RULE..HEADER
  }
  SI_MODEL = DISTANCE ;
}
```

Semantics 162—Arithmetic model LENGTH

— LENGTH as arithmetic model or *limit arithmetic model* (see 10.8.2) in the context of a *layer* (see 8.16)

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 8.17.2). Length shall be measured parallel to the routing direction.

— LENGTH as arithmetic model in the context of a *pattern* (see 8.29)

Length shall represent the distance between the end points of a *pattern* (see 10.20.9) with an associated *shape* annotation value *line* or *jog* (see 8.30.2).

— LENGTH as *limit arithmetic model* (see 10.8.2) in the context of a *rule* (see 8.20)

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 10.20.9) shall be used.

— LENGTH as *header arithmetic model* (see Syntax 89) in the context of a *rule* (see 8.20)

The arithmetic value of *capacitance* (see 10.15.3), *resistance* (see 10.15.4), or *inductance* (see 10.15.5) can be calculated. A design limit for *current* (see 10.15.2), *distance* (see 10.19.9), *overhang* (see 10.19.10), *width* (see 10.19.7), or *extension* (see 10.19.4) can be calculated. The dimension *length* shall represent the distance between the end points of a *pattern* with *shape* annotation value *line* or *end* (see 8.30.2). A *pattern reference* annotation (see 10.20.9), a *model reference* annotation (see 10.9.5) or a *between* annotation (see 10.20.4) 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*. A *between* annotation shall refer to two patterns representing two parallel routing segments

10.19.9 DISTANCE

The arithmetic model *distance* shall be defined as shown in Semantics 163.

The purpose of the arithmetic model *distance* is to define a space in-between two objects, according to the International System of Units (see U.S. National Bureau of Standards, Spec. Pub. 330).

— DISTANCE as arithmetic model or as *limit arithmetic model* (see 10.8.2) in the context of a *rule* (see 8.20)

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 10.20.4) shall be used.

```
KEYWORD DISTANCE = arithmetic_model ;
SEMANTICS DISTANCE {
  CONTEXT { RULE RULE.LIMIT RULE..HEADER }
  VALUETYPE = number ;
  SI_MODEL = DISTANCE ;
}
DISTANCE { UNIT = 10e-6; MIN = 0; }
```

Semantics 163—Arithmetic model DISTANCE

The arithmetic submodels *horizontal*, *vertical*, *acute* and *obtuse* (see 10.22) can be used.

— DISTANCE as *header arithmetic model* (see Syntax 89) in the context of a *rule* (see 8.20)

The arithmetic value of *capacitance* (see 10.15.3), *resistance* (see 10.15.4), or *inductance* (see 10.15.5) can be calculated. A design limit for *current* (see 10.15.2), *length* (see 10.19.8), *overhang* (see 10.19.10), *width* (see 10.19.7), or *extension* (see 10.19.4) can be calculated. The dimension *distance* shall represent the measured distance between two patterns. A *between reference* annotation (see 10.20.4) or *model reference* annotation (see 10.9.5) 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.

10.19.10 OVERHANG

The arithmetic model *overhang* shall be defined as shown in Semantics 164.

```
KEYWORD OVERHANG = arithmetic_model ;
SEMANTICS OVERHANG {
  CONTEXT { RULE RULE.LIMIT RULE..HEADER }
  SI_MODEL = DISTANCE ;
}
```

Semantics 164—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 10.8.2) or as *header arithmetic model* (see Syntax 89) in the context of a *rule* (see 8.20), with similar semantic restrictions as *distance* (see 10.19.9).

Overhang can be interpreted as the distance between the nearest parallel edges in the region of overlap between two objects.

NOTE: The use of the arithmetic model *distance* instead of *overhang* would imply that there is no overlap.

This is illustrated in Figure 43.

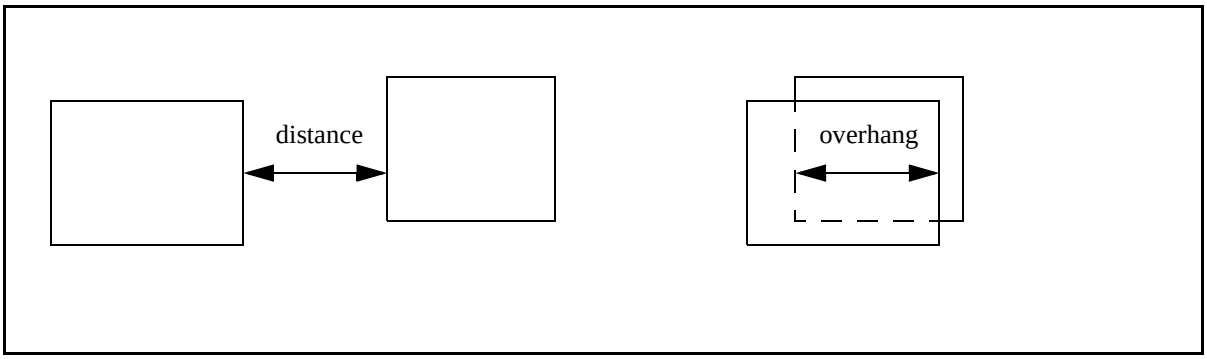

Figure 43—Illustration of DISTANCE versus OVERHANG

10.19.11 DENSITY

The arithmetic model *density* shall be defined as shown in Semantics 165.

```
KEYWORD DENSITY = arithmetic_model ;
SEMANTICS DENSITY {
  CONTEXT { LAYER.LIMIT RULE RULE.LIMIT }
  VALUETYPE = number ;
}
DENSITY { MIN = 0; MAX = 1; }
```

Semantics 165—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 8.16) with *layertype* annotation value *routing* (see 8.17.2), divided by an allocated area wherein the metal segments are found.

— DENSITY as *limit arithmetic model* (see 10.8.2) in the context of a *layer* (see 8.16)

A constant design limit for metal density can be specified.

— DENSITY as arithmetic model or as *limit arithmetic model* (see 10.8.2) in the context of a *rule* (see 8.20)

A design limit or a calculation model for metal density can be specified. A *region reference* annotation (see 8.32.1) can be used to relate the design limit or the calculation model for metal density to a *region* (see 8.31) declared in the context of the same *rule*. A *model reference* annotation (see 10.9.5) can be used to relate a design limit to a related calculation model.

10.20 Annotations related to arithmetic models for layout implementation

10.20.1 CONNECT_RULE annotation

A *connect-rule* annotation shall be defined as shown in Semantics 166.

The purpose of the *connect-rule* annotation is to specify that the arithmetic model *connectivity* (see 10.18.1) is to be interpreted as a requirement for connection rather than an actual connection.

```
KEYWORD CONNECT_RULE = single_value_annotation {
  CONTEXT = arithmetic_model ;
}
SEMANTICS CONNECT_RULE {
  CONTEXT = CONNECTIVITY ;
  VALUES { must_short can_short cannot_short }
}
```

Semantics 166—CONNECT_RULE annotation

The meaning of the annotation values is shown in Table 106.

Table 106—CONNECT_RULE annotation

Annotation value	Description
must_short	Electrical connection required.
can_short	Electrical connection allowed.
cannot_short	Electrical connection disallowed.

Implications between requirements for a connection are shown in Table 107.

Table 107—Implications between CONNECT_RULE specifications

specified rule	must_short			can_short			cannot_short		
implied rule	**1**	**0**	**?**	**1**	**0**	**?**	**1**	**0**	**?**
must_short	**1**	**0**	**?**	?	0	?	0	?	?
can_ short	1	?	?	**1**	**0**	**?**	0	1	?
cannot_short	0	?	?	0	1	?	**1**	**0**	**?**

A set of requirements for a connection that can be inferred by implication according to Table 107 is redundant. A set of requirements contradicting Table 107 shall be a conflict. The application shall be responsible for handling redundant requirements and conflicts.

10.20.2 BETWEEN annotation

A *between* annotation shall be defined as shown in Semantics 167.

The purpose of the *between* annotation is to specify a reference to multiple objects related to an arithmetic model *distance* (see 10.19.9), *length* (see 10.19.8), *overhang* (see 10.19.10), or *connectivity* (see 10.18.1).

10.20.3 BETWEEN annotation for CONNECTIVITY

A *between* annotation shall be subjected to the restriction shown in Semantics 168.

```
KEYWORD BETWEEN = multi_value_annotation {
  CONTEXT = arithmetic_model ;
}
SEMANTICS BETWEEN {
  CONTEXT { DISTANCE LENGTH OVERHANG CONNECTIVITY }
}
```

Semantics 167—BETWEEN annotation

```
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 168—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 10.18.1).

10.20.4 BETWEEN annotation for DISTANCE, LENGTH, OVERHANG

A *between* annotation shall be subjected to the restriction shown in Semantics 169.

```
SEMANTICS DISTANCE.BETWEEN {
  REFERENCETYPE { PATTERN REGION }
}
SEMANTICS LENGTH.BETWEEN {
  REFERENCETYPE { PATTERN REGION }
}
SEMANTICS OVERHANG.BETWEEN {
  REFERENCETYPE { PATTERN REGION }
}
```

Semantics 169—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 10.19.9), *length* (see 10.19.8), or *overhang* (see 10.19.10).

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 Figure 44.

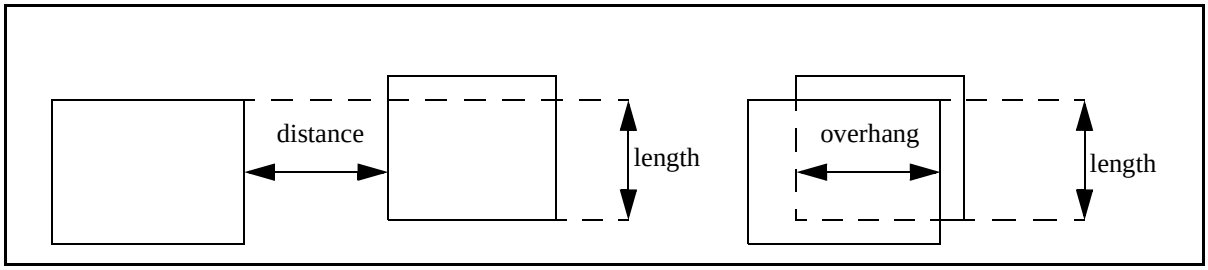

Figure 44—Illustration of DISTANCE versus OVERHANG versus LENGTH

10.20.5 MEASURE annotation

A *measure* annotation shall be defined as shown in Semantics 170.

```
KEYWORD MEASURE = single_value_annotation {
  CONTEXT = arithmetic_model ;
}
SEMANTICS MEASURE {
  CONTEXT { DISTANCE LENGTH OVERHANG }
  VALUETYPE = identifier ;
  VALUES { euclidean horizontal vertical manhattan }
  DEFAULT = euclidean ;
}
```

Semantics 170—DISTANCE_MEASUREMENT annotation

The mathematical description of the annotation values is specified in Table 108.

Table 108—Annotation values for MEASURE

Annotation value	Mathematical description
`euclidean`	*measure* = $\sqrt{x^2 + y^2}$
`manhattan`	*measure* = $x + y$
`horizontal`	*measure* = x

Table 108—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 8.30.2) 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 45.

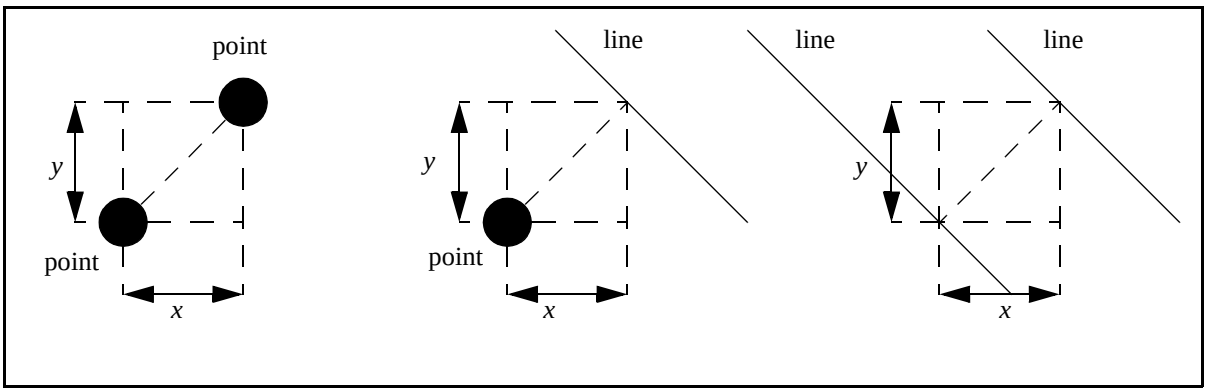

Figure 45—Illustration of MEASURE

Figure 45 shows the distance between two points, between a point and a line, and between two parallel lines.

10.20.6 REFERENCE annotation container

A *reference* annotation container shall be defined as shown in Semantics 171.

```
KEYWORD REFERENCE = annotation_container {
  CONTEXT = arithmetic_model ;
}
SEMANTICS REFERENCE {
  CONTEXT { DISTANCE LENGTH OVERHANG }
  REFERENCETYPE { PATTERN REGION }
}
SEMANTICS REFERENCE.identifier = single_value_annotation {
  VALUETYPE = identifier ;
  VALUES { center origin near_edge far_edge }
  DEFAULT = origin ;
}
```

Semantics 171—REFERENCE annotation container

The purpose of the *reference* annotation container is to specify the reference points for a measurement of *distance* (see 10.19.9).

An annotation within the *reference* annotation container shall associate a *pattern* (see 8.29) or a *region* (see 8.31) with a reference point specified by an annotation value.

The meaning of the annotation values is specified in Table 109.

Table 109—Annotation values for REFERENCE

Annotation value	Description
`origin`	The reference point is the origin of a pattern or a region.
`center`	The reference point is the center of a pattern or a region
`near_edge`	The reference point is the edge of a pattern or a region which is nearest to a parallel edge of another pattern or another region.
`far_edge`	The reference point is the edge of a pattern or a region which is farthest from a parallel edge of another pattern or another 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 9.16) is associated with the pattern or region. A *geometric transformation* (see 9.18) 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 10.19.7) shall be considered either as *near edge* or as *far edge*.
   2) A predefined geometric model *rectangle* (see 9.16) 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 9.16) 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 46.

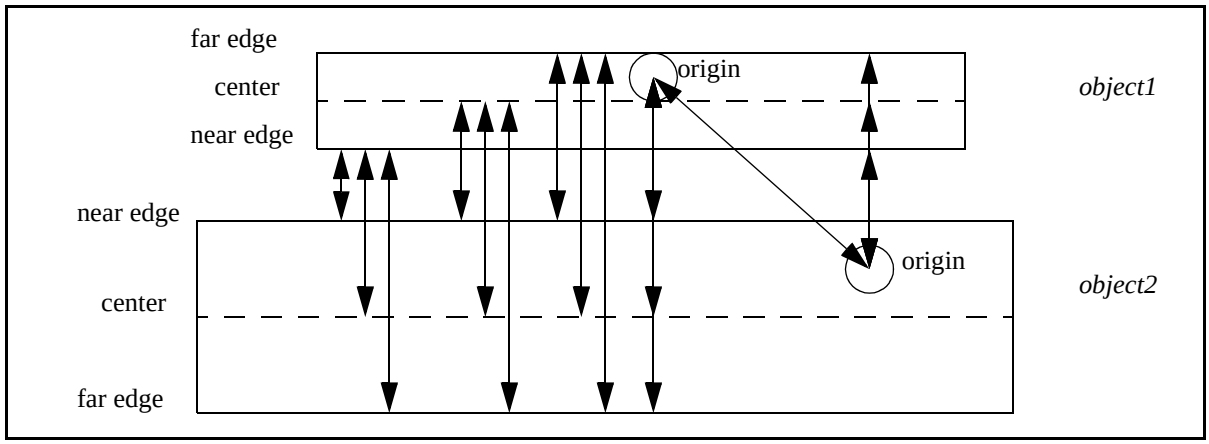

Figure 46—Illustration of REFERENCE for DISTANCE

Figure 46 shows *euclidean* distance between all possible reference points of *object1* and *object2*.

10.20.7 ANTENNA reference annotation

An *antenna* reference annotation shall be defined as shown in Semantics 172.

```
KEYWORD ANTENNA = annotation {
  CONTEXT = arithmetic_model ;
}
SEMANTICS ANTENNA {
  CONTEXT { PIN.SIZE PIN.AREA PIN.PERIMETER }
  REFERENCETYPE = ANTENNA;
}
```

Semantics 172—ANTENNA reference annotation

An *antenna reference* annotation shall be used to relate a calculated *size* (see 10.19.1) or *area* (see 10.19.2) or *perimeter* (see 10.19.3) in the context of the *pin* with a calculation rule for *size* in the context of an *antenna* (see 8.21). A reference to multiple antennas can be made using a *multi-value annotation*.

10.20.8 TARGET annotation

An *target* annotation shall be defined as shown in Semantics 173.

```
KEYWORD TARGET = annotation {
  CONTEXT = arithmetic_model ;
}
SEMANTICS TARGET {
  VALUETYPE = identifier ;
  CONTEXT = PIN.SIZE;
  REFERENCETYPE = PIN.PATTERN;
}
```

Semantics 173—TARGET annotation

The *target* annotation shall be associated with the arithmetic model *size* (see 10.19.1) in the context of a *pin* (see 8.6).

The purpose of the *target* annotation is to specify a *pattern* (see 8.29) in the context of the same *pin* which is the victim of an antenna effect (see 8.21). The referenced pattern shall have a *layer reference* annotation (see 8.17.1) and a *trivial* or a *full arithmetic model* (see Syntax 83 and Syntax 85) for *area* (see 10.19.2) or *perimeter* (see 10.19.3).

An *antenna reference* annotation (see 10.20.7) shall also be associated with the arithmetic model *size*. The referred *antenna* (see 8.21) 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.

10.20.9 PATTERN reference annotation

A *pattern* reference annotation shall be defined as shown in Semantics 174.

```
KEYWORD PATTERN = single_value_annotation {
  CONTEXT = arithmetic_model ;
}
SEMANTICS PATTERN {
  CONTEXT {
    LENGTH WIDTH HEIGHT SIZE AREA THICKNESS
    PERIMETER EXTENSION
  }
  REFERENCETYPE = PATTERN ;
}
```

Semantics 174—PATTERN annotation

The purpose of the *pattern reference* annotation is to relate an arithmetic model or a *header arithmetic model* (see Syntax 89) to a declared *pattern* (see 8.29).

10.21 Arithmetic submodels for timing and electrical data

The arithmetic submodels shown in Table 110 shall be applicable in the context of electrical modeling.

Table 110—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 Semantics 175.

```
KEYWORD HIGH = arithmetic_submodel ;
SEMANTICS HIGH { CONTEXT {
    CLASS.VOLTAGE CLASS.LIMIT.VOLTAGE
    PIN.VOLTAGE PIN.LIMIT.VOLTAGE PIN.CAPACITANCE
    PIN.NOISE PIN.NOISE_MARGIN PIN.LIMIT.NOISE
    LIBRARY.NOISE_MARGIN LIBRARY.LIMIT.NOISE
} }
KEYWORD LOW = arithmetic_submodel ;
SEMANTICS LOW { CONTEXT {
    CLASS.VOLTAGE CLASS.LIMIT.VOLTAGE
    PIN.VOLTAGE PIN.LIMIT.VOLTAGE PIN.CAPACITANCE
    PIN.NOISE PIN.NOISE_MARGIN PIN.LIMIT.NOISE
    LIBRARY.NOISE_MARGIN LIBRARY.LIMIT.NOISE
} }
```

Semantics 175—Arithmetic submodels HIGH and LOW

The arithmetic submodels *rise* and *fall* shall be defined as shown in Semantics 176.

```
KEYWORD RISE = arithmetic_submodel ;
SEMANTICS RISE { CONTEXT {
    FROM.THRESHOLD TO.THRESHOLD PIN.THRESHOLD
    PIN.CAPACITANCE PIN.SLEWRATE PIN.LIMIT.SLEWRATE
    PIN.PULSEWIDTH PIN.LIMIT.PULSEWIDTH
} }
KEYWORD FALL = arithmetic_submodel ;
SEMANTICS FALL { CONTEXT {
    FROM.THRESHOLD TO.THRESHOLD PIN.THRESHOLD
    PIN.CAPACITANCE PIN.SLEWRATE PIN.LIMIT.SLEWRATE
    PIN.PULSEWIDTH PIN.LIMIT.PULSEWIDTH
} }
```

Semantics 176—Arithmetic submodels RISE and FALL

10.22 Arithmetic submodels for physical data

The arithmetic submodels shown in Table 111 shall be applicable in the context of physical modeling.

Table 111—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 Semantics 177.

```
KEYWORD HORIZONTAL = arithmetic_submodel ;
SEMANTICS HORIZONTAL { CONTEXT {
    WIDTH LENGTH EXTENSION DISTANCE OVERHANG
} }
KEYWORD VERTICAL = arithmetic_submodel ;
SEMANTICS VERTICAL { CONTEXT {
    WIDTH LENGTH EXTENSION DISTANCE OVERHANG
} }
KEYWORD ACUTE = arithmetic_submodel ;
SEMANTICS ACUTE { CONTEXT {
    WIDTH LENGTH EXTENSION DISTANCE OVERHANG
} }
KEYWORD OBTUSE = arithmetic_submodel ;
SEMANTICS OBTUSE { CONTEXT {
    WIDTH LENGTH EXTENSION DISTANCE OVERHANG
} }
```

Semantics 177—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 syntax for description of lexical and syntax rules uses the conventions shown in 1.4.

ALF_statement ::= // See Syntax 1 on page 25
ALF_type [[index] ALF_name [index]] [= ALF_value] **;**
| ALF_type [[index] ALF_name [index]] [= ALF_value] **{** { ALF_value | **:** | **;** } **}**
| ALF_type [[index] ALF_name [index]] [= ALF_value] **{** { ALF_statement } **}**

ALF_type ::=
identifier
| **@**
| **:**

ALF_name ::=
identifier
| control_expression

ALF_value ::=
number
| multiplier_prefix_symbol
| identifier
| quoted_string
| bit_literal
| based_literal
| edge_value
| arithmetic_expression
| boolean_expression
| control_expression

ALF_statement_termination ::=
;
| **{** { ALF_value | **:** | **;** } **}**
| **{** { ALF_statement } **}**

character ::= // See Syntax 2 on page 37
whitespace
| letter
| digit
| special

whitespace ::=
space | horizontal_tab | new_line | vertical_tab | form_feed | carriage_return

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 Syntax 3 on page 39
        in_line_comment
      | block_comment
in_line_comment ::=
        / /{character} new_line
      | / /{character} carriage_return
block_comment ::=
        / *{character}* /
delimiter ::=                                                  // See Syntax 4 on page 39
        ( | ) | [ | ] | { | } | : | ; | ,
operator ::=                                                   // See Syntax 5 on page 40
        arithmetic_operator
      | boolean_operator
      | relational_operator
      | shift_operator
      | event_operator
      | meta_operator
arithmetic_operator ::=
        + | - | * | / | % | **
boolean_operator ::=
        && | || | ~& | ~| | ^ | ~^ | ~ | ! | & | |
relational_operator ::=
        == | != | >= | <= | > | <
shift_operator ::=
        << | >>
event_operator ::=
        -> | ~> | <-> | <~> | &> | <&>
meta_operator ::=
        = | ? | @
number ::=                                                     // See Syntax 6 on page 43
        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 ::=
        mantissa [ exponent ]
      | unsigned_integer exponent
```

```
sign ::=
        + | -
mantissa ::=
         . unsigned_integer
        | unsigned_integer . [ unsigned_integer ]
exponent ::=
         E [ sign ] unsigned_integer
        | e [ sign ] unsigned_integer
index_value ::=                                  // See Syntax 7 on page 43
        unsigned_integer | atomic_identifier
index ::=                                        // See Syntax 8 on page 44
        single_index | multi_index
single_index ::=
        [ index_value ]
multi_index ::=
        [ index_value : index_value ]
multiplier_prefix_symbol ::=                     // See Syntax 9 on page 44
         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
multiplier_prefix_value ::=                      // See Syntax 10 on page 45
        unsigned_number | multiplier_prefix_symbol
bit_literal ::=                                  // See Syntax 11 on page 45
         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 Syntax 12 on page 46
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**

boolean_value ::= // See Syntax 13 on page 46
alphanumeric_bit_literal | based_literal | integer

arithmetic_value ::= // See Syntax 14 on page 47
number | identifier | bit_literal | based_literal

edge_literal ::= // See Syntax 15 on page 47
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 ::=
?~ | **?!** | **?-**

edge_value ::= // See Syntax 16 on page 47
(edge_literal **)**

identifier ::= // See Syntax 17 on page 47
atomic_identifier | indexed_identifier | hierarchical_identifier | escaped_identifier

atomic_identifier ::=
non_escaped_identifier | placeholder_identifier

hierarchical_identifier ::=
full_hierarchical_identifier | partial_hierarchical_identifier

non_escaped_identifier ::= // See Syntax 18 on page 48
letter { letter | digit | **_** | **$** | **#** }

placeholder_identifier ::= // See Syntax 19 on page 48
< non_escaped_identifier >

indexed_identifier ::= // See Syntax 20 on page 48
atomic_identifier index

full_hierarchical_identifier ::= // See Syntax 21 on page 49
atomic_identifier [index] . atomic_identifier [index] { . atomic_identifier [index] }

partial_hierarchical_identifier ::= // See Syntax 22 on page 49
atomic_identifier [index] { . atomic_identifier [index] } . .
{ atomic_identifier [index] { . atomic_identifier [index] } . . }
[atomic_identifier [index] { . atomic_identifier [index] }]

escaped_identifier ::= // See Syntax 23 on page 49
\ escapable_character { escapable_character }

escapable_character ::=
letter | digit | special

*keyword*_identifier ::= // See Syntax 24 on page 50
letter { [_] letter }

quoted_string ::= // See Syntax 25 on page 50
" { character } "

string_value ::= // See Syntax 26 on page 51
quoted_string | identifier

generic_value ::= // See Syntax 27 on page 51
number
| multiplier_prefix_symbol
| identifier
| quoted_string
| bit_literal
| based_literal
| edge_value

vector_expression_macro ::= // See Syntax 28 on page 52
. non_escaped_identifier

generic_object ::= // See Syntax 29 on page 53
alias_declaration
| constant_declaration
| class_declaration
| keyword_declaration
| semantics_declaration
| group_declaration
| template_declaration

all_purpose_item ::= // See Syntax 30 on page 53
generic_object
| include_statement
| associate_statement
| annotation
| annotation_container
| arithmetic_model
| arithmetic_model_container
| *all_purpose_item*_template_instantiation

annotation ::= // See Syntax 31 on page 54
single_value_annotation
| multi_value_annotation

single_value_annotation ::=
*annotation*_identifier = annotation_value ;

multi_value_annotation ::=
*annotation*_identifier { annotation_value { annotation_value } }

```
annotation_value ::=
        generic_value
      | control_expression
      | boolean_expression
      | arithmetic_expression
annotation_container ::=                                   // See Syntax 32 on page 54
      annotation_container_identifier { annotation { annotation } }
attribute ::=                                              // See Syntax 33 on page 54
      ATTRIBUTE { identifier { identifier } }
property ::=                                               // See Syntax 34 on page 55
      PROPERTY [ identifier ] { annotation { annotation } }
alias_declaration ::=                                      // See Syntax 35 on page 55
        ALIAS alias_identifier = original_identifier ;
      | ALIAS vector_expression_macro = ( vector_expression ) ;
constant_declaration ::=                                   // See Syntax 36 on page 56
      CONSTANT constant_identifier = constant_value ;
constant_value ::=
      number | based_literal
keyword_declaration ::=                                    // See Syntax 37 on page 56
        KEYWORD keyword_identifier = syntax_item_identifier ;
      | KEYWORD keyword_identifier = syntax_item_identifier { { CONTEXT_annotation } }
semantics_declaration ::=                                  // See Syntax 38 on page 57
        SEMANTICS semantics_identifier = syntax_item_identifier ;
      | SEMANTICS semantics_identifier [ = syntax_item_identifier ] { { semantics_item } }
semantics_item ::=
        CONTEXT_annotation
      | VALUETYPE_single_value_annotation
      | VALUES_multi_value_annotation
      | REFERENCETYPE_annotation
      | DEFAULT_single_value_annotation
      | SI_MODEL_single_value_annotation
class_declaration ::=                                      // See Syntax 39 on page 65
        CLASS class_identifier ;
      | CLASS class_identifier { { class_item } }
class_item ::=
        all_purpose_item
      | geometric_model
      | geometric_transformation
group_declaration ::=                                      // See Syntax 40 on page 67
        GROUP group_identifier { generic_value { generic_value } }
      | GROUP group_identifier { left_index_value : right_index_value }
template_declaration ::=                                   // See Syntax 41 on page 68
      TEMPLATE template_identifier { ALF_statement { ALF_statement } }
template_instantiation ::=                                 // See Syntax 42 on page 69
        static_template_instantiation
      | dynamic_template_instantiation
static_template_instantiation ::=
        template_identifier [ = static ] ;
      | template_identifier [ = static ] { { generic_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 ;
include ::=                                          // See Syntax 43 on page 72
      INCLUDE quoted_string ;
associate ::=                                        // See Syntax 44 on page 72
       ASSOCIATE quoted_string ;
      | ASSOCIATE quoted_string { FORMAT_single_value_annotation }
revision ::=                                         // See Syntax 45 on page 73
      ALF_REVISION string_value
library_specific_object ::=                          // See Syntax 46 on page 75
       library
      | sublibrary
      | cell
      | primitive
      | wire
      | pin
      | pingroup
      | vector
      | node
      | layer
      | via
      | rule
      | antenna
      | site
      | array
      | blockage
      | port
      | pattern
      | region
library ::=                                          // See Syntax 47 on page 76
       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 Syntax 48 on page 78
          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 Syntax 49 on page 88
        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 Syntax 50 on page 89
        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 Syntax 51 on page 110
    PRIMITIVE *primitive*_identifier **{** { primitive_item } **}**
  | **PRIMITIVE** *primitive*_identifier **;**
  | *primitive*_template_instantiation

primitive_item ::=
    all_purpose_item
  | pin
  | pingroup
  | function
  | test

wire ::= // See Syntax 52 on page 110
    WIRE *wire*_identifier **{** { wire_item } **}**
  | **WIRE** *wire*_identifier **;**
  | *wire*_template_instantiation

wire_item ::=
    all_purpose_item
  | node

node ::= // See Syntax 53 on page 112
    NODE *node*_identifier **;**
  | **NODE** *node*_identifier **{** { node_item } **}**
  | *node*_template_instantiation

node_item ::=
  all_purpose_item

vector ::= // See Syntax 54 on page 115
    VECTOR control_expression **;**
  | **VECTOR** control_expression **{** { vector_item } **}**
  | *vector*_template_instantiation

vector_item ::=
    all_purpose_item
  | wire_instantiation

layer ::= // See Syntax 55 on page 121
    LAYER *layer*_identifier **;**
  | **LAYER** *layer*_identifier **{** { layer_item } **}**
  | *layer*_template_instantiation

layer_item ::=
  all_purpose_item

via ::= // See Syntax 56 on page 123
    VIA *via*_identifier **;**
  | **VIA** *via*_identifier **{** { via_item } **}**
  | *via*_template_instantiation

via_item ::=
    all_purpose_item
  | pattern
  | artwork

rule ::= // See Syntax 57 on page 124
    RULE *rule*_identifier **;**
  | **RULE** *rule*_identifier **{** { rule_item } **}**
  | *rule*_template_instantiation

```
rule_item ::=
        all_purpose_item
      | pattern
      | region
      | via_instantiation
antenna ::=                                          // See Syntax 58 on page 125
        ANTENNA antenna_identifier ;
      | ANTENNA antenna_identifier { { antenna_item } }
      | antenna_template_instantiation
antenna_item ::=
        all_purpose_item
      | region
blockage ::=                                         // See Syntax 59 on page 126
        BLOCKAGE blockage_identifier ;
      | BLOCKAGE blockage_identifier { { blockage_item } }
      | blockage_template_instantiation
blockage_item ::=
        all_purpose_item
      | pattern
      | region
      | rule
      | via_instantiation
port ::=                                             // See Syntax 60 on page 126
        PORT port_identifier ;{ { port_item } }
      | PORT port_identifier ;
      | port_template_instantiation
port_item ::=
        all_purpose_item
      | pattern
      | region
      | rule
      | via_instantiation
site ::=                                             // See Syntax 61 on page 127
        SITE site_identifier ;
      | SITE site_identifier { { site_item } }
      | site_template_instantiation
site_item ::=
        all_purpose_item
      | WIDTH_arithmetic_model
      | HEIGHT_arithmetic_model
array ::=                                            // See Syntax 62 on page 129
        ARRAY array_identifier ;
      | ARRAY array_identifier { { array_item } }
      | array_template_instantiation
array_item ::=
        all_purpose_item
      | geometric_transformation
pattern ::=                                          // See Syntax 63 on page 131
        PATTERN pattern_identifier ;
      | PATTERN pattern_identifier { { pattern_item } }
      | pattern_template_instantiation
```

```
pattern_item ::=
        all_purpose_item
      | geometric_model
      | geometric_transformation
region ::=                                          // See Syntax 64 on page 135
        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
function ::=                                        // See Syntax 65 on page 137
        FUNCTION { function_item { function_item } }
      | function_template_instantiation
function_item ::=
        all_purpose_item
      | behavior
      | structure
      | statetable
test ::=                                            // See Syntax 66 on page 137
        TEST { test_item { test_item } }
      | test_template_instantiation
test_item ::=
        all_purpose_item
      | behavior
      | statetable
pin_variable ::=                                    // See Syntax 67 on page 138
        pin_variable_identifier
pin_value ::=
        pin_variable | boolean_value
pin_assignment ::=                                  // See Syntax 68 on page 138
        pin_variable = pin_value ;
behavior ::=                                        // See Syntax 69 on page 140
        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 } }
primitive_instantiation ::=
        primitive_identifier [ identifier ] { pin_value { pin_value } }
      | primitive_identifier [ identifier ] { boolean_assignment { boolean_assignment } }
```

structure ::= // See Syntax 70 on page 141
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 Syntax 71 on page 142
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 Syntax 72 on page 142
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 Syntax 73 on page 143
RANGE { index_value **:** index_value **}**
boolean_expression ::= // See Syntax 74 on page 144
(boolean_expression **)**
| boolean_value
| identifier
| boolean_unary_operator boolean_expression
| boolean_expression boolean_binary_operator boolean_expression
| boolean_expression **?** boolean_expression **:** boolean_expression
boolean_unary_operator ::=
! | **~** | **&** | **~&** | **|** | **~|** | **^** | **~^**
boolean_binary_operator ::=
& | **&&** | **~&** | **|** | **||** | **~|** | **^** | **~^**
| relational_operator

```
        | arithmetic_operator
        | shift_operator
vector_expression ::=                                   // See Syntax 75 on page 154
          ( vector_expression )
        | single_event
        | vector_expression vector_operator vector_expression
        | boolean_expression ? vector_expression : vector_expression
        | boolean_expression control_and vector_expression
        | vector_expression control_and boolean_expression
        | vector_expression_macro
single_event ::=
        edge_literal boolean_expression
vector_operator ::=
          event_operator | event_and | event_or
event_and ::=
        & | &&
event_or ::=
        | | ||
control_and ::=
        & | &&
control_expression ::=
          ( vector_expression )
        | ( boolean_expression )
wire_instantiation ::=                                  // See Syntax 76 on page 167
          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 ;
geometric_model ::=                                     // See Syntax 77 on page 167
          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 Syntax 78 on page 171
          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 Syntax 79 on page 173
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 **;** } **}**

via_instantiation ::= // See Syntax 80 on page 174
*via*_identifier *instance*_identifier **;**
| *via*_identifier *instance*_identifier **{** { geometric_transformation } **}**

arithmetic_expression ::= // See Syntax 81 on page 175
(arithmetic_expression **)**
| arithmetic_value
| identifier
| boolean_expression **?** arithmetic_expression **:** arithmetic_expression
| sign arithmetic_expression
| arithmetic_expression arithmetic_operator arithmetic_expression
| macro_arithmetic_operator **(** arithmetic_expression { **,** arithmetic_expression } **)**

macro_arithmetic_operator ::=
abs | **exp** | **log** | **min** | **max**

arithmetic_model ::= // See Syntax 82 on page 177
trivial_arithmetic_model
| partial_arithmetic_model
| full_arithmetic_model
| *arithmetic_model*_template_instantiation

trivial_arithmetic_model ::= // See Syntax 83 on page 177
*arithmetic_model*_identifier [*name*_identifier] **=** arithmetic_value **;**
| *arithmetic_model*_identifier [*name*_identifier] **=** arithmetic_value
{ { arithmetic_model_qualifier } **}**

partial_arithmetic_model ::= // See Syntax 84 on page 178
*arithmetic_model*_identifier [*name*_identifier] **{** { partial_arithmetic_model_item } **}**

partial_arithmetic_model_item ::=
arithmetic_model_qualifier
| table
| trivial_min-max

full_arithmetic_model ::= // See Syntax 85 on page 178
*arithmetic_model*_identifier [*name*_identifier]
{ { arithmetic_model_qualifier } arithmetic_model_body { arithmetic_model_qualifier } **}**

arithmetic_model_body ::= // See Syntax 86 on page 178
header-table-equation [trivial_min-max]
| min-typ-max
| arithmetic_submodel { arithmetic_submodel }

arithmetic_model_qualifier ::= // See Syntax 87 on page 179
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

header-table-equation ::= // See Syntax 88 on page 179
header table | header equation

header ::= // See Syntax 89 on page 179
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

equation ::= // See Syntax 90 on page 180
EQUATION { arithmetic_expression **}**
| *equation*_template_instantiation

table ::= // See Syntax 91 on page 180
TABLE { arithmetic_value { arithmetic value } **}**

min-typ-max ::= // See Syntax 92 on page 181
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

non_trivial_min ::= // See Syntax 93 on page 182
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 **}**

trivial_min-max ::= // See Syntax 94 on page 182
trivial_min | trivial_max | trivial_min trivial_max

trivial_min ::=
MIN = arithmetic_value **;**

trivial_max ::=
MAX = arithmetic_value **;**

trivial_typ ::=
TYP = arithmetic_value **;**

auxiliary_arithmetic_model ::= // See Syntax 95 on page 183
*arithmetic_model*_identifier **=** arithmetic_value **;**
| *arithmetic_model*_identifier [**=** arithmetic_value]
{ inheritable_arithmetic_model_qualifier { inheritable_arithmetic_model_qualifier } **}**

```
arithmetic_submodel ::=                                          // See Syntax 96 on page 184
        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
arithmetic_model_container ::=                                   // See Syntax 97 on page 184
        limit_arithmetic_model_container
      | early-late_arithmetic_model_container
      | arithmetic_model_container_identifier { arithmetic_model { arithmetic_model } }
limit_arithmetic_model_container ::=                             // See Syntax 98 on page 184
        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 }
early-late_arithmetic_model_container ::=                        // See Syntax 99 on page 185
        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
violation ::=                                                    // See Syntax 100 on page 190
        VIOLATION { violation_item { violation_item } }
      | violation_template_instantiation
violation_item ::=
        MESSAGE_TYPE_single_value_annotation
      | MESSAGE_single_value_annotation
      | behavior
from-to ::=                                                      // See Syntax 101 on page 210
        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
```

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.

```
KEYWORD VALUETYPE = single_value_annotation {        // See Semantics 1 on page 59
  CONTEXT = SEMANTICS;
}
SEMANTICS VALUETYPE {
  VALUES {
    number  signed_integer  unsigned_integer
    multiplier_prefix_value
    identifier  quoted_string string_value
    bit_literal based_literal boolean_value edge_value
    control_expression  boolean_expression
    arithmetic_expression
  }
}
KEYWORD VALUES = multi_value_annotation {            // See Semantics 2 on page 60
  CONTEXT = SEMANTICS;
}
KEYWORD DEFAULT = single_value_annotation {          // See Semantics 3 on page 61
  CONTEXT { SEMANTICS arithmetic_model }
}
KEYWORD CONTEXT = annotation;                        // See Semantics 4 on page 61
SEMANTICS CONTEXT {
  CONTEXT { KEYWORD SEMANTICS }
  VALUETYPE = identifier;
}
KEYWORD REFERENCETYPE = annotation {                 // See Semantics 5 on page 62
  CONTEXT = SEMANTICS;
}
SEMANTICS REFERENCETYPE {
  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 }
}
KEYWORD SI_MODEL = single_value_annotation {         // See Semantics 6 on page 63
  CONTEXT = SEMANTICS;
}
SEMANTICS SI_MODEL {
  VALUES {
    TIME FREQUENCY CURRENT VOLTAGE POWER ENERGY
    RESISTANCE CAPACITANCE INDUCTANCE
    DISTANCE AREA
```

```
  }
}
KEYWORD CLASS = annotation {                          // See Semantics 7 on page 65
  CONTEXT { library_specific_object arithmetic_model }
}
SEMANTICS CLASS { REFERENCETYPE = CLASS; }
KEYWORD USAGE = annotation { CONTEXT = CLASS; }       // See Semantics 8 on page 66
SEMANTICS USAGE {
  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
  }
}
KEYWORD FORMAT = single_value_annotation {            // See Semantics 9 on page 73
  CONTEXT = ASSOCIATE;
}
SEMANTICS FORMAT {
  VALUETYPE = identifier;
  VALUES { vhdl verilog c \c++ alf }
  DEFAULT = alf;
}
KEYWORD LIBRARY = annotation {                        // See Semantics 10 on page 76
  CONTEXT = arithmetic_model;
}
SEMANTICS LIBRARY {
  REFERENCETYPE { LIBRARY SUBLIBRARY }
}
KEYWORD INFORMATION = annotation_container {          // See Semantics 11 on page 77
  CONTEXT { LIBRARY SUBLIBRARY CELL WIRE PRIMITIVE }
}
KEYWORD PRODUCT = single_value_annotation {
  CONTEXT = INFORMATION;
}
SEMANTICS PRODUCT {
  VALUETYPE = string_value; DEFAULT = "";
}
KEYWORD TITLE = single_value_annotation {
  CONTEXT = INFORMATION;
}
SEMANTICS TITLE {
  VALUETYPE = string_value; DEFAULT = "";
}
KEYWORD VERSION = single_value_annotation {
  CONTEXT = INFORMATION;
}
```

```
SEMANTICS VERSION {
  VALUETYPE = string_value; DEFAULT = "";
}
KEYWORD AUTHOR = single_value_annotation {
  CONTEXT = INFORMATION;
}
SEMANTICS AUTHOR {
  VALUETYPE = string_value; DEFAULT = "";
}
KEYWORD DATETIME = single_value_annotation {
  CONTEXT = INFORMATION;
}
SEMANTICS DATETIME {
  VALUETYPE = string_value; DEFAULT = "";
}
KEYWORD CELL = annotation { CONTEXT = arithmetic_model; }
SEMANTICS CELL { REFERENCETYPE = CELL; }            // See Semantics 12 on page 78
KEYWORD CELLTYPE = single_value_annotation {        // See Semantics 13 on page 79
  CONTEXT = CELL;
}
SEMANTICS CELLTYPE {
  VALUETYPE = identifier;
  VALUES {
    buffer combinational multiplexor flipflop latch
    memory block core special
  }
}
KEYWORD RESTRICT_CLASS = annotation {               // See Semantics 14 on page 80
  CONTEXT { CELL CLASS }
}
SEMANTICS RESTRICT_CLASS {
  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 ; }
KEYWORD SWAP_CLASS = annotation {                   // See Semantics 15 on page 81
  CONTEXT = CELL;
}
SEMANTICS SWAP_CLASS {
  REFERENCETYPE = CLASS;
}
KEYWORD SCAN_TYPE = single_value_annotation {       // See Semantics 16 on page 82
  CONTEXT = CELL;
}
SEMANTICS SCAN_TYPE {
  VALUETYPE = identifier;
  VALUES { muxscan clocked lssd control_0 control_1 }
}
```

```
KEYWORD SCAN_USAGE = single_value_annotation {      // See Semantics 17 on page 83
  CONTEXT = CELL;
}
SEMANTICS SCAN_USAGE {
  VALUETYPE = identifier;
  VALUES { input output hold }
}
KEYWORD BUFFERTYPE = single_value_annotation {      // See Semantics 18 on page 83
  CONTEXT = CELL;
}
SEMANTICS BUFFERTYPE {
  VALUETYPE = identifier;
  VALUES { input output inout internal }
  DEFAULT = internal;
}
KEYWORD DRIVERTYPE = single_value_annotation {      // See Semantics 19 on page 84
  CONTEXT = CELL;
}
SEMANTICS DRIVERTYPE {
  VALUETYPE = identifier;
  VALUES { predriver slotdriver both }
}
KEYWORD PARALLEL_DRIVE = single_value_annotation {  // See Semantics 20 on page 85
  CONTEXT = CELL;
}
SEMANTICS PARALLEL_DRIVE {
  VALUETYPE = unsigned_integer;
  DEFAULT = 1;
}
KEYWORD PLACEMENT_TYPE = single_value_annotation {  // See Semantics 21 on page 85
  CONTEXT = CELL;
}
SEMANTICS PLACEMENT_TYPE {
  VALUETYPE = identifier;
  VALUES { pad core ring block connector }
  DEFAULT = core;
}
SEMANTICS CELL.SITE = single_value_annotation;      // See Semantics 22 on page 86
KEYWORD PIN = annotation {                          // See Semantics 23 on page 90
  CONTEXT { arithmetic_model FROM TO }
}
SEMANTICS PIN {
  REFERENCETYPE { PIN PINGROUP PORT NODE }
}
KEYWORD MEMBERS = multi_value_annotation {          // See Semantics 24 on page 90
  CONTEXT = PINGROUP;
}
SEMANTICS MEMBERS {
  REFERENCETYPE = PIN;
}
```

```
KEYWORD VIEW = single_value_annotation {            // See Semantics 25 on page 90
  CONTEXT { PIN PINGROUP }
}
SEMANTICS VIEW {
  VALUES { functional physical both none }
  DEFAULT = both;
}
KEYWORD PINTYPE = single_value_annotation {         // See Semantics 26 on page 91
  CONTEXT = PIN;
}
SEMANTICS PINTYPE {
  VALUETYPE = identifier;
  VALUES { digital analog supply }
  DEFAULT = digital;
}
KEYWORD DIRECTION = single_value_annotation {       // See Semantics 27 on page 92
  CONTEXT = PIN;
}
SEMANTICS DIRECTION {
  VALUES { input output both none }
}
KEYWORD SIGNALTYPE = single_value_annotation {      // See Semantics 28 on page 93
  CONTEXT = PIN;
}
SEMANTICS SIGNALTYPE {
  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 Semantics 29 on page 95
  CONTEXT = PIN;
}
SEMANTICS ACTION {
  VALUES { asynchronous synchronous }
}
KEYWORD POLARITY = single_value_annotation {        // See Semantics 30 on page 96
  CONTEXT = PIN;
}
SEMANTICS POLARITY {
  VALUES { high low rising_edge falling_edge double_edge }
}
KEYWORD CONTROL_POLARITY = annotation_container {   // See Semantics 31 on page 97
  CONTEXT = PIN ;
}
```

```
SEMANTICS
CONTROL_POLARITY.identifier = single_value_annotation {
  VALUES { high low rising_edge falling_edge double_edge }
}
KEYWORD DATATYPE = single_value_annotation {        // See Semantics 32 on page 98
  CONTEXT { PIN PINGROUP }
}
SEMANTICS DATATYPE {
  VALUES { signed unsigned }
}
KEYWORD INITIAL_VALUE = single_value_annotation {   // See Semantics 33 on page 99
  CONTEXT { PIN PINGROUP }
}
SEMANTICS INITIAL_VALUE {
  VALUETYPE = boolean_value;
  DEFAULT = U;
}
KEYWORD SCAN_POSITION = single_value_annotation {   // See Semantics 34 on page 99
  CONTEXT = PIN;
}
SEMANTICS SCAN_POSITION {
  VALUETYPE = unsigned_integer;
  DEFAULT = 0;
}
KEYWORD STUCK = single_value_annotation {           // See Semantics 35 on page 99
  CONTEXT = PIN;
}
SEMANTICS STUCK {
  VALUES { stuck_at_0 stuck_at_1 both none }
  DEFAULT = both;
}
KEYWORD SUPPLYTYPE = annotation {                   // See Semantics 36 on page 100
  CONTEXT { PIN CLASS }
}
SEMANTICS SUPPLYTYPE {
  VALUETYPE = identifier;
  VALUES { power ground reference }
}
KEYWORD SIGNAL_CLASS = annotation {                 // See Semantics 37 on page 101
  CONTEXT { PIN PINGROUP }
}
SEMANTICS SIGNAL_CLASS {
  REFERENCETYPE = CLASS;
}
KEYWORD SUPPLY_CLASS = annotation {                 // See Semantics 38 on page 101
  CONTEXT { PIN CLASS POWER ENERGY }
}
SEMANTICS SUPPLY_CLASS {
  REFERENCETYPE = CLASS;
}
```

```
KEYWORD DRIVETYPE = single_value_annotation {      // See Semantics 39 on page 102
  CONTEXT { PIN CLASS }
}
SEMANTICS DRIVETYPE {
  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 Semantics 40 on page 103
  CONTEXT { PIN PINGROUP }
}
SEMANTICS SCOPE {
  VALUES { behavior measure both none }
  DEFAULT = both;
}
KEYWORD CONNECT_CLASS = single_value_annotation {  // See Semantics 41 on page 104
  CONTEXT = PIN;
}
SEMANTICS CONNECT_CLASS {
  REFERENCETYPE = CLASS;
}
KEYWORD SIDE = single_value_annotation {           // See Semantics 42 on page 105
  CONTEXT { PIN PINGROUP }
}
SEMANTICS SIDE {
  VALUETYPE = identifier;
  VALUES { left right top bottom inside }
}
KEYWORD ROW = annotation {                         // See Semantics 43 on page 105
  CONTEXT { PIN PINGROUP }
}
SEMANTICS ROW {
  VALUETYPE = unsigned_integer;
}
KEYWORD COLUMN = annotation {
  CONTEXT { PIN PINGROUP }
}
SEMANTICS COLUMN {
  VALUETYPE = unsigned_integer;
}
KEYWORD ROUTING_TYPE = single_value_annotation {   // See Semantics 44 on page 106
  CONTEXT { PIN PORT }
}
SEMANTICS ROUTING_TYPE {
  VALUETYPE = identifier;
  VALUES { regular abutment ring feedthrough }
  DEFAULT = regular;
}
```

```
KEYWORD PULL = single_value_annotation {          // See Semantics 45 on page 107
  CONTEXT = PIN;
}
SEMANTICS PULL {
  VALUES { up down both none }
  DEFAULT = none;
}
KEYWORD WIRE = annotation {                       // See Semantics 46 on page 110
  CONTEXT = arithmetic_model;
}
SEMANTICS WIRE {
  REFERENCETYPE = WIRE;
}
KEYWORD WIRETYPE = single_value_annotation {      // See Semantics 47 on page 111
  CONTEXT = WIRE;
}
SEMANTICS WIRETYPE {
  VALUETYPE = identifier;
  VALUES { estimated extracted interconnect load }
}
KEYWORD SELECT_CLASS = annotation {               // See Semantics 48 on page 112
  CONTEXT = WIRE;
}
SEMANTICS SELECT_CLASS {
  REFERENCETYPE = CLASS;
}
KEYWORD NODE = multi_value_annotation {           // See Semantics 49 on page 113
  CONTEXT = arithmetic_model;
}
SEMANTICS NODE {
  REFERENCETYPE { PIN PORT NODE }
}
KEYWORD NODETYPE = single_value_annotation {      // See Semantics 50 on page 113
  CONTEXT = NODE;
}
SEMANTICS NODETYPE {
  VALUETYPE = identifier;
  VALUES { power ground source sink
    driver receiver interconnect }
  DEFAULT = interconnect;
}
KEYWORD NODE_CLASS = annotation {                 // See Semantics 51 on page 115
  CONTEXT = NODE;
}
SEMANTICS NODE_CLASS {
  REFERENCETYPE = CLASS;
}
KEYWORD VECTOR = single_value_annotation {        // See Semantics 52 on page 116
  CONTEXT = arithmetic_model;
}
```

```
SEMANTICS VECTOR {
  VALUETYPE = control_expression;
  REFERENCETYPE = VECTOR;
}
KEYWORD PURPOSE = annotation {                    // See Semantics 53 on page 116
  CONTEXT { VECTOR CLASS }
}
SEMANTICS PURPOSE {
  VALUETYPE = identifier ;
  VALUES { bist test timing power noise reliability }
}
KEYWORD OPERATION = single_value_annotation {     // See Semantics 54 on page 117
  CONTEXT = VECTOR;
}
SEMANTICS OPERATION {
  VALUETYPE = identifier;
  VALUES {
    read write read_modify_write refresh load
    start end iddq
  }
}
KEYWORD LABEL = single_value_annotation {         // See Semantics 55 on page 118
  CONTEXT = VECTOR;
}
SEMANTICS LABEL {
  VALUETYPE = string_value;
}
KEYWORD EXISTENCE_CONDITION = single_value_annotation {
  CONTEXT { VECTOR CLASS }                        // See Semantics 56 on page 118
}
SEMANTICS EXISTENCE_CONDITION {
  VALUETYPE = boolean_expression;
  DEFAULT = 1;
}
KEYWORD EXISTENCE_CLASS = annotation {            // See Semantics 57 on page 119
  CONTEXT { VECTOR CLASS }
}
SEMANTICS EXISTENCE_CLASS {
  REFERENCETYPE = CLASS;
}
KEYWORD                                           // See Semantics 58 on page 119
CHARACTERIZATION_CONDITION = single_value_annotation {
  CONTEXT { VECTOR CLASS }
}
SEMANTICS CHARACTERIZATION_CONDITION {
  VALUETYPE = boolean_expression;
}
KEYWORD CHARACTERIZATION_VECTOR = single_value_annotation {
  CONTEXT { VECTOR CLASS }                        // See Semantics 59 on page 120
}
```

```
SEMANTICS CHARACTERIZATION_VECTOR {
  VALUETYPE = control_expression;
}
KEYWORD CHARACTERIZATION_CLASS = annotation {      // See Semantics 60 on page 120
  CONTEXT { VECTOR CLASS }
}
SEMANTICS CHARACTERIZATION_CLASS {
  REFERENCETYPE = CLASS;
}
KEYWORD MONITOR = annotation {                     // See Semantics 61 on page 120
  CONTEXT { VECTOR CLASS }
}
SEMANTICS MONITOR {
  VALUETYPE = identifier;
}
KEYWORD LAYER = annotation {                       // See Semantics 62 on page 121
  CONTEXT { arithmetic_model PATTERN ARRAY }
}
SEMANTICS LAYER {
  REFERENCETYPE = LAYER;
}
KEYWORD LAYERTYPE = single_value_annotation {      // See Semantics 63 on page 121
  CONTEXT = LAYER;
}
SEMANTICS LAYERTYPE {
  VALUETYPE = identifier;
  VALUES { routing cut substrate dielectric reserved abstract }
}
KEYWORD PITCH = single_value_annotation {          // See Semantics 64 on page 122
  CONTEXT = LAYER;
}
SEMANTICS PITCH {
  VALUETYPE = unsigned_number;
}
KEYWORD PREFERENCE = single_value_annotation {     // See Semantics 65 on page 122
  CONTEXT = LAYER;
}
SEMANTICS PREFERENCE {
  VALUETYPE = identifier;
  VALUES { horizontal vertical acute obtuse }
}
KEYWORD VIA = annotation {                         // See Semantics 66 on page 123
  CONTEXT = arithmetic_model;
}
SEMANTICS VIA {
  REFERENCETYPE = VIA;
}
KEYWORD VIATYPE = single_value_annotation {        // See Semantics 67 on page 124
  CONTEXT = VIA;
}
```

```
SEMANTICS VIATYPE {
  VALUETYPE = identifier;
  VALUES { default non_default partial_stack full_stack }
  DEFAULT = default;
}
KEYWORD PORTTYPE = single_value_annotation {       // See Semantics 68 on page 127
  CONTEXT = PORT;
}
SEMANTICS PORTTYPE {
  VALUETYPE = identifier;
  VALUES { external internal }
  DEFAULT = external;
}
KEYWORD SITE = annotation {                        // See Semantics 69 on page 128
  CONTEXT { CELL ARRAY CLASS }
}
SEMANTICS SITE {
  REFERENCETYPE = SITE;
}
KEYWORD ORIENTATION_CLASS = annotation {           // See Semantics 70 on page 128
  CONTEXT { SITE CELL }
}
SEMANTICS ORIENTATION_CLASS {
  REFERENCETYPE = CLASS;
}
KEYWORD SYMMETRY_CLASS = multi_value_annotation {  // See Semantics 71 on page 128
  CONTEXT = SITE;
}
SEMANTICS SYMMETRY_CLASS {
  REFERENCETYPE = CLASS;
}
KEYWORD ARRAYTYPE = single_value_annotation {      // See Semantics 72 on page 130
  CONTEXT = ARRAY;
}
SEMANTICS ARRAYTYPE {
  VALUETYPE = identifier;
  VALUES { floorplan placement
     global_routing detailed_routing }
}
SEMANTICS ARRAY.LAYER = multi_value_annotation;    // See Semantics 73 on page 130
SEMANTICS ARRAY.SITE = single_value_annotation;    // See Semantics 74 on page 130
KEYWORD PATTERN = annotation {                     // See Semantics 75 on page 131
  CONTEXT = arithmetic_model ;
}
SEMANTICS PATTERN {
  REFERENCETYPE = PATTERN ;
}
KEYWORD SHAPE = single_value_annotation {          // See Semantics 76 on page 131
  CONTEXT = PATTERN;
}
```

```
SEMANTICS SHAPE {
  VALUETYPE = identifier;
  VALUES { line tee cross jog corner end }
  DEFAULT = line;
}
KEYWORD VERTEX = single_value_annotation {          // See Semantics 77 on page 133
  CONTEXT = PATTERN;
}
SEMANTICS VERTEX {
  VALUETYPE = identifier;
  VALUES { round angular }
  DEFAULT = angular;
}
KEYWORD ROUTE = single_value_annotation {           // See Semantics 78 on page 133
  CONTEXT = PATTERN;
}
SEMANTICS ROUTE {
  VALUETYPE = identifier;
  VALUES { horizontal acute vertical obtuse }
}
SEMANTICS PATTERN.LAYER = single_value_annotation; // See Semantics 79 on page 134
KEYWORD REGION = annotation {                       // See Semantics 80 on page 135
  CONTEXT = arithmetic_model ;
}
SEMANTICS REGION {
  REFERENCETYPE = REGION ;
}
KEYWORD BOOLEAN = single_value_annotation {         // See Semantics 81 on page 135
  CONTEXT = REGION ;
}
SEMANTICS BOOLEAN {
  VALUETYPE = boolean_expression ;
}
PRIMITIVE ALF_BUF {                                 // See Semantics 82 on page 162
  PIN in { DIRECTION = input; }
  PIN [1:<bitwidth>] out { DIRECTION = output; }
  GROUP index { 1 : <bitwidth> }
  FUNCTION { BEHAVIOR { out[index] = in ; } }
}
PRIMITIVE ALF_NOT {                                 // See Semantics 83 on page 162
  PIN in { DIRECTION = input; }
  PIN [1:<bitwidth>] out { DIRECTION = output; }
  GROUP index { 1 : <bitwidth> }
  FUNCTION { BEHAVIOR { out[index] = ! in ; } }
}
PRIMITIVE ALF_AND {                                 // See Semantics 84 on page 162
  PIN out { DIRECTION = output; }
  PIN [1:<bitwidth>] in { DIRECTION = input; }
  FUNCTION { BEHAVIOR { out = & in ; } }
}
```

```
PRIMITIVE ALF_NAND {                                  // See Semantics 85 on page 163
  PIN out { DIRECTION = output; }
  PIN [1:<bitwidth>] in { DIRECTION = input; }
  FUNCTION { BEHAVIOR { out = ~& in ; } }
}
PRIMITIVE ALF_OR {                                    // See Semantics 86 on page 163
  PIN out { DIRECTION = output; }
  PIN [1:<bitwidth>] in { DIRECTION = input; }
  FUNCTION { BEHAVIOR { out = | in ; } }
}
PRIMITIVE ALF_NOR {                                   // See Semantics 87 on page 163
  PIN out { DIRECTION = output; }
  PIN [1:<bitwidth>] in { DIRECTION = input; }
  FUNCTION { BEHAVIOR { out = ~| in ; } }
}
PRIMITIVE ALF_XOR {                                   // See Semantics 88 on page 163
  PIN out { DIRECTION = output; }
  PIN [1:<bitwidth>] in { DIRECTION = input; }
  FUNCTION { BEHAVIOR { out = ^ in ; } }
}
PRIMITIVE ALF_XNOR {                                  // See Semantics 89 on page 164
  PIN out { DIRECTION = output; }
  PIN [1:<bitwidth>] in { DIRECTION = input; }
  FUNCTION { BEHAVIOR { out = ~^ in ; } }
}
PRIMITIVE ALF_BUFIF1 {                                // See Semantics 90 on page 164
  PIN out { DIRECTION = output; }
  PIN in  { DIRECTION = input; }
  PIN enable { DIRECTION = input; }
  FUNCTION { BEHAVIOR { out = (enable)? in : 'bZ ; } }
}
PRIMITIVE ALF_BUFIF0 {                                // See Semantics 91 on page 164
  PIN out { DIRECTION = output; }
  PIN in  { DIRECTION = input; }
  PIN enable { DIRECTION = input; }
  FUNCTION { BEHAVIOR { out = (! enable)? in : 'bZ ; } }
}
PRIMITIVE ALF_NOTIF1 {                                // See Semantics 92 on page 164
  PIN out { DIRECTION = output; }
  PIN in  { DIRECTION = input; }
  PIN enable { DIRECTION = input; }
  FUNCTION { BEHAVIOR { out = (enable)? ! in : 'bZ ; } }
}
PRIMITIVE ALF_NOTIF0 {                                // See Semantics 93 on page 165
  PIN out { DIRECTION = output; }
  PIN in  { DIRECTION = input; }
  PIN enable { DIRECTION = input; }
  FUNCTION { BEHAVIOR { out = (! enable)? ! in : 'bZ ; } }
}
PRIMITIVE ALF_MUX {                                   // See Semantics 94 on page 165
  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] ;
    }
  }
}
PRIMITIVE ALF_LATCH {                                  // See Semantics 95 on page 166
  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 ;
      }
    }
  }
}
PRIMITIVE ALF_FLIPFLOP {                               // See Semantics 96 on page 166
  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 ;
      }
    }
  }
}
```

```
KEYWORD DOT = geometric_model;                          // See Semantics 97 on page 168
KEYWORD POLYLINE = geometric_model;
KEYWORD RING = geometric_model;
KEYWORD POLYGON = geometric_model;
KEYWORD POINT_TO_POINT = single_value_annotation {      // See Semantics 98 on page 169
  CONTEXT { POLYLINE RING POLYGON }
}
SEMANTICS POINT_TO_POINT {
  VALUES { direct manhattan }
  DEFAULT = direct;
}
TEMPLATE RECTANGLE {                                    // See Semantics 99 on page 171
  POLYGON {
    POINT_TO_POINT = manhattan;
    COORDINATES { <left> <bottom> <right> <top> }
  }
}
TEMPLATE LINE {                                         // See Semantics 100 on page 171
  POLYLINE {
    POINT_TO_POINT = direct;
    COORDINATES { <x_start> <y_start> <x_end> <y_end> }
  }
}
KEYWORD MIN = arithmetic_submodel {                     // See Semantics 101 on page 183
  CONTEXT { arithmetic_model arithmetic_submodel }
}
KEYWORD MAX = arithmetic_submodel {
  CONTEXT { arithmetic_model arithmetic_submodel }
}
KEYWORD TYP = arithmetic_submodel {
  CONTEXT { arithmetic_model arithmetic_submodel }
}
KEYWORD LIMIT = arithmetic_model_container;             // See Semantics 102 on page 185
KEYWORD EARLY = arithmetic_model_container              // See Semantics 103 on page 185
 { CONTEXT = VECTOR; }
KEYWORD LATE = arithmetic_model_container
 { CONTEXT = VECTOR; }
KEYWORD UNIT = single_value_annotation {                // See Semantics 104 on page 186
  CONTEXT = arithmetic_model ;
}
SEMANTICS UNIT {
  VALUETYPE = multiplier_prefix_value ;
}
KEYWORD CALCULATION = single_value_annotation {         // See Semantics 105 on page 186
  CONTEXT = arithmetic_model ;
}
SEMANTICS CALCULATION {
  CONTEXT = library_specific_object.arithmetic_model ;
  VALUES { absolute incremental }
  DEFAULT = absolute ;
}
```

```
KEYWORD INTERPOLATION = single_value_annotation { // See Semantics 106 on page 187
  CONTEXT = arithmetic_model ;
}
SEMANTICS INTERPOLATION {
  CONTEXT = HEADER.arithmetic_model ;
  VALUES { linear fit ceiling floor }
  DEFAULT = fit ;
}
KEYWORD MODEL = single_value_annotation {         // See Semantics 107 on page 189
  CONTEXT = arithmetic_model ;
}
SEMANTICS MODEL {
  REFERENCETYPE { arithmetic_model arithmetic_submodel }
}
SEMANTICS VIOLATION {                             // See Semantics 108 on page 190
  CONTEXT {
    SETUP HOLD RECOVERY REMOVAL NOCHANGE ILLEGAL
    NOISE_MARGIN LIMIT..
  }
}
SEMANTICS VIOLATION.BEHAVIOR { CONTEXT { VECTOR.. }// See Semantics 109 on page 190
KEYWORD MESSAGE_TYPE = single_value_annotation {  // See Semantics 110 on page 191
  CONTEXT = VIOLATION ;
}
SEMANTICS MESSAGE_TYPE {
  VALUETYPE = identifier ;
  VALUES { information warning error }
}
KEYWORD MESSAGE = single_value_annotation {       // See Semantics 111 on page 192
  CONTEXT = VIOLATION ;
}
SEMANTICS MESSAGE {
  VALUETYPE = quoted_string ;
}
KEYWORD TIME = arithmetic_model ;                 // See Semantics 112 on page 192
SEMANTICS TIME {
  CONTEXT {
    LIBRARY SUBLIBRARY CELL WIRE VECTOR arithmetic_model
    VECTOR.arithmetic_model_container
    VECTOR..HEADER LIMIT..HEADER
  }
  VALUETYPE = number ;
  SI_MODEL = TIME ;
}
TIME { UNIT = NanoSeconds ; }
KEYWORD FREQUENCY = arithmetic_model ;            // See Semantics 113 on page 193
SEMANTICS FREQUENCY {
  CONTEXT {
    LIBRARY SUBLIBRARY CELL WIRE VECTOR arithmetic_model
    VECTOR.arithmetic_model_container
    VECTOR..HEADER LIMIT..HEADER
  }
```

```
  VALUETYPE = number ;
  SI_MODEL = FREQUENCY ;
}
FREQUENCY { UNIT = GigaHertz; MIN = 0; }
KEYWORD DELAY = arithmetic_model ;                // See Semantics 114 on page 194
SEMANTICS DELAY {
  CONTEXT {
    LIBRARY SUBLIBRARY CELL WIRE
    VECTOR VECTOR.EARLY VECTOR.LATE
  }
  SI_MODEL = TIME ;
}
KEYWORD RETAIN = arithmetic_model ;               // See Semantics 115 on page 195
SEMANTICS RETAIN{
  CONTEXT {
    VECTOR VECTOR.EARLY VECTOR.LATE
  }
  SI_MODEL = TIME ;
}
KEYWORD SLEWRATE = arithmetic_model ;             // See Semantics 116 on page 196
SEMANTICS SLEWRATE {
  CONTEXT {
    LIBRARY LIBRARY.LIMIT SUBLIBRARY SUBLIBRARY.LIMIT
    CELL CELL.LIMIT PIN PIN.LIMIT WIRE WIRE.LIMIT
    VECTOR VECTOR.EARLY VECTOR.LATE VECTOR.LIMIT
    VECTOR..HEADER
  }
  SI_MODEL = TIME ;
}
SLEWRATE { MIN = 0; }
KEYWORD SETUP = arithmetic_model ;                // See Semantics 117 on page 197
SEMANTICS SETUP { CONTEXT = VECTOR ; SI_MODEL = TIME ; }
KEYWORD HOLD = arithmetic_model ;
SEMANTICS HOLD { CONTEXT = VECTOR ; SI_MODEL = TIME ; }
KEYWORD RECOVERY = arithmetic_model ;             // See Semantics 118 on page 198
SEMANTICS RECOVERY { CONTEXT = VECTOR ; SI_MODEL = TIME ; }
KEYWORD REMOVAL = arithmetic_model ;
SEMANTICS REMOVAL { CONTEXT = VECTOR ; SI_MODEL = TIME ; }
KEYWORD NOCHANGE = arithmetic_model ;             // See Semantics 119 on page 199
SEMANTICS NOCHANGE { CONTEXT = VECTOR ; SI_MODEL = TIME ; }
NOCHANGE { MIN = 0; }
KEYWORD ILLEGAL = arithmetic_model ;
SEMANTICS ILLEGAL { CONTEXT = VECTOR ; SI_MODEL = TIME ; }
ILLEGAL { MIN = 0; }
KEYWORD PULSEWIDTH=arithmetic_model ;             // See Semantics 120 on page 201
SEMANTICS PULSEWIDTH {
  CONTEXT {
    LIBRARY LIBRARY.LIMIT SUBLIBRARY SUBLIBRARY.LIMIT
    CELL CELL.LIMIT PIN PIN.LIMIT WIRE WIRE.LIMIT
    VECTOR VECTOR..HEADER
```

```
  }
  SI_MODEL = TIME ;
}
PULSEWIDTH { MIN = 0; }
KEYWORD PERIOD = arithmetic_model ;                // See Semantics 121 on page 202
SEMANTICS PERIOD {
  CONTEXT { VECTOR VECTOR.LIMIT VECTOR..HEADER }
  SI_MODEL = TIME ;
}
PERIOD { MIN = 0; }
KEYWORD JITTER = arithmetic_model ;                // See Semantics 122 on page 203
SEMANTICS JITTER {
  CONTEXT { VECTOR VECTOR.LIMIT VECTOR..HEADER }
  SI_MODEL = TIME ;
}
JITTER { MIN = 0; }
KEYWORD SKEW = arithmetic_model ;                  // See Semantics 123 on page 204
SEMANTICS SKEW {
  CONTEXT { VECTOR VECTOR.LIMIT VECTOR..HEADER }
  SI_MODEL = TIME ;
}
SKEW { MIN = 0; }
KEYWORD THRESHOLD = arithmetic_model ;             // See Semantics 124 on page 205
SEMANTICS THRESHOLD {
  CONTEXT { PIN FROM TO }
  VALUETYPE = number ;
}
THRESHOLD { MIN = 0; MAX = 1; }
KEYWORD NOISE = arithmetic_model ;                 // See Semantics 125 on page 206
SEMANTICS NOISE {
  CONTEXT {
    LIBRARY.LIMIT SUBLIBRARY.LIMIT CELL.LIMIT
    PIN PIN.LIMIT VECTOR VECTOR.LIMIT VECTOR..HEADER
  }
  VALUETYPE = number ;
}
KEYWORD NOISE_MARGIN = arithmetic_model ;
SEMANTICS NOISE_MARGIN {
  CONTEXT { CLASS LIBRARY SUBLIBRARY CELL PIN VECTOR }
  VALUETYPE = number ;
}
NOISE_MARGIN { MIN = 0; }
KEYWORD POWER = arithmetic_model ;                 // See Semantics 126 on page 209
SEMANTICS POWER {
  CONTEXT {
    LIBRARY SUBLIBRARY CELL VECTOR
    CLASS.LIMIT CELL.LIMIT
  }
  VALUETYPE = number ;
}
```

```
POWER { UNIT = MilliWatt; }
KEYWORD ENERGY = arithmetic_model { VALUETYPE = number; }
SEMANTICS ENERGY {
  CONTEXT { LIBRARY SUBLIBRARY CELL VECTOR }
  VALUETYPE = number ;
}
ENERGY { UNIT = PicoJoule; }
SEMANTICS FROM {                                  // See Semantics 127 on page 210
  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
  }
}
KEYWORD EDGE_NUMBER = annotation {                // See Semantics 128 on page 211
  CONTEXT { arithmetic_model FROM TO }
}
SEMANTICS EDGE_NUMBER {
  CONTEXT { VECTOR.. }
  VALUETYPE = unsigned_integer ;
  DEFAULT = 0;
}
SEMANTICS FROM.PIN = single_value_annotation {    // See Semantics 129 on page 211
  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 FROM.EDGE_NUMBER = single_value_annotation {
  CONTEXT { TIME DELAY RETAIN SETUP HOLD          // See Semantics 130 on page 211
  RECOVERY REMOVAL NOCHANGE ILLEGAL }
}
SEMANTICS TO.EDGE_NUMBER = single_value_annotation {
  CONTEXT { TIME DELAY RETAIN SETUP HOLD
  RECOVERY REMOVAL NOCHANGE ILLEGAL }
}
SEMANTICS SLEWRATE.PIN = single_value_annotation ;// See Semantics 131 on page 212
SEMANTICS SLEWRATE.EDGE_NUMBER = single_value_annotation ;
SEMANTICS PULSEWIDTH.PIN = single_value_annotation;// See Semantics 132 on page 212
SEMANTICS PULSEWIDTH.EDGE_NUMBER = single_value_annotation;
SEMANTICS SKEW.PIN = multi_value_annotation ;     // See Semantics 133 on page 213
SEMANTICS SKEW.EDGE_NUMBER = multi_value_annotation ;
SEMANTICS NOISE.PIN = single_value_annotation ;   // See Semantics 134 on page 213
SEMANTICS NOISE_MARGIN.PIN = single_value_annotation ;
```

```
KEYWORD MEASUREMENT = single_value_annotation {   // See Semantics 135 on page 213
  CONTEXT = arithmetic_model ;
}
SEMANTICS MEASUREMENT {
  CONTEXT { ENERGY POWER CURRENT VOLTAGE JITTER }
  VALUETYPE = identifier ;
  VALUES {  transient static average absolute_average rms peak }
}
KEYWORD PROCESS = arithmetic_model ;              // See Semantics 136 on page 215
SEMANTICS PROCESS {
  CONTEXT {
    CLASS LIBRARY SUBLIBRARY CELL WIRE HEADER
    arithmetic_model
  }
  VALUETYPE = identifier ;
}
PROCESS { DEFAULT = nom; TABLE { nom snsp snwp wnsp wnwp } }
KEYWORD DERATE_CASE = arithmetic_model ;          // See Semantics 137 on page 216
SEMANTICS DERATE_CASE {
  CONTEXT {
    CLASS LIBRARY SUBLIBRARY CELL WIRE HEADER
    arithmetic_model
  }
  VALUETYPE = identifier ;
}
DERATE_CASE { DEFAULT = nom;
  TABLE { nom bccom wccom bcind wcind bcmil wcmil }
}
KEYWORD TEMPERATURE = arithmetic_model {          // See Semantics 138 on page 216
}
SEMANTICS TEMPERATURE {
  CONTEXT {
    CLASS LIBRARY SUBLIBRARY CELL WIRE
    LIMIT HEADER arithmetic_model
  }
  VALUETYPE = number ;
}
TEMPERATURE { UNIT = 1DegreeCelsius; MIN = -273; }
KEYWORD VOLTAGE = arithmetic_model ;              // See Semantics 139 on page 217
SEMANTICS VOLTAGE {
  CONTEXT {
    CLASS LIBRARY SUBLIBRARY CELL PIN WIRE VECTOR HEADER
    CLASS.LIMIT CELL.LIMIT PIN.LIMIT VECTOR.LIMIT
  }
  VALUETYPE = number ;
}
VOLTAGE { UNIT = 1Volt; }
```

```
KEYWORD CURRENT = arithmetic_model ;              // See Semantics 140 on page 218
SEMANTICS CURRENT {
  CONTEXT {
    LIBRARY SUBLIBRARY CELL WIRE VECTOR HEADER
    CELL.LIMIT VECTOR.LIMIT
    LAYER.LIMIT VIA.LIMIT RULE.LIMIT
  }
  VALUETYPE = number ;
}
CURRENT { UNIT = MilliAmpere; }
KEYWORD CAPACITANCE = arithmetic_model ;          // See Semantics 141 on page 219
SEMANTICS CAPACITANCE {
  CONTEXT {
    LIBRARY SUBLIBRARY CELL CELL.LIMIT PIN PIN.LIMIT
    WIRE LAYER RULE VECTOR HEADER
  }
  VALUETYPE = number ;
  SI_MODEL = CAPACITANCE ;
}
CAPACITANCE { UNIT = PicoFarad; MIN = 0; }
KEYWORD RESISTANCE = arithmetic_model ;           // See Semantics 142 on page 221
SEMANTICS RESISTANCE {
  CONTEXT {
    LIBRARY SUBLIBRARY CELL WIRE LAYER RULE
    CELL.LIMIT VECTOR HEADER
  }
  VALUETYPE = number ;
  SI_MODEL = RESISTANCE ;
}
RESISTANCE { UNIT = KiloOhm; MIN = 0; }
KEYWORD INDUCTANCE = arithmetic_model ;           // See Semantics 143 on page 222
SEMANTICS INDUCTANCE {
  CONTEXT {
    LIBRARY SUBLIBRARY CELL WIRE LAYER RULE
    CELL.LIMIT VECTOR HEADER
  }
  VALUETYPE = number ;
  SI_MODEL = INDUCTANCE ;
}
INDUCTANCE { UNIT = 1e-6; MIN = 0; }
SEMANTICS VOLTAGE.NODE = multi_value_annotation { // See Semantics 144 on page 224
  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 } }
```

```
KEYWORD COMPONENT = single_value_annotation {      // See Semantics 145 on page 225
  CONTEXT = arithmetic_model ;
}
SEMANTICS COMPONENT {
  CONTEXT { CURRENT POWER ENERGY }
  REFERENCETYPE {
    CURRENT VOLTAGE CAPACITANCE RESISTANCE INDUCTANCE
  }
}
SEMANTICS VOLTAGE.PIN = single_value_annotation { // See Semantics 146 on page 225
  CONTEXT { VECTOR VECTOR.LIMIT VECTOR..HEADER } }
SEMANTICS CURRENT.PIN = single_value_annotation {
  CONTEXT { VECTOR VECTOR.LIMIT VECTOR..HEADER } }
SEMANTICS CAPACITANCE.PIN = single_value_annotation {
  CONTEXT { VECTOR VECTOR..HEADER } }
SEMANTICS RESISTANCE.PIN = single_value_annotation {
  CONTEXT { VECTOR } }
KEYWORD FLOW = single_value_annotation {           // See Semantics 147 on page 227
  CONTEXT = arithmetic_model ;
}
SEMANTICS FLOW {
  CONTEXT = CURRENT;
  VALUES { in out }
  DEFAULT = in;
}
KEYWORD DRIVE_STRENGTH = arithmetic_model ;        // See Semantics 148 on page 227
SEMANTICS DRIVE_STRENGTH {
  CONTEXT { CLASS LIBRARY SUBLIBRARY CELL PIN PINGROUP }
  VALUETYPE = number ;
}
DRIVE_STRENGTH { MIN = 0; }
KEYWORD SWITCHING_BITS = arithmetic_model ;        // See Semantics 149 on page 228
SEMANTICS SWITCHING_BITS {
  CONTEXT { VECTOR.POWER.HEADER VECTOR.ENERGY.HEADER }
  VALUETYPE = unsigned_integer ;
}
SEMANTICS SWITCHING_BITS.PIN = single_value_annotation;
KEYWORD CONNECTIVITY = arithmetic_model ;          // See Semantics 150 on page 228
SEMANTICS CONNECTIVITY {
  CONTEXT { LIBRARY SUBLIBRARY CELL RULE ANTENNA HEADER }
  VALUES { 1 0 ? }
}
KEYWORD DRIVER = arithmetic_model {                // See Semantics 151 on page 229
SEMANTICS DRIVER {
  CONTEXT = CONNECTIVITY.HEADER;
  REFERENCETYPE = CLASS ;
}
KEYWORD RECEIVER = arithmetic_model ;
```

```
SEMANTICS RECEIVER {
  CONTEXT = CONNECTIVITY.HEADER;
  REFERENCETYPE = CLASS ;
}
KEYWORD FANOUT = arithmetic_model ;                // See Semantics 152 on page 230
SEMANTICS FANOUT {
  CONTEXT {
    PIN.LIMIT WIRE.SIZE.HEADER WIRE.CAPACITANCE.HEADER
    WIRE.RESISTANCE.HEADER WIRE.INDUCTANCE.HEADER
  }
  VALUETYPE = unsigned_integer ;
}
KEYWORD FANIN = arithmetic_model ;                 // See Semantics 153 on page 231
SEMANTICS FANIN {
  CONTEXT {
    PIN.LIMIT WIRE.SIZE.HEADER WIRE.CAPACITANCE.HEADER
    WIRE.RESISTANCE.HEADER WIRE.INDUCTANCE.HEADER
  }
  VALUETYPE = unsigned_integer ;
}
KEYWORD CONNECTIONS = arithmetic_model ;           // See Semantics 154 on page 231
SEMANTICS CONNECTIONS {
  CONTEXT {
    PIN.LIMIT WIRE.SIZE.HEADER WIRE.CAPACITANCE.HEADER
    WIRE.RESISTANCE.HEADER WIRE.INDUCTANCE.HEADER
  }
  VALUETYPE = unsigned_integer ;
}
KEYWORD SIZE = arithmetic_model ;                  // See Semantics 155 on page 232
SEMANTICS SIZE {
  CONTEXT {
    CELL ANTENNA ANTENNA.LIMIT PIN WIRE
    WIRE.CAPACITANCE.HEADER
    WIRE.RESISTANCE.HEADER
    WIRE.INDUCTANCE.HEADER
  }
  VALUETYPE = number ;
}
SIZE { MIN = 0; }
KEYWORD AREA = arithmetic_model ;                  // See Semantics 156 on page 233
SEMANTICS AREA {
  CONTEXT {
    CELL WIRE WIRE..HEADER LAYER..HEADER
    RULE..HEADER ANTENNA..HEADER
  }
  VALUETYPE = number ;
  SI_MODEL = AREA ;
}
AREA { UNIT = 1e-12; MIN = 0; }
```

```
KEYWORD PERIMETER = arithmetic_model ;          // See Semantics 157 on page 234
SEMANTICS PERIMETER {
  CONTEXT {
    CELL WIRE WIRE..HEADER LAYER..HEADER
    RULE..HEADER ANTENNA..HEADER
  }
  SI_MODEL = DISTANCE ;
}
KEYWORD EXTENSION = arithmetic_model ;          // See Semantics 158 on page 235
SEMANTICS EXTENSION {
  CONTEXT { LAYER PATTERN RULE.LIMIT RULE..HEADER }
  SI_MODEL = DISTANCE ;
}
KEYWORD THICKNESS = arithmetic_model ;          // See Semantics 159 on page 236
SEMANTICS EXTENSION {
  CONTEXT { LAYER RULE..HEADER }
  SI_MODEL = DISTANCE ;
}
KEYWORD HEIGHT = arithmetic_model ;             // See Semantics 160 on page 236
SEMANTICS HEIGHT {
  CONTEXT { CELL SITE REGION LAYER WIRE..HEADER }
  SI_MODEL = DISTANCE ;
}
KEYWORD WIDTH = arithmetic_model ;              // See Semantics 161 on page 237
SEMANTICS WIDTH {
  CONTEXT {
    CELL SITE REGION LAYER LAYER.LIMIT
    PATTERN RULE.LIMIT RULE..HEADER
  }
  SI_MODEL = DISTANCE ;
}
KEYWORD LENGTH = arithmetic_model ;             // See Semantics 162 on page 238
SEMANTICS LENGTH {
  CONTEXT {
    LAYER LAYER.LIMIT PATTERN RULE.LIMIT RULE..HEADER
  }
  SI_MODEL = DISTANCE ;
}
KEYWORD DISTANCE = arithmetic_model ;           // See Semantics 163 on page 239
SEMANTICS DISTANCE {
  CONTEXT { RULE RULE.LIMIT RULE..HEADER }
  VALUETYPE = number ;
  SI_MODEL = DISTANCE ;
}
DISTANCE { UNIT = 10e-6; MIN = 0; }
KEYWORD OVERHANG = arithmetic_model ;           // See Semantics 164 on page 239
SEMANTICS OVERHANG {
  CONTEXT { RULE RULE.LIMIT RULE..HEADER }
  SI_MODEL = DISTANCE ;
}
```

```
KEYWORD DENSITY = arithmetic_model ;              // See Semantics 165 on page 240
SEMANTICS DENSITY {
  CONTEXT { LAYER.LIMIT RULE RULE.LIMIT }
  VALUETYPE = number ;
}
DENSITY { MIN = 0; MAX = 1; }
KEYWORD CONNECT_RULE = single_value_annotation {  // See Semantics 166 on page 241
  CONTEXT = arithmetic_model ;
}
SEMANTICS CONNECT_RULE {
  CONTEXT = CONNECTIVITY ;
  VALUES { must_short can_short cannot_short }
}
KEYWORD BETWEEN = multi_value_annotation {        // See Semantics 167 on page 242
  CONTEXT = arithmetic_model ;
}
SEMANTICS BETWEEN {
  CONTEXT { DISTANCE LENGTH OVERHANG CONNECTIVITY }
}
SEMANTICS ANTENNA.CONNECTIVITY.BETWEEN {          // See Semantics 168 on page 242
  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 DISTANCE.BETWEEN {                      // See Semantics 169 on page 242
  REFERENCETYPE { PATTERN REGION }
}
SEMANTICS LENGTH.BETWEEN {
  REFERENCETYPE { PATTERN REGION }
}
SEMANTICS OVERHANG.BETWEEN {
  REFERENCETYPE { PATTERN REGION }
}
KEYWORD MEASURE = single_value_annotation {       // See Semantics 170 on page 243
  CONTEXT = arithmetic_model ;
}
SEMANTICS MEASURE {
  CONTEXT { DISTANCE LENGTH OVERHANG }
  VALUETYPE = identifier ;
```

```
  VALUES { euclidean horizontal vertical manhattan }
  DEFAULT = euclidean ;
}
KEYWORD REFERENCE = annotation_container {        // See Semantics 171 on page 244
  CONTEXT = arithmetic_model ;
}
SEMANTICS REFERENCE {
  CONTEXT { DISTANCE LENGTH OVERHANG }
  REFERENCETYPE { PATTERN REGION }
}
SEMANTICS REFERENCE.identifier = single_value_annotation {
  VALUETYPE = identifier ;
  VALUES { center origin near_edge far_edge }
  DEFAULT = origin ;
}
KEYWORD ANTENNA = annotation {                    // See Semantics 172 on page 246
  CONTEXT = arithmetic_model ;
}
SEMANTICS ANTENNA
  CONTEXT { PIN.SIZE PIN.AREA PIN.PERIMETER }
  REFERENCETYPE = ANTENNA;
}
KEYWORD TARGET = annotation {                     // See Semantics 173 on page 246
  CONTEXT = arithmetic_model ;
}
SEMANTICS TARGET {
  CONTEXT = PIN.SIZE;
  REFERENCETYPE = PIN.PATTERN;
}
KEYWORD PATTERN = single_value_annotation {{      // See Semantics 174 on page 247
  CONTEXT = arithmetic_model ;
}
SEMANTICS PATTERN {
  CONTEXT {
    LENGTH WIDTH HEIGHT SIZE AREA THICKNESS
    PERIMETER EXTENSION
  }
  REFERENCETYPE = PATTERN ;
}
KEYWORD HIGH = arithmetic_submodel ;              // See Semantics 175 on page 248

SEMANTICS HIGH { CONTEXT {
    CLASS.VOLTAGE CLASS.LIMIT.VOLTAGE
    PIN.VOLTAGE PIN.LIMIT.VOLTAGE PIN.CAPACITANCE
    PIN.NOISE PIN.NOISE_MARGIN PIN.LIMIT.NOISE
    LIBRARY.NOISE_MARGIN LIBRARY.LIMIT.NOISE
} }
KEYWORD LOW = arithmetic_submodel ;
SEMANTICS LOW { CONTEXT {
    CLASS.VOLTAGE CLASS.LIMIT.VOLTAGE
    PIN.VOLTAGE PIN.LIMIT.VOLTAGE PIN.CAPACITANCE
```

```
    PIN.NOISE PIN.NOISE_MARGIN PIN.LIMIT.NOISE
    LIBRARY.NOISE_MARGIN LIBRARY.LIMIT.NOISE
} }
KEYWORD RISE = arithmetic_submodel ;              // See Semantics 176 on page 248
SEMANTICS RISE { CONTEXT {
    FROM.THRESHOLD TO.THRESHOLD PIN.THRESHOLD
    PIN.CAPACITANCE PIN.SLEWRATE PIN.LIMIT.SLEWRATE
    PIN.PULSEWIDTH PIN.LIMIT.PULSEWIDTH
} }
KEYWORD FALL = arithmetic_submodel ;
SEMANTICS FALL { CONTEXT {
    FROM.THRESHOLD TO.THRESHOLD PIN.THRESHOLD
    PIN.CAPACITANCE PIN.SLEWRATE PIN.LIMIT.SLEWRATE
    PIN.PULSEWIDTH PIN.LIMIT.PULSEWIDTH
} }
KEYWORD HORIZONTAL = arithmetic_submodel ;        // See Semantics 177 on page 249
SEMANTICS HORIZONTAL { CONTEXT {
    WIDTH LENGTH EXTENSION DISTANCE OVERHANG
} }
KEYWORD VERTICAL = arithmetic_submodel ;
SEMANTICS VERTICAL { CONTEXT {
    WIDTH LENGTH EXTENSION DISTANCE OVERHANG
} }
KEYWORD ACUTE = arithmetic_submodel ;
SEMANTICS ACUTE { CONTEXT {
    WIDTH LENGTH EXTENSION DISTANCE OVERHANG
} }
KEYWORD OBTUSE = arithmetic_submodel ;
SEMANTICS OBTUSE { CONTEXT {
    WIDTH LENGTH EXTENSION DISTANCE OVERHANG
} }
```

Annex C

(informative)

ALF library example

This annex shows a sample ALF library.

```
ALF_REVISION "IEEE 1603-2003"

LIBRARY sampleLibrary {

// global units for physical measurements

  TIME { UNIT = 1e-9; }
  FREQUENCY { UNIT = 1e6; }
  DISTANCE { UNIT = 1e-6; }
  AREA { UNIT = 1e-12; }
  VOLTAGE { UNIT = 1; }
  CURRENT { UNIT = 1e-3; }
  ENERGY { UNIT = 1e-12; }
  POWER { UNIT = 1e-3; }
  CAPACITANCE { UNIT = 1e-12; }
  RESISTANCE { UNIT = 1e3; }
  INDUCTANCE { UNIT = 1e-9; }

// global definitions for PVT

  PROCESS {
    TABLE { nom snsp snwp wnsp wnwp }
    DEFAULT = nom;
  }
  DERATE_CASE {
    TABLE { nom bccom wccom bcind wcind bcmil wcmil }
    DEFAULT = nom;
  }
  VOLTAGE VDD {
    HEADER {
      DERATE_CASE {
        TABLE { nom bccom wccom bcind wcind bcmil wcmil }
      }
    }
    TABLE { 1.5 1.7 1.3 1.6 1.4 1.9 1.1 }
    DEFAULT = 1.5;
  }
  TEMPERATURE { MIN = -40; MAX = 125; DEFAULT = 25; }

// global thresholds for timing measurements

  DELAY {
    FROM { THRESHOLD = 0.5; }
    TO { THRESHOLD = 0.5; }
```

```
    }
    SLEWRATE {
      FROM { THRESHOLD { RISE = 0.3; FALL = 0.7; } }
      TO { THRESHOLD { RISE = 0.6; FALL = 0.4; } }
    }

  // templates for cell characterization

    TEMPLATE DelayPowerArc {
      DELAY {
        FROM { PIN = <FromPin>; }
        TO { PIN = <ToPin>; }
        HEADER {
          CAPACITANCE { PIN = <ToPin>; TABLE { 0 0.5 1 } }
          SLEWRATE { PIN = <FromPin>; TABLE { 0.1 1 } }
        } TABLE { <DelayTable> }
      }
      SLEWRATE {
        PIN = <ToPin>;
        HEADER {
          CAPACITANCE { PIN = <ToPin>; TABLE { 0 0.5 1 } }
          SLEWRATE { PIN = <FromPin>; TABLE { 0.1 1 } }
        } TABLE { <SlewTable> }
      }
      RESISTANCE = <Rdriver> {
        PIN = <ToPin>;
      }
      ENERGY {
        HEADER {
          CAPACITANCE { PIN = <ToPin>; TABLE { 0 0.5 1 } }
          SLEWRATE { PIN = <FromPin>; TABLE { 0.1 1 } }
        } TABLE { <PowerTable> }
      }
    }
    TEMPLATE NoisePropagation {
      NOISE {
        PIN = <ToPin>;
        HEADER {
          NOISE H { PIN = <FromPin>; }
          PULSEWIDTH W { PIN = <FromPin>; }
          CAPACITANCE C { PIN = <ToPin>; }
        } EQUATION { <NoiseHight> }
      }
      PULSEWIDTH {
        PIN = <ToPin>;
        HEADER {
          NOISE H { PIN = <FromPin>; }
          PULSEWIDTH W { PIN = <FromPin>; }
          CAPACITANCE C { PIN = <ToPin>; }
        } EQUATION { <NoiseWidth> }
      }
      DELAY {
        FROM { PIN = <FromPin>; }
        TO { PIN = <ToPin>; }
```

```
      HEADER {
        NOISE H { PIN = <FromPin>; }
        PULSEWIDTH W { PIN = <FromPin>; }
        CAPACITANCE C { PIN = <ToPin>; }
      } EQUATION { <NoiseDelay> }
    }
  }
  TEMPLATE SetupHold {
    SETUP {
      FROM { PIN = <DataPin>; EDGE_NUMBER = 0; }
      TO { PIN = <ClockPin>; }
      HEADER {
        SLEWRATE s1 { PIN = <DataPin>; EDGE_NUMBER = 0; TABLE { 0.1 1 } }
        SLEWRATE s2 { PIN = <ClockPin>; TABLE { 0.1 0.2 0.3 } }
      } TABLE { <SetupTable> }
    }
    HOLD {
      TO { PIN = <ClockPin>; }
      FROM { PIN = <DataPin>; EDGE_NUMBER = 1; }
      HEADER {
        SLEWRATE s1 { PIN = <DataPin>; EDGE_NUMBER = 1; TABLE { 0.1 1 } }
        SLEWRATE s2 { PIN = <ClockPin>; TABLE { 0.1 0.2 0.3 } }
      } TABLE { <HoldTable> }
    }
    NOISE_MARGIN = <DataNoiseMargin> {
      PIN = <DataPin>;
      TIME {
        FROM { PIN = <DataPin>; EDGE_NUMBER = 0; }
        TO { PIN = <DataPin>; EDGE_NUMBER = 1; }
      }
    }
  }

// example of combinatorial circuit

  CELL sampleNand2 {
    CELLTYPE = combinational;
    PIN A { DIRECTION = input; CAPACITANCE = 0.01; }
    PIN B { DIRECTION = input; CAPACITANCE = 0.01; }
    PIN Y { DIRECTION = output; LIMIT { CAPACITANCE { MAX = 1.0; } } }
    FUNCTION {
      BEHAVIOR { Y = ! ( A & B ); }
    }
    GROUP AnyInput { A B }
    VECTOR ( 01 AnyInput -> 10 Y ) {
      DelayPowerArc {
        FromPin = AnyInput;
        ToPin = Y;
        Rdriver = 1;
        DelayTable { 0 0.5 1 0 0.5 1 }
        SlewTable { 0 0.5 1 0 0.5 1 }
        PowerTable { 1 1 1 1 1 1 }
      }
    }
```

```
      VECTOR ( 10 AnyInput -> 01 Y ) {
        DelayPowerArc {
          FromPin = AnyInput;
          ToPin = Y;
          Rdriver = 1;
          DelayTable { 0 0.5 1 0 0.5 1 }
          SlewTable { 0 0.5 1 0 0.5 1 }
          PowerTable { 1 1 1 1 1 1 }
        }
      }
      VECTOR ( 1* AnyInput -> *1 AnyInput <&> 0* Y -> *0 Y ) {
        NoisePropagation = dynamic {
          FromPin = AnyInput;
          ToPin = Y;
          NoiseHeight = 0.5*H / (C*(W + 1));
          NoiseWidth  = 0.5*H*W / (C + 1);
          NoiseDelay  = 0.5*C ;
        }
      }
      VECTOR ( 0* AnyInput -> *0 AnyInput <&> 1* Y -> *1 Y ) {
        NoisePropagation = dynamic {
          FromPin = AnyInput;
          ToPin = Y;
          NoiseHeight = 0.5*H / (C*(W + 1));
          NoiseWidth  = 0.5*H*W / (C + 1);
          NoiseDelay  = 0.5*C ;
        }
      }
    } // end CELL sampleNand2

  // example of sequential circuit

    CELL sampleDFlipFlop {
      CELLTYPE = flipflop;
      PIN D {
        DIRECTION = input;
        SIGNALTYPE = data;
        CAPACITANCE = 0.01;
      }
      PIN C {
        DIRECTION = input;
        SIGNALTYPE = clock;
        POLARITY = rising_edge;
        CAPACITANCE = 0.01;
        NOISE_MARGIN { LOW = 0.1; HIGH = 0.2; }
        LIMIT { SLEWRATE { MAX = 0.3; } }
      }
      PIN Q {
        DIRECTION = output;
        SIGNALTYPE = data;
        LIMIT { CAPACITANCE { MAX = 1.0; } }
      }
      FUNCTION {
        BEHAVIOR {
```

```
        @ ( 01 C ) { Q = D; }
      }
    }
    VECTOR ( 01 C -> ?! Q ) {
      DelayPowerArc {
        FromPin = C;
        ToPin = Q;
        Rdriver = 1;
        DelayTable { 0 0.5 1 0 0.5 1 }
        SlewTable { 0 0.5 1 0 0.5 1 }
        PowerTable { 1 1 1 1 1 1 }
      }
    }
    VECTOR ( ?! D -> 01 C -> ?! D ) {
      SetupHold {
        DataPin = D;
        ClockPin = C;
        DataNoiseMargin = 0.2;
        SetupTable { 1 1 1 1 1 1 }
        HoldTable { 0 0 0 0 0 0 }
      }
    }
  } // end CELL sampleDFlipFlop

// template for parametrized megacell

  TEMPLATE \1PortAsyncRAM {
    CELL <RAMInstance> {
      CELLTYPE = memory;
      GROUP Addr { <AddrPins> }
      GROUP Din  { <DataInputs> }
      GROUP Dout { <DataOutputs> }
      PIN Addr { DIRECTION=input;  SIGNALTYPE=address; VIEW=physical; }
      PIN Din  { DIRECTION=input;  SIGNALTYPE=data;    VIEW=physical; }
      PIN Dout { DIRECTION=output; SIGNALTYPE=data;    VIEW=physical; }
      PIN WE   { DIRECTION=input;  SIGNALTYPE=enable;  POLARITY = high; }
      PIN [<DataHigh>:<DataLow>] DataArray [<Rows>:1] {
        DIRECTION=none; VIEW = none;
      }
      PINGROUP [<AddrHigh>:<AddrLow>] AddrBus {
        MEMBERS { <AddrPins> }    VIEW = functional;
      }
      PINGROUP [<DataHigh>:<DataLow>] DinBus  {
        MEMBERS { <DataInputs> }  VIEW = functional;
      }
      PINGROUP [<DataHigh>:<DataLow>] DoutBus {
        MEMBERS { <DataOutputs> } VIEW = functional;
      }
      FUNCTION {
        BEHAVIOR {
          DoutBus = DataArray[AddrBus];
          @ ( WE ) { DataArray[AddrBus] = DinBus; }
        }
      }
```

```
        GROUP AddrIndex { <AddrLow> : <AddrHigh> }
        GROUP DataIndex { <DataLow> : <DataHigh> }
        VECTOR ( 01 AddrBus[AddrIndex] -> ?? DoutBus[DataIndex] ) {
          DELAY {
            FROM { PIN = AddrBus[AddrIndex]; }
            TO { PIN = DoutBus[DataIndex]; }
            HEADER {
              CAPACITANCE { PIN = DoutBus[DataIndex]; }
            } EQUATION { <Doe0> + <Doe1> * CAPACITANCE }
          }
        }
        VECTOR ( ?! DinBus[DataIndex] -> ?! DoutBus[DataIndex] ) {
          EXISTENCE_CONDITION = WE;
          DELAY {
            FROM { PIN = DinBus[DataIndex]; }
            TO { PIN = DoutBus[DataIndex]; }
            HEADER {
              CAPACITANCE { PIN = DoutBus[DataIndex]; }
            } EQUATION { <Dio0> + <Dio1> * CAPACITANCE }
          }
        }
        VECTOR ( *? AddrBus[AddrIndex] -> 01 WE
        -> 10 WE -> ?* AddrBus[AddrIndex] ) {
          SETUP = <AddrSetup> {
            FROM { PIN = AddrBus[AddrIndex]; EDGE_NUMBER = 0; }
            TO { PIN = WE; EDGE_NUMBER = 0; }
          }
          HOLD = <AddrHold> {
            FROM { PIN = WE; EDGE_NUMBER = 1; }
            TO   { PIN = AddrBus[AddrIndex]; EDGE_NUMBER = 1; }
          }
        }
      } // end TEMPLATE \1PortAsyncRAM

    // instance of parametrized megacell

      \1PortAsyncRAM {
        RAMInstance = \1PortAsyncRAM64X8 ;
        AddrPins { Addr5 Addr4 Addr3 Addr2 Addr1 Addr0 }
        DataInputs { Din7 Din6 Din5 Din4 Din3 Din2 Din1 Din0 }
        DataOutputs { Dout7 Dout6 Dout5 Dout4 Dout3 Dout2 Dout1 Dout0 }
        AddrHigh = 5; AddrLow  = 0;
        DataHigh = 7; DataLow  = 0;
        Rows = 64;
        Doe0 = 1; Doe1 = 1;
        Dio0 = 1; Dio1 = 1;
        AddrSetup = 1;
        AddrHold  = 1;
      }

    }
```

Annex D

(informative)

Bibliography

[B1] The IEEE Standard Dictionary of Electrical and Electronics Terms, Sixth Edition.

[B2] OVI, *Advanced Library Format,* Version 1.1.

[B3] Accellera, *Advanced Library Format,* Version 2.0.

[B4] Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, Ca., *SPICE 2G6 User's Guide*.

[B5] IEEE Std. 1481-1999, IEEE Standard for Delay and Power Calculation for Integrated Circuit Design, Clause 9.

[B6] Zvi Kohavi: *Switching and Finite Automata Theory,* McGraw-Hill Publishing Company, ISBN 0-07-035310-7

[B7] Matthew W. Crocker: *Computational Psycholinguistics - An Interdisciplinary Approach to the Study of Language,* Kluwer 1996, ISBN 0-7923-3802-2

[B8] Stroustrup, Bjarne : *The C++ Programming Language (Third Edition and Special Edition)*, Addison-Wesley, ISBN 0-201-88954-4 and 0-201-70073-5.

[B9] Weste, N. H. E., and Eshraghian, K.: *Principles of CMOS VLSI Design*, Addison-Wesley, 1985, 1990, ISBN 0-201-08222-5.