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

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

IEEE P1603 Draft 4

April 17, 2002

put in IEEE verbage

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

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

Revision history:

IEEE P1596 Draft 0	August 19, 2001
IEEE P1603 Draft 1	September 17, 2001
IEEE P1603 Draft 2	November 12, 2001
IEEE P1603 Draft 3	January 4, 2002

Table of Contents

List of Figures

List of Tables

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

1. Introduction

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

1.1 Motivation

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

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

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

1.2 Goals

The basic goals of the proposed library standard are

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

1.3 Target applications

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

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

Design automation enabled tremendous progress in efficiency, productivity, and the ability to deal with complexity, yet it did not change the fundamental requirements for ASIC design. Therefore, ALF needs to provide models with *functional* information and *performance* information, primarily including timing and power. Signal integrity characteristics, such as noise margin can also be included under performance category. Such information is typically found in any databook for analog cells. At deep sub-micron levels, digital cells behave similar to analog cells as electronic devices bound by physical laws and therefore are not infinitely robust against noise.

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

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

Table 1—Target applications and models supported by ALF

Application	Functional model	Performance model	Physical model
Simulation	Derived from ALF	N/A	N/A
Synthesis	Supported by ALF	Supported by ALF	Supported by ALF
Design for test	Supported by ALF	N/A	N/A

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

Application	Functional model	Performance model	Physical model
Design planning	Supported by ALF	Supported by ALF	Supported by ALF
Timing analysis	N/A	Supported by ALF	N/A
Power analysis	N/A	Supported by ALF	N/A
Signal integrity	N/A	Supported by ALF	N/A
Layout	N/A	N/A	Supported by ALF

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

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

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

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

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

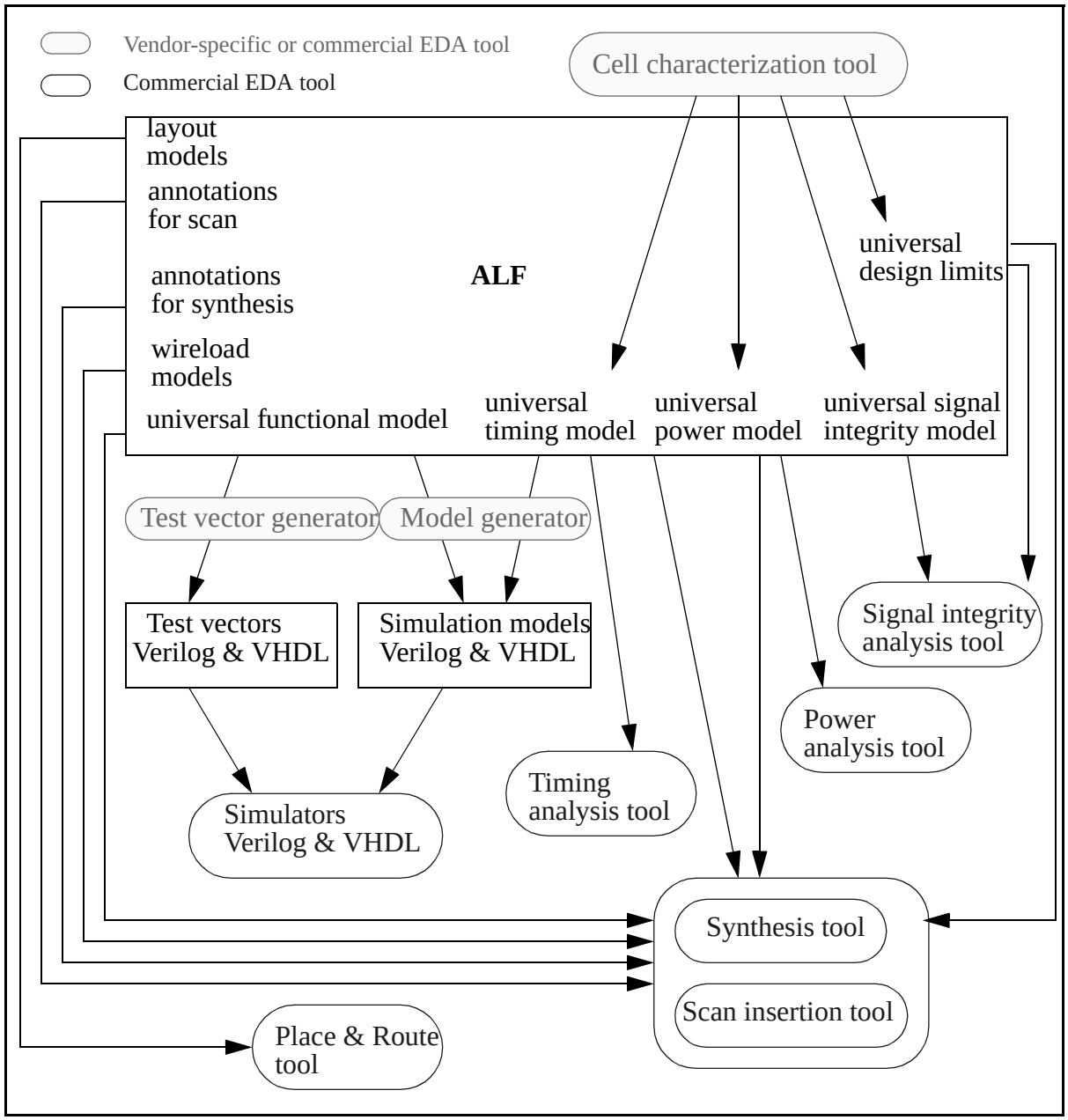


Figure 1—ALF and its target applications

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

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

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

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

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

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

1.4 Conventions

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

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

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

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

1.5 Contents of this standard

The organization of the remainder of this standard is

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

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

2. References

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

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

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

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

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

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

3. Definitions

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

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

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

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

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

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

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

3.6 ...

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

3.8 ...

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

3.10 ...

4. Acronyms and abbreviations

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

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

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

5. ALF language construction principles and overview

Add lead-in text

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

5.1 ALF meta-language

The following Syntax 1— establishes an ALF meta-language.

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

Syntax 1—syntax construction for ALF meta-language

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

The *ALF type* is defined by a *keyword* (see Section 6.12 on page 36) eventually in conjunction with an *index* (see Section 7.8 on page 40) or by the *operator* "**@**" (Section 6.4 on page 28) or by the *delimiter* "**:**" (see Section 6.3 on page 27). The usage of keyword, index, operator, or delimiter as ALF type is defined by ALF language rules concerning the particular ALF type.

The *ALF name* is defined by an *identifier* (see Section 6.11 on page 34) eventually in conjunction with an index or by a *control expression* (see Section 10.9 on page 160). Depending on the ALF type, the ALF name is mandatory or optional or not applicable. The usage of identifier, index, or control expression as ALF name is defined by ALF language rules concerning the particular ALF type.

The *ALF value* is defined by an identifier, a *number* (see Section 6.5 on page 31), an *arithmetic expression* (see Section 11.1 on page 163), a *boolean expression* (see Section 10.7 on page 159), or a control expression. Depending on the type of the ALF statement, the ALF value is mandatory or optional or not applicable. The usage of identifier, number, arithmetic expression, boolean expression or control expression as ALF value is defined by ALF language rules concerning the particular ALF type.

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

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

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

Examples

a) ALF statement describing an unnamed object without value:

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

b) ALF statement describing an unnamed object with value:

```
ARBITRARY_ALF_TYPE = arbitrary_ALF_value;
```

or

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

c) ALF statement describing a named object without value:

```
ARBITRARY_ALF_TYPE arbitrary_ALF_name;
```

or

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

d) ALF statement describing a named object with value:

```
ARBITRARY_ALF_TYPE arbitrary_ALF_name = arbitrary_ALF_value;
```

or

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

5.2 Categories of ALF statements

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

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

Table 2—Categories of ALF statements

category	purpose	syntax particularity
generic object	provide a definition for use within other ALF statements	Statement is atomic, semi-compound or compound. Name is mandatory. Value is either mandatory or not applicable.
library-specific object	describe the contents of a IC technology library	Statement is atomic or compound. Name is mandatory. Value does not apply. Category of parent is exclusively *library-specific object*
arithmetic model	describe an abstract mathematical quantity that can be calculated and eventually measured within the design of an IC	Statement is atomic or compound. Name is optional. Value is mandatory, if atomic.
arithmetic submodel	describe an arithmetic model under a specific measurement condition	Statement is atomic or compound. Name does not apply. Value is mandatory, if atomic. Category of parent is exclusively *arithmetic model*
arithmetic model container	provide a context for an arithmetic model	Statement is compound. Name and value do not apply. Category of child is exclusively *arithmetic model*
geometric model	describe an abstract geometrical form used in physical design of an IC	Statement is semi-compound or compound. Name is optional. Value does not apply.
annotation	provide a qualifier or a set of qualifiers for an ALF statement	Statement is atomic, semi-compound or compound. Name does not apply. Value is mandatory, if atomic or compound. Value does not apply, if semi-compound. Category of child is exclusively *annotation*
annotation container	provide a context for an annotation	Statement is compound. Name and value do not apply. Category of child is exclusively *annotation*
auxiliary statement	provide an additional description within the context of a library-specific object, an arithmetic model, an arithmetic submodel, geometric model or another auxliary statement	dependent on subcategory

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

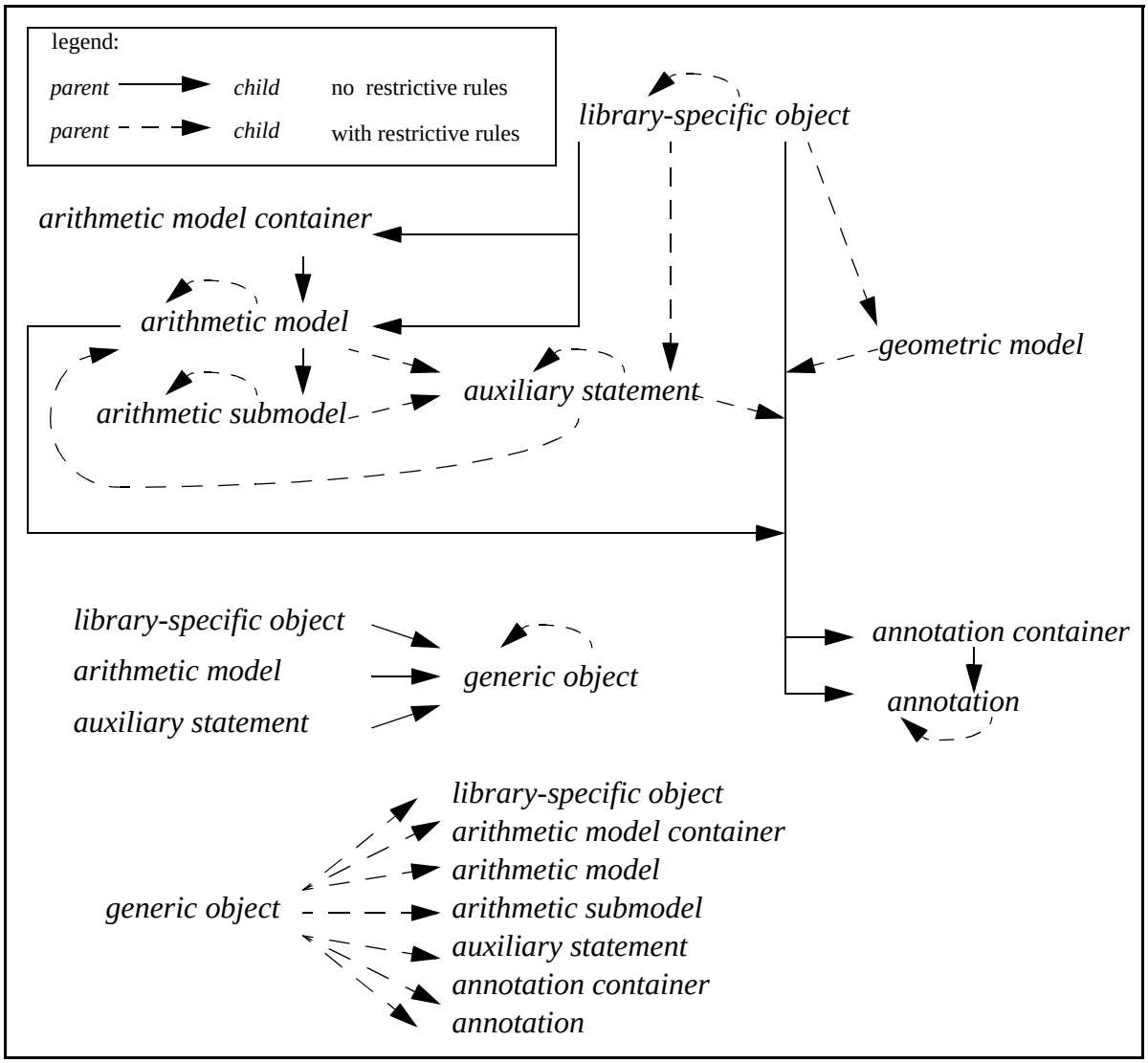


Figure 2—Parent/child relationship between ALF statements

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

5.3 Generic objects and library-specific objects

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

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

Table 3—Generic objects

keyword	item	section
ALIAS	alias declaration	
CONSTANT	constant declaration	
CLASS	class declaration	
GROUP	group declaration	
KEYWORD	keyword declaration	
TEMPLATE	template declaration	

The following 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 4—Library-specific objects

keyword	item	section
LIBRARY	library	
SUBLIBRARY	sublibrary	
CELL	cell	
PRIMITIVE	primitive	
WIRE	wire	
PIN	pin	
PINGROUP	pin group	
VECTOR	vector	
NODE	node	
LAYER	layer	
VIA	via	
RULE	rule	
ANTENNA	antenna	
SITE	site	

Table 4—Library-specific objects

keyword	item	section
ARRAY	array	
BLOCKAGE	blockage	
PORT	port	
PATTERN	pattern	
REGION	region	new proposal for IEEE

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

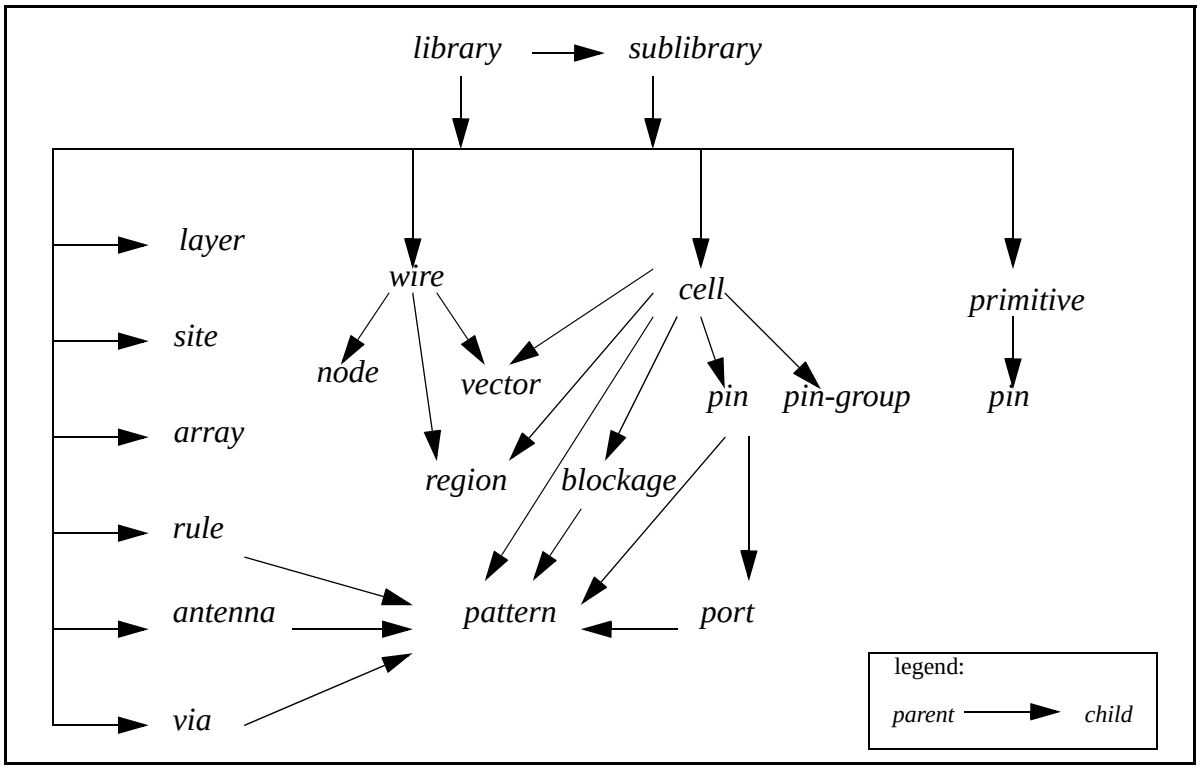


Figure 3—Parent/child relationship amongst library-specific objects

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

5.4 Singular statements and plural statements

Auxiliary statements with predefined keywords are devided 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*.

The following Table 5— lists the singular statements.

Table 5—Singular statements

keyword	item	value	complexity	section
FUNCTION	function	N/A	compound	
TEST	test	N/A	compound	
RANGE	range	N/A	semi-compound	
FROM	from	N/A	compound	
TO	to	N/A	compound	
VIOLATION	violation	N/A	compound	
HEADER	header	N/A	compound (or semi-compound?)	
TABLE	table	N/A	semi-compound	
EQUATION	equation	N/A	semi-compound	
BEHAVIOR	behavior	N/A	compound	
STRUCTURE	structure	N/A	compound	
NON_SCAN_CELL	non-scan cell	optional	compound or semi-compound	
ARTWORK	artwork	mandatory	compound or atomic	

The following Table 6— lists the plural statements.

Table 6—Plural statements

keyword	item	name	complexity	section
STATETABLE	state table	optional	semi-compound	
@	control statement	mandatory	compound	
:	alternative control statement	mandatory	compound	

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

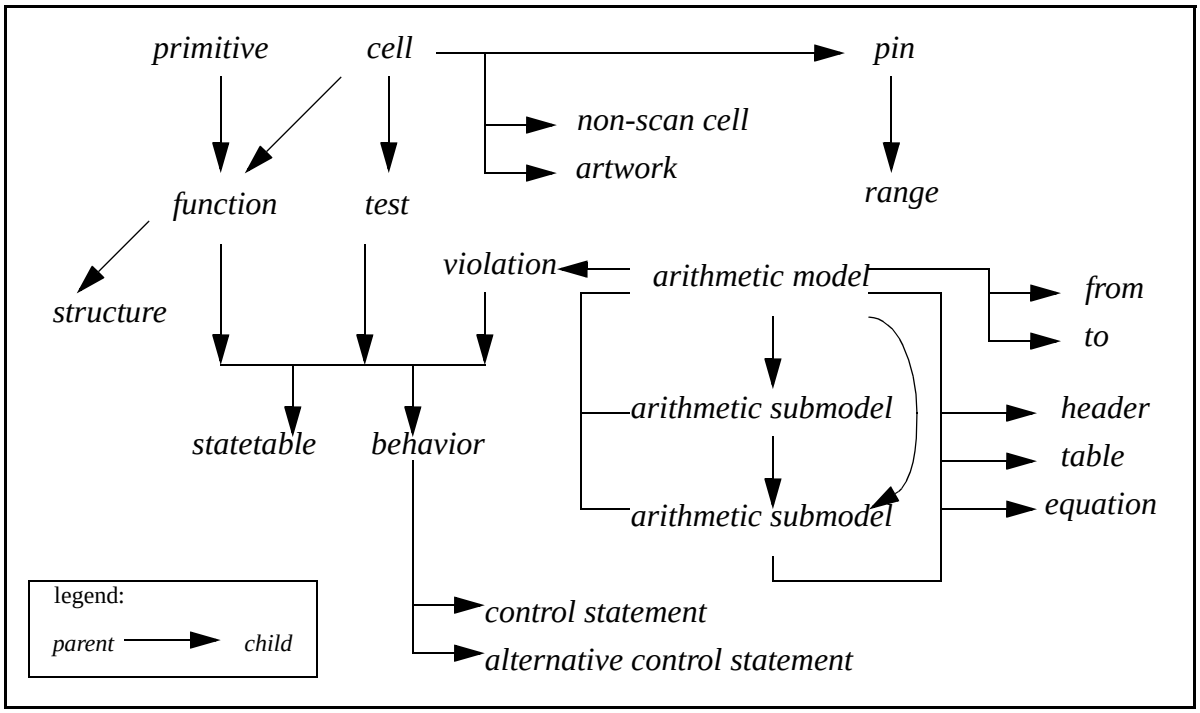


Figure 4—Parent/child relationship involving singular statements and plural statements

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

5.5 Instantiation statement and assignment statement

Auxiliary statements without predefined keywords use the name of an object as keyword. Such statements are devided 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.

The following Table 7— lists the instantiation statements.

Table 7—Instantiation statements

item	name	value	section
cell instantiation	optional	N/A	
primitive instantiation	optional	N/A	
template instantiation	N/A	optional	
via instantiation	mandatory	N/A	
wire instantiation	mandatory	N/A	proposed for IEEE

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

The following Table 8— lists the assignment statements.

Table 8—Assignment statements

item	section
pin assignment	
boolean assignment	
arithmetic assignment	

The following Figure 5— illustrates the parent/child relationship involving instantiation and assigment statements.

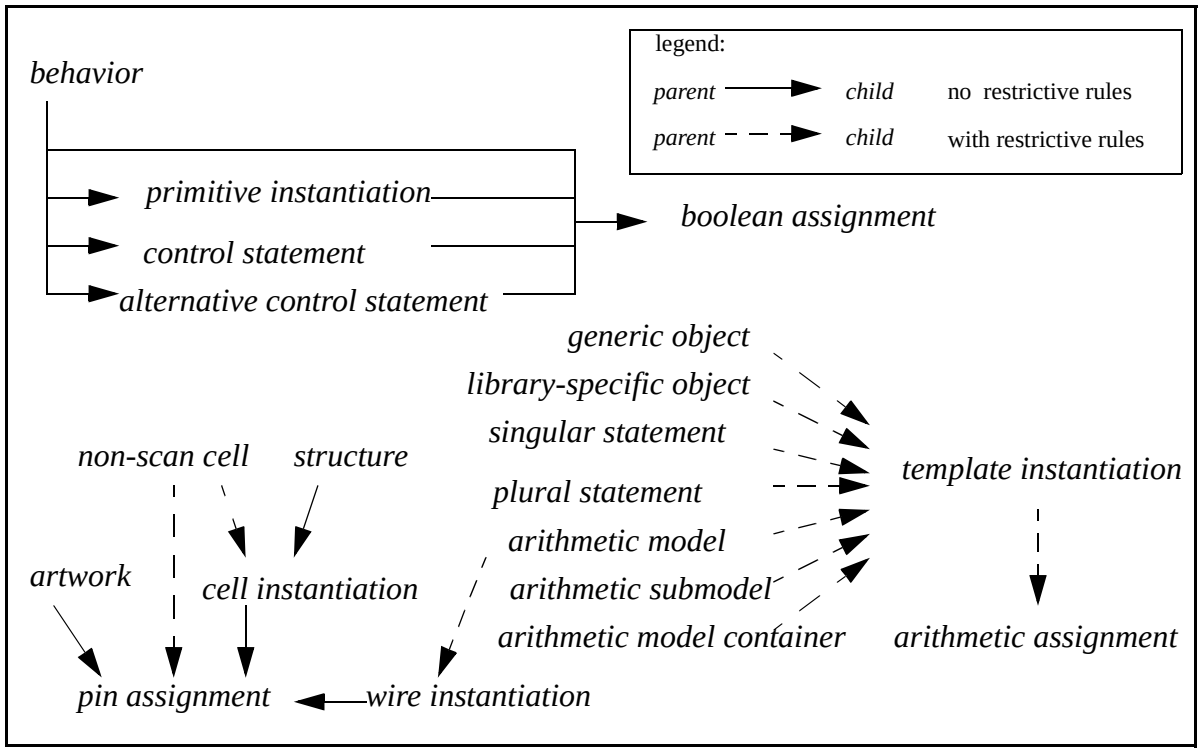


Figure 5—Parent/child relationship involving instantiation and assignment statements

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

5.6 Annotation, arithmetic model, and related statements

Multiple keywords are predefined in the categories *arithmetic model*, *arithmetic model container*, *arithmetic submodel*, *annotation*, *annotation container*, and *geometric model*. Their semantics are established within the context of their parent. Therefore they are called *context-sensitive keywords*. In addition, the ALF language allows additional definition of keywords in these categories.

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

Table 9—Other categories of ALF statements

item	section
arithmetic model	
arithmetic submodel	
arithmetic model container	
annotation	
annotation container	
geometric model	

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

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

Table 10—Annotations and annotation containers with generic keyword

keyword	item / subcategory	section
PROPERTY	one_level_annotation_container	
ATTRIBUTE	multi_value_annotation	
INFORMATION	one_level_annotation_container	

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

Table 11—Keywords related to arithmetic model

keyword	item / category	section
LIMIT	arithmetic model container	

Table 11—Keywords related to arithmetic model

keyword	item / category	section
MIN	arithmetic submodel, operator within *arithmetic expression*	
MAX	arithmetic submodel, operator within *arithmetic expression*	
TYP	arithmetic submodel	
DEFAULT	arithmetic submodel, annotation	
ABS	operator within *arithmetic expression*	
EXP	operator within *arithmetic expression*	
LOG	operator within *arithmetic expression*	

The definitions of other predefined keywords, especially in the category arithmetic model, can be self-described in ALF using the *keyword declaration* statement (see Section 8.4 on page 48).

5.7 Statements for parser control

The following provides a reference to statements used for ALF parser control.

Table 12—Statements for ALF parser control

keyword	statement	section
INCLUDE	include statement	
ASSOCIATE	associate statement	
ALF_REVISION	revision statement	

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

5.8 Name space and visibility of statements

The following rules for name space and visibility shall apply:

a) A statement shall be visible within its parent statement, but not outside its parent statement.
b) A statement visible within another statement shall also be visible within a child of that other statement.
c) All objects (i.e., generic objects and library-specific objects) shall share a common name space within their scope of visibility. No object shall use the same name as any other visible object. Conversely, an object may 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 may 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 may use the same optional name as any other statement with optional name outside the scope of its visibility.

6. Lexical rules

This section discusses the lexical rules.

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

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

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

6.1 Character set

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

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

```
character ::=
      whitespace
    | letter
    | digit
    | special
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 ::=
      & | | | ^ | ~ | + | - | * | / | % | ? | ! | : | ; | , | ” | ’ | @ | = | \ | . | $ | _ | #
    | ( | ) | < | > | [ | ] | { | }
whitespace ::=
      space | vertical_tab | horizontal_tab | new_line | carriage_return | form_feed
```

Syntax 2—ASCII character

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

Table 13—List of whitespace characters

Name	ASCII code (octal)
space	200
horizontal tab	011
new line	012
vertical tab	013

Table 13—List of whitespace characters (Continued)

Name	ASCII code (octal)
form feed	014
carriage return	015

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

Table 14—List of special characters

Symbol	Name
&	amperesand
|	??? bar
^	??? hyphen
~	tilde
+	plus
-	minus
*	asterix
/	divider
%	percent
?	question mark
!	exclamation mark
:	colon
;	semicolon
,	comma
”	double quote
’	single quote
@	??? at
=	equal
\	escape character
.	dot
$	dollar
_	underscore
#	??? sharp
(|)	parenthesis (open | close)
< | >	angular bracket (open | close)

Table 14—List of special characters (Continued)

Symbol	Name
[| **]**	square bracket (open | close)
{ | **}**	curly brace (open | close)

6.2 Comment

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

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

Syntax 3—Comment

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

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

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

6.3 Delimiter

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

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

Syntax 4—Delimiter

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

6.4 Operator

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

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

Syntax 5—Operator

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

6.4.1 Arithmetic operator

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

Table 15—List arithmetic operators

Symbol	Operator name	unary / binary	section
+	plus	binary	
-	minus	both	
*	multiply	binary	
/	divide	binary	
%	modulo	binary	
**	power	binary	

Arithmetic operators shall be used to specify arithmetic operations.

6.4.2 Boolean operator

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

Table 16—List of boolean operators

Symbol	Operator name	unary / binary	section
!	logical invert	unary	
&&	logical and	binary	
||	logical or	binary	
~	vector invert	unary	
&	vector and	both	
~&	vector nand	both	
|	vector or	both	
~|	vector nor	both	
^	exclusive or	both	
~^	exclusive nor	both	

Boolean operators shall be used to specify boolean operations.

6.4.3 Relational operator

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

Table 17—List of relational operators

Symbol	Operator name	unary / binary	section
==	equal	binary	
!=	not equal	binary	
>	greater	binary	
<	lesser	binary	
>=	greater or equal	binary	
<=	lesser or equal	binary	

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

6.4.4 Shift operator

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

Table 18—List of shift operators

Symbol	Operator name	unary / binary	section
<<	shift left	binary	
>>	shift right	binary	

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

6.4.5 Event sequence operator

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

Table 19—List of event sequence operators

Symbol	Operator name	unary / binary	section
->	immediately followed by	binary	
~>	eventually followed by	binary	
<->	immediately following each other	binary	
<~>	eventually following each other	binary	
&>	simultaneous or immediately followed by	binary	
<&>	simultaneous or immediately following each other	binary	

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

6.4.6 Meta operator

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

Table 20—List of meta operators

Symbol	Operator name	unary / binary	section
=	assignment	binary	
?	condition	binary	
@	control	unary	

Meta operators shall be used to specify transactions between variables.

6.5 Number

Numbers shall be divided into subcategories *signed number* and *unsigned number*, as shown in Syntax 6.

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

Syntax 6—Signed and unsigned numbers

Alternatively, numbers shall be divided into subcategories *integer* and *real*, as shown in Syntax 7—.

```
number ::=
      integer | real
integer ::=
      signed_integer | unsigned_integer
real ::=
      signed_real | unsigned_real
```

Syntax 7—Integer and real numbers

Numbers shall be used to represent numerical quantities.

6.6 Unit symbol

A *unit symbol* shall be defined as shown in .

```
unit_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 8—Unit symbol

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

Table 21—UNIT symbol

leading character	lexical value	numerical value
F	femto	1e-15
P	pico	1e-12
N	nano	1e-9
U	micro	1e-6
M	milli	1e-3
unity	one	1
K	kilo	1e+3
MEG	mega	1e+6
G	giga	1e+9

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

6.7 Bit literal

Bit literals shall be divided into subcategories *numeric bit literal* and *symbolic bit literal*, as shown in Syntax 9.

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

Syntax 9—Bit literal

Bit literals shall be used to specify scalar values within a boolean system.

6.8 Based literal

Based literals shall be divided into subcategories *binary based literal*, *octal based literal*, *decimal based literal*, and *hexadecimal based literal*, as shown in Syntax 10.

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

Syntax 10—Based literal

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

6.9 Edge literal

Edge literals shall be divided into subcategories *bit edge literal*, *based edge literal*, and *symbolic edge literal*, as shown in Syntax 11—.

```
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 11—Edge literal

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

6.10 Quoted string

A *quoted string* shall be a sequence of zero or more characters enclosed between two double quote characters, as shown in Syntax 12.

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

Syntax 12—Quoted string

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

Table 22—Character symbols within a quoted string

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

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

6.11 Identifier

Identifiers shall be divided into the subcategories *non-escaped identifier*, *escaped identifier*, *placeholder identifier*, and *hierarchical identifier*, as shown in Syntax 13.

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

Syntax 13—Identifier

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

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

6.11.1 Non-escaped identifier

A *non-escaped identifier* shall be defined as shown in Syntax 14.

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

Syntax 14—Non-escaped identifier

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

6.11.2 Escaped identifier

An *escaped identifier* shall be defined as shown in Syntax 15.

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

Syntax 15—Escaped identifier

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

6.11.3 Placeholder identifier

A *placeholder identifier* shall be defined as a non-escaped identifier enclosed by angular brackets without whitespace, as shown in Syntax 16.

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

Syntax 16—Placeholder identifier

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

6.11.4 Hierarchical identifier

A *hierarchical identifier* shall be defined as shown in Syntax 17.

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

Syntax 17—Hierarchical identifier

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

Example

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

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

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

6.12 Keyword

Keywords shall be lexically equivalent to non-escaped identifiers. Predefined keywords are listed in Table 3—, Table 4—, Table 5—, Table 6—, Table 10—, and Table 11—. Additional keywords are predefined in section ...

The predefined keywords in this standard follow a more restrictive lexical rule than general non-escaped identifiers, as shown in Syntax 18—.

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

Syntax 18—Keyword

**Should this be a normative rule or a recommended practice to follow for additional keyword definitions? **

Note: This document presents keywords in all-uppercase letters for clarity.

6.13 Rules for whitespace usage

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

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

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

6.14 Rules against parser ambiguity

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

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

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

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

7. Auxiliary Syntax Rules

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

7.1 All-purpose value

An *all-purpose value* shall be defined as shown in Syntax 19.

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

Syntax 19—All purpose value

7.2 Unit value

A *unit value* shall be defined as shown in .

```
unit_value ::=
      unsigned_number | unit_symbol
```

Syntax 20—Unit value

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

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

7.3 String

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

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

Syntax 21—String value

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

7.4 Arithmetic value

An *arithmetic value* shall be defined as shown in Syntax 22.

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

Syntax 22—Arithmetic value

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

7.5 Boolean value

A *boolean value* shall be defined as shown in Syntax 23.

```
boolean_value ::=
    bit_literal | based_literal | unsigned_integer
```

Syntax 23—Boolean value

A boolean value shall represent the contents of a pin variable (see Section 7.9 on page 41).

7.6 Edge value

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

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

Syntax 24—Edge value

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

7.7 Index value

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

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

Syntax 25—Index value

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

7.8 Index

An *index* shall be defined as shown in Syntax 26.

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

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

Syntax 26—Index

7.9 Pin variable and pin value

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

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

Syntax 27—Pin variable

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

A pin value shall represent the actual value or a pointer to the actual value associated with a pin variable. The actual value is a boolean value. A pin variable represents a pointer to the actual value.

7.10 Pin assignment

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

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

Syntax 28—Pin assignment

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

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

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

7.11 Annotation

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

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

Syntax 29—Annotation

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

7.12 Annotation container

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

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

Syntax 30—Annotation container

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

7.13 ATTRIBUTE statement

An *attribute* statement shall be defined as shown in Syntax 31.

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

Syntax 31—ATTRIBUTE statement

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

Example

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

7.14 PROPERTY statement

A *property* statement shall be defined as shown in Syntax 32.

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

Syntax 32—PROPERTY statement

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

Example

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

7.15 INCLUDE statement

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

include ::=
INCLUDE quoted_string **;**

Syntax 33—INCLUDE statement

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

Example

```
LIBRARY myLib {
     INCLUDE “templates.alf”;
```

```
        INCLUDE “technology.alf”;
        INCLUDE “primitives.alf”;
        INCLUDE “wires.alf”;
        INCLUDE “cells.alf”;
    }
```

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

7.16 REVISION statement

A *revision statement* shall be defined as shown in Syntax 29

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

Syntax 34—Revision statement

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

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

Table 23—Legal string values within the REVISION statement

string value	revision or version
“1.1”	Version 1.1 by Open Verilog International, released on April 6, 1999
“2.0”	Version 2.0 by Accellera, released on December 14, 2000
“P1603.2002-04-16”	IEEE draft version as described in this document
TBD	IEEE 1603 release version

The revision statement shall be optional, as the application program parsing the ALF file may provide other means of specifying the revision or version of the file to be parsed. If a revision statement is encountered while a revision has already been specified to the parser (e.g. if an included file is parsed), the parser shall be responsible to decide whether the newly encountered revision is compatible with the originally specified revision and then either proceed assuming the original revision or abandon.

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

7.17 Generic object

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

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

Syntax 35—Generic object

7.18 Library-specific object

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

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

Syntax 36—Library-specific object

7.19 All purpose item

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

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

Syntax 37—All purpose item

8. Generic objects and related statements

Add lead-in text

8.1 ALIAS declaration

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

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

Syntax 38—ALIAS declaration

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

Example

```
ALIAS reset = clear;
```

8.2 CONSTANT declaration

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

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

Syntax 39—CONSTANT declaration

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

Example

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

8.3 CLASS declaration

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

```
class_declaration ::=
    CLASS class_identifier ;
  | CLASS identifier { all_purpose_items }
```

Syntax 40—CLASS declaration

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

Example

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

8.4 KEYWORD declaration

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

keyword_declaration ::=
KEYWORD *keyword*_identifier **=** *syntax_item*_identifier **;**
| **KEYWORD** *keyword*_identifier **=** *syntax_item*_identifier **{** annotation { annotation } **}**

Syntax 41—KEYWORD declaration

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

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

Table 24—Syntax item identifier

identifier	semantic meaning
annotation	The keyword shall specify an *annotation* (see Section 7.11)
single_value_annotation	The keyword shall specify a *single value annotation* (see Section 7.11)
multi_value_annotation	The keyword shall specify a *multi_value_annotation* (see Section 7.11)
annotation_container	The keyword shall specify an *annotation container* (see Section 7.12)
arithmetic_model	The keyword shall specify an *arithmetic model* (see)
arithmetic_submodel	The keyword shall specify an *arithmetic submodel* (see)
arithmetic_model_container	The keyword shall specify an *arithmetic model container* (see)

8.5 Annotations for a KEYWORD

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

8.5.1 VALUETYPE annotation

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

Table 25—VALUETYPE annotation

syntax item identifier	set of legal values for VALUETYPE	default value for VALUETYPE	comment
annotation or single_value_annotation or multi_value_annotation	number, identifier, quoted_string, edge_value, pin_variable, control_expression, boolean_expression, arithmetic_expression	identifier	see Syntax 29, definition of *annotation value*
annotation_container	N/A	N/A	an *annotation container* (see Syntax 30) has no value
arithmetic_model	number, identifier, bit_literal, based_literal	number	see Syntax 22, definition of *arithmetic value*
arithmetic_submodel	N/A	N/A	an *arithmetic submodel* (see) shall always have the same valuetype as its parent arithmetic mdel
arithmetic_model_container	N/A	N/A	an *arithmetic model container* (see) has no value

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

Example:

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

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

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

8.5.2 VALUES annotation

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

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

Example:

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

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

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

8.5.3 DEFAULT annotation

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

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

Example:

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

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

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

8.5.4 CONTEXT annotation

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

Example:

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

```
     }
}
```

The following change would legalize the example above:

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

8.5.5 SI_MODEL annotation

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

8.6 GROUP declaration

A *group* shall be declared as shown in Syntax 42.

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

Syntax 42—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 goup declaration) may be re-used as name of another statement. As a consequence, the other statement shall be interpreted as multiple statements wherein the group identifier within each replication shall be replaced by the all-purpose value. On the other hand, no name of any visible statement shall be allowed to be re-used as group identifier.

Examples

The following example shows substitution involving group values.

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

```
        PIN data3 { DIRECTION = input ; }
    }
```

The following example shows substitution involving index values.

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

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

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

8.7 TEMPLATE declaration

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

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

Syntax 43—TEMPLATE declaration

A template declaration shall be used to specify one or more ALF statements with variable contents that can be used many times. A template instantiation (see Section 8.8) shall specify the usage of such an ALF statement.

Within the template declaration, the variable contents shall be specified by a placeholder identifier (see Section 6.11.3).

8.8 TEMPLATE instantiation

A *template* shall be instantiated in form of a *static template instantiation* or a *dynamic template instantiation*, as shown in Syntax 44

```
template_instantiation ::=
      static_template_instantiation
    | dynamic_template_instantiation

static_template_instantiation ::=
      template_identifier [ = STATIC ] ;
    | template_identifier [ = STATIC ] { { all_purpose_value } }
    | template_identifier [ = STATIC ] { { annotation } }

dynamic_template_instantiation ::=
      template_identifier = DYNAMIC { { dynamic_template_instantiation_item } }

dynamic_template_instantiation_item ::=
      annotation
    | arithmetic_model
```

Syntax 44—TEMPLATE instantiation

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

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

The following rules shall apply:

a) A static template instantiation shall be used when the replacement value of any placeholder identifier can be determined during compilation of the library. Only a matching identifier shall be considered a legal annotation identifier. Each occurence of the placeholder identifier shall be replaced by the annotation value associated with the annotation identifier.
b) A dynamic template instantiation shall be used when the replacement value of at least one placeholder identifier can only determined during runtime of the application. Only a matching identifier shall be considered a legal annotation identifier, or alternatively, a arithmetic model identifier, or alternatively, a legal arithmetic value.
c) Multiple replacement values within a multi-value annotation shall be legal if and only if the syntax rules for the ALF statement within the template declaration allow substitution of multiple values for one placeholder identifier.
d) In the case replacement by order, subsequently occuring placeholder identifiers in the template declaration shall be replaced by subsequently occuring all-purpose values in the template instantiation. If a placeholder identifier occurs more than once within the template declaration, all occurences of that placeholder identifier shall be immediately replaced by the same all-purpose value. The first amongst the remaining placeholder identifiers shall then be considered the next placeholder to be replaced by the next all-purpose value.

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

Examples

The following example illustrates rule a).

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

The following example illustrates rule b).

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

The following example illustrates rule c).

```
TEMPLATE moreAnnotations {
     KEYWORD annotation3 = annotation ;
     KEYWORD annotation4 = annotation ;
     annotation3 { <someValue> }
     annotation4 = <yetAnotherValue> ;
}
moreAnnotations {
     someValue { value1 value2 }
     yetAnotherValue = value3 ;
}
```

```
// semantically equivalent statement:
KEYWORD annotation3 = annotation ;
KEYWORD annotation4 = annotation ;
annotation3 { value1 value2 }
annotation4 = value3 ;
```

The following example illustrates rule e).

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

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

The following example illustrates rule d).

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

```
context2 { annotation5 = all ; }
// Both everything (without brackets) and <nothing> (with brackets)
// are legal values for annotation5.
```

9. Library-specific objects and related statements

Add lead-in text

9.1 LIBRARY and SUBLIBRARY declaration

A *library* and a *sublibrary* shall be declared as shown in Syntax 45.

```
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 45—LIBRARY and SUBLIBRARY declaration

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

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

9.2 Annotations for LIBRARY and SUBLIBRARY

9.2.1 INFORMATION annotation container

An *information* annotation container shall be defined using ALF language as shown in Syntax 46.

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.

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

Syntax 46—INFORMATION statement

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

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

Table 26—Annotations within an INFORMATION statement

annotation identifier	semantics of annotation value
PRODUCT	a code name of a product described herein
TITLE	a descriptive title of the product described herein
VERSION	a version number of the product description
AUTHOR	the name of a person or company generating this product description
DATETIME	date and time of day when this product description was created

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

Example

```
LIBRARY myProduct {
    INFORMATION {
        PRODUCT = p10sc;
        TITLE = “0.10 standard cell”;
        VERSION = “v2.1.0”;
        AUTHOR = “Major Asic Vendor, Inc.”;
```

```
            DATETIME = “Mon Apr 8 18:33:12 PST 2002”;
        }
    }
```

9.3 CELL declaration

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

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

Syntax 47—CELL declaration

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

9.4 CELL instantiation

A *cell* shall be instantiated as shown in .

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

Syntax 48—CELL instantiation

9.5 Annotations for a CELL

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

9.5.1 CELLTYPE annotation

A *celltype* annotation shall be defined using ALF language as shown in .

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

Syntax 49— annotation

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

Table 27—CELLTYPE annotation values

Annotation value	Description
`buffer`	Cell is a buffer, inverting or non-inverting.
`combinational`	Cell is a combinational logic element.
`multiplexor`	Cell is a multiplexor.
`flipflop`	Cell is a flip-flop.
`latch`	Cell is a latch.
`memory`	Cell is a memory or a register file.
`block`	Cell is a hierarchical block, i.e., a complex element which can be represented as a netlist. 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 can be represented as a netlist. At least one instance of the netlist is not a library element, i.e., there is no `CELL` model, but a `PRIMITIVE` model for that instance.
`special`	Cell is a special element, which can only be used in certain application contexts not describable by the `FUNCTION` statement. Examples: busholders, protection diodes, and fillcells.

9.5.2 SWAP_CLASS annotation

A *swap_class* annotation shall be defined using ALF language as shown in .

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

Syntax 50— annotation

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

Cell-swapping is only allowed under the following conditions:

— the RESTRICT_CLASS annotation (see 9.5.3) authorizes usage of the cell
— the cells to be swapped are compatible from an application standpoint (functional compatibility for synthesis and physical compatibility for layout)

9.5.3 RESTRICT_CLASS annotation

A *xxx* annotation shall be defined using ALF language as shown in .

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

Syntax 51— annotation

The value is the name of a declared CLASS. Multi-value annotation can be used. Cells referring to a particular class can be used in design tools identified by the value. The restricted annotations are shown in Table 28.

Table 28—Predefined values for RESTRICT_CLASS

Annotation string	Description
synthesis	Use restricted to logic synthesis.
scan	Use restricted to scan synthesis.
datapath	Use restricted to datapath synthesis.
clock	Use restricted to clock tree synthesis.
layout	Use restricted to layout, i.e., place & route.

User-defined values are also possible. If a cell has no or only unknown values for RESTRICT_CLASS, the application tool shall not modify any instantiation of that cell in the design. However, the cell shall still be considered for analysis.

9.5.4 SCAN_TYPE annotation

A *xxx* annotation shall be defined using ALF language as shown in .

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

Syntax 52— annotation

can take the values shown in Table 29.

Table 29—SCAN_TYPE annotations for a CELL object

Annotation string	Description
`muxscan`	A multiplexor for normal data and scan data.
`clocked`	A special scan clock.
`lssd`	Combination between flip-flop and latch with special clocking (level sensitive scan design).
`control_0`	Combinational scan cell, controlling pin shall be `0` in scan mode.
`control_1`	Combinational scan cell, controlling pin shall be `1` in scan mode.

9.5.5 SCAN_USAGE annotation

A *xxx* annotation shall be defined using ALF language as shown in .

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

Syntax 53— annotation

can take the values shown in Table 30.

Table 30—SCAN_USAGE annotations for a CELL object

Annotation string	Description
`input`	Primary input in a chain of cells.
`output`	Primary output in a chain of cells.
`hold`	Holds intermediate value in the scan chain.

The `SCAN_USAGE` applies for a special cell which is designed to be the primary input, output or intermediate stage of a scan chain. It also applies for macro blocks with connected scan chains in case there are particular scan-ordering requirements.

9.5.6 BUFFERTYPE annotation

A *xxx* annotation shall be defined using ALF language as shown in .

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

Syntax 54— annotation

can take the values shown in Table 31.

Table 31—BUFFERTYPE annotations for a CELL object

Annotation string	Description
`input`	Cell has at least one external (off-chip) input pin.
`output`	Cell has at least one external (off-chip) output pin.
`inout`	Cell has at least one external (off-chip) bidirectional pin.
`internal`	Cell has only internal (on-chip) pins.

9.5.7 DRIVERTYPE annotation

A *xxx* annotation shall be defined using ALF language as shown in .

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

Syntax 55— annotation

can take the values shown in Table 32.

Table 32—DRIVERTYPE annotations for a CELL object

Annotation string	Description
`predriver`	Cell is a predriver, i.e., the core part of an IO buffer.
`slotdriver`	Cell is a slotdriver, i.e., the pad of an IO buffer with off-chip connection.
`both`	Cell is both a predriver and a slot driver, i.e., a complete IO buffer.

NOTE—`DRIVERTYPE` applies only for cells with `BUFFERTYPE = input | output | inout`.

9.5.8 PARALLEL_DRIVE annotation

A *xxx* annotation shall be defined using ALF language as shown in .

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

Syntax 56— annotation

specifies the number of parallel drivers. This shall be greater than zero (`0`) ; the default is `1`.

9.5.9 PLACEMENT_TYPE annotation

A *xxx* annotation shall be defined using ALF language as shown in .

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

Syntax 57— annotation

The identifiers have the following definitions:

- *pad*: I/O pad, to be placed in the I/O rows
- *core*: regular macro, to be placed in the core rows
- *block*: hierarchical block with regular power structure
- *ring*: macro with built-in power structure
- *connector*: macro at the end of core rows connecting with power or ground

9.5.10 SITE reference annotation

A `CELL` can reference one or more legal placement `SITE`s. Signle-value annotation and multi-value annotation shall be legal.

9.6 ATTRIBUTE values for a CELL

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

The attribute values shown in Table 33 can be used within a CELL with CELLTYPE=memory.

Table 33—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 (e.g., static RAM)
dynamic	Dynamic memory (e.g., dynamic RAM)
asynchronous	Asynchronous memory
synchronous	Synchronous memory

The attributes shown in Table 34 can be used within a CELL with CELLTYPE=block.

Table 34—Attributes within a CELL with CELLTYPE=block

Attribute item	Description
counter	Cell is a complex sequential cell 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 complex sequential cell 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 combinational element performing an addition of two operands.
subtractor	Cell is a subtractor, i.e., a combinational element performing a subtraction of two operands.
multiplier	Cell is a multiplier, i.e., a combinational element performing a multiplication of two operands.
comparator	Cell is a comparator, i.e., a combinational element comparing the magnitude of two operands.
ALU	Cell is an arithmetic logic unit, i.e., a combinational element combining the functionality of adder, subtractor, comparator in a selectable way.

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

Table 35—Attributes within a CELL with CELLTYPE=core

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

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

Table 36—Attributes within a CELL with CELLTYPE=special

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

9.7 PIN declaration

A *pin* shall be declared as a *scalar pin* or as a *vector pin* or a *matrix pin*, as shown in Syntax 58.

A pin shall represent a terminal of an electronic circuit for the purpose of exchanging information with the environment of the electronic circuit. A constant value of information shall be called *state*. A time-dependent value of information shall be called *signal*. A reference to a pin in general shall be established by the pin identifier.

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

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

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

Syntax 58—PIN declaration

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 vector within a matrix is not provided.

Example

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

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

Consider the following pin assignment:

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

This establishes the following exchange of information:

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

9.8 PINGROUP declaration

A *pingroup* shall be declared as a *simple pingroup* or as a *vector pingroup*, as shown in Syntax 59.

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

Syntax 59—PINGROUP declaration

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

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

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

9.9 Annotations for a PIN and a PINGROUP

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

9.9.1 VIEW annotation

A *xxx* annotation shall be defined using ALF language as shown in .

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

Syntax 60— annotation

annotates the view where the pin appears, which can take the values shown in Table 37.

Table 37—VIEW annotations for a PIN object

Annotation string	Description
`functional`	Pin appears in functional netlist.
`physical`	Pin appears in physical netlist.
`both` (default)	Pin appears in both functional and physical netlist.
`none`	Pin does not appear in netlist.

9.9.2 PINTYPE annotation

A *xxx* annotation shall be defined using ALF language as shown in .

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

Syntax 61— annotation

annotates the type of the pin, which can take the values shown in Table 38.

Table 38—PINTYPE annotations for a PIN object

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

9.9.3 DIRECTION annotation

A *xxx* annotation shall be defined using ALF language as shown in .

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

Syntax 62— annotation

annotates the direction of the pin, which can take the values shown in Table 39.

Table 39—DIRECTION annotations for a PIN object

Annotation string	Description
`input`	Input pin.
`output`	Output pin.
`both`	Bidirectional pin.
`none`	No direction can be assigned to the pin.

Table 40 gives a more detailed semantic interpretation for using `DIRECTION` in combination with `PINTYPE`.

Table 40—DIRECTION in combination with PINTYPE

DIRECTION	PINTYPE=digital	PINTYPE=analog	PINTYPE=supply
`input`	Pin receives a digital signal.	Pin receives an analog signal.	Pin is a power sink.
`output`	Pin drives a digital signal.	Pin drives an analog signal.	Pin is a power source.
`both`	Pin drives or receives a digital signal, depending on the operation mode.	Pin drives or receives an analog signal, depending on the operation mode.	Pin is both power sink and source.
`none`	Pin represents either an internal digital signal with no external connection or a feed through.	Pin represents either an internal analog signal with no external connection or a feed through.	Pin represents either an internal power pin with no external connection or a feed through.

For pins with PINTYPE=supply, the DIRECTION describes an electrical characteristic rather than a functional characteristic, since there is no functional definition for DIRECTION. For pins with PINTYPE=digital or analog, the functional definition of DIRECTION actually matches the electrical definition.

Examples

- The power and ground pins of regular cells shall have `DIRECTION=input`.
- A level converter cell shall have a power supply pin with `DIRECTION=input` and another power supply pin with `DIRECTION=output`.
- A level converter can have separate ground pins on the input and output side or a common ground pin with `DIRECTION=both`.
- The power and ground pins of a feed through cell shall have `DIRECTION=none`.

9.9.4 SIGNALTYPE annotation

A *xxx* annotation shall be defined using ALF language as shown in .

`SIGNALTYPE` classifies the functionality of a pin. The currently defined values apply for pins with `PINTYPE=DIGITAL`.

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

Syntax 63— annotation

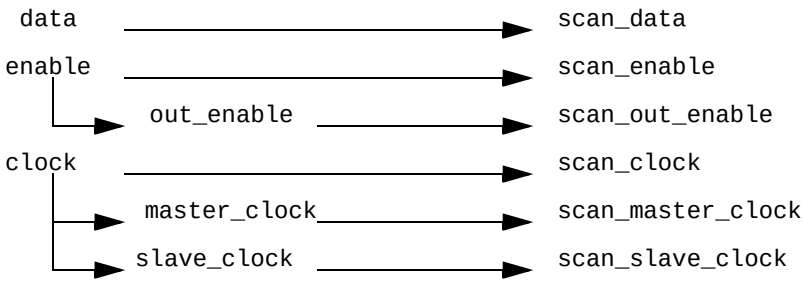
Conceptually, a pin with `PINTYPE = ANALOG` can also have a `SIGNALTYPE` annotation. However, no values are currently defined.

The fundamental `SIGNALTYPE` values are defined in Table 41

Table 41—Fundamental SIGNALTYPE annotations for a PIN object

Annotation string	Description
`data` (default)	General data signal, i.e., a signal that carries information to be transmitted, received, or subjected to logic operations within the `CELL`.
`address`	Address signal of a memory, i.e., an encoded signal, usually a bus or part of a bus, driving an address decoder within the `CELL`.
`control`	General control signal, i.e., an encoded signal that controls at least two modes of operation of the `CELL`, eventually in conjunction with other signals. The signal value is allowed to change during real-time circuit operation.
`select`	Select signal of a multiplexor, i.e., a decoded or encoded signal that selects the data path of a multiplexor or de-multiplexor within the `CELL`. Each selected signal has the same `SIGNALTYPE`.
`enable`	Enables storage of general input data in a sequential cell, i.e., a cell or a flipflop
`tie`	The signal needs to be tied to a fixed value statically in order to define a fixed or programmable mode of operation of the `CELL`, eventually in conjunction with other signals. The signal value is not allowed to change during real-time circuit operation.
`clear`	Clear signal of a flip-flop or latch, i.e., a signal that controls the storage of the value `0` within the `CELL`.
`set`	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`.

Scheme for construction of composite signaltype values:



The composite SIGNALTYPE values are defined in Table 41

Table 42—Composite SIGNALTYPE annotations for a PIN object

Annotation string	Description
scan_data	Scan data signal, i.e., signal is for testing purpose only
out_enable	Enables visibility of general data at the output.
scan_enable	Enables storage of scan input data in a sequential cell, i.e., a cell or a flipflop
scan_out_enable	Enables visibility of scan data at the output.
master_clock	triggers storage of input data in 1st stage of flipflop in a two-phase clocking scheme
slave_clock	triggers data transfer from 1st stage to 2nd stage of flipflop in a two-phase clocking scheme
scan_clock	triggers scan data storage within the CELL.
scan_master_clock	triggers storage of input scan data in 1st stage of flipflop in a two-phase clocking scheme
scan_slave_clock	triggers scan data transfer from 1st stage to 2nd stage of flipflop in a two-phase clocking scheme

"Flipflop", "latch", "multiplexor", and "memory" can be standalone cells or embedded in larger cells. In the former case, the celltype is flipflop, latch, multiplexor, and memory, respectively. In the latter case, the celltype is block or core.

9.9.5 ACTION annotation

A *xxx* annotation shall be defined using ALF language as shown in .

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

Syntax 64— annotation

annotates the action of the signal, which can take the values shown in Table 43.

Table 43—ACTION annotations for a PIN object

Annotation string	Description
`asynchronous`	Signal acts in an asynchronous way, i.e., self-triggered.
`synchronous`	Signal acts in a synchronous way, i.e., triggered by a signal with `SIGNALTYPE CLOCK` or a composite `SIGNALTYPE` with postfix `_CLOCK`.

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

Table 44—ACTION applicable in conjunction with fundamental SIGNALTYPE values

Fundamental SIGNALTYPE	Applicable ACTION	Comment
`data`	no	
`address`	no	
`control`	yes	
`select`	no	
`enable`	yes	
`tie`	no	
`clear`	yes	
`set`	yes	
`clock`	no	Presence of `SIGNALTYPE=clock` conditions the validity of `ACTION=synchronous` for other signals.

9.9.6 POLARITY annotation

A *xxx* annotation shall be defined using ALF language as shown in .

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

Syntax 65— annotation

annotates the polarity of the pin signal.

The polarity of an input pin (i.e., `DIRECTION = input;`) takes the values shown in Table 45.

Table 45—POLARITY annotations for a PIN

Annotation string	Description
`high`	Signal active high or to be driven high.
`low`	Signal active low or to be driven low.
`rising_edge`	Signal sensitive to rising edge.
`falling_edge`	Signal sensitive to falling edge.
`double_edge`	Signal sensitive to any edge.

The `POLARITY` annotation applies only to pins with certain `SIGNALTYPE` values, as shown in Table 46. The rule applies also to any composite `SIGNALTYPE` values based on the fundamental values.

Table 46—POLARITY applicable in conjunction with fundamental SIGNALTYPE values

Fundamental SIGNALTYPE	Applicable POLARITY	Comment
`data`	N/A	
`address`	N/A	
`control`	N/A	CONTROL_POLARITY `high`, `low`
`select`	N/A	
`enable`	`high`, `low`.	
`tie`	`high`, `low`.	
`clear`	`high`, `low`.	
`set`	`high`, `low`.	
`clock`	`high`, `low`, `rising_edge`, `falling_edge`, `double_edge`,	CONTROL_POLARITY can apply

9.9.7 DATATYPE annotation

A *xxx* annotation shall be defined using ALF language as shown in .

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

Syntax 66— annotation

annotates the datatype of the pin, which can take the values shown in Table 47.

Table 47—DATATYPE annotations for a PIN object

Annotation string	Description
`signed`	Result of arithmetic operation is signed 2's complement.
`unsigned`	Result of arithmetic operation is unsigned.

`DATATYPE` is only relevant for bus pins.

9.9.8 INITIAL_VALUE annotation

A *xxx* annotation shall be defined using ALF language as shown in .

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

Syntax 67— annotation

shall be compatible with the buswidth and `DATATYPE` of the signal.

`INITIAL_VALUE` is used for a downstream behavioral simulation model, as far as the simulator (e.g., a VITAL-compliant simulator) supports the notion of initial value.

9.9.9 SCAN_POSITION annotation

A *xxx* annotation shall be defined using ALF language as shown in .

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

Syntax 68— annotation

annotates the position of the pin in scan chain, starting with `1`. Value 0 (default) indicates that the PIN is not on the scan chain.

9.9.10 STUCK annotation

A *xxx* annotation shall be defined using ALF language as shown in .

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

Syntax 69— annotation

annotates the stuck-at fault model as shown in Table 48.

Table 48—STUCK annotations for a PIN object

Annotation string	Description
`stuck_at_0`	Pin can have stuck-at-0 fault.
`stuck_at_1`	Pin can have stuck-at-1 fault.
`both` (default)	Pin can have both stuck-at-0 and stuck-at-1 faults.
`none`	Pin can not have stuck-at faults.

9.9.11 SUPPLYTYPE

A *xxx* annotation shall be defined using ALF language as shown in .

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

Syntax 70— annotation

A `PIN` with `PINTYPE = SUPPLY` shall have a `SUPPLYTYPE` annotation, as shown in.

9.9.12 SIGNAL_CLASS

A *xxx* annotation shall be defined using ALF language as shown in .

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

Syntax 71— annotation

9.9.13 SUPPLY_CLASS

A *xxx* annotation shall be defined using ALF language as shown in .

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

Syntax 72— annotation

9.9.14 DRIVETYPE annotation

A *xxx* annotation shall be defined using ALF language as shown in .

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

Syntax 73— annotation

annotates the drive type for the pin, which can take the values shown in Table 49.

Table 49—DRIVETYPE annotations for a PIN object

Annotation string	Description
`cmos` (default)	Standard cmos signal.
`nmos`	Nmos or pseudo nmos signal.
`pmos`	Pmos or pseudo pmos signal.
`nmos_pass`	Nmos passgate signal.
`pmos_pass`	Pmos passgate signal.
`cmos_pass`	Cmos passgate signal, i.e., the full transmission gate.
`ttl`	TTL signal.
`open_drain`	Open drain signal.
`open_source`	Open source signal.

9.9.15 SCOPE annotation

A *xxx* annotation shall be defined using ALF language as shown in .

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

Syntax 74— annotation

annotates the modeling scope of a pin, which can take the values shown in Table 50.

Table 50—SCOPE annotations for a PIN object

Annotation string	Description
`behavior`	The pin is used for modeling functional behavior and events on the pin are monitored for vector expressions in `BEHAVIOR` statements.
`measure`	Measurements related to the pin can be described, e.g., timing or power characterization, and events on the pin are monitored for vector expressions in `VECTOR` statements.
`both` (default)	The pin is used for functional behavior as well as for characterization measurements.
`none`	No model; only the pin exists.

9.9.16 CONNECT_CLASS annotation

A *xxx* annotation shall be defined using ALF language as shown in .

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

Syntax 75— annotation

annotates a declared class object for connectivity determination.

Connectivity rules involving those classes shall apply for the pin.

9.9.17 SIDE annotation

A *xxx* annotation shall be defined using ALF language as shown in .

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

Syntax 76— annotation

which can take the values shown in Table 51.

Table 51—SIDE annotations for a PIN object

Annotation string	Description
`left`	Pin is on the left side.
`right`	Pin is on the right side.
`top`	Pin is at the top.
`bottom`	Pin is at the bottom.

9.9.18 ROW and COLUMN annotation

A *xxx* annotation shall be defined using ALF language as shown in .

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

Syntax 77— annotation

The following annotation shall be used for a pin in order to indicate the location of the pin within a placement row or column, as shown in .

where *row*`_assignment` applies for pins with `SIDE = right | left` and *column*`_assignment` applies for pins with `SIDE = top | bottom`.

For bus pins, *row*`_assignment` and *column*`_assignment` shall have the form of `multi_value_assignment`s, as shown in .

9.9.19 ROUTING_TYPE annotation

A *xxx* annotation shall be defined using ALF language as shown in .

The identifiers have the following definitions:

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

Syntax 78— annotation

- *regular*: connection by regular routing
- *abutment*: connection by abutment, no routing
- *ring*: pin forms a ring around the block with connection allowed to any point of the ring
- *feedthrough*: both ends of the pin align and can be used for connection

9.9.20 PULL annotation

A *xxx* annotation shall be defined using ALF language as shown in .

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

Syntax 79— annotation

annotates the pull type for the pin, which can take the values shown in Table 52.

Table 52—PULL annotations for a PIN object

Annotation string	Description
`up`	Pullup device connected to pin.
`down`	Pulldown device connected to pin.
`both`	Pullup and pulldown device connected to pin.
`none` (default)	No pull device.

9.10 ATTRIBUTE values for a PIN and a PINGROUP

The attribute values shown in Table 53 can be used within a `PIN` object.

Table 53—Attributes within a PIN object

Attribute item	Description
`SCHMITT`	Schmitt trigger signal.

Table 53—Attributes within a PIN object (Continued)

Attribute item	Description
TRISTATE	Tristate signal.
XTAL	Crystal/oscillator signal.
PAD	Pad going off-chip.

The attributes shown in Table 54 are only applicable for pins within cells with CELLTYPE=memory and certain values of SIGNALTYPE.

Table 54—Attributes for pins of a memory

Attribute item	SIGNALTYPE	Description
ROW_ADDRESS_STROBE	clock	Samples the row address of the memory.
COLUMN_ADDRESS_STROBE	clock	Samples the column address of the memory.
ROW	address	Selects an addressable row of the memory. (PINGROUP)
COLUMN	address	Selects an addressable column of the memory. (PINGROUP)
BANK	address	Selects an addressable bank of the memory. (PINGROUP)

The attributes shown in Table 55 are only applicable for pins representing double-rail signals.

Table 55—Attributes for pins representing double-rail signals

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

The following restrictions apply for double-rail 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 no 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 special pin ATTRIBUTE values shown in Table 56 shall be defined for memory BIST.

Table 56—PIN or PINGROUP attributes for memory BIST

Attribute item	Description
ROW_INDEX	Pin is a bus with a contiguous range of values, indicating a physical row of a memory.
COLUMN_INDEX	Pin is a bus with a contiguous range of values, indicating a physical column of a memory.
BANK_INDEX	Pin is a bus with a contiguous range of values, indicating a physical bank of a memory.
DATA_INDEX	Pin is a bus with a contiguous range of values, indicating the bit position within a data bus of a memory.
DATA_VALUE	Pin represents a value stored in a physical memory location.

These attributes apply to the pins of the BIST wrapper around the memory rather than to the pins of the memory itself.

The BEHAVIOR statement within TEST shall involve the variables declared as PINs with ATTRIBUTE ROW_INDEX, COLUMN_INDEX, BANK_INDEX, DATA_INDEX, or DATA_VALUE.

9.11 PRIMITIVE declaration

A PRIMITIVE shall be declared as shown in Syntax 80.

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

Syntax 80—PRIMITIVE statement

A PRIMITIVE referenced in a CELL can replace the complete set of PIN and FUNCTION definition. PINs can be declared before the reference to the PRIMITIVE, in order to provide supplementary annotations that cannot be inherited from the PRIMITIVE. However, the CELL shall be pin-compatible with the PRIMITIVE.

If the PRIMITIVE or a CELL is referenced in an annotation container such as SCAN, only the subset of PINs used in the non-scan cell shall be compatible with the PINs of the cell.

The pin names can be referenced by order or by name. In the latter case, the LHS is the pin name of the referenced `PRIMITIVE` or `CELL` (e.g., the non-scan cell), the RHS is the pin name of the actual cell. A constant logic value can also appear at the LHS or RHS, indicating a pin needs to be tied to a constant value. If this information is already specified in an annotation inside the `PIN` object itself, referencing between a pin name and a constant value is not necessary.

9.12 WIRE declaration

A *wire* shall be declared as shown in .

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

Syntax 81—WIRE declaration

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

9.12.1 Annotations for a WIRE

9.12.2 SELECT_CLASS annotation

A *xxx* annotation shall be defined using ALF language as shown in .

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

Syntax 82— annotation

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

The purpose of the select class annotation is to enable a convenient interconnect model selection for a given application. The user of the application can select a set of interconnect models by specifying the name of the class rather than specifying the name of each interconnect model.

9.13 NODE declaration

A *node* shall be declared as shown in Syntax 83.

The purpose of a node declaration is to specify an electrical node in the context of a wire declaration or in the context of a cell declaration.

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

Syntax 83—NODE statement

9.13.1 NODETYPE annotation

A *xxx* annotation shall be defined using ALF language as shown in .

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

Syntax 84— annotation

The values shall have the following semantic meaning.

Table 57—NODETYPE annotation values

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

9.13.2 NODE_CLASS annotation

A *xxx* annotation shall be defined using ALF language as shown in .

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

Syntax 85— annotation

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

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

9.14 VECTOR declaration

A *vector* shall be declared as shown in Syntax 86.

vector ::=
VECTOR control_expression **;**
| **VECTOR** control_expression **{** { vector_item } **}**
| *vector*_template_instantiation
vector_item ::=
all_purpose_item

Syntax 86—VECTOR statement

9.15 Annotations for VECTOR

9.15.1 PURPOSE annotation

A *xxx* annotation shall be defined using ALF language as shown in .

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

Syntax 87— annotation

9.15.2 OPERATION annotation

A *xxx* annotation shall be defined using ALF language as shown in .

The `OPERATION` statement inside a `VECTOR` shall be used to indicate the combined definition of signal values or signal changes for certain operations which are not entirely controlled by a single signal.

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

Syntax 88— annotation

The values shall have the following semantic meaning.

Table 58—OPERATION annotation values

Annotation string	Description
`read`	read operation at one address
`write`	write operation at one address
`read_modify_write`	read followed by write of different value at same address
`start`	first operation required in a particular mode
`end`	last operation required in a particular mode
`refresh`	operation required to maintain the contents of the memory without modifying it
`load`	operation for loading control registers
`iddq`	operation for supply current measurements in quiescent state

9.15.3 LABEL annotation

A *xxx* annotation shall be defined using ALF language as shown in .

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

Syntax 89— annotation

ensures SDF matching with conditional delays across Verilog, VITAL, etc.

See the end of B.3 for an example.

9.15.4 EXISTENCE_CONDITION annotation

A *xxx* annotation shall be defined using ALF language as shown in .

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

Syntax 90— annotation

For false-path analysis tools, the existence condition shall be used to eliminate the vector from further analysis if, and only if, the existence condition evaluates to *False*. For applications other than false-path analysis, the existence condition shall be treated as if the boolean expression was a co-factor to the vector itself. The default existence condition is *True*.

Example

```
VECTOR (01 a -> 01 z & (c | !d) ) {
    EXISTENCE_CONDITION = !scan_select;
    DELAY { FROM { PIN=a; } TO { PIN=z; } /* data */ }
}
VECTOR (01 a -> 01 z & (!c | d) ) {
    EXISTENCE_CONDITION = !scan_select;
    DELAY { FROM { PIN=a; } TO { PIN=z; } /* data */ }
}
```

Each vector contains state-dependent delay for the same timing arc. If `!scan_select` evaluates *True*, both vectors are eliminated from timing analysis.

9.15.5 EXISTENCE_CLASS annotation

A *xxx* annotation shall be defined using ALF language as shown in .

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

Syntax 91— annotation

Reference to the same existence class by multiple vectors has the following effects:

— A common mode of operation is established between those vectors, which can be used for selective analysis, for instance mode-dependent timing analysis. The name of the mode is the name of the class.
— A common existence condition is inherited from that existence class, if there is one.

Example

```
CLASS non_scan_mode {
    EXISTENCE_CONDITION = !scan_select;
}
VECTOR (01 a -> 01 z & (c | !d) ) {
    EXISTENCE_CLASS = non_scan_mode;
    DELAY { FROM { PIN=a; } TO { PIN=z; } /* data */ }
```

```
}
VECTOR (01 a -> 01 z & (!c | d) ) {
    EXISTENCE_CLASS = non_scan_mode;
    DELAY { FROM { PIN=a; } TO { PIN=z; } /* data */ }
}
```

Each vector contains state-dependent delay for the same timing arc. If the mode `non_scan_mode` is turned off or if `!scan_select` evaluates *True*, both vectors are eliminated from timing analysis.

9.15.6 CHARACTERIZATION_CONDITION annotation

A *xxx* annotation shall be defined using ALF language as shown in .

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

Syntax 92— annotation

For characterization tools, the characterization condition shall be treated as if the boolean expression was a co-factor to the vector itself. For all other applications, the characterization condition shall be disregarded. The default characterization condition is *True*.

Example

```
VECTOR (01 a -> 01 z & (c | !d) ) {
    CHARACTERIZATION_CONDITION = c & !d;
    DELAY { FROM { PIN=a; } TO { PIN=z; } /* data */ }
}
```

The delay value for the timing arc applies for any of the following conditions: (`c & !d`), (`c & d`), or (`!c & !d`), since they all satisfy (`c | !d`) . However, the only condition chosen for delay characterization is (`c & !d`).

9.15.7 CHARACTERIZATION_VECTOR annotation

A *xxx* annotation shall be defined using ALF language as shown in .

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

Syntax 93— annotation

The characterization vector is provided for the case where the vector expression cannot be constructed using the vector and a boolean co-factor. The use of the characterization vector is restricted to characterization tools in the same way as the use of the characterization condition. Either a characterization condition or a characterization

vector can be provided, but not both. If none is provided, the vector itself shall be used by the characterization tool.

Example

```
VECTOR (01 A -> 01 Z) {
    CHARACTERIZATION_VECTOR = ((01 A & 10 inv_A) -> (01 Z & 10 inv_Z));
}
```

Analysis tools see the signals `A` and `Z`. The signals `inv_A` and `inv_Z` are visible to the characterization tool only.

9.15.8 CHARACTERIZATION_CLASS annotation

A *xxx* annotation shall be defined using ALF language as shown in .

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

Syntax 94— annotation

Reference to the same characterization class by multiple vectors has the following effects:

— A commonality is established between those vectors, which can be used for selective characterization in a way defined by the library characterizer, for instance, to share the characterization task between different teams or jobs or tools.
— A common characterization condition or characterization vector is inherited from that characterization class, if there is one.

9.16 LAYER declaration

A *layer* shall be declared as shown in Syntax 95.

layer ::=
  LAYER *layer*_identifier **;**
  | **LAYER** *layer*_identifier **{** { layer_item } **}**
  | *layer*_template_instantiation
layer_item ::=
  all_purpose_item

Syntax 95—LAYER declaration

LAYER statements shall be in sequential order defined by the manufacturing process, starting bottom-up in the following sequence: one or multiple substrate layers, followed by alternating cut and routing layers, then the dielectric layer. Abstract layers can appear at the end of the sequence.

9.17 Annotations for LAYER

9.17.1 LAYERTYPE annotation

A *xxx* annotation shall be defined using ALF language as shown in .

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

Syntax 96— annotation

The identifiers have the following definitions:

The values shall have the following semantic meaning.

Table 59—LAYERTYPE annotation values

Annotation string	Description
routing	layer provides electrical connections within one plane
cut	layer provides electrical connections between planes
substrate	layer(s) at the bottom
dielectric	provides electrical isolation between planes
reserved	layer is for proprietary use only
abstract	not a manufacturable layer, used for description of boundaries between objects

9.17.2 PITCH annotation

A *xxx* annotation shall be defined using ALF language as shown in .

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

Syntax 97— annotation

The PITCH annotation identifies the routing pitch for a layer with LAYERTYPE=routing.

The pitch is measured between the center of two adjacent parallel wires routed on the layer.

9.17.3 PREFERENCE annotation

A *xxx* annotation shall be defined using ALF language as shown in .

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

Syntax 98— annotation

The purpose is to indicate the preferred routing direction.

9.18 VIA declaration

A *via* shall be declared as shown in Syntax 99.

via ::=
  VIA *via*_identifier **;**
  | **VIA** *via*_identifier **{** { via_item } **}**
  | *via*_template_instantiation
via_item ::=
  all_purpose_item
  | pattern
  | artwork

Syntax 99—VIA statement

The VIA statement shall contain at least three patterns, referring to the cut layer and two adjacent routing layers. Stacked vias can contain more than three patterns.

9.19 VIA instantiation

A *via* shall be instantiated as shown in .

via_instantiation ::=
  *via*_identifier *instance*_identifier **;**
  | *via*_identifier *instance*_identifier **{** { geometric_transformation } **}**

Syntax 100—VIA instantiation

9.20 Annotations for a VIA

9.20.1 VIATYPE annotation

A *xxx* annotation shall be defined using ALF language as shown in .

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

Syntax 101— annotation

The values shall have the following semantic meaning.

Table 60—VIATYPE annotation values

Annotation string	Description
`default`	via can be used per default
`non_default`	via can only be used if authorized by a `RULE`
`partial_stack`	via contains 3 patterns: lower and upper routing layer and cut layer in-between. It can only be used to build stacked vias. The bottom of a stack can be a `default` or a `non_default` via.
`full_stack`	via contains `2N+1` patterns (`N>1`). It describes the full stack from bottom to top.

9.21 RULE declaration

A *rule* shall be declared as shown in Syntax 102.

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

Syntax 102—RULE statement

9.22 ANTENNA declaration

An *antenna* shall be declared as shown in Syntax 103.

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

Syntax 103—ANTENNA declaration

9.23 BLOCKAGE declaration

A *blockage* shall be declared as shown in Syntax 104.

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

Syntax 104—BLOCKAGE statement

9.24 PORT declaration

A *port* shall be declared as shown in Syntax 105.

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

Syntax 105—PORT declaration

A port is a collection of geometries within a pin, representing electrically equivalent points.

9.25 Annotations for PORT

9.25.1 PORT_VIEW annotation

A *xxx* annotation shall be defined using ALF language as shown in .

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

Syntax 106— annotation

The values shall have the following semantic meaning.

Table 61—PORT_VIEW annotation values

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

9.26 SITE declaration

A *site* shall be declared as shown in Syntax 107.

site ::=
SITE *site*_identifier **;**
| **SITE** *site*_identifier **{** { site_item } **}**
| *site*_template_instantiation
site_item ::=
all_purpose_item

Syntax 107—SITE declaration

The arithmetic models WIDTH and HEIGHT within a SITE declaration are deemed mandatory.

9.27 Annotations for SITE

9.27.1 ORIENTATION_CLASS

A *xxx* annotation shall be defined using ALF language as shown in .

9.27.2 SYMMETRY_CLASS

A *xxx* annotation shall be defined using ALF language as shown in .

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

Syntax 108— annotation

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

Syntax 109— annotation

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

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

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

Case 1: no symmetry

Site has legal orientations A and B. Cell has legal orientations A and B. When the site is instantiated in the A orientation, the cell shall be placed in the A orientation.

Case 2: symmetry

Site has legal orientations A and B and symmetry between A and B. Cell has legal orientations A and B. When the site is instantiated in the A orientation, the cell can be placed in the A or B orientation.

9.28 ARRAY declaration

An array shall be declared as shown in Syntax 110.

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

Syntax 110—ARRAY statement

The `geometric_transformations` define the locations of the starting points within the array and the number of repetitions of the components of the array. Details are defined in the next section.

9.29 Annotations for ARRAY

9.29.1 ARRAYTYPE annotation

A *xxx* annotation shall be defined using ALF language as shown in .

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

Syntax 111— annotation

9.30 PATTERN declaration

A *pattern* shall be declared as shown in Syntax 112.

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

Syntax 112—PATTERN declaration

9.31 Annotations for PATTERN

9.31.1 SHAPE annotation

A *xxx* annotation shall be defined using ALF language as shown in .

SHAPE applies only for a PATTERN in a routing layer, as shown in Figure 6. The default is `line`.

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

Syntax 113— annotation

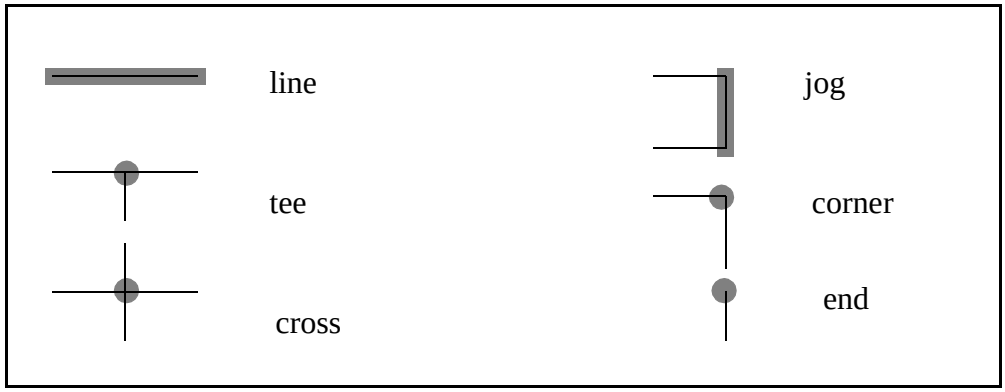


Figure 6—Routing layer shapes

`line` and `jog` represent routing segments, which can have an individual `LENGTH` and `WIDTH`. The `LENGTH` *between* routing segments is defined as the common run length. The `DISTANCE` *between* routing segments is measured orthogonal to the routing direction.

`tee`, `cross`, and `corner` represent intersections between routing segments. `end` represents the end of a routing segment. Therefore, they have points rather than lines as references. The points can have an `EXTENSION`. The `DISTANCE` between points can be measured straight or by using `HORIZONTAL` and `VERTICAL`.

9.31.2 VERTEX annotation

A *xxx* annotation shall be defined using ALF language as shown in .

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

Syntax 114— annotation

The *`vertex`*`_annotation` shall appear only in conjunction with the *`extension`*`_arithmetic_model`. It specifies the form of the extended object, as shown in Figure 7.

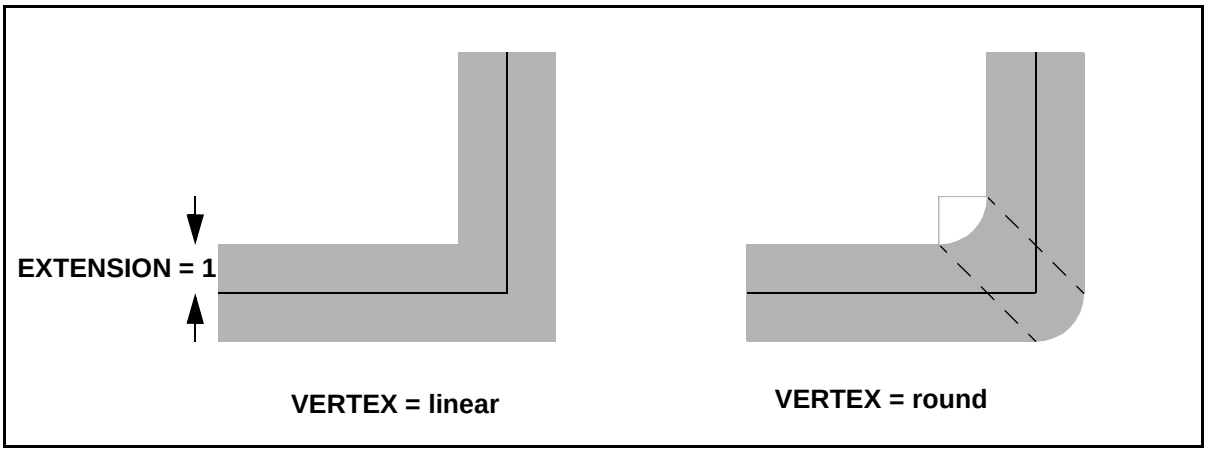


Figure 7—Illustration of VERTEX annotation

9.31.3 LAYER reference annotation

A PATTERN is associated with a LAYER.

9.32 Geometric model

This section defines the geometric model statement and how to predefine commonly used objects (using TEMPLATE).

A geometric model describes the form of an object in a physical library. It is in the context of a pattern, which is associated with physical objects, such as via, blockage, port, rule. Patterns and other physical objects can also be subjected to geometric transformations, as shown in Figure 8.

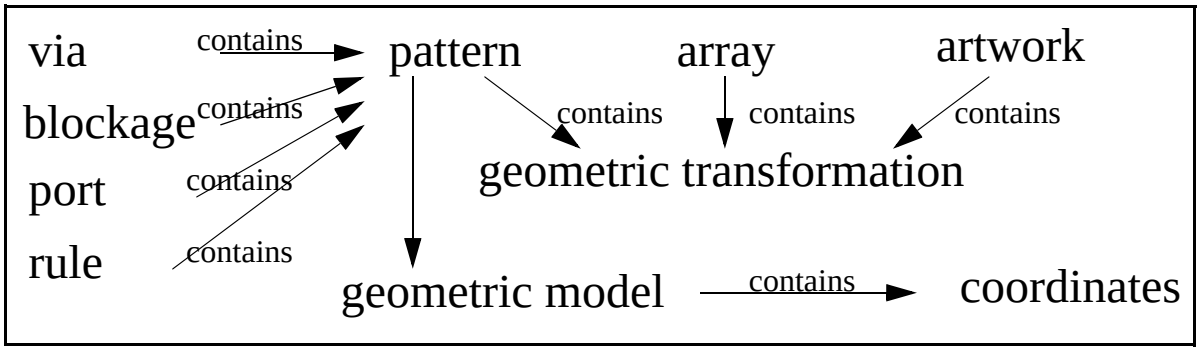


Figure 8—Geometric model and its context

A geometric model is defined as shown in Syntax 115.

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

Syntax 115—Geometric model

The following geometric model identifiers shall be defined.

Table 62—Geometric model identifiers

identifier	Description
`DOT`	describes one point
`POLYLINE`	defined by `N>1` directly connected points, forming an open object
`RING`	defined by `N>1` directly connected points, forming a closed object, i.e., the last point is connected with first point. The object occupies the boundary of the enclosed space.
`POLYGON`	defined by `N>1` connected points, forming a closed object, i.e., the last point is connected with first point. The object occupies the entire enclosed space.

All of these are depicted in Figure 9.

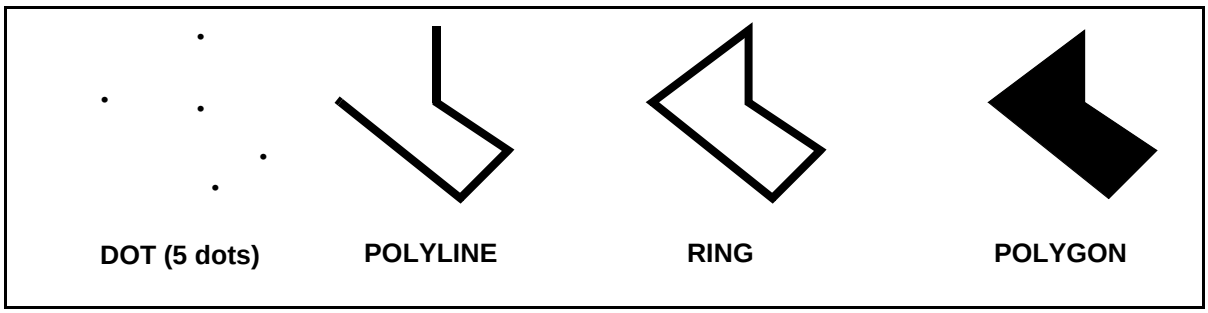


Figure 9—Illustration of geometric models

A *xxx* annotation shall be defined using ALF language as shown in .

The *`point_to_point`*`_annotation` applies for **POLYLINE**, **RING**, and **POLYGON**. It specifies how the connections between points is made. The default is `direct`, which defines a straight connection (see Figure 10). The value `manhattan` specifies a connection by moving in the x-direction first and then moving in the y-direction (see Figure 11). This enables a non-redundant specification of rectilinear objects using `N/2` points instead of `N` points.

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

Syntax 116— annotation

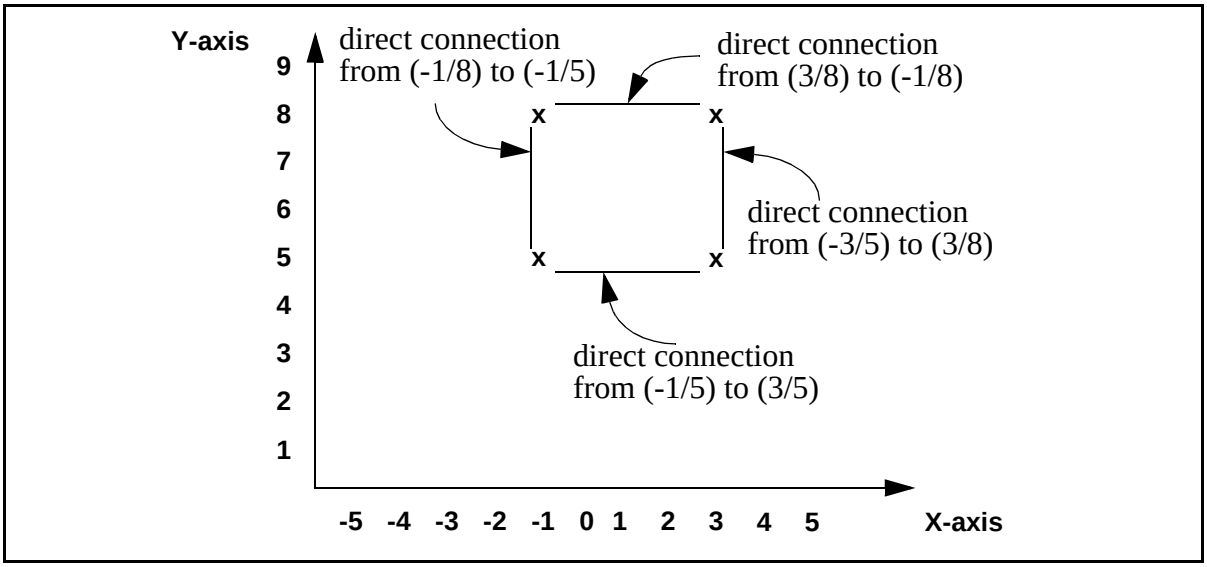


Figure 10—Illustration of direct point-to-point connection

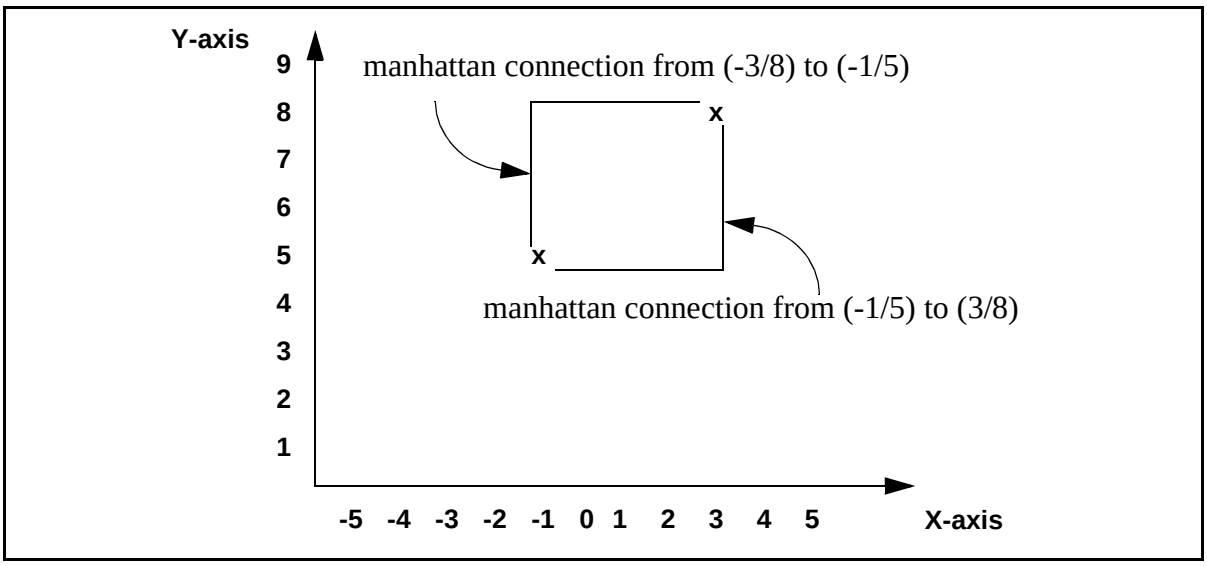


Figure 11—Illustration of manhattan point-to-point connection

Example

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

Both objects describe the same rectangle.

9.33 Predefined geometric models using TEMPLATE

The TEMPLATE construct (see 3.2.6) can be used to predefine some commonly used objects.

The templates RECTANGLE and LINE shall be predefined as follows:

```
TEMPLATE RECTANGLE {
    POLYGON {
       POINT_TO_POINT = manhattan;
       COORDINATES { <left> <bottom> <right> <top> }
    }
}
TEMPLATE LINE {
    POLYLINE {
       POINT_TO_POINT = direct;
       COORDINATES { <x_start> <y_start> <x_end> <y_end> }
    }
}
```

Example 1

The following example shows the instantiation of predefined templates.

```
// same rectangle as in previous example
RECTANGLE {left = -1; bottom = 5; right = 3; top = 8; }
//or
RECTANGLE {-1 5 3 8 }

// diagonals through the rectangle
LINE {x_start = -1; y_start = 5; x_end = 3; y_end = 8; }
LINE {x_start = 3; y_start = 5; x_end = -1; y_end = 8; }
//or
LINE { -1 5 3 8 }
LINE { 3 5 -1 8 }
```

The definitions for predefined templates are fixed. Therefore the keywords RECTANGLE and LINE are reserved. On the other hand, the definitions for user-defined templates are only known by the library supplied by the user.

Example 2

The following example shows some user-defined templates.

```
TEMPLATE HORIZONTAL_LINE {
    POLYLINE {
       POINT_TO_POINT = direct;
       COORDINATES { <left> <y> <right> <y> }
    }
}
TEMPLATE VERTICAL_LINE {
    POLYLINE {
       POINT_TO_POINT = direct;
       COORDINATES { <x> <bottom> <x> <top> }
    }
}
```

Example 3

The following example shows the instantiation of user-defined templates.

```
// lines bounding the rectangle
HORIZONTAL_LINE { y = 5; left = -1; right = 3; }
HORIZONTAL_LINE { y = 8; left = -1; right = 3; }
VERTICAL_LINE { x = -1; bottom = 5; top = 8; }
VERTICAL_LINE { x = 3; bottom = 5; top = 8; }
//or
HORIZONTAL_LINE { 5 -1 3 }
HORIZONTAL_LINE { 8 -1 3 }
VERTICAL_LINE { -1 5 8 }
VERTICAL_LINE { 3 5 8 }
```

9.34 Geometric transformation

A geometric transformation XXX, as shown in Syntax 117.

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 117—Geometric transformation

The `SHIFT` statement defines the horizontal and vertical offset measured between the coordinates of the geometric model and the actual placement of the object. Eventually, a layout tool only supports integer numbers. The numbers are in units of DISTANCE. If only one annotation is given, the default value for the other one is `0`. If the `SHIFT` statement is not given, both values default to `0`.

The ROTATE statement defines the angle of rotation in degrees measured between the orientation of the object described by the coordinates of the geometric model and the actual placement of the object measured in counter-clockwise direction, specified by a number between 0 and 360. Eventually, a layout tool can only support angles which are multiple of 90 degrees. The default is 0. The object shall rotate around its origin.

The FLIP describes a transformation of the specified coordinates by flipping the object around an axis specified by a number between 0 and 180. The number represents the angle of the flipping direction in degrees. Eventually, a layout tool can only support angles which are multiple of 90 degrees. The axis is orthogonal to the flipping direction. The axis shall go through the origin of the object. For example, 0 means flip in horizontal direction, axis is vertical whereas 90 means flip in vertical direction, axis is horizontal.

The purpose of the REPEAT statement is to describe the replication of a physical object in a regular way, for example SITE (see Section 9.12). The REPEAT statement can also appear within a geometric_model. The unsigned number defines the total number of replications. The number 1 means, the object appears just once. If this number is not given, the REPEAT statement defines a rule for an arbitrary number of replications. REPEAT statements can also be nested.

Examples

The following example replicates an object three times along the horizontal axis in a distance of 7 units.

```
REPEAT = 3 {
    SHIFT { HORIZONTAL = 7; }
}
```

The following example replicates an object five times along a 45-degree axis.

```
REPEAT = 5 {
    SHIFT { HORIZONTAL = 4; VERTICAL = 4; }
}
```

The following example replicates an object two times along the horizontal axis and four times along the vertical axis.

```
REPEAT = 2 {
    SHIFT { HORIZONTAL = 5; }
    REPEAT = 4 {
       SHIFT { VERTICAL = 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 { VERTICAL = 6; }
    REPEAT = 2 {
       SHIFT { HORIZONTAL = 5; }
    }
}
```

Rules and restrictions:

— A physical object can contain a `geometric_transformation` statement of any kind, but no more than one of a specific kind.
— The `geometric_transformation` statements shall apply to all `geometric_models` within the context of the object.
— The `geometric_transformation` statements shall refer to the origin of the object, i.e., the point with coordinates `{ 0 0 }`. Therefore, the result of a combined transformation shall be independent of the order in which each individual transformation is applied.

These are demonstrated in Figure 12.

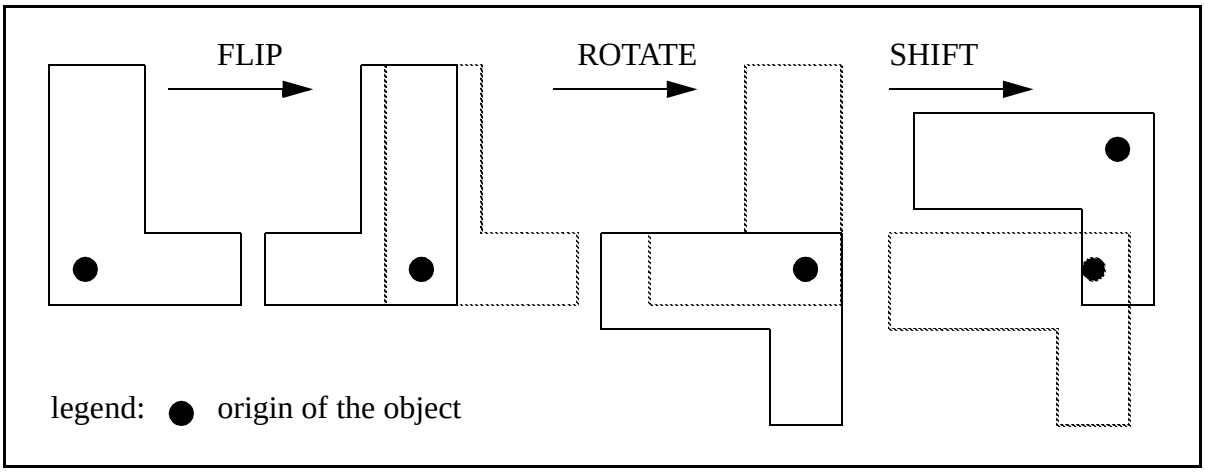


Figure 12—Illustration of FLIP, ROTATE, and SHIFT

9.35 ARTWORK statement

An *artwork* statement shall be defined as shown in Syntax 118.

```
artwork ::=
    ARTWORK = artwork_identifier ;
  | ARTWORK = artwork_identifier { { artwork_item } }
  | artwork_template_instantiation
artwork_item ::=
    geometric_transformation
  | pin_assignment
```

Syntax 118—ARTWORK statement

The `ARTWORK` statement creates a reference between the cell in the library and the original cell imported from a physical layout database (e.g., GDS2).

The `geometric_transformations` define the operations for transformation from the artwork geometry to the actual cell geometry. In other words, the artwork is considered as the original object whereas the cell is the transformed object.

The imported cell can have pins with different names. The LHS of the `pin_assignments` describes the pin names of the original cell, the RHS describes the pin names of the cell in this library. See 11.4 for the syntax of `pin_assignments` .

Example

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

9.36 FUNCTION statement

A FUNCTION statement shall be defined as shown in Syntax 119.

function ::=
FUNCTION { function_item { function_item } **}**
| *function*_template_instantiation
function_item ::=
all_purpose_item
| behavior
| structure
| statetable

Syntax 119—FUNCTION statement

9.37 TEST statement

A TEST statement, shall be defined as shown in Syntax 120.

test ::=
TEST { test_item { test_item } **}**
| *test*_template_instantiation
test_item ::=
all_purpose_item
| behavior
| statetable

Syntax 120—TEST statement

The purpose is to describe the interface between an externally applied test algorithm and the CELL. The BEHAVIOR statement within the TEST statement uses the same syntax as the behavior statement within the FUNCTION statement. However, the set of used variables is different. Both the TEST and the FUNCTION statement shall be self-contained, complete and complementary to each other.

9.38 BEHAVIOR statement

A BEHAVIOR statemen shall be defined as shown in Syntax 121.

```
behavior ::=
    BEHAVIOR { behavior_item { behavior_item }s }
  | behavior_template_instantiation
behavior_item ::=
    boolean_assignment
  | control_statement
  | primitive_instantiation
  | behavior_item_template_instantiation
boolean_assignments ::=
  boolean_assignment  { boolean_assignment }
boolean_assignment ::=
  pin_variable = boolean_expression ;
primitive_instantiation ::=
    primitive_identifier [ identifier ] { pin_value { pin_value } }
  | primitive_identifier [ identifier ]  { boolean_assignments }
control_statement ::=
  @ control_expression { boolean_assignments } { : control_expression { boolean_assignments } }
```

Syntax 121—BEHAVIOR statement

Inside BEHAVIOR, variables that appear at the LHS of an assignment conditionally controlled by a vector expression, as opposed to an unconditional continuous assignment, hold their values, when the vector expression evaluates *False*. Those variables are considered to have latch-type behavior.

Examples

```
BEHAVIOR {
    @(G){
       Q = D;  // both Q and QN have latch-type behavior
       QN = !D;
    }
}
BEHAVIOR {
    @(G){
       Q = D;  // only Q has latch-type behavior
    }
    QN = !Q;
}
```

9.39 STRUCTURE statement

A STRUCTURE statement shall be defined as shown in Syntax 122.

```
structure ::=
    STRUCTURE { named_cell_instantiation { named_cell_instantiation } }
  | structure_template_instantiation
```

Syntax 122—STRUCTURE statement

An optional STRUCTURE statement shall be legal in the context of a FUNCTION. A STRUCTURE statement describes the structure of a complex cell composed of atomic cells, for example I/O buffers, LSSD flip-flops, or clock trees. The STRUCTURE statement shall be legal inside the FUNCTION statement (see 11.17):

The STRUCTURE statement shall describe a netlist of components inside the CELL. The STRUCURE statement shall not be a substitute for the BEHAVIOR statement. If a FUNCTION contains only a STRUCTURE statement

and no BEHAVIOR statement, a behavior description for that particular cell shall be meaningless (e.g., fillcells, diodes, vias, or analog cells).

Timing and power models shall be provided for the CELL, if such models are meaningful. Application tools are not expected to use function, timing, or power models from the instantiated components as a substitute of a missing function, timing, or power model at the top-level. However, tools performing characterization, construction, or verification of a top-level model shall use the models of the instantiated components for this purpose.

Test synthesis applications can use the structural information in order to define a one-to-many mapping for scan cell replacement, such as where a single flip-flop is replaced by a pair of master/slave latches. A macro cell can be defined whose structure is a netlist containing the master and slave latch and this shall contain the NON_SCAN_CELL annotation to define which sequential cells it is replacing. No timing model is required for this macro cell, since it should be treated as a transparent hierarchy level in the design netlist after test synthesis.

NOTES

1—Every *instance*_identifier within a STRUCTURE statement shall be different from each other.

2—The STRUCTURE statement provides a directive to the application (e.g., synthesis and DFT) as to how the CELL is implemented. A CELL referenced in named_cell_instantiation can be replaced by another CELL within the same SWAP_CLASS and RESTRICT_CLASS (recognized by the application).

3—The *cell*_identifier within a STRUCTURE statement can refer to actual cells as well as to primitives. The usage of primitives is recommended in fault modeling for DFT.

4—BEHAVIOR statements also provide the possibility of instantiating primitives. However, those instantiations are for modeling purposes only; they do not necessarily match a physical structure. The STRUCTURE statement always matches a physical structure.

9.40 STATETABLE statement

A STATETABLE statement shall be defined as shown in Syntax 123.

```
statetable ::=
      STATETABLE [ identifier ]
         { statetable_header statetable_row { statetable_row } }
    | statetable_template_instantiation
statetable_header ::=
    input_pin_variables : output_pin_variables ;
statetable_row ::=
    statetable_control_values : statetable_data_values ;
statetable_control_values ::=
    statetable_control_value { statetable_control_value }
statetable_control_value ::=
      bit_literal
    | based_literal
    | unsigned
    | edge_value
statetable_data_values ::=
    statetable_data_value { statetable_data_value }
statetable_data_value ::=
      bit_literal
    | based_literal
    | unsigned
    | ( [ ! ] pin_variable )
    | ( [ ~ ] pin_variable )
```

Syntax 123—STATETABLE statement

The functional description can be supplemented by a STATETABLE, the first row of which contains the arguments that are object IDs of the declared PINs. The arguments appear in two fields, the first is input and the second is output. The fields are separated by a :. The rows are separated by a ;. The arguments can appear in both fields if the PINs have attribute direction=output or direction=both. If direction=output, then the argument has latch-type behavior. The argument on the input field is considered previous state and the argument on the output field is considered the next state. If direction=both, then the argument on the input field applies for input direction and the argument on the output field applies for output direction of the bidirectional PIN.

Example

```
CELL ff_sd {
    PIN  q {DIRECTION=output;}
    PIN  d {DIRECTION=input;}
    PIN cp {DIRECTION=input;
            SIGNALTYPE=clock;
            POLARITY=rising_edge;}
    PIN cd {DIRECTION=input; SIGNALTYPE=clear; POLARITY=low;}
    PIN sd {DIRECTION=input; SIGNALTYPE=set; POLARITY=low;}
    FUNCTION {
       BEHAVIOR {
          @(!cd) {q = 0;} :(!sd) {q = 1;} :(01 cp) {q = d;}
       }
       STATETABLE {
          cd sd  cp  d   q  : q ;
          0  ?   ??  ?   ?  : 0 ;
          1  0   ??  ?   ?  : 1 ;
          1  1   1?  ?   0  : 0 ;
          1  1   ?0  ?   1  : 1 ;
          1  1   1?  ?   0  : 0 ;
          1  1   ?0  ?   1  : 1 ;
          1  1   01  ?   ?  :(d);
       }
    }
}
```

If the output variable with latch-type behavior depends only on the previous state of itself, as opposed to the previous state of other output variables with latch-type behavior, it is not necessary to use that output variable in the input field. This allows a more compact form of the STATETABLE.

Example

```
STATETABLE {
    cd sd  cp  d  : q ;
    0  ?   ??  ?  : 0 ;
    1  0   ??  ?  : 1 ;
    1  1   1?  ?  :(q);
    1  1   ?0  ?  :(q);
    1  1   01  ?  :(d);
}
```

A generic ALF parser shall make the following semantic checks.

— Are all variables of a FUNCTION declared either by declaration as PIN names or through assignment?

— Does the STATETABLE exclusively contain declared PINs?
— Is the format of the STATETABLE, i.e., the number of elements in each field of each row, consistent?
— Are the values consistently either state or transition digits?
— Is the number of digits in each TABLE entry compatible with the signal bus width?

A more sophisticated checker for complete verification of logical consistency of a FUNCTION given in both equation and tabular representation is out of scope for a generic ALF parser, which checks only syntax and compliance to semantic rules. However, formal verification algorithms can be implemented in special-purpose ALF analyzers or model generators/compilers.

9.41 NON_SCAN_CELL statement

A *non-scan cell* statement shall be defined as shown in Syntax 124.

non_scan_cell ::=
NON_SCAN_CELL { unnamed_cell_instantiation { unnamed_cell_instantiation } **}**
| **NON_SCAN_CELL =** unnamed_cell_instantiation
| *non_scan_cell*_template_instantiation

Syntax 124—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 *unnamed cell instantiation* within the non-scan cell statement specifies a cell that is functionally equivalent to the scan cell, if the extra pins are not used. The cell without extra pins is refered to as non-scan cell. The name of the non-scan cell is given by the *cell identifier.*

The pin mapping is given either by order, using *pin value*, or by name, using *pin assignment*. In the former case, the pin values shall refer to pin names of the scan cell. The order of the pin values corresponds to the pin declarations within the non-scan cell. In the latter case, the pin names of the non-scan cell shall appear at the LHS of the assignment, and the pin names of the scan cell shall appear at the RHS of the assignment. The order of the pin assignments is arbitrary.

Example

```
// declaration of a non-scan cell
CELL myNonScanFlop {
    PIN D { DIRECTION=input;  SIGNALTYPE=data; }
    PIN C { DIRECTION=input;  SIGNALTYPE=clock; POLARITY=rising_edge; }
    PIN Q { DIRECTION=output; SIGNALTYPE=data; }
}
// declaration of a scan cell
CELL myScanFlop {
   PIN CK { DIRECTION=input; SIGNALTYPE=clock; }
   PIN DI { DIRECTION=input; SIGNALTYPE=data; }
   PIN SI { DIRECTION=input; SIGNALTYPE=scan_data; }
   PIN SE { DIRECTION=input; SIGNALTYPE=scan_enable; POLARITY=high; }
   PIN DO { DIRECTION=output; SIGNALTYPE=data; }
   // put NON_SCAN_CELL statement here
}
```

The non-scan cell statement with pin mapping by order looks as follows:

```
   NON_SCAN_CELL { myNonScanFlop { DI CK DO } }
// corresponding pins by order:    D  C  Q
```

The non-scan cell statement with pin mapping by name looks as follows:

```
   NON_SCAN_CELL { myNonScanFlop { Q=DO; D=DI; C=CK; } }
```

9.42 RANGE statement

A *range* statement shall be defined as shown in Syntax 125.

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

Syntax 125—RANGE statement

The range statement shall be used to specify a valid address space for elements of a vector- or matrix-pin.

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

$$0 \le a \le 2^b - 1$$

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

where

a is an unsigned number representing the address space within a vector- or matrix-pin,
b is the bitwidth of the vector-or matrix-pin,

and

MSB is the leftmost bit within the vector- or matrix-pin,
LSB is the rightmost bit within the vector or- matrix-pin,

in accordance with Section 7.8 on page 40.

The index values within a range statement shall be bound by the address space *a*, otherwise the range statement shall not be considered valid.

Example

```
PIN [5:8] myVectorPin { RANGE { 3 : 13 } }
```

bitwidth:	$b = 4$
default address space:	$0 \le a \le 15$
address space defined by range statement:	$3 \le a \le 13$

10. Constructs for modeling of digital behavior

Add lead-in text

10.1 Variable declarations

Inside a `CELL` object, the `PIN` objects with the `PINTYPE digital` define variables for `FUNCTION` objects inside the same `CELL`. A *primary input variable* inside a `FUNCTION` shall be declared as a `PIN` with `DIRECTION=input` or `both` (since `DIRECTION=both` is a bidirectional pin). However, it is not required that all declared pins are used in the function. Output variables inside a `FUNCTION` need not be declared pins, since they are implicitly declared when they appear at the left-hand side (LHS) of an assignment.

Example

```
CELL my_cell {
    PIN A {DIRECTION = input;}
    PIN B {DIRECTION = input;}
    PIN C {DIRECTION = output;}
    FUNCTION {
       BEHAVIOR {
          D = A && B;
          C = !D;
       }
    }
}
```

`C` and `D` are output variables that need not be declared prior to use. After implicit declaration, `D` is reused as an input variable. `A` and `B` are primary input variables.

10.2 Boolean value system

this paragraph needs to move into another section

A *bit* literal shall represent a single bit constant, as shown in Table 63.

Table 63—Single bit constants

Literal	Description
`0`	Value is logic zero.
`1`	Value is logic one.
`X` or `x`	Value is unknown.
`L` or `l`	Value is logic zero with weak drive strength.
`H` or `h`	Value is logic one with weak drive strength.
`W` or `w`	Value is unknown with weak drive strength.
`Z` or `z`	Value is high-impedance.
`U` or `u`	Value is uninitialized.

Table 63—Single bit constants (Continued)

Literal	Description
?	Value is any of the above, yet stable.
*	Value can randomly change.

The following symbols within an octal based literal shall represent numerical values, which can be mapped into equivalent symbols within a binary based literal, as shown in .

Table 64—Mapping between octal base and binary base

Octal	Binary (bit literal)	Numerical value
0	**000**	0
1	**001**	1
2	**010**	2
3	**011**	3
4	**100**	4
5	**101**	5
6	**110**	6
7	**111**	7

The following symbols within a hexadecimal based literal shall represent numerical values, which can be mapped into equivalent symbols within an octal based literal and a binary based literal, as shown in .

Table 65—Mapping between hexadecimal base, octal base, and binary base

Hexadecimal	Octal	Binary (bit literal)	Numerical value
0	**00**	**0000**	0
1	**01**	**0001**	1
2	**02**	**0010**	2
3	**03**	**0011**	3
4	**04**	**0100**	4
5	**05**	**0101**	5
6	**06**	**0110**	6
7	**07**	**0111**	7
8	**10**	**1000**	8
9	**11**	**1001**	9

Table 65—Mapping between hexadecimal base, octal base, and binary base (Continued)

Hexadecimal	Octal	Binary (bit literal)	Numerical value
a or **A**	**12**	**1010**	10
b or **B**	**13**	**1011**	11
c or **C**	**14**	**1100**	12
d or **D**	**15**	**1101**	13
e or **E**	**16**	**1110**	14
f or **F**	**17**	**1111**	15

Based literals involving symbolic bit literals shall not be used to represent numerical values. They shall be mapped from one base into another base according to the following rules:

a) A symbolic bit literal in a hexadecimal based literal shall be mapped into two subsequent occurences of the same symbolic bit literal in an octal based literal.
b) A symbolic bit literal in an octal based literal shall be mapped into three subsequent occurences of the same symbolic bit literal in a binary based literal.
c) A symbolic bit literal in an hexadecimal based literal shall be mapped into four subsequent occurences of the same symbolic bit literal in a binary based literal.

Example

`'o2xw0u` is equivalent to **`'b010_xxx_www_000_uuu`**
`'hLux` is equivalent to **`'bLLLL_uuuu_xxxx`**

10.3 Combinational functions

This section defines the different types of combinational functions in ALF.

10.3.1 Combinational logic

Combinational logic can be described by continuous assignments of boolean values (*True* or *False*) to output variables as a function of boolean values of input variables. Such functions can be expressed in either boolean expression format or statetable format.

Let us consider an arbitrary continuous assignment

$$z = f(a_1 .., .. a_n)$$

In a dynamic or simulation context, the left-hand side (LHS) variable *z* is evaluated whenever there is a change in one of the right-hand side (RHS) variables *ai*. No storage of previous states is needed for dynamic simulation of combinational logic.

10.3.2 Boolean operators on scalars

Table 66, Table 67, and Table 68 list unary, binary, and ternary boolean operators on scalars.

Table 66—Unary boolean operators

Operator	Description
!, **~**	Logical inversion.

Table 67—Binary boolean operators

Operator	Description
&&, **&**	Logical AND.
||,		Logical OR.
~^	Logic equivalence (XNOR).
^	Logic anti valence (XOR).

Table 68—Ternary operator

Operator	Description
?	Boolean condition operator for construction of combinational if-then-else clause.
:	Boolean else operator for construction of combinational if-then-else clause.

Combinational if-then-else clauses are constructed as follows:

```
<cond1>? <value1>: <cond2>? <value2>: <cond3>? <value3>: <default_value>
```

If `cond1` evaluates to boolean *True*, then `value1` is the result; else if `cond2` evaluates to boolean *True*, then `value2` is the result; else if `cond3` evaluates to boolean *True*, then `value3` is the result; else `default_value` is the result of this clause.

10.3.3 Boolean operators on words

Table 69 and Table 70 list unary and binary reduction operators on words (logic variables with one or more bits). The result of an expression using these operators shall be a logic value.

Table 69—Unary reduction operators

Operator	Description
&	AND all bits.

Table 69—Unary reduction operators (Continued)

Operator	Description
~&	NAND all bits.
|	OR all bits.
~|	NOR all bits.
^	XOR all bits.
~^	XNOR all bits.

Table 70—Binary reduction operators

Operator	Description
==	Equality for case comparison.
!=	Non-equality for case comparison.
>	Greater.
<	Smaller.
>=	Greater or equal.
<=	Smaller or equal.

Table 71 and Table 72 list unary and binary bitwise operators. The result of an expression using these operators shall be an array of bits.

Table 71—Unary bitwise operators

Operator	Description
~	Bitwise inversion.

Table 72—Binary bitwise operators

Operator	Description
&	Bitwise AND.
|	Bitwise OR.
^	Bitwise XOR.
~^	Bitwise XNOR.

The following arithmetic operators, listed in Table 73, are also defined for boolean operations on words. The result of an expression using these operators shall be an extended array of bits.

Table 73—Binary operators

Operator	Description
<<	Shift left.
>>	Shift right.
+	Addition.
-	Subtraction.
*	Multiplication.
/	Division.
%	Modulo division.

The arithmetic operations addition, subtraction, multiplication, and division shall be *unsigned* if all the operands have the datatype *unsigned*. If any of the operands have the datatype signed, the operation shall be *signed*. See Table 6-25 for the DATATYPE definitions.

10.3.4 Operator priorities

The priority of binding operators to operands in boolean expressions shall be from strongest to weakest in the following order:

a) unary boolean operator (!, ~, &, ~&, |, ~|, ^, ~^)
b) XNOR (~^), XOR (^), relational (>, <, >=, <=, ==, !=), shift (<<, >>)
c) AND (&, &&), NAND (~&), multiply (*), divide (/), modulus (%)
d) OR (|, ||), NOR (~|), add (+), subtract (-)
e) ternary operators (?, :)

10.3.5 Datatype mapping

Logical operations can be applied to scalars and words. For that purpose, the values of the operands are reduced to a system of three logic values in the following way:

H has the logic value 1
L has the logic value 0
W, Z, U have the logic value X
A word has the logic value 1, if the unary OR reduction of all bits results in 1
A word has the logic value 0, if the unary OR reduction of all bits results in 0
A word has the logic value X, if the unary OR reduction of all bits results in X

Case comparison operations can also be applied to scalars and words. For scalars, they are defined in Table 74.

Table 74—Case comparison operators

A	B	A==B	A!=B	A>B	A<B
1	1	1	0	0	0
1	H	0	1	X	X
1	0	0	1	1	0
1	L	0	1	1	0
1	W, U, Z, X	0	1	X	0
H	1	0	1	X	X
H	H	1	0	0	0
H	0	0	1	1	0
H	L	0	1	1	0
H	W, U, Z, X	0	1	X	0
0	1	0	1	0	1
0	H	0	1	0	1
0	0	1	0	0	0
0	L	0	1	X	X
0	W, U, Z, X	0	1	0	X
L	1	0	1	0	1
L	H	0	1	0	1
L	0	0	1	X	X
L	L	1	0	0	0
L	W, U, Z, X	0	1	0	X
X	X	1	0	X	X
X	U	X	X	X	X
X	0, 1, H, L, W, Z	0	1	X	X
W	W	1	0	X	X
W	U	X	X	X	X
W	0, 1, H, L, X, Z	0	1	X	X
Z	Z	1	0	X	X
Z	U	X	X	X	X
Z	0, 1, H, L, X, W	0	1	X	X

Table 74—Case comparison operators (Continued)

A	B	A==B	A!=B	A>B	A<B
U	0, 1, H, L, X, W, Z, U	X	X	X	X

For word operands, the operations > and < are performed after reducing all bits to the 3-value system first and then interpreting the resulting number according to the datatype of the operands. For example, if datatype is *signed*, `'b1111` is smaller than `'b0000`; if datatype is *unsigned*, `'b1111` is greater than `'b0000`. If two operands have the same value `'b1111` and a different datatype, the unsigned `'b1111` is greater than the signed `'b1111`.

The operations >= and <= are defined in the following way:

```
(a >= b) === (a > b) || (a == b)
(a <= b) === (a < b) || (a == b)
```

10.3.6 Rules for combinational functions

If a boolean expression evaluates *True*, the assigned output value is `1`. If a boolean expression evaluates *False*, the assigned output value is `0`. If the value of a boolean expression cannot be determined, the assigned output value is `X`. Assignment of values other than `1`, `0`, or `X` needs to be specified explicitly.

For evaluation of the boolean expression, input value `'bH` shall be treated as `'b1`. Input value `'bL` shall be treated as `'b0`. All other input values shall be treated as `'bX`.

Examples

In equation form, these rules can be expressed as follows.

```
BEHAVIOR {
    Z = A;
}
```

is equivalent to

```
BEHAVIOR {
    Z = A ? 'b1 : 'b0;
}
```

More explicitly, this is also equivalent to

```
BEHAVIOR {
    Z = (A=='b1 || A=='bH)? 'b1 : (A=='b0 || A=='bL)? 'b0 : 'bX;
}
```

In table form, this can be expressed as follows:

```
STATETABLE {
    A   :   Z;
    ?   :   (A);
}
```

which is equivalent to

```
STATETABLE {
    A   :   Z;
    0   :   0;
    1   :   1;
}
```

More explicitly, this is also equivalent to

```
STATETABLE {
    A   :   Z;
    0   :   0;
    L   :   0;
    1   :   1;
    H   :   1;
    X   :   X;
    W   :   X;
    Z   :   X;
    U   :   X;
}
```

10.3.7 Concurrency in combinational functions

Multiple boolean assignments in combinational functions are understood to be concurrent. The order in the functional description does not matter, as each boolean assignment describes a piece of a logic circuit. This is illustrated in Figure 13.

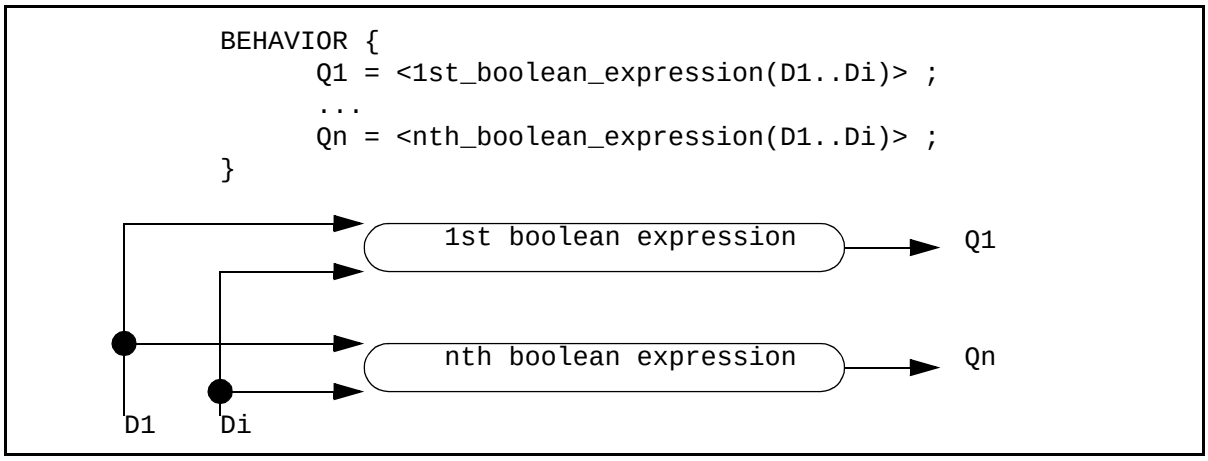


Figure 13—Concurrency for combinational logic

10.4 Sequential functions

This section defines the different types of sequential functions in ALF.

10.4.1 Level-sensitive sequential logic

In sequential logic, an output variable *zj* can also be a function of itself, i.e., of its previous state. The sequential assignment has the form

$$z_j = f(a_1 ..,.. a_n , z_1 ..,.. z_m)$$

The RHS cannot be evaluated continuously, since a change in the LHS as a result of a RHS evaluation shall trigger a new RHS evaluation repeatedly, unless the variables attain stable values. Modeling capabilities of sequential logic with continuous assignments are restricted to systems with oscillating or self-stabilizing behavior.

However, using the concept of *triggering conditions* for the LHS enables everything which is necessary for modeling *level-sensitive* sequential logic. The expression of a triggered assignment can look like this:

$$@\ g(b_1 ..,.. b_k)\ z_j = f(a_1 ..,.. a_n , z_1 ..,.. z_m)$$

The evaluation of *f* is activated whenever the *triggering function g* is *True*. The evaluation of *g* is self-triggered, i.e. at each time when an argument of *g* changes its value. If *g* is a boolean expression like *f*, we can model all types of *level-sensitive sequential logic*.

During the time when *g* is *True*, the logic cell behaves exactly like combinational logic. During the time when *g* is *False*, the logic cell holds its value. Hence, one memory element per state bit is needed.

10.4.2 Edge-sensitive sequential logic

In order to model *edge-sensitive sequential logic*, notations for logical transitions and logical states are needed.

If the triggering function *g* is sensitive to logical transitions rather than to logical states, the function *g* evaluates to *True* only for an infinitely small time, exactly at the moment when the transition happens. The sole purpose of *g* is to trigger an assignment to the output variable through evaluation of the function *f* exactly at this time.

Edge-sensitive logic requires storage of the previous output state and the input state (to detect a transition). In fact, all implementations of edge-triggered flip-flops require at least two storage elements. For instance, the most popular flip-flop architecture features a master latch driving a slave latch.

Using transitions in the triggering function for value assignment, the functionality of a positive edge triggered flip-flop can be described as follows in ALF:

```
@ (01 CP) {Q = D;}
```

which reads “at rising edge of `CP`, assign `Q` the value of `D`”.

If the flip-flop also has an asynchronous direct clear pin (`CD`), the functional description consists of either two concurrent statements or two statements ordered by priority, as shown in Figure 14.

```
// concurrent style
@ (!CD) {Q = 0;}
@ (01 CP && CD) {Q = D;}
// priority (if-then-else) style
@ (!CD) {Q = 0;} : (01 CP) {Q = D;}
```

Figure 14—Model of a flip-flop with asynchronous clear in ALF

The following two examples show corresponding simulation models in Verilog and VHDL.

```
// full simulation model

always @(negedge CD or posedge CP) begin
   if ( ! CD ) Q <= 0;
   else if (CP && !CP_last_value) Q <= D;
   else Q <= 1'bx;
end
always @ (posedge CP or negedge CP) begin
   if (CP===0 | CP===1'bx) CP_last_value <= CP ;
end

// simplified simulation model for synthesis

always @(negedge CD or posedge CP) begin
   if ( ! CD ) Q <= 0;
   else Q <= D;
end
```

Figure 15—Model of a flip-flop with asynchronous clear in Verilog

```
// full simulation model

process (CP, CD) begin
   if (CD = '0') then
      Q <= '0';
   elsif (CP'last_value = '0' and CP = '1' and CP'event) then
      Q <= D;
   elsif (CP'last_value = '0' and CP = 'X' and CP'event) then
      Q <= 'X';
   elsif (CP'last_value = 'X' and CP = '1' and CP'event) then
      Q <= 'X';
   end if;
end process;

// simplified simulation model for synthesis

process (CP, CD) begin
   if (CD = '0') then
      Q <= '0';
   elsif (CP = '1' and CP'event) then
      Q <= D;
   end if;
end process;
```

Figure 16—Model of a flip-flop with asynchronous clear in VHDL

The following differences in modeling style can be noticed: VHDL and Verilog provide the list of sensitive signals at the beginning of the `process` or `always` block, respectively. The information of level-or edge-sensitiv-

ity shall be inferred by `if-then-else` statements inside the block. ALF shows the level-or-edge sensitivity as well as the priority directly in the triggering expression. Verilog has another particularity: The sensitivity list indicates whether at least one of the triggering signals is edge-sensitive by the use of `negedge` or `posedge`. However, it does not indicate which one, since either none or all signals shall have `negedge` or `posedge` qualifiers.

Furthermore, `posedge` is any transition with `0` as initial state *or* `1` as final state. A positive-edge triggered flip-flop shall be inferred for synthesis, yet this flip-flop shall only work correctly if both the initial state is `0` *and* the final state is `1`. Therefore, a simulation model for verification needs to be more complex than the model in the synthesizeable RTL code.

In Verilog, the extra non-synthesizeable code needs to also reproduce the relevant previous state of the clock signal, whereas VHDL has built-in support for `last_value` of a signal.

10.4.3 Unary operators for vector expressions

A transition operation is defined using unary operators on a scalar net. The scalar constants (see 6.8) shall be used to indicate the start and end states of a transition on a scalar net.

`bit bit` // apply transition from bit value to bit value

For example,

01 is a transition from `0` to `1`.

No whitespace shall be allowed between the two scalar constants. The transition operators shown in Table 75 shall be considered legal.

Table 75—Unary vector operators on bits

Operator	Description
01	Signal toggles from `0` to `1`.
10	Signal toggles from `1` to `0`.
00	signal remains `0`.
11	Signal remains `1`.
0?	Signal remains `0` or toggles from `0` to arbitrary value.
1?	Signal remains `1` or toggles from `1` to arbitrary value.
?0	Signal remains `0` or toggles from arbitrary value to `0`.
?1	Signal remains `1` or toggles from arbitrary value to `1`.
??	Signal remains constant or toggles between arbitrary values.
0*	A number of arbitrary signal transitions, including possibility of constant value, with the initial value `0`.
1*	A number of arbitrary signal transitions, including possibility of constant value, with the initial value `1`.
?*	A number of arbitrary signal transitions, including possibility of constant value, with arbitrary initial value.

Table 75—Unary vector operators on bits (Continued)

Operator	Description
***0**	A number of arbitrary signal transitions, including possibility of constant value, with the final value `0`.
***1**	A number of arbitrary signal transitions, including possibility of constant value, with the final value `1`.
***?**	A number of arbitrary signal transitions, including possibility of constant value, with arbitrary final value.

Unary operators for transitions can also appear in the STATETABLE.

Transition operators are also defined on words (and can appear the in STATETABLE as well):

```
'base word 'base word
```

In this context, the transition operator shall apply transition from first word value to second word value.

For example,

'hA'h5 is a transition of a 4-bit signal from 'b1010 to 'b0101.

No whitespace shall be allowed between *base* and *word*.

The unary and binary operators for transition, listed in Table 76 and Table 77 respectively, are defined on bits and words.

Table 76—Unary vector operators on bits or words

Operator	Description
?-	No transition occurs.
??	Apply arbitrary transition, including possibility of constant value.
?!	Apply arbitrary transition, excluding possibility of constant value.
?~	Apply arbitrary transition with all bits toggling.

10.4.4 Basic rules for sequential functions

A sequential function is described in equation form by a boolean assignment with a condition specified by a boolean expression or a vector expression. If the condition evaluates to `1` (*True*), the boolean assignment is activated and the assigned output values follows the rules for combinational functions. If the vector expression evaluates to `0` (*False*), the output variables hold their assigned value from the previous evaluation.

For evaluation of a condition, the value `'bH` shall be treated as *True*, the value `'bL` shall be treated as *False*. All other values shall be treated as the unknown value `'bX`.

Example

The following behavior statement

```
BEHAVIOR {
    @ (E) {Z = A;}
}
```

is equivalent to

```
BEHAVIOR {
    @ (E=='b1 || E=='bH) {Z = A;}
}
```

The following statetable statement, describing the same logic function

```
STATETABLE {
    E   A   :   Z;
    0   ?   :   (Z);
    1   ?   :   (A);
}
```

is equivalent to

```
STATETABLE {
    E   A   :   Z;
    0   ?   :   (Z);
    L   ?   :   (Z);
    1   ?   :   (A);
    H   ?   :   (A);
}
```

For edge-sensitive and higher-order event sensitive functions, transitions from or to `'bL` shall be treated like transitions from or to `'b0`, and transitions from or to `'bH` shall be treated like transitions from or to `'b1`.

Not every transition can trigger the evaluation of a function. The set of vectors triggering the evaluation of a function are called *active vectors*. From the set of active vectors, a set of *inactive vectors* can be derived, which shall clearly not trigger the evaluation of a function. There are is also a set of ambiguous vectors, which can trigger the evaluation of the function.

The set of active vectors is the set of vectors for which both observed states before and after the transition are known to be logically equivalent to the corresponding states defined in the vector expression.

The set of inactive vectors is the set of vectors for which at least one of the observed states before or after the transition is known to be not logically equivalent to the corresponding states defined in the vector expression.

Example

For the following sequential function

```
@ (01 CP) { Z = A; }
```

the active vectors are

```
('b0'b1 CP)
('b0'bH CP)
```

```
('bL'b1 CP)
('bL'bH CP)
```

and the inactive vectors are

```
('b1'b0 CP)
('b1'bL CP)
('b1'bX CP)
('b1'bW CP)
('b1'bZ CP)
('bH'b0 CP)
('bH'bL CP)
('bH'bX CP)
('bH'bW CP)
('bH'bZ CP)
('bX'b0 CP)
('bX'bL CP)
('bW'b0 CP)
('bW'bL CP)
('bZ'b0 CP)
('bZ'bL CP)
('bU'b0 CP)
('bU'bL CP)
```

and the ambiguous vectors are

```
('b0'bX CP)
('b0'bW CP)
('b0'bZ CP)
('bL'bX CP)
('bL'bW CP)
('bL'bZ CP)
('bX'b1 CP)
('bW'b1 CP)
('bZ'b1 CP)
('bX'bH CP)
('bW'bH CP)
('bZ'bH CP)
('bX'bW CP)
('bX'bZ CP)
('bW'bX CP)
('bW'bZ CP)
('bZ'bX CP)
('bZ'bW CP)
('bU'bX CP)
('bU'bW CP)
('bU'bZ CP)
```

For vectors using exclusively based literals, the set of active vectors is the vector itself, the set of inactive vectors is any vector with at least one different literal, and the set of ambiguous vectors is empty.

Therefore, ALF does not provide a default behavior for ambiguous vectors, since the behavior for each vector can be explicitly defined in vectors using based literals.

10.4.5 Concurrency in sequential functions

The principle of concurrency applies also for edge-sensitive sequential functions, where the triggering condition is described by a vector expression rather than a boolean expression. In edge-sensitive logic, the target logic variable for the boolean assignment (LHS) can also be an operand of the boolean expression defining the assigned value (RHS). Concurrency implies that the RHS expressions are evaluated immediately *before* the triggering edge, and the values are assigned to the LHS variables immediately *after* the triggering edge. This is illustrated in Figure 17.

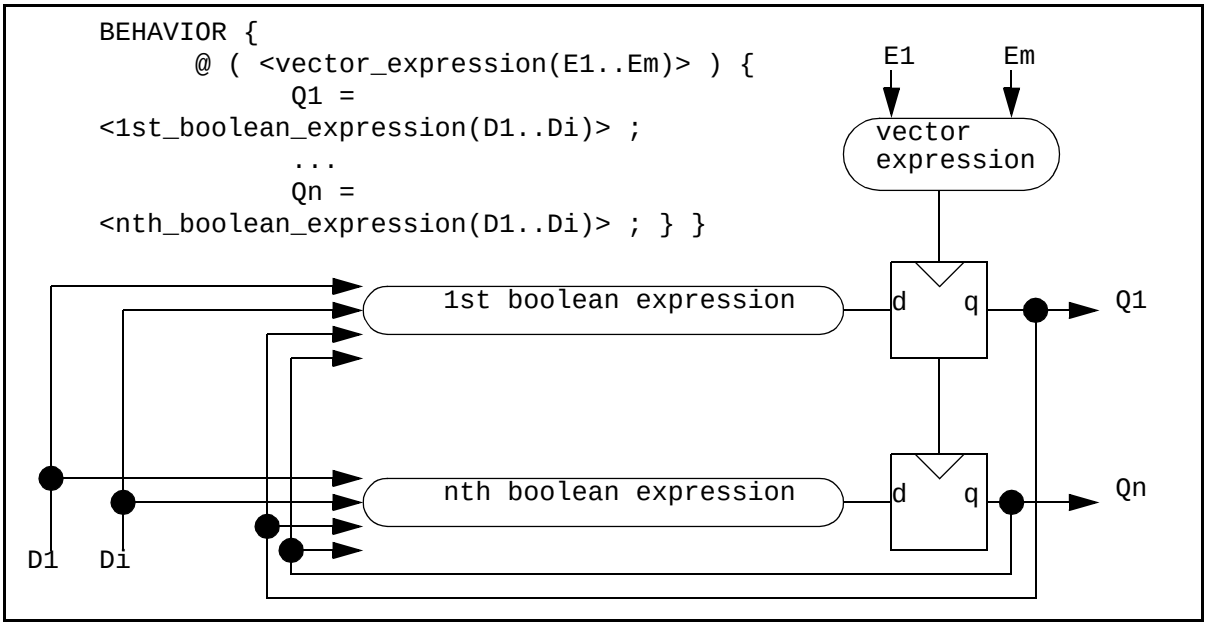


Figure 17—Concurrency for edge-sensitive sequential logic

Statements with multiple concurrent conditions for boolean assignments can also be used in sequential logic. In that case conflicting values can be assigned to the same logic variable. A default conflict resolution is not provided for the following reasons.

- Conflict resolution might not be necessary, since the conflicting situation is prohibited by specification.
- For different types of analysis (e.g., logic simulation), a different conflict resolution behavior might be desirable, while the physical behavior of the circuit shall not change. For instance, pessimistic conflict resolution always assigns X, more accurate conflict resolution first checks whether the values are conflicting. Different choices can be motivated by a trade-off in analysis accuracy and runtime.
- If complete library control over analysis is desired, conflict resolution can be specified explicitly.

Example

```
BEHAVIOR {
    @ ( <condition_1> ) { Q = <value_1>; }
    @ ( <condition_2> ) { Q = <value_2>; }
}
```

Explicit pessimistic conflict resolution can be described as follows:

```
BEHAVIOR {
    @ ( <condition_1> && <condition_2>  ) { Q = 'bX; }
    @ ( <condition_1> && ! <condition_2>) { Q = <value_1>; }
    @ ( <condition_2> && ! <condition_1>) { Q = <value_2>; }
}
```

Explicit accurate conflict resolution can be described as follows:

```
BEHAVIOR {
    @ ( <condition_1> && <condition_2>  ) {
       Q = (<value_1>==<value_2>)? <value_1> : 'bX;
    }
    @ ( <condition_1> && ! <condition_2>) { Q = <value_1>; }
    @ ( <condition_2> && ! <condition_1>) { Q = <value_2>; }
}
```

Since the conditions are now rendered mutually exclusive, equivalent descriptions with priority statements can be used. They are more elegant than descriptions with concurrent statements.

```
BEHAVIOR {
    @ ( <condition_1> && <condition_2>  ) {
       Q = <conflict_resolution_value>;
    }
    : ( <condition_1> ) { Q = <value_1>; }
    : ( <condition_2> ) { Q = <value_2>; }
}
```

Given the various explicit description possibilities, the standard does not prescribe a default behavior. The model developer has the freedom of incomplete specification.

10.4.6 Initial values for logic variables

Per definition, all logic variables in a behavioral description have the initial value U which means “uninitialized”. This value cannot be assigned to a logic variable, yet it can be used in a behavioral description in order to assign other values than U after initialization.

Example

```
BEHAVIOR {
    @ ( Q1 == 'bU ) { Q1 = 'b1 ; }
    @ ( Q2 == 'bU ) { Q2 = 'b0 ; }
    // followed by the rest of the behavioral description
}
```

A template can be used to make the intent more obvious, for example:

```
TEMPLATE VALUE_AFTER_INITIALIZATION {
    @ ( <logic_variable> == 'bU ) { <logic_variable> = <initial_value> ; }
}
BEHAVIOR {
    VALUE_AFTER_INITIALIZATION ( Q1 'b1' )
    VALUE_AFTER_INITIALIZATION ( Q2 'b0' )
    // followed by the rest of the behavioral description
}
```

Logic variables in a vector expression shall be declared as PINs. It is possible to annotate initial values directly to a pin. Such variables shall never take the value U. Therefore vector expressions involving U for such variables (see the previous example) are meaningless.

Example

```
PIN Q1 { INITIAL_VALUE = 'b1 ; }
PIN Q2 { INITIAL_VALUE = 'b0 ; }
```

10.5 Higher-order sequential functions

This section defines the different types of higher-order sequential functions in ALF.

10.5.1 Vector-sensitive sequential logic

Vector expressions can be used to model generalized higher order sequential logic; they are an extension of the boolean expressions. A *vector expression* describes sequences of logical events or transitions in addition to static logical states. A vector expression represents a description of a logical stimulus without timescale. It describes the order of occurrence of events.

The -> operator (*followed by*) gives a general capability of describing a sequence of events or a vector. For example, consider the following vector expression:

```
01 A -> 01 B
```

which reads "rising edge on A is followed by rising edge on B".

A vector expression is evaluated by an event sequence detection function. Like a single event or a transition, this function evaluates *True* only at an infinitely short time when the event sequence is detected, as shown in Figure 18.

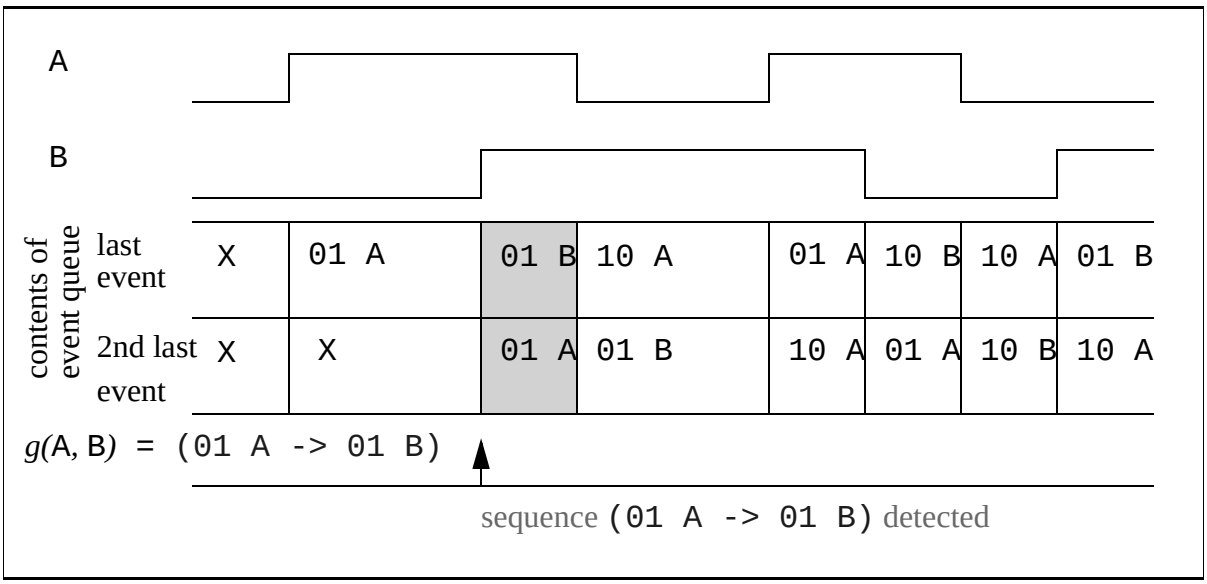


Figure 18—Example of event sequence detection function

The event sequence detection mechanism can be described as a queue that sorts events according to their order of arrival. The event sequence detection function evaluates *True* at exactly the time when a new event enters the queue and forms the required sequence, i.e., *the sequence specified by the vector expression* with its preceding events.

A vector-sensitive sequential logic can be called *(N+1) order sequential logic*, where *N* is the number of events to be stored in the queue. The implementation of *(N+1)* order sequential logic requires *N* memory elements for the event queue and one memory element for the output itself.

A sequence of events can also be gated with static logical conditions. In the example,

```
(01 CP -> 10 CP) && CD
```

the pin `CD` shall have `state 1` from some time before the rising edge at `CP` to some time after the falling edge of `CP`. The pin `CD` can not go low (`state 0`) after the rising edge of `CP` and go high again before the falling edge of `CP` because this would insert events into the queue and the sequence "rising edge on `CP` followed by falling edge on `CP`" would not be detected.

The formal calculation rules for general vector expressions featuring both states and transitions are detailed in 10.5.2 and 10.5.3.

The concept of vector expression supports functional modeling of devices featuring digital communication protocols with arbitrary complexity.

10.5.2 Canonical binary operators for vector expressions

The following canonical binary operators are necessary to define sequences of transitions:

- `vector_followed_by` for completely specified sequence of events
- `vector_and` for simultaneous events
- `vector_or` for alternative events
- `vector_followed_by` for incompletely specified sequence of events

The symbols for the boolean operators for `AND` and `OR` are overloaded for `vector_and` and `vector_or`, respectively. The new symbols for the `vector_followed_by` operators are shown in Table 77.

Table 77—Canonical binary vector operators

Operator	Operands	LHS, RHS commutative	Description
->	2 vector expressions	No	Left-hand side (LHS) transition *is followed by* Right-hand side (RHS) transition, no transition can occur in-between.
&&, &	2 vector expressions	Yes	LHS *and* RHS transition *occur simultaneously.*
||, |	2 vector expressions	Yes	LHS *or* RHS transition *occur alternatively.*
~>	2 vector expressions	No	Left-hand side (LHS) transition *is followed by* Right-hand side (RHS) transition, other transitions can occur in-between.

Per definition, the -> and ~> operators shall not be commutative, whereas the && and || operators on events shall be commutative.

```
01 a && 01 b === 01 b && 01 a
01 a || 01 b === 01 b || 01 a
```

The -> and ~> operators shall be freely associative.

```
01 a -> 01 b -> 01 c === (01 a -> 01 b) -> 01 c === 01 a -> (01 b -> 01 c)
01 a ~> 01 b ~> 01 c === (01 a ~> 01 b) ~> 01 c === 01 a ~> (01 b ~> 01 c)
```

The && operator is defined for single events and for event sequences with the same number of -> operators each.

```
(01 A1 .. -> ... 01 AN) & (01 B1 .. -> ... 01 BN)
===
01 A1 & 01 B1 ... -> ... 01 AN & 01 BN
```

The || operator reduces the set of edge operators (unary vector operators) to canonical and non-canonical operators.

```
(?? a) === (?! a)||(?- a)  //a does or does not change its value
```

Hence ?? is non-canonical, since it can be defined by other operators.

If <value1><value2> is an edge operator consisting of two based literals value1 and value2 and word is an expression which can take the value value1 or value2, then the following vector expressions are considered equivalent:

```
<value1><value2> <word>
    ===   10 (<word> == <value1>) && 01 (<word> == <value2>)
    ===   01 (<word> != <value1>) && 01 (<word> == <value2>)
    ===   10 (<word> == <value1>) && 10 (<word> != <value2>)
    ===   01 (<word> != <value1>) && 10 (<word> != <value2>)
// all expressions describe the same event:
// <word> makes a transition from <value1> to <value2>
```

Hence vector expressions with edge operators using based literals can be reduced to vector expressions using only the edge operators 01 and 10.

10.5.3 Complex binary operators for vector expressions

Table 78 defines the complex binary operators for vector operators.

Table 78—Complex binary vector operators

Operator	Operands	LHS, RHS commutative	Description
<->	2 vector expressions	Yes	LHS transition follows or is followed by RHS transition.
&>	2 vector expressions	No	LHS transition *is followed by or occurs simultaneously* with RHS transition.

Table 78—Complex binary vector operators (Continued)

Operator	Operands	LHS, RHS commutative	Description
<&>	2 vector expressions	Yes	LHS transition *follows or is followed by or occurs simultaneously* with RHS transition.

The following expressions shall be considered equivalent:

```
(01 a <-> 01 b) === (01 a -> 01 b)||(01 b -> 01 a)
(01 a  &> 01 b) === (01 a -> 01 b)||(01 a && 01 b)
(01 a <&> 01 b) === (01 a -> 01 b)||(01 b -> 01 a)||(01 a && 01 b)
```

By their symmetric definition, the `<->` and `<&>` operators are commutative.

```
01 a <-> 01 b === 01 b <-> 01 a
01 a <&> 01 b === 01 b <&> 01 a
```

The commutative complex binary vector operators are defined in Table 77. The commutativity rules are only defined for two operands:

— commutative “followed by”:

```
vect_expr1 <-> vect_expr2 ===
    vect_expr1 -> vect_expr2 // vect_expr1 occurs first
|   vect_expr2 -> vect_expr1 // vect_expr2 occurs first
```

— commutative “followed by or simultaneously occurring”:

```
vect_expr1 <&> vect_expr2 ===
    vect_expr1 -> vect_expr2 // vect_expr1 occurs first
|   vect_expr2 -> vect_expr1 // vect_expr2 occurs first
|   vect_expr1 && vect_expr2 // both occur simultaneously
```

10.5.4 Extension to N operands

This section defines how to use *N* operands.

A `complex_vector_expression` of the form

```
vector_expression { <-> vector_expression }
```

shall be commutative for all operands. The `complex_vector_expression` describes alternative event sequences in which the temporal order of each constituent `vector_expression` is completely permutable, excluding simultaneous occurrence of each constituent `vector_expression`.

A `complex_vector_expression` of the form

```
vector_expression { <&> vector_expression }
```

shall be commutative for all operands. The `complex_vector_expression` describes alternative event sequences in which the temporal order of each constituent `vector_expression` is completely permutable, including simultaneous occurrence of each constituent `vector_expression`.

Example

```
01 A <-> 01 B <-> 01 C ===
    01 A -> 01 B -> 01 C
|   01 B -> 01 C -> 01 A
|   01 C -> 01 A -> 01 B
|   01 C -> 01 B -> 01 A
|   01 B -> 01 A -> 01 C
|   01 A -> 01 C -> 01 B

01 A <&> 01 B <&> 01 C ===
    01 A -> 01 B -> 01 C
|   01 B -> 01 C -> 01 A
|   01 C -> 01 A -> 01 B
|   01 C -> 01 B -> 01 A
|   01 B -> 01 A -> 01 C
|   01 A -> 01 C -> 01 B
|   01 A && 01 B -> 01 C
|   01 A -> 01 B && 01 C
|   01 B && 01 C -> 01 A
|   01 B -> 01 C && 01 A
|   01 C && 01 A -> 01 B
|   01 C -> 01 A && 01 B
|   01 A && 01 B && 01 C
```

10.5.4.1 Boolean rules

The following rule applies for a boolean `AND` operation with three operands:

```
rule 1:
A & B & C === (A & B) & C | A & (B & C)
```

A corresponding rule also applies to the commutative followed-by operation with three operands:

```
rule 2:
01 A <-> 01 B <-> 01 C ===
    (01 A <-> 01 B) <-> 01 C
|   01 A <-> (01 B <-> 01 C)
```

The alternative boolean expressions `(A & B) & C` and `A & (B & C)` in `rule 1` are equivalent. Therefore, `rule 1` can be reduced to the following:

```
rule 3:
A & B & C === (A & B) & C === (B & C) & A
```

A corresponding rule does *not* apply to complex vector operands, since each expression with associated operands generates only a subset of permutations:

```
(01 A <-> 01 B) <-> 01 C ===
    (01 A <-> 01 B) -> 01 C)
|   (01 C -> (01 A <-> 01 B)) ===
    01 A -> 01 B -> 01 C
|   01 B -> 01 A -> 01 C
```

```
|   01 C -> 01 A -> 01 B
|   01 C -> 01 B -> 01 A
```

The permutations

```
01 A -> 01 C -> 01 B
01 B -> 01 C -> 01 A
```

are missing.

```
01 A <-> (01 B <-> 01 C) ===
    (01 A -> (01 B <-> 01 C))
|   ((01 B <-> 01 C) -> 01 A) ===
    01 A -> 01 B -> 01 C
|   01 A -> 01 C -> 01 B
|   01 B -> 01 C -> 01 A
|   01 C -> 01 B -> 01 A
```

The permutations

```
|   01 B -> 01 A -> 01 C
|   01 C -> 01 A -> 01 B
```

are missing.

10.5.5 Operators for conditional vector expressions

The definitions of the &&, ?, and : operators are also overloaded to describe a *conditional vector expression* (involving boolean expressions and vector expressions), as shown in Table 79. The clauses are boolean expressions; while vector expressions are subject to those clauses.

Table 79—Operators for conditional vector expressions

Operator	Operands	LHS, RHS commutative	Description
&&, &	1 vector expression, 1 boolean expression	Yes	Boolean expression (LHS or RHS) is *True* while sequence of transitions, defined by vector expression (RHS or LHS) occurs.
?	1 vector expression, 1 boolean expression	No	Boolean condition operator for construction of if-then-else clause involving vector expressions.
:	1 vector expression, 1 boolean expression	No	Boolean else operator for construction of if-then-else clause involving vector expressions.

An example for conditional vector expression using && is given below:

```
(01 a && !b)    // a rises while b==0
```

The order of the operands in a conditional vector expression using && shall not matter.

```
<vector_exp> && <boolean_exp> === <boolean_exp> && <vector_exp>
```

The && operator is still commutative in this case, although one operand is a boolean expression defining a static state, the other operand is a vector expression defining an event or a sequence of events. However, since the operands are distinguishable per se, it is not necessary to impose a particular order of the operands.

An example for conditional vector expression using `?` and `:` is given below.

```
!b ? 01 a : c ? 10 b : 01 d
===
!b & 01 a | !(!b) & c & 10 b | !(!b) & !c & 01 d
```

This example shows how a conditional vector expression using ternary operators can be expressed with alternative conditional vector expressions.

A conditional vector expression can be reduced to a non-conditional vector expression in some cases (see 10.6.11).

Every binary vector operator can be applied to a conditional vector expression.

10.5.6 Operators for sequential logic

Table 80 defines the complex binary operators for vector operators.

Table 80—Operators for sequential logic

Operator	Description
@	Sequential `if` operator, followed by a boolean logic expression (for level-sensitive assignment) or by a vector expression (for edge-sensitive assignment).
:	Sequential `else if` operator, followed by a boolean logic expression (for level-sensitive assignment) or by a vector expression (for edge-sensitive assignment) with lower priority.

Sequential assignments are constructed as follows:

```
@ ( <trigger1> ) { <action1> } : ( <trigger2> ) { <action2> } :
  ( <trigger3> ) { <action3> }
```

If `trigger1` event is detected, then `action1` is performed; else if `trigger2` event is detected, then `action2` is performed; else if `trigger3` event is detected, then `action3` is performed as a result of this clause.

10.5.7 Operator priorities

The priority of binding operators to operands in non-conditional vector expressions shall be from strongest to weakest in the following order:

a) unary vector operators (edge literals)

b) complex binary vector operators (<->, &>, <&>)
c) vector AND (&, &&)
d) vector_followed_by operators (->, ~>)
e) vector OR (|, ||)

10.5.8 Using PINs in VECTORs

A VECTOR defines state, transition, or sequence of transitions of pins that are controllable and observable for characterization.

Within a CELL, the set of PINs with SCOPE=behavior or SCOPE=measure or SCOPE=both is the default set of variables in the event queue for vector expressions relevant for BEHAVIOR or VECTOR statements or both, respectively.

For detection of a sequence of transitions it is necessary to observe the set of variables in the event queue. For instance, if the set of pins consists of A, B, C, D, the vector expression

```
(01 A -> 01 B)
```

implies no transition on A, B, C, D occurs between the transitions 01 A and 01 B.

The default set of pins applies only for vector expressions without conditions. The conditional event AND operator limits the set of variables in the event queue. In this case, only the state of the condition and the variables appearing in the vector expression are observed.

Example

```
(01 A -> 01 B) && (C | D)
```

No transition on A, B occurs between 01 A and 01 B, and (C | D) needs to stay *True* in-between 01 A and 01 B as well. However, C and D can change their values as long as (C | D) is satisfied.

10.6 Modeling with vector expressions

Vector expressions provide a formal language to describe digital waveforms. This capability can be used for functional specification, for timing and power characterization, and for timing and power analysis.

In particular, vector expressions add value by addressing the following modeling issues:

— *Functional specification*: complex sequential functionality, e.g., bus protocols.
— *Timing analysis*: complex timing arcs and timing constraints involving more than two signals.
— *Power analysis*: temporal and spatial correlation between events relevant for power consumption.
— *Circuit characterization and test*: specification of characterization and/or test vectors for particular timing, power, fault, or other measurements within a circuit.

Like boolean expressions, vector expressions provide the means for describing the functionality of digital circuits in various contexts without being self-sufficient. Vector expressions enrich this functional description capability by adding a “dynamic” dimension to the otherwise “static” boolean expressions.

The following subsections explain the semantics of vector expressions step-by-step. The vector expression concept is explained using terminology from simulation event reports. However, the application of vector expressions is not restricted to post-processing event reports.

Some application tools (e.g., power analysis tools) can actually evaluate vector expressions during post-processing of event reports from simulation. Other application tools, especially simulation model generators, need to respect the causality between the triggering events and the actions to be triggered. While it is semantically impossible to describe cause and effect in the same vector expression for the purpose of functional modeling, both cause and effect can appear in a vector expression used for a timing arc description.

ALF does not make assumption about the physical nature of the event report. Vector expressions can be applied to an actual event report written in a file, to an internal event queue within a simulator, or to a hypothetical event report which is merely a mathematical concept.

10.6.1 Event reports

This section describes the terminology of event reports from simulation, which is used to explain the concept of ALF vector expressions. The intent of ALF vector expressions is not to *replace* existing event report formats. Non-pertinent details of event report formats are not described here.

Simulation events (e.g., from Verilog or VHDL) can be reported in a value change dump (VCD) file, which has the following general form:

```
<time1>
    <variableA> <stateU>
    <variableB> <stateV>
    ...
<time2>
    <variableC> <stateW>
    <variableD> <stateX>
    ...
<time3> ...
```

The set of variables for which simulation events are reported, i.e., the *scope* of the event report needs to be defined beforehand. Each variable also has a definition for the *set of states* it can take. For instance, there can be binary variables, 16-bit integer variables, 1-bit variables with drive-strength information, etc. Furthermore, the initial state of each variable shall be defined as well. In an ALF context, the terms *signal* and *variable* are used interchangeably. In VHDL, the corresponding term is *signal*. In Verilog, there is no single corresponding term. All `input`, `output`, `wire`, and `reg` variables in Verilog correspond to a `signal` in VHDL.

The time values `<time1>`, `<time2>`, `<time3>`, etc. shall be in increasing order. The order in which simultaneous events are reported does not matter. The number of time points and the number of simultaneous events at a certain time point are unlimited.

In the physical world, each event or change of state of a variable takes a certain amount of time. A variable cannot change its state more than once at a given point in time. However, in simulation, this time can be smaller than the resolution of the time scale or even zero (`0`). Therefore, a variable can change its state more than once at a given point in simulation time. Those events are, strictly speaking, not simultaneous. They occur in a certain order, separated by an infinitely small delta-time. Multiple simultaneous events of the same variable are not reported in the VCD. Only the final state of each variable is reported.

A VCD file is the most compact format that allows reconstruction of entire waveforms for a given set of variables. A more verbose form is the test pattern format.

```
<TIME>  <variableA> <variableB> <variableC> <variableD>
<time1> <stateU>    <stateV>    ...         ...
<time2> <stateU>    <stateV>    <stateW>    <stateX>
<time3> ...         ...         ...         ...
```

The test pattern format reports the state of each variable at every point in time, regardless of whether the state has changed or not. Previous and following states are immediately available in the previous and next row, respectively. This makes the test pattern format more readable than the VCD and well-suited for taking a snapshot of events in a time window.

An example of an event report in VCD format:

```
// initial values
A 0   B 1   C 1   D X   E 1
// event dump
109   A 1   D 0
258   B 0
573   C 0
586   A 0
643   A 1
788   A 0   B 1   C 1
915   A 1
1062  E 0
1395  B 0   C 0
1640  A 0   D 1
// end of event dump
```

An example of an event report in test pattern format:

```
time   A   B   C   D   E
0      0   1   1   X   1
109    1   1   1   0   1
258    1   0   1   0   1
573    1   0   0   0   1
586    0   0   0   0   1
643    1   0   0   0   1
788    0   1   1   0   1
915    1   1   1   0   1
1062   1   1   1   0   0
1395   1   0   0   0   0
1640   0   0   0   1   0
```

Both VCD and test pattern formats represent the same amount of information and can be translated into each other.

10.6.2 Event sequences

For specification of a functional waveform (e.g., the write cycle of a memory), it is not practical to use an event report format, such as a VCD or test pattern format. In such waveforms, there is no absolute time. And the relative time, for example, the setup time between address change and write enable change, can vary from one instance to the other.

The main purpose of `vector_expressions` is waveform specification capability. The following operators can be used:

— `vector_unary` (also called *edge operator* or *unary vector operator*)
The edge operator is a prefix to a variable in a vector expression. It contains a pair of states, the first being the previous state, the second being the new state. Edge operators can describe a change of state or no change of state.

— `vector_and` (also called *simultaneous event operator*)
This operator uses the overloaded symbol `&` or `&&` interchangeably. The `&` operator is the separator between simultaneously occurring events
— `vector_followed_by` (also called *followed-by operator*)
The "immediately followed-by operator" using the symbol `->` is treated first. The `->` operator is the separator between consecutively occurring events.

These operators are necessary and sufficient to describe the following subset of `vector_expressions`:

a) vector_single_event
A change of state in a single variable, for example:
```
01 A
```
b) vector_event
A simultaneous change of state in one or more variables, for example:
```
01 A & 10 B
```
c) vector_event_sequence
Subsequently occurring changes of state in one or more variables, for example:
```
01 A & 10 B -> 10 A
```

The `vector_and` operator has a higher binding priority than the `vector_followed_by` operator.

We can now express the pattern of the sample event report in a `vector_event_sequence` expression:

```
01 A & X0 D -> 10 B -> 10 C -> 10 A -> 01 A
-> 10 A & 01 B & 01 C -> 01 A -> 10 E -> 10 B & 10 C -> 10 A & 01 D
```

We can define the *length* of a `vector_event_sequence` expression as the number of subsequent events described in the `vector_event_sequence` expression. The length is equal to the number of `->` operators plus one (`1`).

Although the vector expression format contains an inherent redundancy, since the old state of each variable is always the same as the new state of the same variable in a previous event, it is more human-readable, especially for waveform description. On the other hand, it is more compact than the test pattern format. For short event sequences, it is even more compact than the VCD, since it eliminates the declaration of initial values. To be accurate, for variables with exactly one event the vector expression is more compact than the VCD. For variables with more than one event the VCD is more compact than the vector expression. In summary, the vector expression format offers readability similar to the test pattern format and compactness close to the VCD format.

10.6.3 Scope and content of event sequences

The *scope* applicable to a vector expression defines the set of variables in the event report. The *content* of a vector expression is the set of variables that appear in the vector expression itself. The content of a vector expression shall be a subset of variables within scope.

— `PIN`s with the annotation `SCOPE = BEHAVIOR` are applicable variables for vector expressions within the context of `BEHAVIOR`.
— `PIN`s with the annotation `SCOPE = MEASURE` are applicable variables for vector expressions within the context of `VECTOR`.
— `PIN`s with the annotation `SCOPE = BOTH` are applicable variables for all vector expressions.

A `vector_event_sequence` expression is an event pattern without time, containing only the variables within its own content. This event pattern is evaluated against the event report containing all variables within scope. The vector expression is *True* when the event pattern matches the event report.

Example

```
time  A   B   C   D   E   // scope is A, B, C, D, E
0     0   1   1   X   1
109   1   1   1   0   1
258   1   0   1   0   1
573   1   0   0   0   1
586   0   0   0   0   1
643   1   0   0   0   1
788   0   1   1   0   1
915   1   1   1   0   1
1062  1   1   1   0   0
1395  1   0   0   0   0
1640  0   0   0   1   0
```

Consider the following vector expressions in the context of the sample event report:

```
01 A                                          //(1) content is A
//event pattern expressed by (1):
//   A
//   0
//   1
```

`(1)` is *True* at time 109, time 643, and time 915.

```
10 B -> 10 C                                  //(2) content is B, C
//event pattern expressed by (2):
//   B   C
//   1   1
//   0   1
//   0   0
```

`(2)` is *True* at time 573.

```
10 A -> 01 A                                  //(3) content is A
//event pattern expressed by (3):
//   A
//   1
//   0
//   1
```

`(3)` is *True* at time 643 and time 915.

```
01 D                                          //(4) content is D
//event pattern expressed by (4):
//   D
//   0
//   1
```

`(4)` is *True* at time 1640.

```
01 A -> 10 C                                  //(5) content is A, C
//event pattern expressed by (5):
//   A   C
```

```
//   0   1
//   1   1
//   1   0
```

`(5)` is not be *True* at any time, since the event pattern expressed by `(5)` does not match the event report at any time.

10.6.4 Alternative event sequences

The following operator can be used to describe alternative events:

> `vector_or`, also called *event-or operator* or *alternative-event operator*, using the overloaded symbol `|` or `||` interchangeably. The `|` operator is the separator between alternative events or alternative event sequences.

In analogy to boolean operators, `|` has a lower binding priority than & and `->`. Parentheses can be used to change the binding priority.

Example

```
(01 A -> 01 B) | 10 C === 01 A -> 01 B | 10 C
01 A -> (01 B | 10 C) === 01 A -> 01 B | 01 A -> 10 C
```

Consider the following vector expressions in the context of the sample event report:

```
01 A | 10                                          //(6)
//event pattern expressed by (6):
//   A
//   0
//   1
//alternative event pattern expressed by (6):
//   C
//   1
//   0
```

`(6)` is *True* at time 109, time 573, time 643, time 915, and time 1395.

```
10 B -> 10 C | 10 A -> 01 A                        //(7)
//event pattern expressed by (7):
//   B   C
//   1   1
//   0   1
//   0   0
//alternative event pattern expressed by (7):
//   A
//   1
//   0
//   1
```

`(7)` is *True* at time 573, time 643, and time 915.

```
01 D | 10 B -> 10 C                                //(8)
```

```
//event pattern expressed by (8):
//   D
//   0
//   1
//alternative event pattern expressed by (8):
//   B   C
//   1   1
//   0   1
//   0   0
```

(8) is *True* at time 573 and time 1640.

```
10 B -> 10 C | 10 A                              //(9)
//event pattern expressed by (9):
//   B   C
//   1   1
//   0   1
//   0   0
//alternative event pattern expressed by (9):
//   A
//   1
//   0
```

(9) is *True* at time 573, time 586, time 788, and time 1640.

The following operators provide a more compact description of certain alternative event sequences:

— &> events occur simultaneously or follow each other in the order RHS after LHS
— <-> a LHS event followed by a RHS event or a RHS event followed by a LHS event
— <&> events occur simultaneously or follow each other in arbitrary order

Example

```
01 A &> 01 C    ===   01 A & 01 C | 01 A -> 01 C
01 A <-> 01 C   ===   01 A -> 01 C | 01 C -> 01 A
01 A <&> 01 C   ===   01 A <-> 01 C | 01 A & 01 C
```

The binding priority of these operators is higher than of & and ->.

10.6.5 Symbolic edge operators

Alternative events of the same variable can be described in a even more compact way through the use of edge operators with symbolic states. The symbol ? stands for “any state”.

— edge operator with ? as the previous state:
  transition from any state to the defined new state
— edge operator with ? as the next state:
  transition from the defined previous state to any state.

Both edge operators include the possibility no transition occurred at all, i.e., the previous and the next state are the same. This situation can be explicitly described with the following operator:

edge operator with next state = previous state, also called *non-event operator*
The operand stays in the state defined by the operator.

The following symbolic edge operators also can be used:

a) `?-` no transition on the operand
b) `?!` transition from any state to any state different from the previous state
c) `??` transition from any state to any state or no transition on the operand
d) `?~` transition from any state to its bitwise complementary state

Example

Let `A` be a logic variable with the possible states `1`, `0`, and `X`.

```
?0 A === 00 A | 10 A | X0 A
?1 A === 01 A | 11 A | X1 A
?X A === 0X A | 1X A | XX A
0? A === 00 A | 01 A | 0X A
1? A === 10 A | 11 A | 1X A
X? A === X0 A | X1 A | XX A
?! A === 01 A | 0X A | 10 A | 1X A | X0 A | X1 A
?~ A === 01 A | 10 A | XX A
?? A === 00 A | 01 A | 0X A | 10 A | 11 A | 1X A | X0 A | X1 A | XX A
?- A === 00 A | 11 A | XX A
```

For variables with more possible states (e.g., logic states with different drive strength and multiple bits) the explicit description of alternative events is quite verbose. Therefore the symbolic edge operators are useful for a more compact description.

This completes the set of `vector_binary` operators necessary for the description of a subset of `vector_expressions` called `vector_complex_event` expressions. All `vector_binary` operators have two `vector_complex_event` expressions as operands. The set of `vector_event_sequence` expressions is a subset of `vector_complex_event` expressions. Every `vector_complex_event` expression can be expressed in terms of alternative `vector_event_sequence` expressions. The latter could be called *minterms*, in analogy to boolean algebra.

10.6.6 Non-events

A `vector_single_event` expression involving a non-event operator is called a *non-event*. A rigorous definition is required for `vector_complex_event` expressions containing non-events. Consider the following example of a flip-flop with clock input `CLK` and data output `Q`.

```
01 CLK -> 01 Q   // (i)
01 CLK -> 00 Q   // (ii)
```

The vector expression `(i)` describes the situation where the output switches from `0` to `1` after the rising edge of the clock. The vector expression `(ii)` describes the situation where the output remains at `0` after the rising edge of the clock.

How is it possible to decide whether `(i)` or `(ii)` is *True*, without knowing the delay between `CLK` and `Q`? The only way is to wait until any event occurs after the rising edge of `CLK`. If the event is not on `Q` and the state of `Q` is `0` during that event, then `(ii)` is *True*.

Hence, a non-event is *True* every time when another event happens and the state of the variable involved in the non-event satisfies the edge operator of the non-event.

Example

```
time  A   B   C   D   E
0     0   1   1   X   1
109   1   1   1   0   1
258   1   0   1   0   1
573   1   0   0   0   1
586   0   0   0   0   1
643   1   0   0   0   1
788   0   1   1   0   1
915   1   1   1   0   1
1062  1   1   1   0   0
1395  1   0   0   0   0
1640  0   0   0   1   0
```

The test pattern format represents an event, for example `01 A`, in no different way than a non-event, for example `11 E`. This non-event is *True* at times 109, 258, 573, 586, 643, 788, and 915; in short, every time when an event happens while `E` is constant `1`.

10.6.7 Compact and verbose event sequences

A `vector_event_sequence` expression in a compact form can be transformed into a verbose form by padding up every `vector_event` expression with non-events. The next state of each variable within a `vector_event` expression shall be equal to the previous state of the same variable in the subsequent `vector_event` expression.

Example

```
01 A -> 10B === 01 A & 11 B -> 11 A & 10 B
```

A vector expression for a complete event report in compact form resembles the VCD, whereas the verbose form looks like the test pattern.

```
// compact form
01 A & X0 D -> 10 B -> 10 C -> 10 A -> 01 A
-> 10 A & 01 B & 01 C -> 01 A -> 10 E
-> 10 B & 10 C -> 10 A & 01 D
===
// verbose form
?0 A & ?1 B & ?1 C & ?X D & ?1 E   ->
01 A & 11 B & 11 C & X0 D & 11 E   ->
11 A & 10 B & 11 C & 00 D & 11 E   ->
11 A & 00 B & 10 C & 00 D & 11 E   ->
10 A & 00 B & 00 C & 00 D & 11 E   ->
01 A & 00 B & 00 C & 00 D & 11 E   ->
10 A & 01 B & 01 C & 00 D & 11 E   ->
01 A & 11 B & 11 C & 00 D & 11 E   ->
11 A & 11 B & 11 C & 00 D & 10 E   ->
11 A & 10 B & 10 C & 00 D & 00 E   ->
10 A & 00 B & 00 C & 01 D & 00 E
```

The transformation rule needs to be slightly modified in case the compact form contains a `vector_event` expression consisting only of non-events. By definition, the non-event is *True* only if a real event happens simultaneously with the non-event. Padding up a `vector_event` expression consisting of non-events with other non-events make this impossible. Rather, this `vector_event` expression needs to be padded up with unspeci-

fied events, using the ?? operator. Eventually, unspecified events can be further transformed into partly specified events, if a former or future state of the involved variable is known.

Example

```
01 A -> 00 B
=== 01 A & 00 B -> ?? A & 00 B
```

In the first transformation step, the unspecified event ?? A is introduced.

```
01 A & 00 B -> ?? A & 00 B
=== 01 A & 00 B -> 1? A & 00 B
```

In the second step, this event becomes partly specified. ?? A is bound to be 1? A due to the previous event on A.

10.6.8 Unspecified simultaneous events within scope

Variables which are within the scope of the vector expression yet do not appear in the vector expression, can be used to pad up the vector expression with unspecified events as well. This is equivalent to omitting them from the vector expression.

Example

```
01 A -> 10 B    // let us assume a scope containing A, B, C, D, E
===
01 A & 10 B & ?? C & ?? D & ?? E -> 11 A & 10 B & ?? C & ?? D & ?? E
```

This definition allows unspecified events to occur *simultaneously* with specified events or specified non-events. However, it disallows unspecified events to occur *in-between* specified events or specified non-events.

At first sight, this distinction seems to be arbitrary. Why not disallow unspecified events altogether? Yet there are several reasons why this definition is practical.

If a vector expression disallows simultaneously occurring unspecified events, the application tool has the burden not only to match the pattern of specified events with the event report but also to check whether the other variables remain constant. Therefore, it is better to specify this extra pattern matching constraint explicitly in the vector expression by using the ?- operator.

There are many cases where it actually does not matter whether simultaneously occurring unspecified events are allowed or disallowed:

- *Case 1*: Simultaneous events are impossible by design of the flip-flop. For instance, in a flip-flop it is impossible for a triggering clock edge 01 CK and a switch of the data output ? Q to occur at the same time. Therefore, such events can not appear in the event report. It makes no difference whether 01 CK & ?- Q, 01 CK & ?? Q, or 01 CK is specified. The only occurring event pattern is 01 CK & ?- Q and this pattern can be reliably detected by specifying 01 CK.
- *Case 2*: Simultaneous events are prohibited by design. For instance, in a flip-flop with a positive setup time and positive hold time, the triggering clock edge 01 CK and a switch of the data input ?! D is a timing violation. A timing checker tool needs the violating pattern specified explicitly, i.e., 01 CK & ?! D. In this context, it makes sense to specify the non-violating pattern also explicitly, i.e., 01 CK & ?- D. The pattern 01 CK by itself is not applicable.
- *Case 3*: Simultaneous events do not occur in correct design. For instance, power analysis of the event 01 CK needs no specification of ?! D or ?- D. In the analysis of an event report with timing violations, the

power analysis is less accurate anyway. In the analysis of the event report for the design without timing violation, the only occurring event pattern is `01 CK & ?- D` and this pattern can be reliably detected by specifying `01 CK`.[2]

— *Case 4*: The effects of simultaneous events are not modeled accurately. This is the case in static timing analysis and also to some degree in dynamic timing simulation. For instance, a `NAND` gate can have the inputs `A` and `B` and the output `Z`. The event sequence exercising the timing arc `01 A -> 10 Z` can only happen if `B` is constant `1`. No event on `B` can happen in-between `01 A` and `10 Z`. Likewise, the timing arc `01 B -> 10 Z` can only happen if `A` is constant `1` and no event happens in-between `01 B` and `10 Z`. The timing arc with simultaneously switching inputs is commonly ignored. A tool encountering the scenario `01 A & 01 B -> 10 Z` has no choice other than treating it arbitrarily as `01 A -> 10 Z` or as `01 B -> 10 Z`.

— *Case 5*: The effects of simultaneous events are modeled accurately. Here it makes sense to specify all scenarios explicitly, e.g., `01 A & ?- B -> 10 Z`, `01 A &?! B -> 10 Z`, `?- A & 01 B -> 10 Z`, etc., whereas the patterns `01 A -> 10 Z` and `01 B -> 10 Z` by themselves apply only for less accurate analysis (see *Case 4*).

There is also a formal argument why unspecified events on a vector expression need to be allowed rather than disallowed. Consider the following vector expressions within the scope of two variables `A` and `B`.

```
01 A          // (i)
01 B          // (ii)
01 A & 01 B   // (iii)
```

The natural interpretation here is `(iii)` === `(i)` & `(ii)`. This interpretation is only possible by allowing simultaneously occurring unspecified events.

Allowing simultaneously occurring unspecified events, the vector expressions `(i)` and `(ii)`, respectively, are interpreted as follows:

```
01 A & ?? B   // (i')
?? A & 01 B   // (ii')
```

Disallowing simultaneously occurring unspecified events, the vector expressions `(i)` and `(ii)`, respectively, are interpreted as follows:

```
01 A & ?- B   // (i'')
?- A & 01 B   // (ii'')
```

The vector expressions `(i')` and `(ii')` are compatible with `(iii)`, whereas `(i'')` and `(ii'')` are not.

10.6.9 Simultaneous event sequences

The semantic meaning of the "simultaneous event operator" can be extended to describe simultaneously occurring *event sequences*, by using the following definition:

```
(01 A#1 .. -> ... 01 A#N) & (01 B#1 .. -> ... 01 B#N)
=== 01 A#1 & 01 B#1 ... -> ... 01 A#N & 01 B#N
```

This definition is analogous to scalar multiplication of vectors with the same number of indices. The number of indices corresponds to the number of `vector_event` expressions separated by `->` operators. If the number of

[2]The power analysis tool relates to a timing constraint checker in a similar way as a parasitic extraction tool relates to a DRC tool. If the layout has DRC violations, for instance shorts between nets, the parasitic extraction tool shall report inaccurate wire capacitance for those nets. After final layout, the DRC violations shall be gone and the wire capacitance shall be accurate.

`->` in both vector expressions is not the same, the shorter vector expression can be left-extended with unspecified events, using the `??` operator, in order to align both vector expressions.

Example

```
(01 A -> 01 B -> 01 C) & (01 D -> 01 E)
=== (01 A -> 01 B -> 01 C) & (?? D -> 01 D -> 01 E)
=== 01 A & ?? D -> 01 B & 01 D -> 01 C & 01 E
=== 01 A -> 01 B & 01 D -> 01 C & 01 E
```

The easiest way to understand the meaning of "simultaneous event sequences" is to consider the event report in test pattern format. If each `vector_event_sequence` expression matches the event report in the same time window, then the event sequences happen simultaneously.

```
time  A   B   C   D   E
0     0   1   1   X   1
109   1   1   1   0   1
258   1   0   1   0   1
573   1   0   0   0   1
586   0   0   0   0   1
643   1   0   0   0   1
788   0   1   1   0   1
915   1   1   1   0   1
1062  1   1   1   0   0
1395  1   0   0   0   0
1640  0   0   0   1   0
```

Example

```
01 A -> 10 B === 01 A & 11 B -> 11 A & 10 B      // (10a)
// event pattern expressed by (10a):
//   A   B
//   0   1
//   1   1
//   1   0
X0 D -> 00 D                                    // (10b)
// event pattern expressed by (10b):
//   D
//   X
//   0
//   0
(01 A -> 10 B) & (X0 D -> 00 D)                 // (10) === (10a)&(10b)
```

Both `(10a)` and `(10b)` are *True* at time 258. Therefore `(10)` is *True* at time 258.

```
10 C
=== ?? C -> ?? C -> 10 C
=== ?? C -> ?1 C -> 10 C                        // (11a)
// event pattern expressed by (11a):
//   C
//   ?
//   ?
//   1
//   0
```

(11a) is left-extended to match the length of the following (11b).

```
01 A -> 00 D -> 11 E ===
   01 A & 00 D & ?? E
-> ?? A & 00 D & ?? E
-> ?? A & ?? D & 11 E
===
   01 A & 00 D & ?? E
-> 1? A & 00 D & ?1 E
-> ?? A & 0? D & 11 E                          // (11b)
// event pattern expressed by (11b):
//   A   D   E
//   0   0   ?
//   1   0   ?
//   ?   0   1
//   ?   ?   1
```

(11b) contains explicitly specified non-events. The non-event 00 D calls for the unspecified events ?? A and ?? E. The non-event 00 E calls for the unspecified events ?? A and ?? D. By propagating well-specified previous and next states to subsequent events, some unspecified events become partly specified.

```
10 C & (01 A -> 00 D -> 11 E)                  // (11) === (11a)&(11b)
```

(11a) is *True* at time 573 and time 1395. (11b) is *True* at time 573 and time 915. Therefore, (11) is *True* at time 573.

10.6.10 Implicit local variables

Until now, vector expressions are evaluated against an event report containing all variables within the scope of a cell. It is practical for the application to work with only one event report per cell or, at most, two event reports if the set of variables for BEHAVIOR (scope=behavior) and VECTOR (scope=measure) is different. However, for complex cells and megacells, it is sometimes necessary to change the scope of event observation, depending on operation modes. Different modes can require a different set of variables to be observed in different event reports.

The following definition allows to *extend* the scope of a vector expression locally:

> Edge operators apply not only to variables, but also to boolean expressions involving those variables. Those boolean expressions represent *implicit local variables* that are visible only within the vector expression where they appear.

Suppose the local variables (A & B), (A | B) are inserted into the event report:

```
time  A   B   C   D   E   A&B   A|B
0     0   1   1   X   1    0     1
109   1   1   1   0   1    1     1
258   1   0   1   0   1    0     1
573   1   0   0   0   1    0     1
586   0   0   0   0   1    0     0
643   1   0   0   0   1    0     1
788   0   1   1   0   1    0     1
915   1   1   1   0   1    1     1
1062  1   1   1   0   0    1     1
```

```
1395  1   0   0   0   0    0     1
1640  0   0   0   1   0    0     0
```

Example

```
01 (A & B)                                          // (12)
// event pattern expressed by (12):
//   A&B
//   0
//   1
```

(12) is *True* at time 109 and time 915.

```
10 (A | B)                                          // (13)
// event pattern expressed by (13):
//   A|B
//    1
//    0
```

(13) is *True* at time 586 and time 1640.

```
01 (A & B) -> 10 B                                  // (14)
// event pattern expressed by (14):
//   B   A&B
//   1    0
//   1    1
//   0    1
```

(14) is *True* at time 258.

```
10 ( A & B) & 10 B -> 10 C                          // (15)
// event pattern expressed by (15):
//   B   C   A&B
//   1   1    1
//   0   1    0
//   0   0    0
```

(15) is *True* at time 573.

```
10 (A & B) -> 10 (A | B)                            // (16)
// event pattern expressed by (16):
//   A&B   A|B
//    1     1
//    0     1
//    0     0
```

(16) is *True* at time 1640.

10.6.11 Conditional event sequences

The following definition *restricts* the scope of a vector expression locally:

`vector_boolean_and`, also called *conditional event operator*

This operator is defined between a vector expression and a boolean expression, using the overloaded symbol & or &&. The scope of the vector expression is restricted to the variables and eventual implicit local variables appearing within that vector expression. The boolean expression shall be *True* during the entire vector expression. The boolean expression is called the *Existence Condition* of the vector expression.[3]

Vector expressions using the `vector_boolean_and` operator are called `vector_conditional_event` expressions. Scope and contents of such expressions are identical, as opposed to non-conditional `vector_complex_event` expressions, where the content is a subset of the scope.

Example

```
(10 (A & B) -> 10 (A | B)) & !D                    // (17)
// event pattern expressed by (17):
//   A&B   A|B
//    1     1
//    0     1
//    0     0
// event report without C, E:
time  A   B   D   A&B   A|B
0     0   1   X    0     1
109   1   1   0    1     1
258   1   0   0    0     1
586   0   0   0    0     0
643   1   0   0    0     1
788   0   1   0    0     1
915   1   1   0    1     1
1062  1   1   0    1     1
1395  1   0   0    0     1
1640  0   0   1    0     0
```

`(17)` contains the same `vector_complex_event` expression as `(16)`. However, although `(16)` is not *True* at time 586, `(17)` is *True* at time 586, since the scope of observation is narrowed to `A`, `B`, `A&B`, and `A|B` by the existence condition `!D`, which is statically *True* while the specified event sequence is observed.

Within, and only within, the narrowed scope of the `vector_conditional_event` expression, `(17)` can be considered equivalent to the following:

```
(10 (A & B) -> 10 (A | B)) & !D
===
(10 (A & B) -> 10 (A | B)) & (11 (!D) -> 11 (!D))
===
10 (A & B) & 11 (!D) -> 10 (A | B) & 11 (!D)
```

The transformation consists of the following steps:

a) Transform the boolean condition into a non-event.
For example, `!D` becomes `11 (!D)`.

[3]An Existence Condition can also appear as annotation to a `VECTOR` object instead of appearing in the vector expression. This enables recognition of existence conditions by application tools which can not evaluate vector expressions (e.g., static timing analysis tools). However, for tools that can evaluate vector expressions, there is no difference between existence condition as a co-factor in the vector expression or as an annotation.

b) Left-extend the `vector_single_event` expression containing the non-event in order to match the length of the `vector_complex_event` expression.
For example, `11 (!D)` becomes `11 (!D) -> 11 (!D)` to match the length of `10 (A & B) -> 10 (A | B)`.

c) Apply scalar multiplication rule for simultaneously occurring event sequences.

Thus, a `vector_conditional_event` expression can be transformed into an equivalent `vector_complex_event` expression, but the change of scope needs to be kept in mind. An operator which can express the change of scope in the vector expression language is defined in 10.6.13. This can make the transformation more rigorous.

Regardless of scope, the transformation from `vector_conditional_event` expression to `vector_complex_event` expression also provides the means of detecting ill-specified `vector_conditional_event` expressions.

Example

```
(10 A -> 01 B -> 01 A) & A
===
10 A & 11 A -> 01 B & 11 A -> 01 A & 11 A
```

The first expression `10 A & 11 A` and the third expression `01 A & 11 A` within the `vector_complex_event` expression are contradictory. Hence, the `vector_conditional_event` expression can never be *True*.

10.6.12 Alternative conditional event sequences

All `vector_binary` operators, in particular the `vector_or` operator, can be applied to `vector_conditional_event` expressions as well as to `vector_complex_event` expressions.

Consider again the event report:

```
time  A   B   C   D   E
0     0   1   1   X   1
109   1   1   1   0   1
258   1   0   1   0   1
573   1   0   0   0   1
586   0   0   0   0   1
643   1   0   0   0   1
788   0   1   1   0   1
915   1   1   1   0   1
1062  1   1   1   0   0
1395  1   0   0   0   0
1640  0   0   0   1   0
```

Concurrent alternative `vector_conditional_event` expressions can be paraphrased in the following way:

```
IF <boolean_expression1> THEN <vector_expression1>
OR IF <boolean_expression2> THEN <vector_expression2>
... OR IF <boolean_expressionN> THEN <vector_expressionN>
```

The conditions can be *True* within overlapping time windows and thus the vector expressions are evaluated concurrently. The `vector_boolean_and` operator and `vector_or` operator describe such vector expressions.

Example

```
C&(01 A -> 10 B) | !D&(10 B -> 10 A) | E&(10 B -> 10 C) // (18)
// Event pattern expressed by (18):
//   A   B   C
//   0   1   1
//   1   1   1
//   1   0   1
```

`(18)` is *True* at time 258 because of `C & (01 A -> 10 B)`.

```
// Alternative event pattern expressed by (18):
//   A   B   D
//   1   1   0
//   1   0   0
//   0   0   0
```

`(18)` is also *True* at time 586 because of `!D & (10 B -> 10 A)`.

```
// Alternative event pattern expressed by (18):
//   B   C   E
//   1   1   1
//   0   1   1
//   0   0   1
```

`(18)` is also *True* at time 573 because of `E & (10 B -> 10 C)`.

Prioritized alternative `vector_conditional_event` expressions can be paraphrased in the following way:

```
IF <boolean_expression1> THEN <vector_expression1>
ELSE IF <boolean_expression2> THEN <vector_expression2>
... ELSE IF <boolean_expressionN> THEN <vector_expressionN>
(optional) ELSE <vector_expressiondefault>
```

Only the vector expression with the highest priority *True* condition is evaluated. The `vector_boolean_cond` operator and `vector_boolean_else` operator are used in ALF to describe such vector expressions.

Example

```
C? (01 A -> 10 B): !D? (10 B -> 10 A): E? (10 B -> 10 C) // (19)
```

The prioritized alternative `vector_conditional_event` expression can be transformed into concurrent alternative `vector_conditional_event` expression as shown:

```
C ? (01 A -> 10 B) : !D ? (10 B -> 10 A) : E ? (10 B -> 10 C)
===
C & (01 A -> 10 B)
| !C & !D & (10 B -> 10 A)
| !C & !(!D) & E & (10 B -> 10 C)
```

`(19)` is *True* at time 258 because of `C & (01 A -> 10 B)`, but not at time 586 because of higher priority `C` while `!D & (10 B -> 10 A)`, nor at time 573 because of higher priority `!D` while `E & (10 B -> 10 C)`.

10.6.13 Change of scope within a vector expression

Conditions on vector expressions redefine the scope of vector expressions locally. The following definition can be used to change the scope even within a part of a vector expression. For this purpose, the symbolic state * can be used, which means “don’t care about events”. This is different from the symbolic state ? which means “don’t care about state”. When the state of a variable is *, arbitrary events occurring on that variable are disregarded.

— Edge operator with * as next state:
The variable to which the operator applies is no longer within the scope of the vector expression.
— Edge operator with * as previous state:
The variable to which the edge operator applies is now within the scope of the vector expression.

As opposed to ?, * stands for an infinite variety of possibilities.

Example

Let A be a logic variable with the possible states 1, 0, and X.

```
*0 A ===
00 A | 10 A | X0 A
| 00 A -> 00 A | 10 A -> 00 A | X0 A -> 00 A
| 01 A -> 10 A | 11 A -> 10 A | X1 A -> 10 A
| 0X A -> X0 A | 1X A -> X0 A | XX A -> X0 A
| 00 A -> 00 A -> 00 A | ...

0* A ===
00 A | 01 A | 0X A
| 00 A -> 00 A | 00 A -> 01 A | 00 A -> 0X A
| 01 A -> 10 A | 01 A -> 11 A | 01 A -> 1X A
| 0X A -> X0 A | 0X A -> X1 A | 0X A -> XX A
| 00 A -> 00 A -> 00 A | ...
```

A vector expression with an infinite variety of possible event sequences cannot be directly matched with an event report. However, there are feasible ways to implement event sequence detection involving *. In principle, there is a “static” and “dynamic” way. The following parts of the vector expression are separated by * *sub-sequences* of events.

— “Static” event sequence detection with *:
The event report with all variables can be maintained, but certain variables are masked for the purpose of detection of certain sub-sequences.
— “Dynamic” event sequence detection with *:
The event report shall contain the set of variables necessary for detection of a relevant sub-sequence. When such a sub-sequence is detected, the set of variables in the event report shall change until the next sub-sequence is detected, etc.

Examples

```
01 A -> 1* B -> 10 C                                  // (20)
// Event pattern expressed by (20):
//   A   B   C
//   0   1   1
//   1   1   1
//   1   *   1
//   1   *   0
```

```
// pattern for 1st sub-sequence:
//   A   B   C
//   0   1   1
//   1   1   1
//   1   *   1
// pattern for 2nd sub-sequence:
//   A   B   C
//   1   *   1
//   1   *   0
```

The event report with masking relevant for `(20)`:

```
time  A   B   C   D   E
0     0   1   1   X   1
109   1   1   1   0   1
258   1   *   1   0   1   // detection of 1st sub-sequence
573   1   *   0   0   1   // detection of 2nd sub-sequence
586   0   0   0   0   1
643   1   0   0   0   1
788   0   1   1   0   1
915   1   1   1   0   1
1062  1   *   1   0   0   // detection of 1st sub-sequence
1395  1   *   0   0   0   // detection of 2nd sub-sequence
1640  0   0   0   1   0
```

`(20)` is *True* at time 573 and time 1395. The first sub-sequence `01 A -> 1* B` is detected at time 258, since * maps to any state. From time 258 onwards, `B` is masked. The second sub-sequence `10 C` is detected at time 573. From time 573 onwards, `B` is unmasked. The first sub-sequence is detected again at time 1062. The second sub-sequence is detected again at time 1395.

```
01 A & 1* E -> 10 C                              // (21)
// Event pattern expressed by (21):
//   A   C   E
//   0   1   1
//   1   1   *
//   1   0   *
// pattern for 1st sub-sequence:
//   A   C   E
//   0   1   1
//   1   1   *
// pattern for 2nd sub-sequence:
//   A   C   E
//   1   1   *
//   1   0   *
```

The event report with masking relevant for `(21)`:

```
time  A   B   C   D   E
0     0   1   1   X   1
109   1   1   1   0   *   // detection of 1st sub-sequence
258   1   0   1   0   *   // abortion of detection process
573   1   0   0   0   1
586   0   0   0   0   1
643   1   0   0   0   1
```

```
788   0   1   1   0   1
915   1   1   1   0   *   // detection of 1st sub-sequence
1062  1   1   1   0   *   // disregard event out of scope
1395  1   0   0   0   0   // detection of 2nd sub-sequence
1640  0   0   0   1   0
```

(21) is *True* at time 1395. The first sub-sequence `01 A & 1* E` is detected at time 109. From time 109 onwards, `E` is masked. The event on `B` at time 258 aborts continuation of the detection process and triggers restart from the beginning. The first sub-sequence is detected again at time 915. From time 915 onwards, `E` is masked. The event at time 1062 is therefore out of scope. The second sub-sequence `10 C` is detected at time 1395.

```
01 A -> *1 B -> 10 B & 10 C                    // (22)
// Event pattern expressed by (22):
//   A   B   C
//   0   *   1
//   1   *   1
//   1   1   1
//   1   0   0
// pattern for 1st sub-sequence:
//   A   B   C
//   0   *   1
//   1   *   1
// pattern for 2nd sub-sequence:
//   A   B   C
//   1   *   1
//   1   1   1
//   1   0   0
```

The event report with masking relevant for `(22)`:

```
time  A   B   C   D   E
0     0   1   1   X   1
109   1   1   1   0   1   // detection of 1st sub-sequence
258   1   0   1   0   1   // abort
573   1   *   0   0   1
586   0   *   0   0   1
643   1   *   0   0   1
788   0   *   1   0   1
915   1   *   1   0   1   // detection of 1st sub-sequence
1062  1   1   1   0   0   // continue
1395  1   0   0   0   0   // detection of 2nd sub-sequence
1640  0   0   0   1   0
```

`(22)` is *True* at time 1395. The first sub-sequence `01 A` is detected at time 109. Therefore, `B` is unmasked. Since `B=0` at time 258, the attempt to detect the second sub-sequence is aborted and the detection process restarts from the beginning. The first sub-sequence `01 A` is detected again at time 109. The second sub-sequence `*1 B -> 10 B & 10 C` is detected at time 1395.

```
01 A -> 1? A & 0* B & 1* E -> 10 C              // (23)
// Event pattern expressed by (23):
//   A   B   C   E
//   0   0   1   1
//   1   0   1   1
```

```
//   1   *   1   *
//   1   *   0   *
// pattern for 1st sub-sequence:
//   A   B   C   E
//   0   0   1   1
//   1   0   1   1
//   ?   *   1   *
// pattern for 2nd sub-sequence:
//   A   B   C   E
//   ?   *   1   *
//   ?   *   0   *
```

The event report with masking relevant for (23):

```
time  A   B   C   D   E
0     0   1   1   X   1
109   1   1   1   0   1
258   1   0   1   0   1
573   1   0   0   0   1
586   0   0   0   0   1
643   1   0   0   0   1
788   0   *   1   0   *   // detection of 1st sub-sequence
915   1   *   1   0   *   // abort
1062  1   1   1   0   0
1395  1   0   0   0   0
1640  0   0   0   1   0
```

`(23)` is not *True* at any time. The first sub-sequence is detected at time 788. The event at time 915 does not match the expected second sub-sequence.

10.6.14 Sequences of conditional event sequences

The symbol `*` can be used to describe the scope of a vector expression directly in the vector expression language. This is particularly useful for sequences of `vector_conditional_event` expressions.

In reusing `(17)` as example:

```
(10 (A & B) -> 10 (A | B)) & !D
```

the scope of the sample event report contains contain the variables `A`, `B`, `C`, `D`, and `E`. The `vector_conditional_event` expression `(17)` contains only the variables `A`, `B`, and `D` and the implicit local variables `A&B` and `A|B`. Therefore, the global variables `C` and `E` are out of scope within `(17)`. The implicit local variables `A&B` and `A|B` are in scope within, and only within, `(17)`.

Now consider a *sequence* of `vector_conditional_event` expressions, where variables move in and out of scope. With the following formalism, it is possible to transform such a sequence into an equivalent `vector_complex_event` expression, allowing for a change of scope within each `vector_conditional_event` expression.

```
<vector_conditional_event#1> .. -> .. <vector_conditional_event#N>
```

where

```
<vector_conditional_event#i>
=== <vector_complex_event#i> & <boolean_expression#i> // 1 ≤ i ≤ N
```

The principle is to decompose each `vector_conditional_event` expression into a sequence of three vector expressions *prefix*, *kernel*, and *postfix* and then to reassemble the decomposed expressions.

```
<vector_conditional_event#i>
=== <prefix#i> -> <kernel#i> -> <postfix#i> // 1 ≤ i ≤ N
```

a) Define the prefix for each `vector_conditional_event` expression.
The *prefix* is a `vector_event` expression defining all implicit local variables.

Example

```
*? (A&B) & *? (A|B)
```

b) Define the kernel for each `vector_conditional_event` expression.
The *kernel* is the `vector_complex_event` expression equivalent to the `vector_conditional_event` expression.

```
<vector_complex_event#i> & <boolean_expression#i>
===    <vector_complex_event#i>
&    (11 <boolean_expression#i> ..->.. 11 <boolean_expression#i>)
```

The kernel can consist of one or several alternative `vector_event_sequence` expressions. Within each `vector_event_sequence` expression, the same set of global variables are pulled out of scope at the first `vector_event` expression and pushed back in scope at the last `vector_event` expression.

Example

```
?* C & ?* E   // global variables out of scope
& 10 (A & B) & 11 (!D) -> 10 (A | B) & 11 (!D)
& *? C & *? E   // global variables back in scope
```

c) Define the postfix for each `vector_conditional_event` expression.
The *postfix* is a `vector_event` expression removing all implicit local variables.

Example

```
?* (A&B) & ?* (A|B)
```

d) Join the subsequent `vector_complex_event` expressions with the `vector_and` operator between prefix#i+1and kernel#i and also between postfix#i and kernel#i+1.

```
.. <vector_conditional_event#i> -> <vector_conditional_event#i+1> ..
===   .. <prefix#i>
   -> <postfix#i-1> & <kernel#i> & <prefix#i+1>
   -> <postfix#i> & <kernel#i+1> & <prefix#i+2>
      -> <postfix#i+1> ..
```

The complete example:

```
(10 (A & B) -> 10 (A | B)) & !D
===
*? (A&B) & *? (A|B)
-> ?* C & ?* E
& 10 (A & B) & 11 (!D) -> 10 (A | B) & 11 (!D)
& *? C & *? E
-> ?* (A&B) & ?* (A|B)
```

NOTE —The in-and-out-of-scope definitions for global variables are within the kernel, whereas the in-and-out-of-scope definitions for global variables are within the prefix and postfix. In this way, the resulting `vector_complex_event` expression contains the same uninterrupted sequence of events as the original sequence of `vector_conditional_event` expressions.

10.6.15 Incompletely specified event sequences

So far the vector expression language has provided support for *completely specified event sequences* and also the capability to put variables temporarily in and out of scope for event observation. As opposed to changing the scope of event observation, *incompletely specified event sequences* require continuous observation of all variables while allowing the occurrence of intermediate events between the specified events. The following operator can be used for that purpose:

> `vector_followed_by`, also called *followed-by operator*, using the symbol `~>`.
> The `~>` operator is the separator between consecutively occurring events, with possible unspecified events in-between.

Detection of event sequences involving `~>` requires detection of the sub-sequence before `~>`, setting a flag, detection of the sub-sequence after `~>`, and clearing the flag.

This can be illustrated with a sample event report:

```
time  A   B   C   D   E
0     0   1   1   X   1
109   1   1   1   0   1   // 01 A detected, set flag
258   1   0   1   0   1
573   1   0   0   0   1   // 10 C detected, clear flag
586   0   0   0   0   1
643   1   0   0   0   1   // 01 A detected, set flag
788   0   1   1   0   1
915   1   1   1   0   1   // 01 A detected again
1062  1   1   1   0   0
1395  1   0   0   0   0   // 10 C detected, clear flag
1640  0   0   0   1   0
```

Example

```
01 A ~> 10 C                                          // (24)
// as opposed to previous example (5):01 A -> 10 C
```

`(24)` is *True* at time 573 because of `01 A` at time 109 and `10 C` at time 573. It is *True* again at time 1395 because of `01 A` at time 643 and `10 C` at 1395. On the other hand, `(5)` is never *True* because there are always events in-between `01 A` and `10 C`.

Vector expressions consisting of `vector_event` expressions separated by `->` or by `~>` are called `vector_event_sequence` expressions, using the same syntax rules for the two different `vector_followed_by` operators. Consequently, all vector expressions involving `vector_event_sequence` expressions and `vector_binary` operators are called `vector_complex_event` expressions.

However, only a subset of the semantic transformation rules can be applied to vector expressions containing `~>`.

Associative rule applies for both `->` and `~>`.

```
(01 A ~> 01 B) ~> 01 C === 01 A ~> (01 C ~> 01 B ~> 01 C)
(01 A -> 01 B) -> 01 C === 01 A -> (01 C -> 01 B -> 01 C)
(01 A ~> 01 B) -> 01 C === 01 A ~> (01 C ~> 01 B -> 01 C)
(01 A -> 01 B) ~> 01 C === 01 A -> (01 C -> 01 B ~> 01 C)
```

Distributive rule applies for both `->` and `~>`.

```
(01 A | 01 B) -> 01 C === 01 A ~> 01 C | 01 B -> 01 C
(01 A | 01 B) ~> 01 C === 01 A ~> 01 C | 01 B ~> 01 C
(01 A | 01 B) -> 01 C === 01 A ~> 01 C | 01 B -> 01 C
```

Scalar multiplication rule applies only for `->`. The transformation involving `~>` is more complicated.

```
(01 A -> 01 B) & (01 C -> 01 D)
=== (01 A & 01 C) -> (01 B & 01 D)

(01 A ~> 01 B) & (01 C -> 01 D)
=== (01 A & 01 C) -> (01 B & 01 D)
|    01 A ~> 01 C -> (01 B & 01 D)

(01 A ~> 01 B) & (01 C ~> 01 D)
=== (01 A & 01 C) ~> (01 B & 01 D)
|    01 A ~> 01 C ~> (01 B & 01 D)
|    01 C ~> 01 A ~> (01 B & 01 D)
```

Transformation of `vector_conditional_event` expressions into `vector_complex_event` expressions applies only for `->`.

```
(01 A -> 01 B) & C
=== 01 A & 11 C -> 01 B & 11 C

(01 A ~> 01 B) & C
~~===~~ 01 A & 11 C ~> 01 B & 11 C
```

Since the `~>` operator allows intermediate events, there is no way to express the continuously *True* condition `C`.

10.6.16 How to determine well-specified vector expressions

By defining semantics for

> alternative `vector_event_sequence` expressions

and establishing calculation rules for

> transforming `vector_complex_event` expressions into alternative `vector_event_sequence` expressions

and for

> transforming alternative `vector_conditional_event` expressions into alternative `vector_complex_event` expressions,

semantics are now defined for all vector expressions.

The calculation rules also provide means to determine whether a vector expression is well-specified or ill-specified. An ill-specified vector expression is contradictory in itself and can therefore never be *True*.

Once a vector expression is reduced to a set of alternative `vector_event_sequence` expressions, two criteria define whether a vector expression is well-defined or not.

— Compatibility between subsequent events on the same variable:
The next state of earlier event shall be compatible with previous state of later event. This check applies only if no `~>` operator is found between the events.
— Compatibility between simultaneous events on the same variable:
Both the previous and next state of both events shall be compatible. Such events commonly occur as intermediate calculation results within vector expression transformation.

The following compatibility rules apply:

a) `?` is compatible with any other state. If the other state is `*`, the resulting state is `?`. Otherwise, the resulting state is the other state.
b) `*` is compatible with any other state. The resulting state is the other state.
c) Any other state is only compatible with itself.

Examples

```
01 A -> 01 B -> 10 A
```

The next state of `01 A` is compatible with the previous state of `10 A`.

```
0X A -> 01 B -> 10 A
```

The next state of `0X A` is not compatible with the previous state of `10 A`.

```
0X A ~> 01 B -> 10 A
```

Compatibility check does not apply, since intermediate events are allowed.

```
01 A & 10 A
```

Both the previous and next state of `A` are contradictory; this results in an impossible event.

```
?1 A & 1? A
```

Both previous and next state of `A` are compatible; this results in the non-event `11 A`.

10.7 Boolean expression language

The boolean expression language XXX, as shown in Syntax 126.

10.8 Vector expression language

The vector expression language XXX, as shown in Syntax 127.

```
boolean_expression ::=
      ( boolean_expression )
    | pin_value
    | boolean_unary boolean_expression
    | boolean_expression boolean_binary boolean_expression
    | boolean_expression ? boolean_expression :
        { boolean_expression ? boolean_expression : }
        boolean_expression
boolean_unary ::=
      !
    | ~
    | &
    | ~&
    | |
    | ~|
    | ^
    | ~^
boolean_binary ::=
      &
    | &&
    | |
    | ||
    | ^
    | ~^
    | !=
    | ==
    | >=
    | <=
    | >
    | <
    | +
    | -
    | *
    | /
    | %
    | >>
    | <<
```

Syntax 126—Boolean expression langauge

10.9 Control expression semantics

**Syntax 127 also shows the control expression syntax (at the bottom); is this deliberate??

```
vector_expression ::=
      ( vector_expression )
    | vector_unary boolean_expression
    | vector_expression vector_binary vector_expression
    | boolean_expression ? vector_expression :
        { boolean_expression ? vector_expression : }
        vector_expression
    | boolean_expression control_and vector_expression
    | vector_expression control_and boolean_expression
vector_unary ::=
    edge_literal
vector_binary ::=
      &
    | &&
    | |
    | ||
    | ->
    | ~>
    | <->
    | <~>
    | &>
    | <&>
control_and ::=
    & | &&
control_expression ::=
      ( vector_expression )
    | ( boolean_expression )
```

Syntax 127—Vector expression language

11. Constructs for electrical and physical modeling

Add lead-in text

11.1 Arithmetic expression

An arithmetic expression shall be defined as shown in Syntax 128.

```
arithmetic_expression ::=
      ( arithmetic_expression )
    | arithmetic_value
    | { boolean_expression ? arithmetic_expression : } arithmetic_expression
    | [ unary_arithmetic_operator ] arithmetic_operand
    | arithmetic_operand binary_arithmetic_operator arithmetic_operand
    | macro_arithmetic_operator ( arithmetic_operand { , arithmetic_operand } )
arithmetic_operand ::=
      arithmetic_expression
```

Syntax 128—Arithemetic expression

An *unary arithmetic operator* shall be defined as shown in Syntax 129.

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

Syntax 129—Unary arithmetic operator

The following Table 81 defines the semantics of unary arithmetic operators.

Table 81—Unary arithmetic operators

Operator	Description	Comment
+	Positive sign	neutral operator
-	Negative sign	

A *binary arithmetic operator* shall be defined as shown in Syntax 130.

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

Syntax 130—Binary arithmetic operator

The following Table 82 defines the semantics of binary arithmetic operators.

Table 82—Binary arithmetic operators

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

A *macro arithmetic operator* shall be defined as shown in Syntax 131.

```
macro_arithmetic_operator ::=
    abs
  | exp
  | log
  | min
  | max
```

Syntax 131—Macro arithmetic operator

The following Table 83 defines the semantics of macro arithmetic operators.

Table 83—Macro arithmetic operators

Operator	Description	Comment
log	Natural logarithm	1 operand, operand > 0
exp	Natural exponential	1 operand
abs	Absolute value	1 operand
min	Minimum	N>1 operands
max	Maximum	N>1 operands

The priority of operators in arithmetic expressions shall be from strongest to weakest in the following order:

a) unary arithmetic operator (+, -)
b) power (**)
c) multiplication (*), division (/), modulo division (%)
d) addition (+), subtraction (-)

Examples for arithmetic expressions

```
1.24
- Vdd
C1 + C2
MAX ( 3.5*C , -Vdd/2 , 0.0 )
(C > 10) ? Vdd**2 : 1/2*Vdd - 0.5*C
```

11.2 Arithmetic model

An *arithmetic model* shall be defined as a *trivial arithmetic_model*, a *partial arithmetic model*, or a *full arithmetic model*, as shown in Syntax 132.

```
arithmetic_model ::=
      trivial_arithmetic_model
    | partial_arithmetic_model
    | full_arithmetic_model
    | arithmetic_model_template_instantiation
```

Syntax 132—Arithmetic model statement

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

A *trivial arithmetic model* shall be defined as shown in Syntax 133.

```
trivial_arithmetic_model ::=
      nonescaped_identifier [ name_identifier ] = arithmetic_value ;
    | nonescaped_identifier [ name_identifier ] = arithmetic_value { { model_qualifier } }
```

Syntax 133—Trivial arithmetic model

No mathemetical operation is necessary to evaluate a trivial arithmetic model. The arithmetic value associated with the arithmetic model represents the evaluation result. One or more *model qualifier* statements can be associated with a trivial arithmetic model.

A *partial arithmetic model* shall be defined as shown in Syntax 134.

```
partial_arithmetic_model ::=
      nonescaped_identifier [ name_identifier ] { { partial_arithmetic_model_item } }
partial_arithmetic_model_item ::=
      model_qualifier
    | table
    | trivial_min-max
```

Syntax 134—Partial arithmetic model

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

The purpose of a partial arithmetic model is to specify one or more *model qualifier* statements, a *table* statement, or a *trivial min-max* statement. The specification contained within a partial arithmetic model can be inherited by another arithmetic model of the same type, according to the following rules:

a) If the partial arithmetic model has no name, the specification shall be inherited by all arithmetic models of the same type appearing within the same parent statement or within a descendant of the same parent statement.

b) If the partial arithmetic model has a name, the specification shall be only inherited by an arithmetic model containing a reference to the partial arithmetic model, using the *model reference annotation* (see NEW SUBSECTION).

c) An arithmetic model can override an inherited specification by its own specification.

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

```
full_arithmetic_model ::=
    nonescaped_identifier [ name_identifier ] { { model_qualifier } model_body { model_qualifier } }
model_body ::=
    header-table-equation [ trivial_min-max ]
  | min-typ-max
  | arithmetic_submodel { arithmetic_submodel }
```

Syntax 135—Full arithmetic model

The *model body* specifies mathematical data associated with the arithmetic model. The data is represented either by a *header-table-equation* statement, or by a *min-typ-max* statement, or by one or more *arithmetic submodel* statements.

The mathematical operation or the arithmetic value for evaluation of the arithmetic model can be contained within one or more arithmetic submodels (see NEW SUBSECTION). The selection of an applicable submodel is controlled by the semantics of the keyword that identifies the type of the arithmetic submodel.

11.3 HEADER, TABLE, and EQUATION

A *header-table-equation* statement shall be defines as shown in

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

Syntax 136—

A header-table-equation statement specifies a procedure for evaluation of the mathemetical data.

11.3.1 HEADER statement

A *header* statement shall be defined as shown in Syntax 137.

```
header ::=
    HEADER { partial_arithmetic_model { partial_arithmetic_model } }
```

Syntax 137—HEADER statement

Each partial arithmetic model within the header statement shall represent a *dimension* of an arithmetic model.

11.3.2 TABLE statement

A *table* statement shall be defined as shown in Syntax 138.

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

Syntax 138—TABLE statement

A table statement within a partial arithmetic model shall define the set of legal values for an arithmetic model that inherits the specification of the partial arithmetic model.

A table statement within a full arithmetic model shall represent a lookup table. If the model body contains a table statement, each dimension within the header statement shall also contain a table statement.

The mathemetical relation between a lookup table and its dimensions shall be established as follows:

$$S = \prod_{i=1}^{N} S(i) \qquad N \geq 1$$

$$S \geq 1$$

$$0 \leq P \leq S - 1$$

$$P = \sum_{i=1}^{N} P(i) \prod_{k=1}^{i-1} S(k)$$

$$S(i) \geq 1$$

$$0 \leq P(i) \leq S(i) - 1$$

where

N denotes the number of dimensions
S denotes the size of the lookup table, i.e., the number of arithmetic values within the lookup table
P denotes the position of an arithmetic value within the lookup table
S(*i*) denotes the size of a dimension, i.e., the number of arithmetic values in the table within a dimension
P(*i*) denotes the position of an arithmetic value within a dimension

A dimension can be either discrete or continuous. In the latter case, interpolation and extrapolation of table values is allowed, and the arithmetic values in this dimension shall appear in strictly monotonous ascending order.

11.3.3 EQUATION statement

An *equation* statement shall be defined as shown in Syntax 139.

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

Syntax 139—EQUATION statement

The arithmetic expression within the equation statement shall represent the mathematical operation for evaluation of the arithmetic model.

Each dimension shall be involved in the arithmetic expression. The arithmetic expression shall refer to a dimension by name, if a name identifier exists or by type otherwise. Consequently, the type or the name of a dimension shall be unique.

11.4 Statements related to arithmetic model

11.4.1 Model qualifier

A *model qualifier* statement shall be defined as shown in

```
model_qualifier ::=
      annotation
    | annotation_container
    | event_reference
    | from-to
    | auxiliary_arithmetic_model
    | violation
```

Syntax 140—Model Qualifier statement

11.4.2 Auxiliary arithmetic model

An *auxiliary arithmetic model* shall be defined as shown in

```
auxiliary_arithmetic_model ::=
      nonescaped_identifier = arithmetic_value ;
    | nonescaped_identifier [ = arithmetic_value ] { auxiliary_qualifier { auxiliary_qualifier } }
auxiliary_qualifier
      annotation
    | annotation_container
    | event_reference
    | from-to
```

Syntax 141—Auxiliary arithmetic model

An auxiliary arithmetic model can be considered as a special case of either a trivial arithmetic model or a partial arithmetic model, since the rule for *auxiliary qualifier* is a true subset of the rule for *model qualifier.* In particular, the items *auxiliary arithmetic model* and *violation* are disallowed in the rule for auxiliary qualifier.

11.4.3 Arithmetic submodel

An *arithmetic submodel* shall be defined as shown in Syntax 142.

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

Syntax 142—Arithmetic submodel

11.4.4 MIN-MAX statement

A *min-max* statement shall be defined as shown in

```
min-max ::=
      min [ max ]
    | max [ min ]
min ::=
      MIN = arithmetic_value ;
    | MIN = arithmetic_value { violation }
    | MIN { [ violation ] header-table-equation }
max ::=
      MAX = arithmetic_value ;
    | MAX = arithmetic_value { violation }
    | MAX { [ violation ] header-table-equation }
```

Syntax 143—MIN-MAX statement

11.4.5 MIN-TYP-MAX statement

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

```
min-typ-max ::=
      [ min-max ] typ [ min-max ]
typ ::=
      TYP = arithmetic_value ;
    | TYP { header-table-equation }
```

Syntax 144—MIN-TYP-MAX statement

11.4.6 Trivial MIN-MAX statement

A *trivial min-max* statement shall be defined as shown in

```
trivial_min-max ::=
      trivial_min [ trivial_max ]
    | trivial_max [ trivial_min ]
trivial_min ::=
      MIN = arithmetic_value ;
trivial_max ::=
      MAX = arithmetic_value ;
```

Syntax 145—

A trivial min-max statement defines the legal range of values for an arithmetic model. The arithmetic value associated with the *trivial min* statement represent the smallest legal number. The arithmetic value associated with the *trivial max* statement represents the greatest legal number. Per default, the range includes between negative and positive infinity.

A trivial min-max statement within a dimension of a full arithmetic model defines the range of validity of a particular dimension. An application tool can still evaluate the header-table-equation statement outside the range of validity, however, the accuracy of the evaluation can not be guaranteed.

The following semantic restrictions shall apply:

a) A partial arithmetic model that is not a dimension of a lookup table can either contain a trivial min-max statement or a table statement but not both.
b) If a syntax rule allows both partial arithmetic model and full arithmetic model, a trivial min-max statement shall be interpreted as a min-typ-max statement, if the arithmetic model contains neither a header-table-equation statement nor a arithmetic submodel and no other arithmetic model can inherit the trivial min-max statement.

Rule a) is established because a trivial min-max statement would be redundant or eventually contradictory to a table statement, since the table statement already defines a discrete set of legal values.

Rule b) is established because the syntax rule for trivial min-max statement is a true subset of the syntax rule for min-typ-max statement.

11.4.7 Arithmetic model container

An arithmetic model container shall be defined as shown in Syntax 146.

```
arithmetic_model_container ::=
    arithmetic_model_container_identifier { arithmetic_model { arithmetic_model } }
```

Syntax 146—Arithmetic model container

11.4.8 LIMIT statement

A *limit statement* shall be defined as shown in .

```
limit ::=
    LIMIT { limit_item { limit_item } }
limit_item ::=
    limit_arithmetic_model
limit_arithmetic_model ::=
    nonescaped_identifier [ name_identifier ] { { model_qualifier } limit_arithmetic_model_body }
limit_arithmetic_model_body ::=
    limit_arithmetic_submodel { limit_arithmetic_submodel }
  | min_max
limit_arithmetic_submodel ::=
    nonescaped_identifier { [ violation ] min-max }
```

Syntax 147—LIMIT statement

11.4.9 Event reference statement

An *event reference* statement shall be defined as shown in .

```
event_reference ::=
    PIN_reference_single_value_annotation [ EDGE_NUMBER_single_value_annotation ]
```

Syntax 148—Event reference statement

11.4.10 FROM and TO statements

A *from* statement and a *to* statement shall be defined as shown in Syntax 149.

```
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 ::=
      event_reference
    | THRESHOLD_arithmetic_model
```

Syntax 149—FROM and TO statements

The event refered by the from-statement and the to-statement, respectively, shall be called *from-event* and *to-event*, respectively.

The from-and to-statements are subjected to the following semantic restriction.

```
SEMANTICS FROM {
  CONTEXT {
    TIME DELAY RETAIN SLEWRATE PULSSEWIDTH
    SETUP HOLD RECOVERY REMOVAL NOCHANGE ILLEGAL SKEW
  }
}
SEMANTICS TO {
  CONTEXT {
    TIME DELAY RETAIN SLEWRATE PULSSEWIDTH
    SETUP HOLD RECOVERY REMOVAL NOCHANGE ILLEGAL SKEW
  }
}
```

Syntax 150— Semantic restriction

11.4.11 EARLY and LATE statements

An *early* statement and a *late* statement shall be defined as shown in Syntax 151.

11.4.12 VIOLATION statement

A *violation* statement shall be defined as shown in Syntax 152.

A violation statement is subjected to the following semantic restriction.

```
early-late ::=
    early late
early ::=
  EARLY { early-late_item { early-late_item } }
late ::=
  LATE { early-late_item { early-late_item } }
early-late_item ::=
    DELAY_arithmetic_model
  | RETAIN_arithmetic_model
  | SLEWRATE_arithmetic_model
```

Syntax 151—EARLY and LATE statements

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

Syntax 152—VIOLATION statement

```
SEMANTICS VIOLATION {
  CONTEXT {
    SETUP HOLD RECOVERY REMOVAL SKEW NOCHANGE ILLEGAL
    LIMIT.arithmetic_model
    LIMIT.arithmetic_model.MIN
    LIMIT.arithmetic_model.MAX
    LIMIT.arithmetic_model.arithmetic_submodel
    LIMIT.arithmetic_model.arithmetic_submodel.MIN
    LIMIT.arithmetic_model.arithmetic_submodel.MAX
  }
}
```

Syntax 153— Semantic restriction

A violation statement can contain a behavior statement with the following semantic restriction.

```
SEMANTICS VIOLATION.BEHAVIOR {
  CONTEXT {
    VECTOR.arithmetic_model
    VECTOR.LIMIT.arithmetic_model
    VECTOR.LIMIT.arithmetic_model.MIN
    VECTOR.LIMIT.arithmetic_model.MAX
    VECTOR.LIMIT.arithmetic_model.arithmetic_submodel
    VECTOR.LIMIT.arithmetic_model.arithmetic_submodel.MIN
    VECTOR.LIMIT.arithmetic_model.arithmetic_submodel.MAX
  }
}
```

Syntax 154— Semantic restriction

The *violation* statement can contain a *message-type* annotation and a *message* annotation.

A *xxx* annotation shall be defined using ALF language as shown in .

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

Syntax 155— annotation

A *xxx* annotation shall be defined using ALF language as shown in .

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

Syntax 156— annotation

11.5 Annotations for arithmetic models

11.5.1 UNIT annotation

A *xxx* annotation shall be defined using ALF language as shown in .

```
KEYWORD UNIT = annotation {
  CONTEXT = arithmetic_model ;
  VALUETYPE = unit_value ;
  DEFAULT = 1 ;
}
```

Syntax 157— annotation

11.5.2 CALCULATION annotation

A *xxx* annotation shall be defined using ALF language as shown in .

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

Syntax 158— annotation

The meaning of the annotation values is shown in .

Table 84—

annotation value	description
absolute	The arithmetic model data is complete within itself
incremental	The arithmetic model data shall be combined with other arithmetic model data

11.5.3 INTERPOLATION annotation

A *xxx* annotation shall be defined using ALF language as shown in .

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

Syntax 159— annotation

The interpolation annotation shall apply for a dimension of a lokup 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 85—

annotation value	description
linear	linear interpolation shall be used
ceiling	the next greater value in the table shall be used
floor	the next lesser value in the table shall be used
fit	linear or higher-order interpolation shall be used

The mathematical operations for *floor*, *ceiling*, and *linear* are specified as follows:

floor $y(x) = y(x^-)$

ceiling $y(x) = y(x^+)$

linear $$y(x) = \frac{(x - x^-) \cdot y(x^+) + (x^+ - x) \cdot y(x^-)}{x^+ - x^-}$$

where

x denotes the value in a dimension subjected to interpolation.
x^- and x^+ denote two subsequent values in the table associated with that dimension.
x^- denotes the value to the left of *x*, such that $x^- < x$, or else x^- denotes the smallest value in the table.
x^+ denotes the value to the right of *x*, such that $x < x^+$, or else x^+ denotes the largest value in the table.
y denotes the evaluation result of the arithmetic model.

The mathematical operation for *fit* can be chosen by the application, as long as the following conditions are satisfied:

$y(x)$ is a continuous function of order $N>0$.
The first N-1 derivatives of $y(x)$ are continuous.
$y(x)$ is bound by $y(x^-)$ and $y(x^+)$.
In case of monotony, $y(x)$ is also bound by linear interpolation applied to the left and the right neighbor of *x*.
In case of monotonous derivative, $y(x)$ is also bound by linear interpolation applied to *x* itself.

These conditions are illustrated in the following figure.

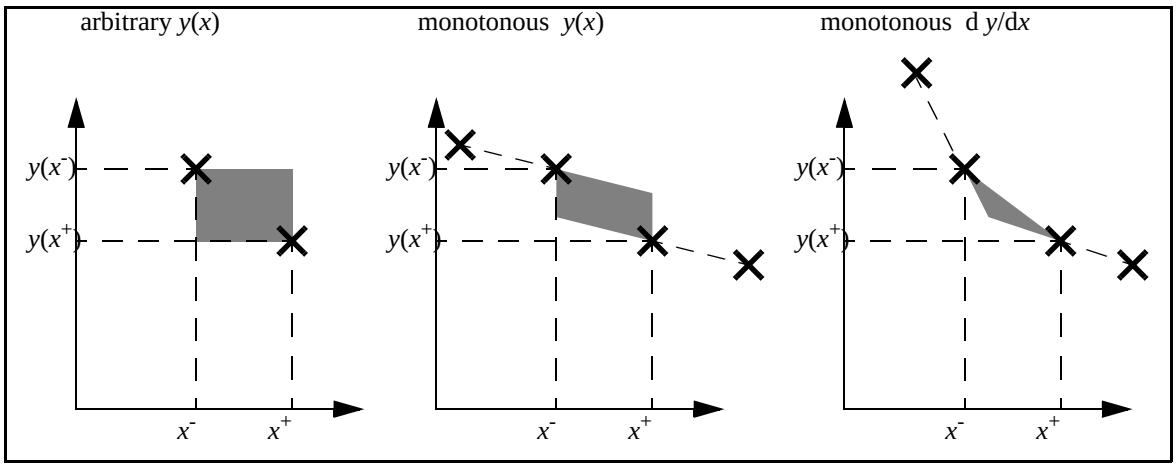


Figure 19—Bounding regions for y(x) with INTERPOLATION=fit

11.5.4 DEFAULT annotation

A *xxx* annotation shall be defined using ALF language as shown in .

```
KEYWORD DEFAULT = single_value_annotation {
  CONTEXT { arithmetic_model KEYWORD }
  VALUETYPE = all_purpose_value ;
}
```

Syntax 160— annotation

11.6 TIME

A *xxx* statement shall be defined using ALF language as shown in .

```
KEYWORD TIME = arithmetic_model {
  VALUETYPE = number ;
}
TIME { UNIT = 1e-9; }
```

Syntax 161— statement

A time statement can have a from-to statement as model qualifier.

11.6.1 TIME in context of a VECTOR declaration

A time statement can be a child or a grandchild of a vector declaration. In particular, the parent of the time statement can be a limit statement. In the context of a limit statement, the time statement shall specify a smallest required time or a largest allowed time interval. Otherwise, the time statement shall specify an actually measured time interval.

If the vector declaration involves a vector expression, from-to statements featuring event reference statements shall be used as model qualifier. The time statement shall model the measured time interval between the refered events.

If the vector declaration involves a boolean expression, the time statement applies to a time interval during which the boolean expression is true. A from-to statement shall not be used as model qualifier.

11.6.2 TIME in context of a HEADER statement

A time statement can be child of a header statement, thus representing a dimension of an arithmetic model.

If the arithmetic model is not a child of a limit statement, the time dimension shall be used to describe a quantity changing over time, which can be visualized by a waveform.

If the arithmetic model is a child of a vector declaration, either a from statement or a to statement can be used as model qualifier to define a temporal relationship between a refered event and the time dimension.

If the arithmetic model is a child of a limit statement, the time dimension shall be used to describe a dependency between a limit for a measured quantity and the expected lifetime of an electronic circuit. A from-to statement shall not be used as model qualifier.

11.6.3 TIME as auxiliary arithmetic model

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

A *measurement* annotation (see Section 11.29.1) shall be used in conjunction with the time statement. The time statement shall specify the time interval during which the measurement is taken.

If the parent arithmetic model is a child of a vector declaration, a from-to statement can be used to define a temporal relationship between one or two events in the vector expression and the time interval.

11.7 FREQUENCY

A *xxx* statement shall be defined using ALF language as shown in .

```
KEYWORD FREQUENCY = arithmetic_model {
  VALUETYPE = number ;
}
FREQUENCY { UNIT = 1e9; MIN = 0; }
```

Syntax 162— statement

11.7.1 FREQUENCY in context of a VECTOR declaration

A frequency statement can be a child or a grandchild of a vector declaration. In particular, the parent of the frequency statement can be a limit statement. In the context of a limit statement, the frequency statement shall specify a smallest required occurence frequency or a largest allowed occurency frequency of the vector. Otherwise, the frequency statement shall specify an actually measured occurence frequency of the vector.

11.7.2 FREQUENCY in context of a HEADER statement

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

If the arithmetic model is a child of a vector declaration, the frequency dimension shall represent the occurence frequency of the vector.

If the arithmetic model is not a child of a vector declaration, the frequency dimension shall be used to describe a spectral properties of the arithmetic model in the frequency domain.

11.7.3 FREQUENCY as auxiliary arithmetic model

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

A *measurement* annotation (see Section 11.29.1) shall be used in conjunction with the frequency statement. The frequency statement shall specify the repetition frequency of the measurement.

A frequency statement can substitute a time statement in the capacity of an auxiliary arithmetic model, if no from-to statement is used as model qualifier. In this case, the measurement repetition frequency f and the measurement time interval t can be related by the equation $f = 1 / t$.

11.8 DELAY

A *delay* statement shall be defined using ALF language as shown in .

```
KEYWORD DELAY = arithmetic_model {
  SI_MODEL = TIME ;
}
```

Syntax 163— statement

11.8.1 DELAY in context of a VECTOR declaration

A delay statement can be a child or a grandchild of a vector declaration involving a vector expression. A delay statement shall have a from-to statement featuring event references as model qualifier. The delay statement shall define the measured time interval between a from-event and a to-event. Both events shall be part of the vector expression. A causal relationship between the from-event and the to-event is implied.

A delay statement with an incomplete model qualifier featuring only a from statement or only a to statement can be used to specify an incremental time interval to be added to another time interval. The calculation annotation (see Section 11.5.2) shall be used in conjunction with such an incomplete model qualifier.

11.8.2 DELAY in context of a library-specific object declaration

A delay statement can be a child of a library-specific object which can be a parent of a vector. Possible parents of a vector include library, sublibrary, cell and wire. Within such a context, a delay statement can not have an event reference within a from-to statement as model qualifier. A from-to statement can only feature threshold statements. The specification given by the threshold statements shall be inherited by delay statements which are child of a vector.

11.9 RETAIN

A *retain* statement shall be defined using ALF language as shown in .

```
KEYWORD RETAIN = arithmetic_model {
  SI_MODEL = TIME ;
}
```

Syntax 164— statement

A retain statement can be a child or a grandchild of a vector declaration involving a vector expression. A retain statement can be used as a substitution for a delay statement in the case where the vector expression features more than one possible to-event. Retain represents the time interval between the from-event and the earliest to-event. Later to-events can be involved in a delay statement.

Retain in conjunction with delay is illustrated in Figure 20.

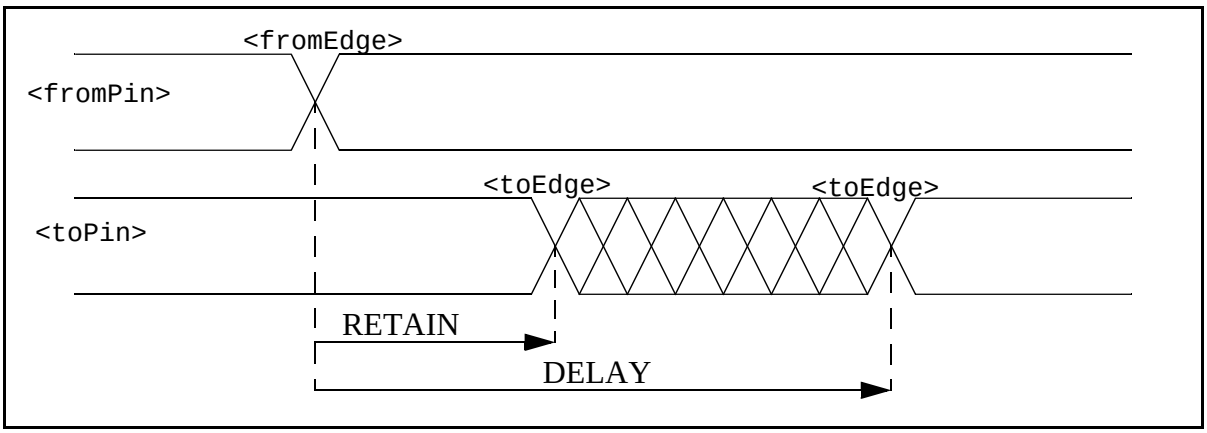


Figure 20—RETAIN and DELAY

11.10 SLEWRATE

A *xxx* statement shall be defined using ALF language as shown in .

```
KEYWORD SLEWRATE = arithmetic_model {
  SI_MODEL = TIME ;
}
SLEWRATE { MIN = 0; }
```

Syntax 165— statement

Slewrate shall define the duration of a single event, measured between two reference transition points. If the parent of the slewrate statement is a limit statement, the slewrate statement defines a minimum required or a maximum allowed duration of an event. Otherwise, slewrate defines the actually measured duration of an event.

11.10.1 SLEWRATE in context of a VECTOR declaration

A slewrate statement can be a child or a granddchild of a vector declaration. Slewrate can also be a dimension of an arithmetic model in the context of a vector.

The slewrate statement can have an event reference statement and a from-to statement without event reference as model qualifier. The from-and the to-statement can involve threshold statements.

11.10.2 SLEWRATE in context of a PIN declaration

A slewrate statement can be a child or a grandchild of a pin declaration. In this context, no from-to statement and no event-reference statement is allowed as model qualifier.

The slewrate statement can have a rise statement or a fall statement as arithmetic submodel.

11.10.3 SLEWRATE in context of a library-specific object declaration

A slewrate statement can be a child of a library-specific object which can be a parent of a vector. Possible parents of a vector include library, sublibrary, cell and wire. Within such a context, a slewrate statement can not have an event reference as model qualifier. A from-to statement with threshold statements can be used as model qualifier. The specification given by the threshold statements can be inherited by slewrate statements which are child of a vector.

The slewrate statement can have a rise statement or a fall statement as arithmetic submodel.

11.11 SETUP and HOLD

A *setup* and a *hold* statement shall be defined using ALF language as shown in .

11.11.1 SETUP in context of a VECTOR declaration

A setup statement can be a child of a vector declaration. Setup represents the minimal required time interval between a signal event and a synchronization event such that the signal is already stable when the synchroniza-

```
KEYWORD SETUP = arithmetic_model {
  SI_MODEL = TIME ;
}
KEYWORD HOLD = arithmetic_model {
  SI_MODEL = TIME ;
}
```

Syntax 166— statement

tion event occurs. The signal event and the synchronization event shall be represented as a from-event and a to-event, respectively, within a from-to statement.

11.11.2 HOLD in context of a VECTOR declaration

A hold statement can be a child of a vector declaration.Hold represents the minimal required time interval between a synchronization event and a signal event such that the synchronization event occurs while the signal is still stable. The synchronization event and the signal event shall be represented as a from-event and a to-event, respectively, within a from-to statement.

11.11.3 SETUP and HOLD in context of the same VECTOR declaration

A setup and a hold statement can be a child of the same vector, provided the vector expression features at least one synchronization event and two signal events related to the synchronization event. The sum of the time intervals represented by setup and hold represents a minimum required stability interval for the signal. This interval shall be greater than zero.

Setup in conjunction with hold is illustrated in Figure 21.

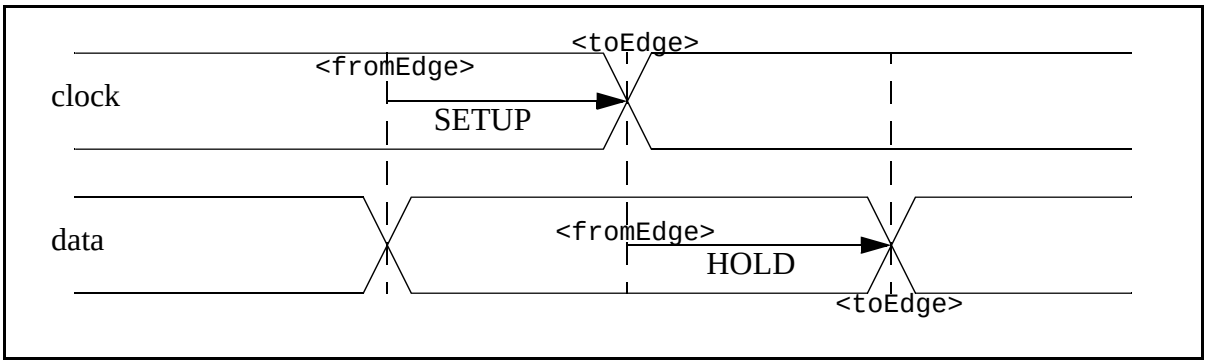


Figure 21—SETUP and HOLD

11.12 RECOVERY and REMOVAL

A *recovery* and a *removal* statement shall be defined using ALF language as shown in .

```
KEYWORD RECOVERY = arithmetic_model {
  SI_MODEL = TIME ;
}
KEYWORD REMOVAL = arithmetic_model {
  SI_MODEL = TIME ;
}
```

Syntax 167— statement

11.12.1 RECOVERY in context of a VECTOR declaration

A recovery statement can be a child of a vector declaration. Recovery represents the minimal required time interval between a controlling event with higher priority and a controlling event with lower priority such that the signal with higher priority is already inactive when the event on the signal with lower priority occurs. The event with higher priority and the event with lower priority shall be represented as a from-event and a to-event, respectively, within a from-to statement.

11.12.2 REMOVAL in context of a VECTOR declaration

A removal statement can be a child of a vector declaration. Removal represents the minimal required time interval between a controlling event with lower priority and a controlling event with higher priority such that the signal with higher priority is still active when the event with lower priority occurs. The event with higher priority and the event with lower priority shall be represented as a from-event and a to-event, respectively, within a from-to statement.

11.12.3 RECOVERY and REMOVAL in context of the same VECTOR declaration

A recovery and a removal statement can be a child of the same vector, provided the vector expression features at least one event with lower priority and two alternative events with highwe priority. The sum of the time intervals represented by recovery and removal represents a minimum required stability interval for the signal with higher priority. This interval shall be greater than zero.

Recovery in conjunction with removal is illustrated in Figure 22.

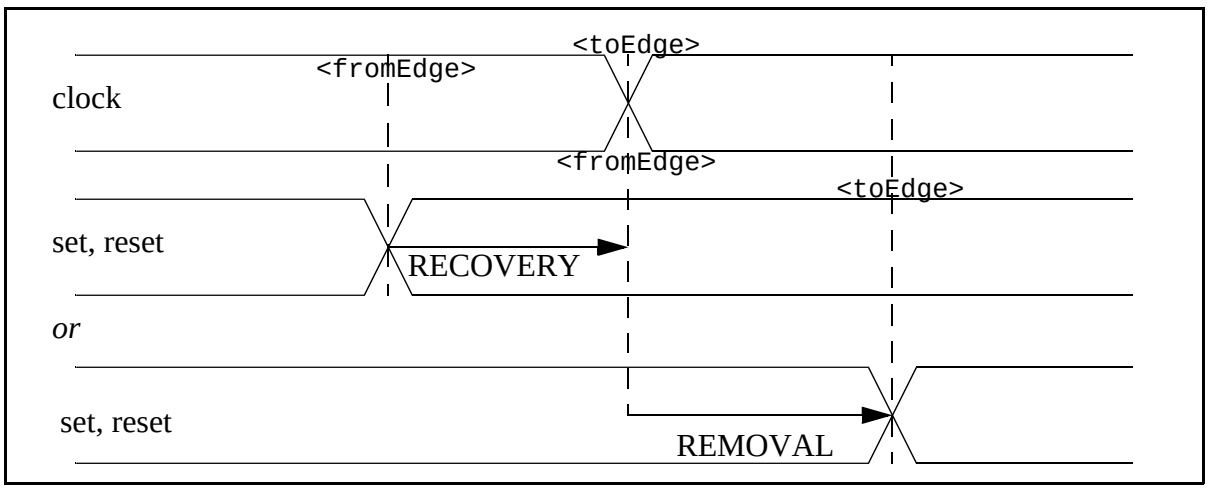


Figure 22—RECOVERY and REMOVAL

11.13 NOCHANGE and ILLEGAL

A *nochange* and an *illegal* statement shall be defined using ALF language as shown in .

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

Syntax 168— statement

11.13.1 NOCHANGE in context of a VECTOR declaration

A nochange statement can be a child of a vector declaration.

If the vector declaration involves a boolean expression, nochange shall specify a minimum required time interval during which the boolean expression is true. Nochange as a partial arithmetic model shall indicate a requirement for the boolean expression to be forever true.

If the vector declaration involves a vector expression, nochange as a partial arithmetic model shall indicate a requirement for the vector expression to be observed as specified. An optional from-to statement as model qualifier can indicate a requirement for the part of the vector expression within the time interval between the from-event and the to-event to be observed as specified. Nochange as a full arithmetic model or as a trivial arithmetic model shall furthermore specify a minimum required duration of the vector expression or part thereof.

11.13.2 ILLEGAL in context of a VECTOR declaration

An illegal statement can be a child of a vector declaration.

If the vector declaration involves a boolean expression, illegal shall specify a maximum allowed time interval during which the boolean expression is true. Illegal as a partial arithmetic model shall indicate a requirement for the boolean expression to be never true.

If the vector declaration involves a vector expression, illegal as a partial arithmetic model shall indicate that the vector expression is not allowed to occur. An optional from-to statement as model qualifier can indicate that a part of the vector expression within the time interval between the from-event and the to-event is not allowed to occur. Illegal as a full arithmetic model or as a trivial arithmetic model shall furthermore specify a maximum tolerated duration of the vector expression or part thereof.

11.14 SKEW

A *xxx* statement shall be defined using ALF language as shown in .

A skew statement can be a child of a vector declaration.

```
KEYWORD SKEW = arithmetic_model {
  SI_MODEL = TIME ;
}
SKEW { MIN = 0; }
```

Syntax 169— statement

11.14.1 SKEW involving two signals

A skew statement can specify a maximum allowed time interval between a from-event and a to-event. In this case, a from-to statement is mandatory as model qualifier. The vector declaration shall specify a vector expression such that the to-event cannot occur before the from-event.

11.14.2 SKEW involving multiple signals

A skew statement can specify a maximum allowed time separation between multiple events. In this case, a multi-value annotation containing pin references is mandatory as model qualifier. Optionally, this multi-value annotation can be accompanied by another multi-value annotation containing a matching number of edge numbers. The vector declaration shall specify a vector expression such that all events can occur simultaneously.

11.15 PULSEWIDTH

A *xxx* statement shall be defined using ALF language as shown in .

```
KEYWORD PULSEWIDTH = arithmetic_model {
  SI_MODEL = TIME ;
}
PULSEWIDTH { MIN = 0; }
```

Syntax 170— statement

A pulsewidth statement shall define the time interval between two consecutive events on the same signal. If the parent of the pulsewidth statement is a limit statement, pulsewidth defines a minimum required or a maximum allowed duration of the time interval. Otherwise, pulsewidth defines the actually measured time interval.

11.15.1 PULSEWIDTH in context of a VECTOR declaration

A pulsewidth statement can be a child of a vector declaration. Pulsewidth can also be a dimension of an arithmetic model in the context of a vector.

The pulsewidth statement can have an event-reference statement and a from-to statement without event reference as model qualifier. The from-and the to-statement can involve threshold statements. The event reference shall refer to the first of two consecutive events.

11.15.2 PULSEWIDTH in context of a PIN declaration

A pulsewidth statement can be a child or a grandchild of a pin declaration. In this context, no from-to statement and no event-reference statement is allowed as model qualifier.

The pulsewidth statement can have a rise statement and/or a fall statement as arithmetic submodel. The switching direction indicated by rise or fall shall refer to the first of two consequtive events.

11.15.3 PULSEWIDTH in context of a library-specific object declaration

A pulsewidth statement can be a child of a library-specific object which can be a parent of a vector. Possible parents of a vector include library, sublibrary, cell and wire. Within such a context, a pulsewidth statement can not have an event reference as model qualifier. A from-to statement with threshold statements can be used as model qualifier. The specification given by the threshold statements can be inherited by pulsewidth statements which are child of a vector.

The pulsewidth statement can have a rise statement or a fall statement as arithmetic submodel. The switching direction indicated by rise or fall shall refer to the first of two consequtive events.

11.16 PERIOD

A *xxx* statement shall be defined using ALF language as shown in .

```
KEYWORD PERIOD = arithmetic_model {
  SI_MODEL = TIME ;
}
PERIOD { MIN = 0; }
```

Syntax 171— statement

A period statement can be a child or a grandchild of a vector. Period can also be a dimension of an arithmetic model in the context of a vector. Period shall define the time interval between two consecutive occurences of a periodically repeating vector.

If the parent of the period statement is a limit statement, period defines a minimum required or a maximum allowed time interval. Otherwise, period defines the actually measured time interval.

11.17 JITTER

A *xxx* statement shall be defined using ALF language as shown in .

```
KEYWORD JITTER = arithmetic_model {
  SI_MODEL = TIME ;
}
JITTER { MIN = 0; }
```

Syntax 172— statement

A jitter statement can be a child or a grandchild of a vector. Jitter can also be a dimension of an arithmetic model in the context of a vector. Jitter shall define the variability of a time interval between two consecutive occurences of the periodically repeating vector.

If the parent of the jitte statement is a limit statement, jitter defines a minimum required or a maximum allowed variability of the time interval. Otherwise, jitter defines the actually measured variability of the time interval.

The measurement annotation (see Section 11.29.1) is applicable as model qualifier.

11.18 THRESHOLD

A *xxx* statement shall be defined using ALF language as shown in .

```
KEYWORD THRESHOLD = arithmetic_model {
  CONTEXT { PIN FROM TO }
}
THRESHOLD { MIN = 0; MAX = 1; }
```

Syntax 173— statement

The THRESHOLD represents a reference voltage level for timing measurements, normalized to the signal voltage swing and measured with respect to the logic 0 voltage level, as shown in Figure 23.

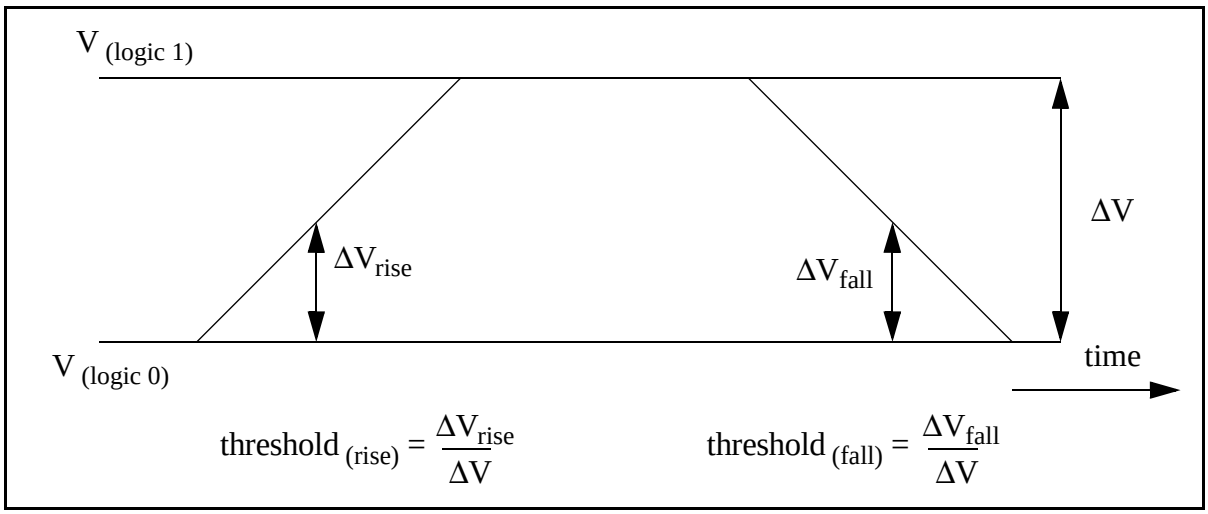


Figure 23—THRESHOLD measurement definition

The voltage levels for logic 1 and 0 represent a full voltage swing.

Different threshold data for RISE and FALL can be specified or else the data shall apply for both rising and falling transitions.

The THRESHOLD statement has the form of an arithmetic model. If the submodel keywords RISE and FALL are used, it has the form of an arithmetic model container.

The THRESHOLD statement can appear in the context of a FROM or TO container. In this case, it specifies the applicable reference for the start and end point of the timing measurement, respectively.

The THRESHOLD statement can also appear in the context of a PIN. In this case, it specifies the applicable reference for the start or end point of timing measurements indicated by the PIN annotation inside a FROM or TO container, unless a THRESHOLD is specified explicitly inside the FROM or TO container.

If both the RISE and FALL thresholds are specified and the switching direction of the applicable pin is clearly indicated in the context of a VECTOR, the RISE or FALL data shall be applied accordingly.

If thresholds are needed for exact definition of the model data, the FROM and TO containers shall each contain an arithmetic model for THRESHOLD.

FROM and TO containers with THRESHOLD definitions, yet without PIN annotations, can appear within unnamed timing model definitions in the context of a VECTOR, CELL, WIRE, SUBLIBRARY, or LIBRARY object for the purpose of specifying global threshold definitions for all timing models within scope of the definition. The following priorities apply:

a) THRESHOLD in the HEADER of the timing model
b) THRESHOLD in the FROM or TO statement within the timing model
c) THRESHOLD for timing model definition in the context of the same VECTOR
d) THRESHOLD within the PIN definition
e) THRESHOLD for timing model definition in the context of the same CELL or WIRE
f) THRESHOLD for timing model definition in the context of the same SUBLIBRARY
g) THRESHOLD for timing model definition in the context of the same LIBRARY
h) THRESHOLD for timing model definition outside LIBRARY

11.19 Annotations related to timing data

11.19.1 PIN reference annotation

If the timing measurements or timing constraints, respectively, apply semantically for two pins (see 11.9.1.1), the FROM and TO containers shall each contain the PIN annotation.

Otherwise, if the timing measurements or timing constraints apply semantically only to one pin (see 11.9.1.3), the PIN annotation shall be outside the FROM or TO container.

The following semantic restrictions shall apply.

```
SEMANTICS PIN = single_value_annotation {
  CONTEXT {
    FROM TO SLEWRATE PULSEWIDTH
    CAPACITANCE RESISTANCE INDUCTANCE VOLTAGE CURRENT
  }
}
SEMANTICS SKEW.PIN = multi_value_annotation ;
```

Syntax 174— Semantic restriction

11.19.2 EDGE_NUMBER annotation

A *xxx* statement shall be defined using ALF language as shown in .

The EDGE_NUMBER annotation within the context of a timing model shall specify the edge where the timing measurement applies. The timing model shall be in the context of a VECTOR. The EDGE_NUMBER shall have an unsigned value pointing to exactly one of subsequent vector_single_event expressions applicable to the referenced pin. The EDGE_NUMBER shall be counted individually for each pin which appears in the VECTOR, starting with zero (0).

```
KEYWORD EDGE_NUMBER = annotation {
  CONTEXT { FROM TO SLEWRATE PULSEWIDTH SKEW }
  VALUETYPE = unsigned_integer ;
  DEFAULT = 0;
}
SEMANTICS EDGE_NUMBER = single_value_annotation {
  CONTEXT { FROM TO SLEWRATE PULSEWIDTH }
}
SEMANTICS SKEW.EDGE_NUMBER = multi_value_annotation ;
```

Syntax 175— statement

If the timing measurements or timing constraints, apply semantically to two pins (see 11.9.1.1), the `EDGE_NUMBER` annotation shall be legal inside the `FROM` or `TO` container in conjunction with the `PIN` annotation.

Otherwise, if the timing measurements or timing constraints apply semantically only to one pin (see 11.9.1.3), the `EDGE_NUMBER` annotation shall be legal outside the `FROM` or `TO` container in conjunction with the `PIN` annotation.

11.20 PROCESS

A *xxx* statement shall be defined using ALF language as shown in .

```
KEYWORD PROCESS = arithmetic_model {
  VALUETYPE = identifier ;
}
PROCESS { DEFAULT = nom; TABLE { nom snsp snwp wnsp wnwp } }
```

Syntax 176— statement

The following identifiers can be used as predefined process corners:

`?n?p` process definition with transistor strength

where `?` can be

`s` strong
`w` weak

The possible process name combinations are shown in Table 86.

Table 86—Predefined process names

Process name	Description
`snsp`	Strong NMOS, strong PMOS.
`snwp`	Strong NMOS, weak PMOS.

Table 86—Predefined process names (Continued)

Process name	Description
wnsp	Weak NMOS, strong PMOS.
wnwp	Weak NMOS, weak PMOS.

11.21 DERATE_CASE

A *xxx* statement shall be defined using ALF language as shown in .

```
KEYWORD DERATE_CASE = arithmetic_model {
  VALUETYPE = identifier ;
}
DERATE_CASE { DEFAULT = nom;
  TABLE { nom bccom wccom bcind wcind bcmil wcmil }}
}
```

Syntax 177— statement

The following identifiers can be used as predefined derating cases:

nom	nominal case
bc?	prefix for best case
wc?	prefix for worst case

where ? can be

com	suffix for commercial case
ind	suffix for industrial case
mil	suffix for military case

The possible derating case combinations are defined in Table 87.

Table 87—Predefined derating cases

Derating case	Description
bccom	Best case commercial.
bcind	Best case industrial.
bcmil	Best case military.
wccom	Worst case commercial.
wcind	Worst case military.
wcmil	Worst case military.

11.22 TEMPERATURE

A *xxx* statement shall be defined using ALF language as shown in .

```
KEYWORD TEMPERATURE = arithmetic_model {
  VALUETYPE = number ;
}
TEMPERATURE { MIN = -273; }
```

Syntax 178— statement

TEMPERATURE can be used as argument in the HEADER of an arithmetic model for timing or electrical data. It can also be used as an arithmetic model with DERATE_CASE as argument, in order to describe what temperature applies for the specified derating case.

11.23 PIN-related arithmetic models for electrical data

Arithmetic models for electrical data can be associated with a pin of a cell. Their meaning is illustrated in Figure 24.

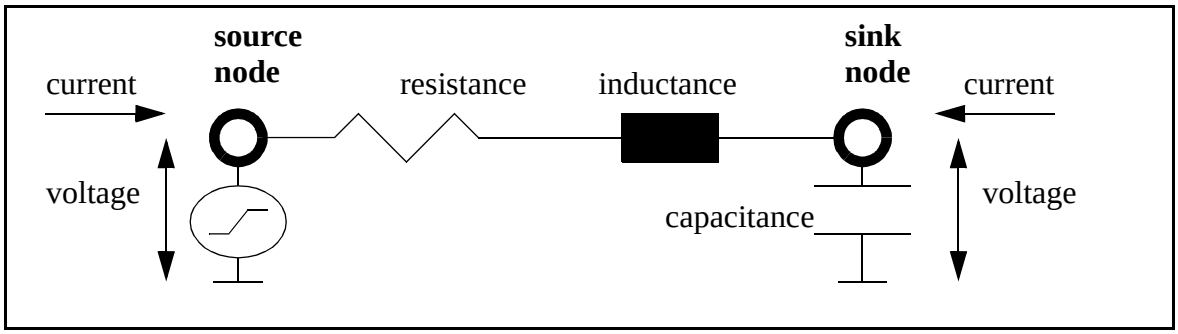


Figure 24—General representation of electrical models around a pin

A pin is represented as a source node and a sink node. For pins with DIRECTION=input, the source node is externally accessible. For pins with DIRECTION=output, the sink node is externally accessible.

11.23.1 CAPACITANCE, RESISTANCE, and INDUCTANCE

A *xxx* statement shall be defined using ALF language as shown in .

RESISTANCE and INDUCTANCE apply between the source and sink node. CAPACITANCE applies between the sink node and ground. By default, the values for resistance, inductance and capacitance shall be zero (0).

11.23.2 VOLTAGE and CURRENT

A *xxx* statement shall be defined using ALF language as shown in .

```
KEYWORD CAPACITANCE = arithmetic_model {
  VALUETYPE = number ;
}
KEYWORD RESISTANCE = arithmetic_model {
  VALUETYPE = number ;
}
KEYWORD INDUCTANCE = arithmetic_model {
  VALUETYPE = number ;
}
CAPACITANCE { UNIT = 1e-12; MIN = 0; }
RESISTANCE { UNIT = 1e3; MIN = 0; }
INDUCTANCE { UNIT = 1e-6; MIN = 0; }
```

Syntax 179— statement

```
KEYWORD VOLTAGE = arithmetic_model {
  VALUETYPE = number ;
}
KEYWORD CURRENT = arithmetic_model {
  VALUETYPE = number ;
}
VOLTAGE { UNIT = 1; }
CURRENT { UNIT = 1e-3; }
```

Syntax 180— statement

VOLTAGE and CURRENT can be measured at either source or sink node, depending on which node is externally accessible. However, a voltage source can only be connected to a source node. The sense of measurement for voltage shall be from the node to ground. The sense of measurement for current shall be *into* the node.

11.23.3 Context-specific semantics

An arithmetic model for VOLTAGE, CURRENT, SLEWRATE, RESISTANCE, INDUCTANCE, and CAPACITANCE can be associated with a PIN in one of the following ways.

a) A model in the context of a PIN

Example

```
PIN my_pin {
    CAPACITANCE = 0.025;
```

b) A model in the context of a CELL, WIRE, or VECTOR with PIN annotation

Example

```
VOLTAGE = 1.8 { PIN = my_pin; }
```

The model in the context of a PIN shall be used if the data is completely confined to the pin. That means, no argument of the model shall make reference to any pin, since such reference implies an external dependency. A

model with dependency only on environmental data not associated with a pin (e.g., TEMPERATURE, PROCESS, and DERATE_CASE) can be described within the context of the PIN.

A model with dependency on external data applied to a pin (e.g., load capacitance) shall be described outside the context of the PIN, using a PIN annotation. In particular, if the model involves a dependency on logic state or logic transition of other PINs, the model shall be described within the context of a VECTOR.

Figure 25 illustrates electrical models associated with input and output pins.

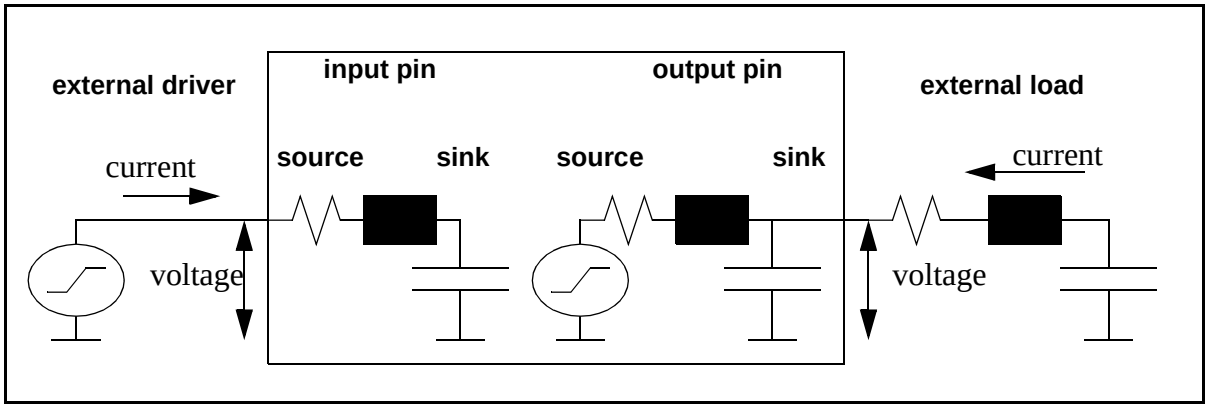


Figure 25—Electrical models associated with input and output pins

Table 88 and Table 89 define how models are associated with the pin, depending on the context.

Table 88—Direct association of models with a PIN

Model	Model in context of PIN	Model in context of CELL, WIRE, and VECTOR with PIN annotation
CAPACITANCE	Pin self-capacitance.	Externally controlled capacitance at the pin, e.g., voltage-dependent.
INDUCTANCE	Pin self-inductance.	Externally controlled inductance at the pin, e.g., voltage-dependent.
RESISTANCE	Pin self-resistance.	Externally controlled resistance at the pin, e.g., voltage-dependent, in the context of a VECTOR for timing-arc specific driver resistance.
VOLTAGE	Operational voltage measured at pin.	Externally controlled voltage at the pin.
CURRENT	Operational current measured into pin.	Externally controlled current into pin.
SAME_PIN_TIMING_MEASUREMENT	For model definition, default, etc.; not for the timing arc.	In context of VECTOR for timing arc, other context for definition, default, etc.
SAME_PIN_TIMING_CONSTRAINT	For model definition, default, etc.; not for the timing arc.	In context of VECTOR for timing arc, other context for definition, default, etc.

Table 89—External association of models with a PIN

Model / Context	LIMIT within PIN or with PIN annotation	Model argument with PIN annotation
CAPACITANCE	Min or max limit for applicable load.	Load for model characterization.
INDUCTANCE	Min or max limit for applicable load.	Load for model characterization.
RESISTANCE	Min or max limit for applicable load.	Load for model characterization.
VOLTAGE	Min or max limit for applicable voltage.	Voltage for model characterization.
CURRENT	Min or max limit for applicable current.	Current for model characterization.
SAME_PIN_TIMING_MEASUREMENT	Currently applicable for min or max limit for SLEWRATE.	Stimulus with SLEWRATE for model characterization.
SAME_PIN_TIMING_CONSTRAINT	N/A, since the keyword means a min or max limit by itself.	N/A

Example

```
CELL my_cell {
    PIN pin1 { DIRECTION=input; CAPACITANCE = 0.05; }
    PIN pin2 { DIRECTION=output; LIMIT { CAPACITANCE { MAX=1.2; } } }
    PIN pin3 { DIRECTION=input; }
    PIN pin4 { DIRECTION=input; }
    CAPACITANCE {
       PIN=pin3;
       HEADER { VOLTAGE { PIN=pin4; } }
       EQUATION { 0.25 + 0.34*VOLTAGE }
    }
}
```

The capacitance on `pin1` is `0.05`. The maximum allowed load capacitance on `pin2` is `1.2`. The capacitance on `pin3` depends on the voltage on pin4.

11.24 POWER and ENERGY

A *xxx* statement shall be defined using ALF language as shown in .

```
KEYWORD POWER = arithmetic_model {
  VALUETYPE = number ;
}
KEYWORD ENERGY = arithmetic_model {
  VALUETYPE = number ;
}
POWER { UNIT = 1e-3; }
ENERGY { UNIT = 1e-12; }
```

Syntax 181— statement

The purpose of power calculation is to evaluate the electrical power supply demand and electrical power dissipation of an electronic circuit. In general, both power supply demand and power dissipation are the same, due to the energy conservation law. However, there are scenarios where power is supplied and dissipated locally in different places. The power models in ALF shall be specified in such a way that the total power supply and dissipation of a circuit adds up correctly to the same number.

Example

> A capacitor `C` is charged from `0` volt to `V` volt by a switched DC source. The energy supplied by the source is $C*V^2$. The energy stored in the capacitor is $1/2*C*V^2$. Hence the dissipated energy is also $1/2*C*V^2$. Later the capacitor is discharged from `V` volt to `0` volt. The supplied energy is `0`. The dissipated energy is $1/2*C*V^2$. A supply-oriented power model can associate the energy $E_1=C*V^2$ with the charging event and $E_2=0$ with the discharging event. The total energy is $E=E_1+E_2=C*V^2$. A dissipation-oriented power model can associate the energy $E_3=1/2*C*V^2$ with both the charging and discharging event. The total energy is also $E=2*E_3=C*V^2$.

In many cases, it is not so easy to decide when and where the power is supplied and where it is dissipated. The choice between a supply-oriented and dissipation-oriented model or a mixture of both is subjective. Hence the ALF language provides no means to specify, which modeling approach is used. The choice is up to the model developer, as long as the energy conservation law is respected.

`POWER` and/or `ENERGY` models shall be in the context of a `CELL` or within a `VECTOR`. The total energy and/or power of a cell shall be calculated by combining the data of all models within the scope of the `CELL` or the `VECTOR`s within the cell.

The data for `POWER` and/or `ENERGY` shall be positive when energy is actually supplied to the `CELL` and/or dissipated within the `CELL`. The data shall be negative when energy is actually supplied or restored by the `CELL`.

11.25 FLUX and FLUENCE

A *xxx* statement shall be defined using ALF language as shown in .

```
KEYWORD FLUX = arithmetic_model {
  VALUETYPE = number ;
}
KEYWORD FLUENCE = arithmetic_model {
  VALUETYPE = number ;
}
FLUX { UNIT = 1e-3; }
FLUENCE { UNIT = 1e-12; }
```

Syntax 182— statement

The purpose of hot electron calculation is to evaluate the damage done to the performance of an electronic device due to the hot electron effect. The hot electron effect consists in accumulation of electrons trapped in the gate oxide of a transistor. The more electrons are trapped, the more the device slows down. At a certain point, the performance specification no longer is met and the device is considered to be damaged.

FLUX and/or FLUENCE models shall be in the context of a CELL or within a VECTOR. Total fluence and/or flux of a cell shall be calculated by combining the data of all models within the scope of the CELL or the VECTORs within the cell.

Both FLUX and FLUENCE are measures for hot electron damage. FLUX relates to FLUENCE in the same way as POWER relates to ENERGY.

11.26 DRIVE_STRENGTH

A *xxx* statement shall be defined using ALF language as shown in .

```
KEYWORD DRIVE_STRENGTH = arithmetic_model {
  VALUETYPE = number ;
}
DRIVE_STRENGTH { MIN = 0; }
```

Syntax 183— statement

DRIVE_STRENGTH is a unit-less, abstract measure for the drivability of a PIN. It can be used as a substitute of driver RESISTANCE. The higher the DRIVE_STRENGTH, the lower the driver RESISTANCE. However, DRIVE_STRENGTH can only be used within a coherent system of calculation models, since it does not represent an absolute quantity, as opposed to RESISTANCE. For example, the weakest driver of a library can have drive strength 1, the next stronger driver can have drive strength 2 and so forth. This does not necessarily mean the resistance of the stronger driver is exactly half of the resistance of the weaker driver.

An arithmetic model for conversion from DRIVE_STRENGTH to RESISTANCE can be given to relate the quantity DRIVE_STRENGTH across technology libraries.

Example

```
SUBLIBRARY high_speed_library {
    RESISTANCE {
       HEADER { DRIVE_STRENGTH } EQUATION { 800 / DRIVE_STRENGTH }
    }
    CELL high_speed_std_driver {
       PIN Z { DIRECTION = output; DRIVE_STRENGTH = 1; }
    }
}
SUBLIBRARY low_power_library {
    RESISTANCE {
       HEADER { DRIVE_STRENGTH } EQUATION { 1600 / DRIVE_STRENGTH }
    }
    CELL low_power_std_driver {
       PIN Z { DIRECTION = output; DRIVE_STRENGTH = 1; }
    }
}
```

Drive strength 1 in the high speed library corresponds to 800 ohm. Drive strength 1 in the low power library corresponds to 1600 ohm.

NOTE—Any particular arithmetic model for RESISTANCE in either library shall locally override the conversion formula from drive strength to resistance.

11.27 SWITCHING_BITS

A *xxx* statement shall be defined using ALF language as shown in .

```
KEYWORD SWITCHING_BITS = arithmetic_model {
  VALUETYPE = unsigned_integer ;
}
```

Syntax 184— statement

The quantity SWITCHING_BITS applies only for bus pins. The range is from 0 to the width of the bus. Usually, the quantity SWITCHING_BITS is not calculated by an arithmetic model, since the number of switching bits on a bus depends on the functional specification rather than the electrical specification. However, SWITCHING_BITS can be used as argument in the HEADER of an arithmetic model to calculate electrical quantities, for instance, energy consumption.

Example

```
CELL my_rom {
    PIN [3:0] addr { DIRECTION=input;  SIGNALTYPE=address; }
    PIN [7:0] dout { DIRECTION=output; SIGNALTYPE=data; }
    VECTOR ( ?! addr -> ?! dout ) {
       ENERGY {
          HEADER {
             SWITCHING_BITS addr_bits { PIN = addr; }
             SWITCHING_BITS dout_bits { PIN = dout; }
          }
          EQUATION { 0.45*LOG(addr_bits) + 2.6*dout_bits }
       }
    }
}
```

The energy consumption of my_rom depends on the number of switching data bits and on the logarithm of the number of switching address bits.

11.28 NOISE and NOISE MARGIN

A *xxx* statement shall be defined using ALF language as shown in .

11.28.1 NOISE MARGIN

Noise margin is defined as the maximal allowed difference between the ideal signal voltage under a well-specified operation condition and the actual signal voltage normalized to the ideal voltage swing. This is illustrated in Figure 26.

```
KEYWORD NOISE = arithmetic_model {
  VALUETYPE = number ;
}
KEYWORD NOISE_MARGIN = arithmetic_model {
  VALUETYPE = number ;
}
NOISE { MIN = 0; }
NOISE_MARGIN { MIN = 0; MAX = 1; }
```

Syntax 185— statement

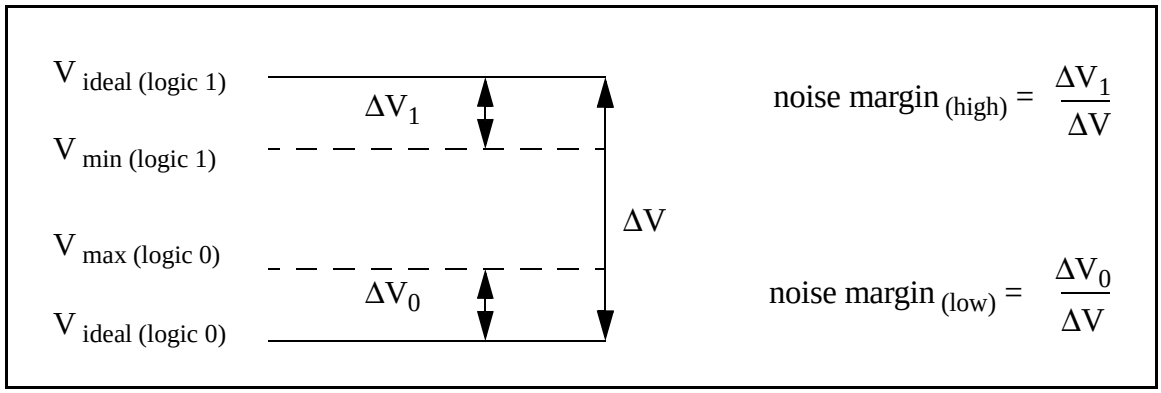


Figure 26—Definition of noise margin

NOISE_MARGIN is a pin-related quantity. It can appear either in the context of a PIN statement or in the context of a VECTOR statement with PIN annotation. It can also appear in the global context of a CELL, SUBLIBRARY, or LIBRARY statement.

If a NOISE_MARGIN statement appears in multiple contexts, the following priorities apply:

a) NOISE_MARGIN with PIN annotation in the context of the VECTOR, NOISE_MARGIN with PIN annotation in the context of the CELL, or NOISE_MARGIN in the context of the PIN
b) NOISE_MARGIN without PIN annotation in the context of the CELL
c) NOISE_MARGIN in the context of the SUBLIBRARY
d) NOISE_MARGIN in the context of the LIBRARY
e) NOISE_MARGIN outside the LIBRARY

11.28.2 NOISE

Noise is defined as thee actual measured noise against which the noise margin is compared.

11.29 Annotations and statements related to electrical models

11.29.1 MEASUREMENT annotation

A *xxx* statement shall be defined using ALF language as shown in .

```
KEYWORD MEASUREMENT = single_value_annotation {
  VALUETYPE = identifier ;
  VALUES {
    transient static average absolute_average rms peak
  }
  CONTEXT {
    ENERGY POWER CURRENT VOLTAGE FLUX FLUENCE JITTER
  }
}
```

Syntax 186— statement

Arithmetic models can have a `MEASUREMENT` annotation. This annotation indicates the type of measurement used for the computation in arithmetic model.

The meaning of the annotation values is shown in Table 90.

Table 90—MEASUREMENT annotation

Annotation string	Description
`transient`	Measurement is a transient value.
`static`	Measurement is a static value.
`average`	Measurement is an average value.
`rms`	Measurement is an root mean square value.
`peak`	Measurement is a peak value.

Their mathematical definitions are shown in Figure 27.

`transient` $\int_{(t=0)}^{(t=T)} dE(t)$

`average` $\frac{\int_{(t=0)}^{(t=T)} E(t)dt}{T}$

`static` $E = constant$

`rms` $\sqrt{\frac{\int_{(t=0)}^{(t=T)} E(t)^2 dt}{T}}$

`peak` $max(|E(t)|) \cdot \mathrm{sgn}E(t) \qquad t = T$

Figure 27—Mathematical definitions for MEASUREMENT annotations

Arithmetic models with certain values of `MEASUREMENT` annotation can also have *either* `TIME` *or* `FREQUENCY` as auxiliary arithmetic models.

The semantics are defined in Table 91.

Table 91—Semantic interpretation of MEASUREMENT, TIME, or FREQUENCY

MEASUREMENT annotation	Semantic meaning of TIME	Semantic meaning of FREQUENCY
`transient`	Integration of analog measurement is done during that time window.	Integration of analog measurement is repeated with that frequency.
`static`	N/A	N/A
`average`	Average value is measured over that time window.	Average value measurement is repeated with that frequency.
`rms`	Root-mean-square value is measured over that time window.	Root-mean-square measurement is repeated with that frequency.
`peak`	Peak value occurs at that time (only within context of `VECTOR`).	Observation of peak value is repeated with that frequency.

In the case of `average` and `rms`, the interpretation `FREQUENCY = 1 / TIME` is valid. Either one of these annotations shall be mandatory. The values for `average` measurements and for `rms` measurements scale linearly with `FREQUENCY` and `1 / TIME`, respectively.

In the case of `transient` and `peak`, the interpretation `FREQUENCY = 1 / TIME` is not valid. Either one of these annotations shall be optional. The values do not necessarily scale with `TIME` or `FREQUENCY`. The `TIME` or `FREQUENCY` annotations for `transient` measurements are purely informational.

11.29.2 TIME to peak measurement

For a model in the context of a `VECTOR`, with a `peak` measurement, the `TIME` annotation shall define the time between a reference event within the `vector_expression` and the instant when the peak value occurs.

For that purpose, either the `FROM` or the `TO` statement shall be used in the context of the `TIME` annotation, containing a `PIN` annotation and, if necessary, a `THRESHOLD` and/or an `EDGE_NUMBER` annotation.

If the `FROM` statement is used, the start point shall be the reference event and the end point shall be the occurrence time of the peak, as shown in Figure 28.

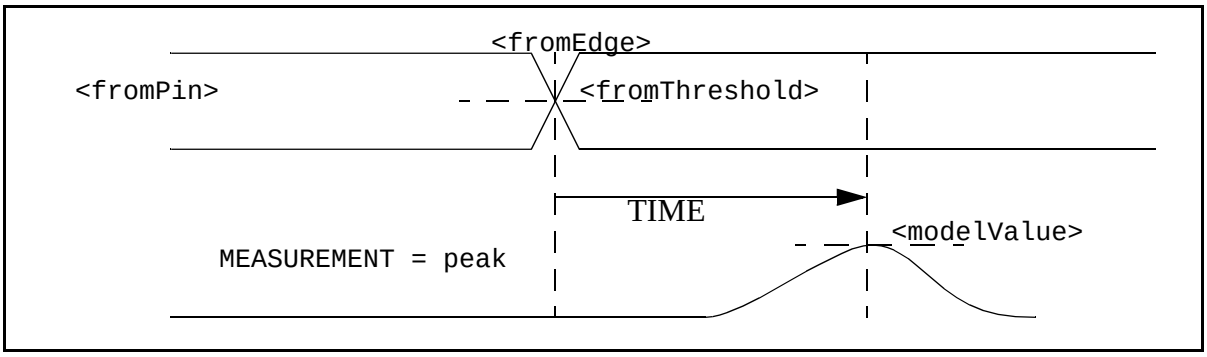


Figure 28—Illustration of time to peak using FROM statement

If the TO statement is used, the start point shall be the occurrence time of the peak and the end point shall be the reference event, as shown in Figure 29.

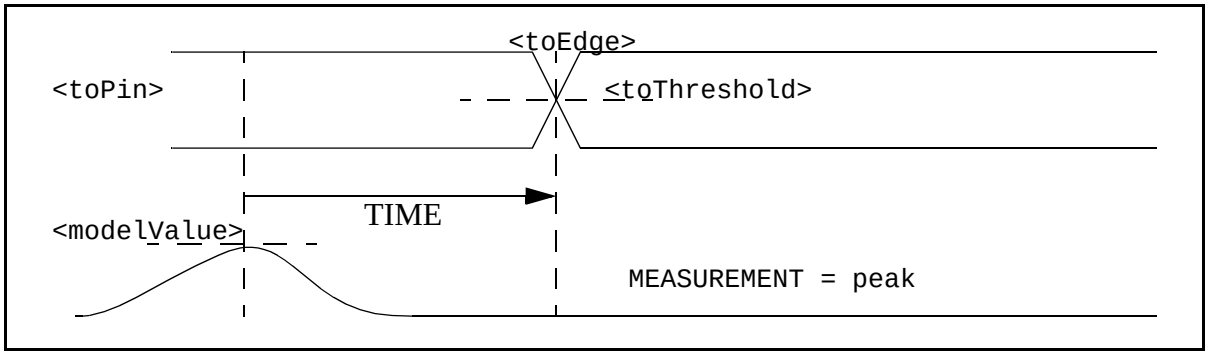


Figure 29—Illustration of time to peak using TO statement

11.30 CONNECTIVITY

A *xxx* statement shall be defined using ALF language as shown in .

```
KEYWORD CONNECTIVITY = arithmetic_model {
  VALUETYPE = boolean ;
  VALUES { 1 0 ? }
}
```

Syntax 187— statement

A *xxx* statement shall be defined using ALF language as shown in .

```
KEYWORD DRIVER = arithmetic_model {
  VALUETYPE = identifier ;
  CONTEXT = CONNECTIVITY.HEADER
}
KEYWORD RECEIVER = arithmetic_model {
  VALUETYPE = identifier ;
  CONTEXT = CONNECTIVITY.HEADER
}
```

Syntax 188— statement

Connectivity can also be described as a lookup table model. This description is usually more compact than the description using the BETWEEN statements.

The connectivity model can have the arguments shown in Table 92 in the HEADER.

Table 92—Arguments for connectivity

Argument	Value type	Description
DRIVER	string	Dimension of connectivity function.
RECEIVER	string	Dimension of connectivity function.

Each dimension shall contain a TABLE.

The connectivity model specifies the allowed and disallowed connections amongst drivers or receivers in one-dimensional tables or between drivers and receivers in two-dimensional tables.The boolean literals in the table refer to the CONNECT_RULE as shown in Table 93.

Table 93—Boolean literals in non-interpolateable tables

Boolean literal	Description
1	CONNECT_RULE is *True*.
0	CONNECT_RULE is *False*.
?	CONNECT_RULE does not apply.

11.31 SIZE

A *xxx* statement shall be defined using ALF language as shown in .

```
KEYWORD SIZE = arithmetic_model {
  VALUETYPE = unsigned_number ;
}
```

Syntax 189— statement

11.32 AREA

A *xxx* statement shall be defined using ALF language as shown in .

```
KEYWORD AREA = arithmetic_model {
  VALUETYPE = unsigned_number ;
}
```

Syntax 190— statement

11.33 WIDTH

A *xxx* statement shall be defined using ALF language as shown in .

```
KEYWORD WIDTH = arithmetic_model {
  VALUETYPE = unsigned_number ;
}
```

Syntax 191— statement

11.34 HEIGHT

A *xxx* statement shall be defined using ALF language as shown in .

```
KEYWORD HEIGHT = arithmetic_model {
  VALUETYPE = unsigned_number ;
}
```

Syntax 192— statement

11.35 LENGTH

A *xxx* statement shall be defined using ALF language as shown in .

```
KEYWORD LENGTH = arithmetic_model {
  VALUETYPE = unsigned_number ;
}
```

Syntax 193— statement

11.36 DISTANCE

A *xxx* statement shall be defined using ALF language as shown in .

```
KEYWORD DISTANCE = arithmetic_model {
  VALUETYPE = unsigned_number ;
}
```

Syntax 194— statement

11.37 OVERHANG

A *xxx* statement shall be defined using ALF language as shown in .

```
KEYWORD OVERHANG = arithmetic_model {
  VALUETYPE = unsigned_number ;
}
```

Syntax 195— statement

11.38 PERIMETER

A *xxx* statement shall be defined using ALF language as shown in .

```
KEYWORD PERIMETER = arithmetic_model {
  VALUETYPE = unsigned_number ;
}
```

Syntax 196— statement

11.39 EXTENSION

A *xxx* statement shall be defined using ALF language as shown in .

```
KEYWORD EXTENSION = arithmetic_model {
  VALUETYPE = unsigned_number ;
}
```

Syntax 197— statement

11.40 THICKNESS

A *xxx* statement shall be defined using ALF language as shown in .

```
KEYWORD THICKNESS = arithmetic_model {
  VALUETYPE = unsigned_number ;
}
```

Syntax 198— statement

11.41 Annotations for physical models

11.41.1 CONNECT_RULE annotation

A *xxx* statement shall be defined using ALF language as shown in .

```
KEYWORD CONNECT_RULE = single_value_annotation {
  VALUETYPE = identifier ;
  VALUES { must_short can_short cannot_short }
  CONTEXT = CONNECTIVITY;
}
```

Syntax 199— statement

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

Table 94—CONNECT_RULE annotation

Annotation string	Description
must_short	Electrical connection required.
can_short	Electrical connection allowed.
cannot_short	Electrical connection disallowed.

It is not necessary to specify more than one rule between a given set of objects. If one rule is specified to be *True*, the logical value of the other rules can be implied shown in Table 95.

Table 95—Implications between connect rules

must_short	cannot_short	can_short
False	*False*	*True*
False	*True*	*False*
True	*False*	N/A

11.41.2 BETWEEN annotation

A *xxx* statement shall be defined using ALF language as shown in .

If the BETWEEN statement contains only one identifier, than the CONNECTIVITY shall apply between multiple instances of the same object.

```
KEYWORD BETWEEN = multi_value_annotation {
  VALUETYPE = identifier ;
  CONTEXT { DISTANCE LENGTH OVERHANG CONNECTIVITY }
}
```

Syntax 200— statement

The `BETWEEN` statement within `DISTANCE` or `LENGTH` shall identify the objects for which the measurement applies.

If the `BETWEEN` statement contains only one identifier, than the `DISTANCE` or `LENGTH`, respectively, shall apply between multiple instances of the same object, as shown in the following example and Figure 30.

Example

```
DISTANCE = 4 { BETWEEN { object1 object2 } }
LENGTH = 2 { BETWEEN { object1 object2 } }
```

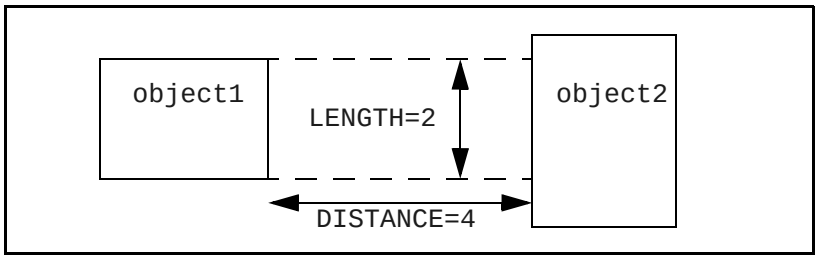


Figure 30—Illustration of LENGTH and DISTANCE

11.41.3 DISTANCE-MEASUREMENT annotation

A *xxx* statement shall be defined using ALF language as shown in .

```
KEYWORD DISTANCE_MEASUREMENT = single_value_annotation {
  VALUETYPE = identifier ;
  VALUES { euclidean horizontal vertical manhattan }
  DEFAULT = euclidean ;
  CONTEXT = DISTANCE ;
}
```

Syntax 201— statement

The mathematical definitions for distance measurements between two points with differential coordinates Δx and Δy are:

- *euclidean* distance = $(\Delta x^2 + \Delta y^2)^{1/2}$
- *horizontal* distance = Δx
- *vertical* distance = Δy
- *manhattan* distance = $\Delta x + \Delta y$

11.41.4 REFERENCE annotation container

A *xxx* statement shall be defined using ALF language as shown in .

```
KEYWORD REFERENCE = annotation_container {
  CONTEXT = DISTANCE ;
}
SEMANTICS REFERENCE.identifier = single_value_annotation {
  VALUETYPE = identifier ;
  VALUES { center origin near_edge far_edge }
  DEFAULT = origin ;
}
```

Syntax 202— statement

The meaning of the annotation values is illustrated in Figure 31.

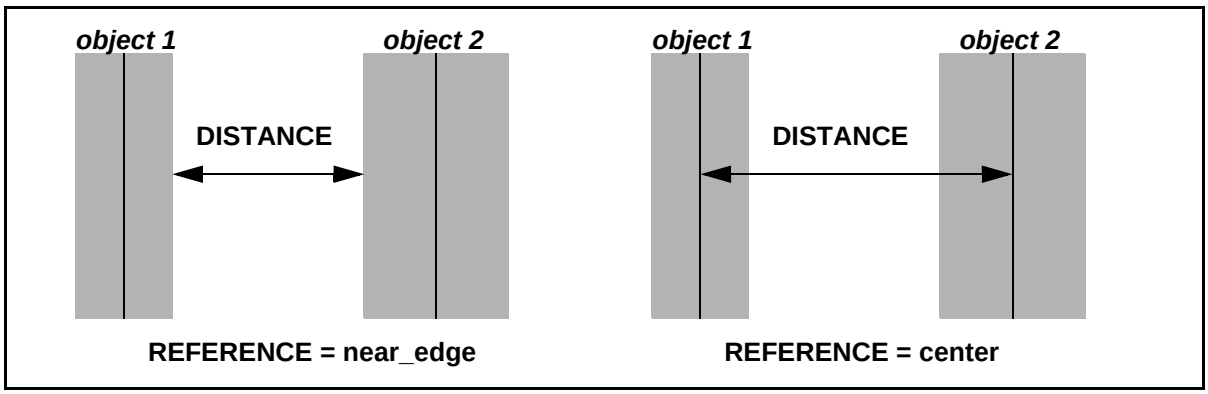


Figure 31—Illustration of REFERENCE for DISTANCE

11.41.5 ANTENNA reference annotation

A *xxx* statement shall be defined using ALF language as shown in .

```
SEMANTICS ANTENNA = annotation {
  VALUETYPE = identifier ;
  CONTEXT { PIN.SIZE PIN.AREA PIN.PERIMETER }
}
```

Syntax 203— statement

In hierarchical design, a `PIN` with physical `PORT`s can be abstracted. Therefore, an arithmetic model for `SIZE`, `AREA`, `PERIMETER`, etc. **relevant?? for certain antenna rules can be precalculated. An `ANTENNA` statement within the arithmetic model enables references to the set of antenna rules for which the arithmetic model applies

Example

```
CELL cell1 {
    PIN pin1 {
       AREA poly_area = 1.5 {
          LAYER = poly;
          ANTENNA { individual_m1 individual_via1 }
       }
       AREA m1_area = 1.0 {
          LAYER = metal1;
          ANTENNA { individual_m1 }
       }
       AREA via1_area = 0.5 {
          LAYER = via1;
          ANTENNA { individual_via1 }
       }
    }
}
```

The area `poly_area` is used in the rules `individual_m1` and `individual_via1`.
The area `m1_area` is used in the rule `individual_m1` only.
The area `via1_area` is used in the rule `individual_via1` only.

The case with diffusion is illustrated in the following example:

```
CELL my_diode {
    CELLTYPE = special; ATTRIBUTE { DIODE }
    PIN my_diode_pin {
       AREA = 3.75 {
          LAYER = diffusion;
          ANTENNA { rule1_for_diffusion rule2_for_diffusion }
       }
    }
}
```

11.41.6 PATTERN reference annotation

A *xxx* statement shall be defined using ALF language as shown in .

```
SEMANTICS PATTERN = single_value_annotation {
  VALUETYPE = identifier ;
  CONTEXT {
    LENGTH WIDTH HEIGHT SIZE AREA THICKNESS
    PERIMETER EXTENSION
  }
}
```

Syntax 204— statement

Reference to a PATTERN shall be legal within arithmetic models, if the pattern and the model are within the scope of the same parent object.

11.42 Arithmetic submodels for timing and electrical data

The arithmetic submodels shown in Table 96 are only applicable in the context of electrical modeling.

Table 96—Submodels applicable for timing and electrical modeling

Object	Description
HIGH	Applicable for electrical data measured at a logic high state of a pin.
LOW	Applicable for electrical data measured at a logic low state of a pin.
RISE	Applicable for electrical data measured during a logic low to high transition of a pin.
FALL	Applicable for electrical data measured during a logic high to low transition of a pin.

11.43 Arithmetic submodels for physical data

The arithmetic submodels shown in Table 97 are only applicable in the context of physical modeling.

Table 97—Submodels applicable for physical modeling

Object	Description
HORIZONTAL	Applicable for layout measurements in 0 degree, i.e., horizontal direction.
VERTICAL	Applicable for layout measurements in 90 degree, i.e., vertical direction.
ACUTE	Applicable for layout measurements in 45 degree direction.
OBTUSE	Applicable for layout measurements in 135 degree direction.

Annex A

(informative)

Syntax rule summary

This summary replicates the syntax detailed in the preceding clauses. If there is any conflict, in detail or completeness, the syntax presented in the clauses shall considered as the normative definition.

The current ordering is as each item appears in its subchapter; this needs to be updated to be complete.

A.1 Lexical definitions

any_character ::= (see 6.2.3)
  reserved_character
| nonreserved_character
| escape_character
| whitespace

reserved_character ::= (see 6.2.3)
  & | **|** | **^** | **~** | **+** | **-** | ***** | **/** | **%** | **?** | **!** | **=** | **<** | **>** | **:** | **(** | **)** | **[** | **]** | **{** | **}** | **@** | **;** | **,** | **.** | **”** | **’**

nonreserved_character ::= (see 6.2.4)
  letter | digit | **_** | **$** | **#**

letter ::=
  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**
| **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**

escape_character ::= (see 6.2.5)
  ****

delimiter ::= (see 6.3)
  reserved_character
| **&&** | **~&** | **||** | **~|** | **~^** | **==** | **!=** | ****** | **>=** | **<=** | **?!** | **?~** | **?-** | **??** | **?*** | ***?**
| **->** | **<->** | **&>** | **<&>** | **>>** | **<<**

comment ::= (see 6.2)
  single_line_comment
| block_comment

integer ::= (see 6.5)
  [sign] unsigned

sign ::=
  + | **-**

unsigned ::=
  digit { **_** | digit }

non_negative_number ::=
  unsigned [**.** unsigned]
| unsigned [**.** unsigned] **E** [sign] unsigned

number ::=
  [sign] non_negative_number

bit_literal ::= (see 6.7)
  numeric_bit_literal

| alphabetic_bit_literal
| dont_care_literal
| random_literal

numeric_bit_literal ::=
0 | **1**

alphabetic_bit_literal ::=
X | **Z** | **L** | **H** | **U** | **W**
| **x** | **z** | **l** | **h** | **u** | **w**

dont_care_literal ::=
?

random_literal ::=
*****

based_literal ::= (see 6.8)
binary_base { **_** | binary_digit }
| octal_base { **_** | octal_digit }
| decimal_base { **_** | digit }
| hex_base { **_** | hex_digit }

binary_base ::=
'B | **'b**

binary_digit ::=
bit_literal

octal_base ::=
'O | **'o**

octal_digit ::=
binary_digit | **2** | **3** | **4** | **5** | **6** | **7**

decimal_base ::=
'D | **'d**

hex_base ::=
'H | **'h**

hex_digit ::=
octal_digit | **8** | **9** | **A** | **B** | **C** | **D** | **E** | **F** | **a** | **b** | **c** | **d** | **e** | **f**

edge_literal ::= (see 6.9)
bit_edge_literal
| word_edge_literal
| symbolic_edge_literal

bit_edge_literal ::=
bit_literal bit_literal

word_edge_literal ::=
based_literal based_literal

symbolic_edge_literal ::=
?? | **?~** | **?!** | **?-**

quoted_string ::= (see 6.10)
" { any_character } **"**

identifiers ::= (see 6.11)
identifier { identifier }

identifier ::=
nonescaped_identifier
| escaped_identifier
| placeholder_identifier
| hierarchical_identifier

nonescaped_identifier ::= (see 6.11.1)
nonreserved_character { nonreserved_character }

escaped_identifier ::= (see 6.11.2)
    escape_character escaped_characters
escaped_characters ::=
    escaped_character { escaped_character }
escaped_character ::=
      nonreserved_character
    | reserved_character
    | escape_character
placeholder_identifier ::= (see 6.11.3)
    < nonescaped_identifier >
hierarchical_identifier ::= (see 6.11.4)
    identifier . { identifier . } identifier
arithmetic_values ::= (see 6.6.1)
    arithmetic_value { arithmetic_value }
arithmetic_value ::=
      number
    | identifier
    | pin_value
string_value ::= (see 6.6.2)
      quoted_string
    | identifier
edge_values ::= (see 6.6.3)
    edge_value { edge_value }
edge_value ::=
    (edge_literal)
index_value ::= (see 6.6.4)
      unsigned
    | identifier

A.2 Auxiliary definitions

index ::= (see 7.1.1)
      [index_range]
    | [index_value]
index_range ::= (see 7.1.2)
    index_value : index_value
pin_assignments ::= (see 7.2.1)
    pin_assignment { pin_assignment }
pin_assignment ::=
    pin_variable = pin_value ;
pin_variables ::= (see 7.2.2)
    pin_variable { pin_variable }
pin_variable ::=
    *pin_variable*_identifier [index]
pin_values ::= (see 7.2.3)
    pin_value { pin_value }
pin_value ::=
      pin_variable
    | bit_literal
    | based_literal
    | unsigned

```
annotation ::=                                                    (see 7.3.1)
        one_level_annotation
      | two_level_annotation
      | multi_level_annotation
one_level_annotations ::=
        one_level_annotation { one_level_annotation }
one_level_annotation ::=
        single_value_annotation
      | multi_value_annotation
single_value_annotation ::=
        identifier = annotation_value ;
multi_value_annotation ::=
        identifier { annotation_values }
two_level_annotations ::=
        two_level_annotation { two_level_annotation }
two_level_annotation ::=
        one_level_annotation
      | identifier [ = annotation_value ]
          { one_level_annotations }
multi_level_annotations ::=
        multi_level_annotation { multi_level_annotation }
multi_level_annotation ::=
        one_level_annotation
      | identifier [ = annotation_value ]
          { multi_level_annotations }
annotation_values ::=                                             (see 7.3.2)
        annotation_value { annotation_value }
annotation_value ::=
        index_value
      | string_value
      | edge_value
      | pin_value
      | arithmetic_value
      | boolean_expression
      | control_expression
all_purpose_items ::=                                             (see 7.19)
        all_purpose_item { all_purpose_item }
all_purpose_item ::=
        include
      | alias
      | constant
      | attribute
      | property
      | class_declaration
      | keyword_declaration
      | group_declaration
      | template_declaration
      | template_instantiation
      | annotation
      | arithmetic_model
      | arithmetic_model_container
```

A.3 Generic definitions

include ::= (see 8.1)
INCLUDE quoted_string **;**

alias ::= (see 8.1)
ALIAS identifier **=** identifier **;**

constant ::= (see 8.2)
CONSTANT identifier **=** arithmetic_value **;**

attribute ::= (see 8.4)
ATTRIBUTE { identifiers **}**

property ::= (see 8.5)
PROPERTY [identifier] **{** one_level_annotations **}**

class_declaration ::= (see 8.3)
CLASS identifier **;**
| **CLASS** identifier **{** all_purpose_items **}**

keyword_declaration ::= (see 8.4)
KEYWORD context_sensitive_keyword **=** *syntax_item*_identifier **;**

group_declaration ::= (see 8.6)
GROUP *group*_identifier **{** annotation_values **}**
| **GROUP** *group*_identifier **{** index_value **:** index_value **}**

template_declaration ::= (see 8.7)
TEMPLATE *template*_identifier **{** template_items **}**

template_items ::=
template_item { template_item }

template_item ::=
all_purpose_item
| cell
| library
| node
| pin
| pin_group
| primitive
| sublibrary
| vector
| wire
| antenna
| array
| blockage
| layer
| pattern
| port
| rule
| site
| via
| function
| non_scan_cell
| test
| range
| artwork
| from
| to
| illegal
| violation
| header

| table
| equation
| arithmetic_submodel
| behavior_item
| geometric_model

template_instantiation ::=
static_template_instantiation
| dynamic_template_instantiation

static_template_instantiation ::=
*template*_identifier [**= static**] **;**
| *template*_identifier [**= static**] **{** annotation_values **}**
| *template*_identifier [**= static**]**{** one_level_annotations **}**

dynamic_template_instantiation ::=
*template*_identifier **= dynamic**
{ dynamic_template_instantiation_items **}**

dynamic_template_instantiation_items ::=
dynamic_template_instantiation_item
{ dynamic_template_instantiation_item }

dynamic_template_instantiation_item ::=
one_level_annotation
| arithmetic_model

A.4 Library definitions

library ::= (see 9.1)
LIBRARY *library*_identifier **{** library_items **}**
| **LIBRARY** *library*_identifier **;**
| *library*_template_instantiation

library_items ::=
library_item { library_item }

library_item ::=
sublibrary
| sublibrary_item

library ::=
SUBLIBRARY *sublibrary*_identifier **{** sublibrary_items **}**
| **SUBLIBRARY** *sublibrary*_identifier **;**
| *sublibrary*_template_instantiation

sublibrary_items ::= (see 9.2.2)
sublibrary_item { sublibrary_item }

sublibrary_item ::=
all_purpose_item
| cell
| primitive
| wire
| layer
| via
| rule
| antenna

| array
| site

*INFORMATION*_two_level_annotation ::= (see 9.2.3)
INFORMATION { *information*_one_level_annotations **}**

*information*_one_level_annotations ::=
*information*_one_level_annotation
{ *information*_one_level_annotation }

*information*_one_level_annotation ::=
*AUTHOR*_one_level_annotation
| *VERSION*_one_level_annotation
| *DATETIME*_one_level_annotation
| *PROJECT*_one_level_annotation

cell ::= (see 9.3.1)
CELL *cell*_identifier **{** cell_items **}**
| **CELL** *cell*_identifier **;**
| *cell*_template_instantiation

cell_items ::=
cell_item { cell_item }

cell_item ::=
all_purpose_item
| pin
| pin_group
| primitive
| function
| non_scan_cell
| test
| vector
| wire
| blockage
| artwork

non_scan_cell ::= (see 9.41)
NON_SCAN_CELL { unnamed_cell_instantiations **}**
| **NON_SCAN_CELL =** unnamed_cell_instantiation
| *non_scan_cell*_template_instantiation

unnamed_cell_instantiations ::=
unnamed_cell_instantiation { unnamed_cell_instantiation }

unnamed_cell_instantiation ::=
*cell*_identifier **{** pin_values **}**
| *cell*_identifier **{** pin_assignments **}**

pin ::= (see 9.4.1)
PIN [**[** index_range **]**] *pin*_identifier [**[** index_range **]**] **{** pin_items **}**
| **PIN** [**[** index_range **]**] *pin*_identifier [**[** index_range **]**] **;**
| *pin*_template_instantiation

pin_item ::=
all_purpose_item
| range
| port
| pin_instantiation

pin_items ::=
pin_item { pin_item }

pin_instantiation ::=
pin_variable **{** pin_items **}**

range ::= (see 9.42)
RANGE { index_range **}**

pin_group ::= (see 9.8)
PIN_GROUP [**[** index_range **]**] *pin_group*_identifier **{** pin_group_items **}**
| *pin_group*_template_instantiation

```
pin_group_items ::=
        pin_group_item { pin_group_item }
pin_group_item ::=
         all_purpose_item
        | range
wire ::=                                                    (see 9.5.1)
         WIRE wire_identifier { wire_items }
        | WIRE wire_identifier ;
        | wire_template_instantiation
wire_items ::=
        wire_item { wire_item }
wire_item ::=
         all_purpose_item
        | node
node ::=                                                    (see 9.13)
         NODE node_identifier { node_items }
        | NODE node_identifier ;
        | node_template_instantiation
node_items ::=
        node_item { node_item }
node_item ::=
        all_purpose_item
vector ::=                                                  (see 9.14)
         VECTOR control_expression { vector_items }
        | VECTOR control_expression ;
        | vector_template_instantiation
vector_items ::=
        vector_item { vector_item }
vector_item ::=
         all_purpose_item
        | illegal
illegal ::=                                                 (see 9.6.2)
         ILLEGAL { illegal_items }
        | illegal_template_instantiation
illegal_items ::=
        illegal_item { illegal_item }
illegal_item ::=
         all_purpose_item
        | violation
layer ::=                                                   (see 9.16)
         LAYER layer_identifier { layer_items }
        | LAYER layer_identifier ;
        | layer_template_instantiation
layer_items ::=
        layer_item { layer_item }
layer_item ::=
        all_purpose_item
via ::=                                                     (see 9.8.1)
         VIA via_identifier { via_items }
        | VIA via_identifier ;
        | via_template_instantiation
via_items ::=
        via_item { via_item }
```

```
via_item ::=
        all_purpose_item
      | pattern
      | artwork
via_instantiations ::=
      via_instantiation { via_instantiation }
via_instantiation ::=
      via_identifier instance_identifier { geometric_transformations }
rule ::=                                                    (see 9.21)
        RULE rule_identifier { rule_items }
      | RULE rule_identifier ;
      | rule_template_instantiation
rule_items ::=
      rule_item { rule_item }
rule_item ::=
        all_purpose_item
      | pattern
      | via_instantiation
antenna ::=                                                 (see 9.22)
        ANTENNA antenna_identifier { antenna_items }
      | ANTENNA antenna_identifier ;
      | antenna_template_instantiation
antenna_items ::=
      antenna_item { antenna_item }
antenna_item ::=
      all_purpose_item
blockage ::=                                                (see 9.23)
        BLOCKAGE blockage_identifier { blockage_items }
      | BLOCKAGE blockage_identifier ;
      | blockage_template_instantiation
blockage_items ::=
      blockage_item { blockage_item }
blockage_item ::=
        all_purpose_item
      | pattern
      | rule
      | via_instantiation
port ::=                                                    (see 9.24)
        PORT port_identifier { port_items }
      | PORT port_identifier ;
      | port_template_instantiation
port_items ::=
      port_item { port_item }
port_item ::=
        all_purpose_item
      | pattern
      | rule
      | via_instantiation

site ::=                                                    (see 9.26)
        SITE site_identifier { site_items }
      | SITE site_identifier ;
```

| *site*_template_instantiation

site_items ::=
site_item { site_item }

site_item ::=
all_purpose_item
| *ORIENTATION_CLASS*_one_level_annotation
| *SYMMETRY_CLASS*_one_level_annotation

array ::= (see 9.28)
ARRAY *array*_identifier **{** array_items **}**
| **ARRAY** *array*_identifier **;**
| *array*_template_instantiation

array_items ::=
array_item { array_item }

array_item ::=
all_purpose_item
| *PURPOSE*_single_value_annotation
| geometric_transformation

pattern ::= (see 9.30)
PATTERN *pattern*_identifier **{** pattern_items **}**
| **PATTERN** *pattern*_identifier **;**
| *pattern*_template_instantiation

pattern_items ::=
pattern_item { pattern_item }

pattern_item ::=
all_purpose_item
| *SHAPE*_single_value_annotation
| *LAYER*_single_value_annotation
| *EXTENSION*_single_value_annotation
| *VERTEX*_single_value_annotation
| geometric_model
| geometric_transformation

artwork ::= (see 9.35)
ARTWORK = *artwork*_identifier **{** artwork_items **}**
| **ARTWORK =** *artwork*_identifier **;**
| *artwork*_template_instantiation

artwork_items ::=
artwork_item { artwork_item }

artwork_item ::=
geometric_transformation
| pin_assignment

geometric_model ::= (see 9.32)
nonescaped_dentifier [*geometric_model*_identifier]
{ geometric_model_items **}**
| *geometric_model_t*emplate_instantiation

geometric_model_items ::=
geometric_model_item { geometric_model_item }

geometric_model_item ::=
all_purpose_item
| *POINT_TO_POINT*_one_level_annotation
| coordinates

coordinates ::=
COORDINATES { *x*_number *y*_number { *x*_number *y*_number } **}**

geometric_transformations ::= (see 9.34)

geometric_transformation { geometric_transformation }

geometric_transformation ::=
*SHIFT*_two_level_annotation
| *ROTATE*_one_level_annotation
| *FLIP*_one_level_annotation
| repeat

repeat ::=
REPEAT [**=** unsigned] **{**
*shift*_two_level_annotation
[repeat]
}

function ::= (see 9.36)
FUNCTION { function_items **}**
| *function*_template_instantiation

function_items ::=
function_item { function_item }

function_item ::=
all_purpose_item
| behavior
| structure
| statetable

test ::= (see 9.37)
TEST { test_items **}**
| *test*_template_instantiation

test_items ::=
test_item { test_item }

test_item ::=
all_purpose_item
| behavior
| statetable

behavior ::= (see 9.38)
BEHAVIOR { behavior_items **}**
| *behavior*_template_instantiation

behavior_items ::=
behavior_item { behavior_item }

behavior_item ::=
boolean_assignments
| control_statement
| primitive_instantiation
| *behavior_item*_template_instantiation

boolean_assignments ::=
boolean_assignment { boolean_assignment }

boolean_assignment ::=
pin_variable **=** boolean_expression **;**

primitive_instantiation ::=
*primitive*_identifier [identifier] **{** pin_values **}**
| *primitive*_identifier [identifier]
{ boolean_assignments **}**

control_statement ::=
@ control_expression **{** boolean_assignments **}**
{ **:** control_expression **{** boolean_assignments **}** }

structure ::= (see 9.39)
STRUCTURE { named_cell_instantiations **}**
| *structure*_template_instantiation

named_cell_instantiations ::=
    named_cell_instantiation { named_cell_instantiation }
named_cell_instantiation ::=
     *cell*_identifier *instance*_identifier **{** pin_values **}**
    | *cell*_identifier *instance*_identifier **{** pin_assignments **}**
violation ::= (see 9.40)
     VIOLATION { violation_items **}**
    | *violation*_template_instantiation
violation_items ::=
    violation_item { violation_item }
violation_item ::=
     *MESSAGE_TYPE*_single_value_annotation
    | *MESSAGE*_single_value_annotation
    | behavior
statetable ::= (see 9.40)
     STATETABLE [identifier]
       { statetable_header statetable_row { statetable_row } **}**
    | *statetable*_template_instantiation
statetable_header ::=
    *input*_pin_variables **:** *output*_pin_variables **;**
statetable_row ::=
    statetable_control_values **:** statetable_data_values **;**
statetable_control_values ::=
    statetable_control_value { statetable_control_value }
statetable_control_value ::=
     bit_literal
    | based_literal
    | unsigned
    | edge_value
statetable_data_values ::=
    statetable_data_value { statetable_data_value }
statetable_data_value ::=
     bit_literal
    | based_literal
    | unsigned
    | **(** [**!**] pin_variable **)**
    | **(** [**~**] pin_variable **)**
primitive ::= (see 9.11)
     PRIMITIVE *primitive*_identifier **{** primitive_items **}**
    | **PRIMITIVE** *primitive*_identifier **;**
    | *primitive*_template_instantiation
primitive_items ::=
    primitive_item { primitive_item }
primitive_item ::=
     all_purpose_item
    | pin
    | pin_group
    | function
    | test

A.5 Control definitions

boolean_expression ::= (see 10.7)
(boolean_expression **)**
| pin_value
| boolean_unary boolean_expression
| boolean_expression boolean_binary boolean_expression
| boolean_expression **?** boolean_expression **:**
{ boolean_expression **?** boolean_expression **:** }
boolean_expression

boolean_unary ::=
!
| ~
| **&**
| **~&**
| |
| ~|
| ^
| ~^

boolean_binary ::=
&
| **&&**
| |
| ||
| ^
| ~^
| **!=**
| ==
| >=
| <=
| >
| <
| +
| -
| *
| /
| **%**
| >>
| <<

vector_expression ::= (see 10.8)
(vector_expression **)**
| vector_unary boolean_expression
| vector_expression vector_binary vector_expression
| boolean_expression **?** vector_expression **:**
{ boolean_expression **?** vector_expression **:** }
vector_expression
| boolean_expression control_and vector_expression
| vector_expression control_and boolean_expression

vector_unary ::=
edge_literal

vector_binary ::=
&
| **&&**
| **|**
| **||**
| **->**
| **~>**
| **<->**
| **<~>**
| **&>**
| **<&>**

control_and ::=
& | **&&**

control_expression ::=
(vector_expression **)**
| **(** boolean_expression **)**

A.6 Arithmetic definitions

arithmetic_expression ::= (see 11.1)
(arithmetic_expression **)**
| arithmetic_value
| [arithmetic_unary] arithmetic_expression
| arithmetic_expression arithmetic_binary
arithmetic_expression
| boolean_expression **?** arithmetic_expression **:**
{ boolean_expression **?** arithmetic_expression **:** }
arithmetic_expression
| arithmetic_macro
(arithmetic_expression { **,** arithmetic_expression } **)**

arithmetic_unary ::=
sign

arithmetic_binary ::=
+
| **-**
| *****
| **/**
| ******
| **%**

arithmetic_macro ::=
abs
| **exp**
| **log**
| **min**
| **max**

arithmetic_models ::= (see 11.2.2)
arithmetic_model { arithmetic_model }

arithmetic_model ::=
partial_arithmetic_model
| non_trivial_arithmetic_model
| trivial_arithmetic_model

```
        | assignment_arithmetic_model
        | arithmetic_model_template_instantiation
partial_arithmetic_model ::=                                        (see 11.2.3)
        nonescaped_identifier [ arithmetic_model_identifier ] { partial_arithmetic_model_items }
partial_arithmetic_model_items ::=
        partial_arithmetic_model_item { partial_arithmetic_model_item }
partial_arithmetic_model_item ::=
          any_arithmetic_model_item
        | table
non_trivial_arithmetic_model ::=                                    (see 11.2.4)
        nonescaped_identifier [ arithmetic_model_identifier ] {
           [ any_arithmetic_model_items ]
           arithmetic_body
           [ any_arithmetic_model_items ]
        }
trivial_arithmetic_model ::=                                        (see 11.2.5)
          nonescaped_identifier [ arithmetic_model_identifier ] = arithmetic_value ;
        | nonescaped_identifier [ arithmetic_model_identifier ] = arithmetic_value
            { any_arithmetic_model_items }
assignment_arithmetic_model ::=                                     (see 11.2.6)
        arithmetic_model_identifier = arithmetic_expression ;
any_arithmetic_model_items ::=                                      (see 11.2.7)
        any_arithmetic_model_item { any_arithmetic_model_item }
any_arithmetic_model_item ::=
          all_purpose_item
        | from
        | to
        | violation
arithmetic_submodels ::=                                            (see 11.3.1)
        arithmetic_submodel { arithmetic_submodel }
arithmetic_submodel ::=
          non_trivial_arithmetic_submodel
        | trivial_arithmetic_submodel
        | arithmetic_submodel_template_instantiation
non_trivial_arithmetic_submodel ::=                                 (see 11.3.2)
        nonescaped_identifier {
           [ any_arithmetic_submodel_items ]
           arithmetic_body
           [ any_arithmetic_submodel_items ]
         }
trivial_arithmetic_submodel ::=                                     (see 11.3.3)
          nonescaped_identifier = arithmetic_value ;
        | nonescaped_identifier = arithmetic_value {  any_arithmetic_submodel_items }
any_arithmetic_submodel_items ::=                                   (see 11.3.4)
        any_arithmetic_submodel_item { any_arithmetic_submodel_item }
any_arithmetic_submodel_item ::=
          all_purpose_item
        | violation
arithmetic_body ::=                                                 (see 11.4.1)
          arithmetic_submodels
        | table_arithmetic_body
        | equation_arithmetic_body
table_arithmetic_body ::=
        header table [ equation ]
```

equation_arithmetic_body ::=
[header] equation [table]

header ::= (see 11.4.2)
HEADER { identifiers **}**
| **HEADER {** header_arithmetic_models **}**
| *header*_template_instantiation

header_arithmetic_models ::=
header_arithmetic_model { header_arithmetic_model }

header_arithmetic_model ::=
non_trivial_arithmetic_model
| partial_arithmetic_model

table ::= (see 11.3.2)
TABLE { arithmetic_values **}**
| *table*_template_instantiation

equation ::= (see 11.3.3)
EQUATION { arithmetic_expression **}**
| *equation*_template_instantiation

arithmetic_model_container ::= (see 11.5)
*arithmetic_model_container*_identifier **{** arithmetic_models **}**

from ::= (see 11.4.10)
FROM { from_to_items **}**

to ::=
TO { from_to_items **}**

from_to_items ::=
from_to_item { from_to_item }

from_to_item ::=
*PIN*_single_value_annotation
| *EDGE*_single_value_annotation
| *THRESHOLD*_arithmetic_model

*EARLY*_arithmetic_model_container ::= (see 11.4.11)
EARLY { *early_late*_arithmetic_models **}**

*LATE*_arithmetic_model_container ::=
LATE { *early_late*_arithmetic_models **}**

*early_late*_arithmetic_models ::=
*early_late*_arithmetic_model { *early_late*_arithmetic_model }

*early_late*_arithmetic_model ::=
*DELAY*_arithmetic_model
| *RETAIN*_arithmetic_model
| *SLEWRATE*_arithmetic_model

Annex B

(informative)

Bibliography

[B1] Ratzlaff, C. L., Gopal, N., and Pillage, L. T., “RICE: Rapid Interconnect Circuit Evaluator,” *Proceedings of 28th Design Automation Conference*, pp. 555–560, 1991.

[B2] SPICE 2G6 User’s Guide.

[B3] Standard Delay Format Specification, Version 3.0, Open Verilog International, May 1995.

[B4] The IEEE Standard Dictionary of Electrical and Electronics Terms, Sixth Edition.

Index

B

P

Q

R

S

T

U

V

W